

1. Documentation	3
1.1 LoopBack	3
1.1.1 Using the LoopBack sample app	4
1.1.2 Creating a LoopBack application	7
1.1.2.1 Configuration file reference	13
1.1.3 Working with models and data sources	13
1.1.3.1 Exposing models over a REST API	14
1.1.3.2 Adding logic to a model	15
1.1.3.3 Creating model relations	17
1.1.3.4 Built-in models	20
1.1.3.4.1 Access control models	20
1.1.3.4.2 Access token model	26
1.1.3.4.3 Application model	26
1.1.3.4.4 Email model	29
1.1.3.4.5 User model	29
1.1.4 Data sources and connectors	33
1.1.4.1 Data Source Juggler	42
1.1.4.1.1 Datasource Juggler API	44
1.1.4.2 Oracle connector	64
1.1.4.2.1 Installing the Oracle connector	67
1.1.4.2.2 Oracle connector API	69
1.1.4.2.3 Oracle discovery API	76
1.1.4.3 MySQL connector	78
1.1.4.3.1 MySQL connector API	85
1.1.4.3.2 MySQL discovery API	87
1.1.4.4 MongoDB connector	88
1.1.4.4.1 MongoDB connector API	89
1.1.4.5 REST connector	92
1.1.4.5.1 REST connector API	95
1.1.4.5.2 REST resource API	99
1.1.4.5.3 Request builder API	101
1.1.4.6 Memory connector	106
1.1.5 Authentication and authorization	107
1.1.5.1 Creating and authenticating users	109
1.1.5.2 Controlling data access	111
1.1.5.3 Access control example app	114
1.1.5.4 Advanced topics	116
1.1.6 Creating push notifications	118
1.1.6.1 Push notifications for Android apps	122
1.1.6.2 Push notifications for iOS apps	126
1.1.6.3 Push notification API	128
1.1.6.3.1 Installation API	128
1.1.6.3.2 Notification API	129
1.1.6.3.3 PushManager API	130
1.1.7 LoopBack Definition Language	131
1.1.7.1 LDL data types	132
1.1.7.2 Model definition reference	134
1.1.8 LoopBack API	142
1.1.8.1 Access context API	142
1.1.8.2 AccessToken API	144
1.1.8.3 Application API	145
1.1.8.4 ACL API	146
1.1.8.5 App API	148
1.1.8.6 Data Source	154
1.1.8.7 DataSource model API	157
1.1.8.8 Email API	157
1.1.8.9 GeoPoint API	158
1.1.8.10 LoopBack object API	159
1.1.8.11 Model	161
1.1.8.11.1 Remote methods and hooks	170
1.1.8.12 Role API	173
1.1.8.13 Token API	176
1.1.8.14 User API	176
1.1.9 REST API	178
1.1.9.1 Access token REST API	179
1.1.9.2 ACL REST API	180
1.1.9.3 Application REST API	181
1.1.9.4 Email REST API	182
1.1.9.5 Installation REST API	183
1.1.9.6 Model REST API	186
1.1.9.7 Push Notification REST API	196
1.1.9.8 Role REST API	196

1.1.9.9 User REST API	197
1.1.10 Client SDKs	199
1.1.10.1 Android SDK	200
1.1.10.2 iOS SDK	205

Documentation

StrongLoop Suite documentation

StrongLoop Suite includes:

- [LoopBack](#), an open-source mobile backend framework for Node.js.
- [StrongNode](#), professional support for Node.js, plus cluster management and other powerful modules.
- [StrongOps](#), a built-in monitoring and management console.

To get up and running quickly, see [Getting started](#).

What's new

LoopBack now enables you to send push notifications to mobile apps. See [Creating push notifications](#) for details. Client SDKs have been updated to support push notifications:

- [Android SDK](#) (version 1.2)
- [iOS SDK](#) (version 1.2)

StrongLoop now supports the [Digital Ocean](#) cloud platform.

See [What's new](#) for a complete list.

LoopBack

LoopBack is a Node.js mobile backend framework that you can run in the cloud or on-premises. For more information on the advantages of using LoopBack, see [StrongLoop | LoopBack](#).

To gain a basic understanding of key LoopBack concepts, read the following section. Then, dive right into creating an app in [Creating a LoopBack application](#).

Overview

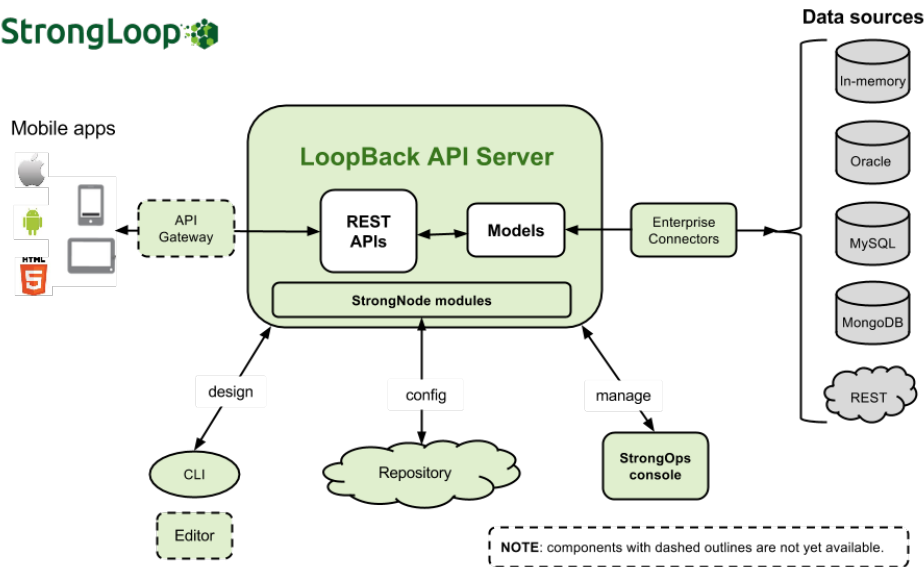
See also the [LoopBack API](#) reference.

LoopBack consists of:

- A library of Node.js modules for connecting mobile apps to data sources such as databases and REST APIs.
- A command line tool, `slc lb`, for creating and working with LoopBack applications.
- Client SDKs for native and web-based mobile clients.

As illustrated in the diagram below, a LoopBack application has three components:

- Models that represent business data and behavior.
- Data sources and connectors. Data sources are databases or other backend services such as REST APIs, SOAP web services, or storage services. Connectors provide apps access to enterprise data sources such as Oracle, MySQL, and MongoDB.
- Mobile clients using the LoopBack client SDKs.



An app 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.

Mobile clients can call LoopBack server APIs directly using [Strong Remoting](#), a pluggable transport layer that enables you to provide backend APIs over REST, WebSockets, and other transports.

Using the LoopBack sample app

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

- [Prerequisites](#)
- [Creating and running the sample app](#)
 - [About the sample app](#)
 - [Connecting to other data sources](#)
- [Using the API Explorer](#)
- [Next steps](#)

Prerequisites

You must have the `git` command-line tool installed to run the sample application. If needed, download it at <http://git-scm.com/downloads> and install it.

On Linux systems, you must have root privileges to write to `/usr/share`.



If you don't have a C compiler (Visual C++ on Windows or XCode on OSX) and command-line "make" tools installed, you will see errors such as these:

```
xcode-select: Error: No Xcode is selected. Use xcode-select -switch <path-to-xcode>, or see
the xcode-select manpage (man xcode-select) for further information.
...
Unable to load native module uvmon; some features may be unavailable without compiling it.
memwatch must be installed to use the instances feature StrongOps not configured to monitor.
...
```

You will still be able to run the sample app, but StrongOps will not be able to collect certain statistics.

Creating and running the sample app

If you have not already done so, follow these steps to run the LoopBack sample app:

1. Follow the instructions in [Getting started](#) to install Node on your local machine or cloud platform.
2. Create the sample app with this command.

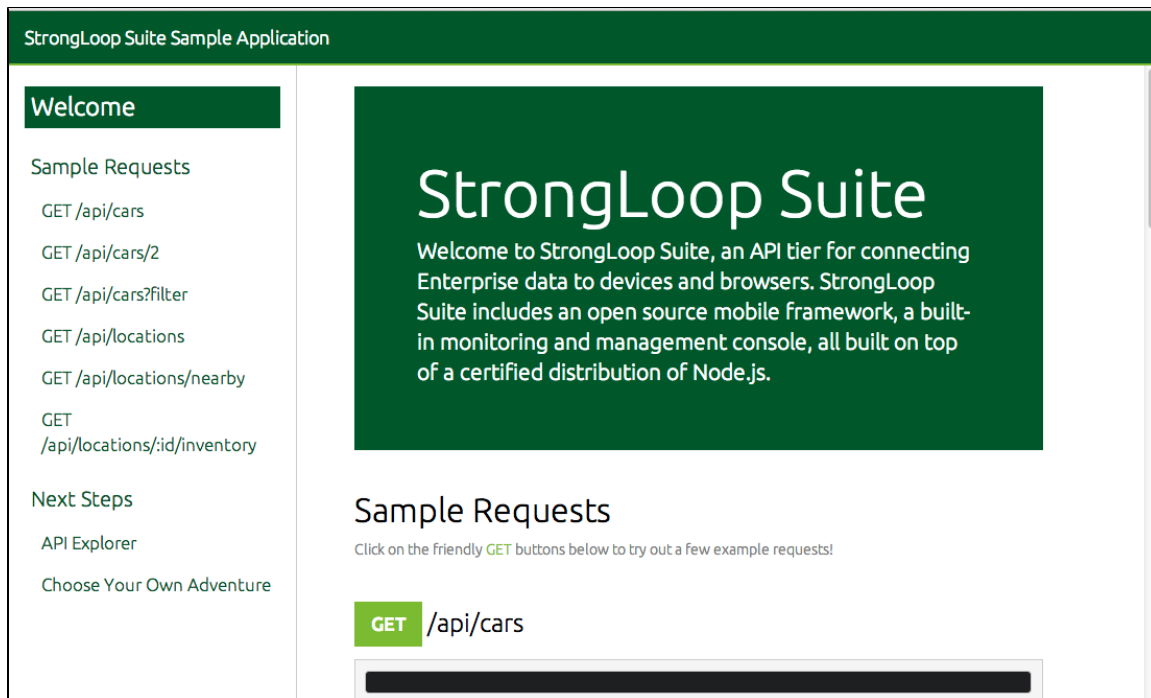
```
$ slc example
```

This command creates the sample app in a new directory (by default `sls-sample-app`) and installs all of its dependencies. See [slc example](#) for more information.

3. Run the sample application by entering these commands:

```
$ cd sls-sample-app
$ slc run app
```

4. 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 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.

Using the API Explorer

Follow these steps to explore the sample 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 screenshot shows the StrongLoop API Explorer interface. At the top, there's a green header with the StrongLoop logo and the text "StrongLoop API Explorer". To the right of the header is a search bar containing "/api/swagger/resources" and a green "Explore" button. Below the header, there's a table listing REST API endpoints grouped by model names. The models listed are /cars, /customers, /inventory, /locations, and /notes. Each model has a row with links for "Show/Hide", "List Operations", "Expand Operations", and "Raw". At the bottom of the table, there's a note: "[BASE URL: http://demo.strongloop.com:3000/api]".

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.

The screenshot shows the StrongLoop API Explorer interface for the `/locations` model. At the top, there's a green header with the StrongLoop logo and the text "StrongLoop API Explorer". To the right of the header is a search bar containing "/api/swagger/resources" and a green "Explore" button. Below the header, there's a table listing REST API endpoints grouped by model names. The models listed are /cars, /customers, /inventory, /locations, and /notes. Each model has a row with links for "Show/Hide", "List Operations", "Expand Operations", and "Raw". At the bottom of the table, there's a note: "[BASE URL: http://demo.strongloop.com:3000/api]".

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.

The screenshot shows the StrongLoop API Explorer interface for the `GET /locations/{id}` operation. The interface displays the following information:

- HTTP Verb:** GET
- Path:** /locations/{id}
- Description:** Find a model instance by id from the data source
- Response Class:** Model
- Model:** Model Schema
- any:** any
- Response Content Type:** application/json
- Parameters:** A table with columns: Parameter, Value, Description, Parameter Type, Data Type. The table contains one row: id, 2, Model id, path, any.
- Buttons:** Try it out! (highlighted), Hide Response

4. Invoke an operation: fill in the parameters, then click the Try it out! button. You'll see something like this:

Request URL
http://localhost:3000/locations/2
Response Body
<pre>{ "id": "2", "street": "390 Lang Road", "city": "Burlingame", "zipcode": "94010", "name": "Bay Area Firearms", "geo": { "lat": 37.5074391, "lng": -122.3301437 }, "state": "CA" }</pre>
Response Code
200
Response Headers
<pre>Access-Control-Allow-Origin: * Date: Tue, 17 Sep 2013 06:03:03 GMT Connection: keep-alive X-Powered-By: Express Content-Length: 199 Content-Type: application/json; charset=utf-8</pre>

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

Next steps

To gain a deeper understanding of LoopBack and how it works, read [Working with Models and data sources](#) and [Data sources and connectors](#).

Creating a LoopBack application

- Creating a new application
 - Create a new application
 - Run empty application
 - A quick look under the hood
 - Examining the application code
- Initializing the application
- Creating models
 - Creating a model using slc
 - Defining a model manually
- Getting a reference to models
- Attaching a model to a data source
 - Add the data source
 - Add data source credentials
 - Make the model use the data source connector
 - Setting the data source connector programmatically
- Exposing models over a REST API
- Sanitizing and validating models

This article walks through some of the initial steps to create your own LoopBack application.

Creating a new application



If you've already followed the steps in [Getting started](#) to create a new (empty) LoopBack application, you can skip this section and proceed to [Initializing the application](#).

Create a new application

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

```
$ slc lb project myapp
```

Replace "myapp" with the name of your application. This command creates a new directory called `myapp` (or the application 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 app
```

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

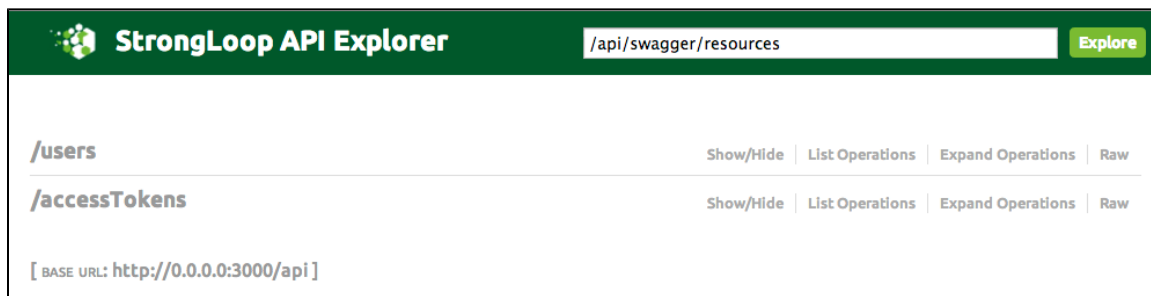
Now, load this URL in your browser: <http://0.0.0.0:3000/>

You'll see the basic JSON data returned by the empty app; something like this:

```
{
  "started": "2014-01-04T00:00:58.383Z",
  "uptime": 15.991
}
```

Load this URL in your browser to view the API explorer: <http://0.0.0.0:3000/explorer>.

The API explorer will show you the REST APIs of the built-in user and access token models.



A quick look under the hood

As mentioned before, `slc` creates a number of files and directories for a LoopBack application; specifically:

- `app.js` - application main program file.
- `app.json` - configuration file that specifies the TCP port and IP address where the application listens.
- `datasources.json` - data source configuration file.
- `/models` directory - The application will load all JavaScript files in this directory when it starts. See [Initializing the application](#) for details.
- `models.json` - Contains model definitions created with `slc lb model`. You can also edit this file manually to add models. See [Creating models](#) for more information.
- `/node_modules` directory, containing a directory for each Node module upon which LoopBack depends by default. See [npm-folders](#) for more information.
- `package.json` - application metadata file that identifies the project, lists project dependencies, and provides other metadata such as a description, version and so on. For more information, see [package.json](#).

Examining the application code

Open `app.js` in your preferred editor. You'll see the following extension points marked; these are places where you can add your own application code:

- Request pre-processing
- Request handlers
- Error handling

Initializing the application

The `app.boot()` method sets the application root directory; the function takes an optional argument that is either a string or an object:

- If string, then it uses the argument as the application root directory.
- If object, then it looks for `model`, `dataSources`, and `appRootDir` properties of the object. If the object has no `appRootDir` property then it sets the [current working directory](#) as the application root directory.

The application looks in the application root directory for `/models` and `/boot` directories. If the argument to `app.boot()` is an object, then it uses the `models` and `dataSource` properties to define models and data sources.

Then it:

1. Creates data sources using the `datasources.json` file in the application directory (or the `options.dataSources` object if provided in the argument).
2. Creates models using the `models.json` file in the application directory (or the `options.models` object if provided in the argument).
3. Loads all JavaScript files in the `appRootDir/models` directory with `require()`.
4. Loads all JavaScript files in the `appRootDir/boot` directory with `require()`.

The `slc lb project` command puts a call to `app.boot(__dirname)` in the main `app.js` program file. Note that `__dirname` is the Node global variable for the directory that contains the current script.



Don't combine using `app.boot()` and `app.model(name, config)` in multiple files because it may result in models being undefined due to race conditions. To avoid this when using `app.boot()` make sure all models are passed as part of the models definition.

Creating models

A LoopBack model displays information to the user and interacts with backend systems. If your app connects to a relational database, then a model generally corresponds to a table. See [Working with models and data sources](#) for more information. You can create a model using the `slc` command line tool and manually in JavaScript.

Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

Creating a model using slc

The easiest way to create a model is with the `slc` command-line tool. 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 `slc` in interactive mode:

```
$ slc lb model book -i
```

You'll be prompted to enter details:

```
Plural Name (books):
```

Simply press RETURN here, since you want to use the default plural "books."

```
Property Names:
Example:
  title author description totalPages genre
```

Enter the property (field) names, separated by spaces, all on a single line; You can enter the suggested names or your own.

Next, you'll be prompted for the data type of each of the fields:

```
Select a type for the property "title":
```

1. string
2. number
3. boolean
4. object
5. array
6. buffer
7. date

```
Type the number or name. Type enter to select the default (string).
```

Following the example, you can press RETURN to accept the default (string) for all the properties except `totalPages`; for that, enter 2 or type 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.

Defining a model manually

Instead of using the `slc` tool, you can create a model programmatically. 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` model 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 models

Once you've created your models (either with `models.json` or programmatically), 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 `models.json`:

```
var loopback = require('loopback'); var app = module.exports = loopback(); ... myModel = app.myModelName;
```

Then `myModel` will contain a reference to the model called `myModelName` created in `models.json`.

For example, if a model named "products" is defined in `models.json`, then you can get a reference to it as follows:

```
productModel = app.products;
```

To get all models, including those created programmatically, instead use:

```
loopback.getModel( 'myModelName' );
```

Attaching a model to a data source

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, `slic` creates and uses the [memory connector](#), which is suitable for development. To use a different datasource:

1. Use `slic lb datasource` to add the data source to `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 the data source

To add a new data source to an application, use this command:

```
slic lb datasource ds_name --connector conn_name
```

Where *ds_name* is the name of the new datasource to create and *conn_name* is the name of the connector to use: "mysql", "mongodb", "oracle", "memory", or "rest". For example:

```
$ slic lb datasource mysql --connector mysql
```

This adds an entry such as the following to `datasources.json`:

```
"mysql1": {  
  "connector": "mysql"  
}
```

This example creates a MySQL data source called "mysql1". The initial 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:

```
"mysql1": {  
  "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 connector

Edit `models.json` to set the connector 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 "mysql":

```

"book": {
  "properties": {
    "title": {
      "type": "string"
    },
    "author": {
      "type": "string"
    },
    "desc": {
      "type": "string"
    },
    "pgs": {
      "type": "number"
    },
    "genre": {
      "type": "string"
    }
  },
  "public": true,
  "dataSource": "mysql",
  "plural": "books"
}

```

Then the books model would use the MySQL connector instead of the memory connector.

Setting the data source connector programmatically

For example, as shown below, the [MongoDB Connector](#), mixes in a `create` method that you can use to store a new product in the database.

```

// Attach data sources
var db = loopback.createDataSource({
  connector: require('loopback-connector-mongodb')
});

// Enable the model to use the MongoDB API
Product.attachTo(db);

// Create a new product in the database
Product.create({ name: 'widget', price: 99.99 }, function(err, widget) {
  console.log(widget.id); // The product's id, added by MongoDB
});

```

Exposing models over a REST API

To expose a model to mobile clients, use one of LoopBack's remoting middleware modules. This example uses the `app.rest` middleware to expose the `Product` Model's API over REST. For more details, see [Exposing models over a REST API](#).

For more information on LoopBack's REST API, see [REST API](#).

```
// Create a LoopBack application
var app = loopback();

// Use the REST remoting middleware
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.

For information on creating remote methods in general, see [Defining remote methods](#).

Sanitizing and validating models

A *schema* provides a description of a model written in LoopBack Definition Language (LDL), a specific form of JSON. Once a schema is defined for a model, the model validates and sanitizes data before passing it on to a data store such as a database. A model with a schema is referred to as a *static 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.

Configuration file reference



Work in progress. Not ready for public yet.

When you create an application with the `slc lb project` command, the tool creates several configuration files:

- **app.json** - general application configuration
- **datasources.json** - data source configuration
- **models.json** - model configuration
- **package.json** - application dependencies.

Working with models and data sources

A LoopBack model consists of:

- Application data.
- Validation rules.
- Data access capabilities.
- Business logic.

Apps use the model API to display information to the user and interact with backend systems. LoopBack supports both "dynamic"

schema-less models and "static", schema-driven models.

Dynamic models require only a name. The format of the data are specified completely and flexibly by the client application. Well-suited for data that originates on the client, dynamic models enable you to persist data both between accessTokens and between devices without involving a schema.

Static models require more code up front, with the format of the data specified completely in JSON. Well-suited to both existing data and large, intricate datasets, static models provide structure and consistency to their data, preventing bugs that can result from unexpected data in the database.

For an introduction to defining a model, see [Defining a LoopBack model](#).

For more information, see:

- The model [REST API](#)
- The [LoopBack Definition Language Guide](#)
- [Node.js model API](#)
- [Built-in models](#)
- Using [remote methods](#) to expose custom behavior to clients
- [Creating model relations](#)

Exposing models over a REST API

- [Overview](#)
 - [Using the REST Router](#)
- [CRUD remote methods](#)
- [Defining remote methods](#)

Overview

The REST API enables clients to interact with LoopBack models using HTTP. A client can be a web browser, a JavaScript program, a mobile SDK, a curl script, or anything that can act as an HTTP client. 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 `/<Model.settings.plural | pluralized(Model.modelName)>`, for example, `/locations`, to the base URL such as <http://localhost:3000/>.

Using the REST Router

To expose models over REST, use the `loopback.rest` router:

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

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

CRUD remote methods

As an example, consider a simple model called `Location` (that provides rental locations) to illustrate the REST API exposed by LoopBack. For a model backed by a data source that supports create, read, update, and delete (CRUD) operations, you'll see the following endpoints:

Model API	HTTP Method	Example Path
<code>Model.create</code>	POST	<code>/locations</code>
<code>Model.upsert</code>	PUT	<code>/locations/:id</code>
<code>Model.exists</code>	GET	<code>/locations/:id/exists</code>
<code>Model.findById</code>	GET	<code>/locations/:id</code>
<code>Model.find</code>	GET	<code>/locations</code>
<code>Model.findOne</code>	GET	<code>/locations/findOne</code>
<code>Model.deleteById</code>	DELETE	<code>/locations/:id</code>
<code>Model.count</code>	GET	<code>/locations/count</code>

Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

Model.prototype.updateAttributes	PUT	/locations/:id
----------------------------------	-----	----------------

Defining remote methods

A remote method is one that can be called through a REST API endpoint. Use `loopback.remoteMethod()` to define a remote method. For example, the following examples shows how to expose a remote method, `stats`, for a model called `product` (defined in the main program file, `app.js`):

```
var loopback = require('loopback');
var app = require('../app');
var product = app.models.product;

product.stats = function(callback) {
  product.count(function(err, count) {
    if(err) {
      callback(err);
    } else {
      callback(null, {
        totalAvailable: count
      });
    }
  });
};

loopback.remoteMethod(product.stats, {
  returns: {arg: 'stats', type: 'object'}
});
```

Here's a more complex example of exposing a JavaScript method over a REST API:

```
loopback.remoteMethod(
  Location.nearby,
  {
    description: 'Find nearby locations around the geo point',
    accepts: [
      {arg: 'here', type: 'GeoPoint', required: true, description: 'geo location (lat & lng)'},
      {arg: 'page', type: 'Number', description: 'number of pages (page size=10)'},
      {arg: 'max', type: 'Number', description: 'max distance in miles'}
    ],
    returns: {arg: 'locations', root: true},
    http: {verb: 'POST', path: '/nearby'}
  }
);
```

The remoting is defined using the following properties:

- **description:** Description of the REST API
- **accepts:** An array of parameters, each parameter has a name, a type, and an optional description
- **returns:** Description of the return value
- **http:** Binding to the HTTP endpoint, including the verb and path

Adding logic to a model

- [Overview](#)
- [Creating a model constructor from a data source](#)
- [Attaching the model to a data source](#)
- [Manually adding methods to the model constructor](#)

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 model relations

You can define the following relations between models:

- [belongsTo](#)
- [hasMany](#)
- [hasMany through](#)
- [hasAndBelongsToMany](#)

Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

REVIEW COMMENT

Need more context and overview, e.g. description of:

- Why create model relations
- How model relations are used

belongsTo

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.



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

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

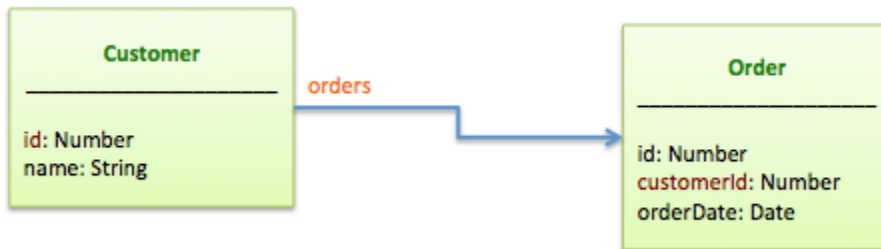
Order.belongsTo(Customer);
```

The code above basically says Order has a reference called `customer` to User using the `customerId` property of Order as the foreign key. Now we can access the customer in one of the following styles:

```
order.customer(callback); // Get the customer for the order
order.customer(); // Get the customer for the order synchronously
order.customer(customer); // Set the customer for the order
```

hasMany

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 containing customers and orders, a customer has zero or more orders.



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

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

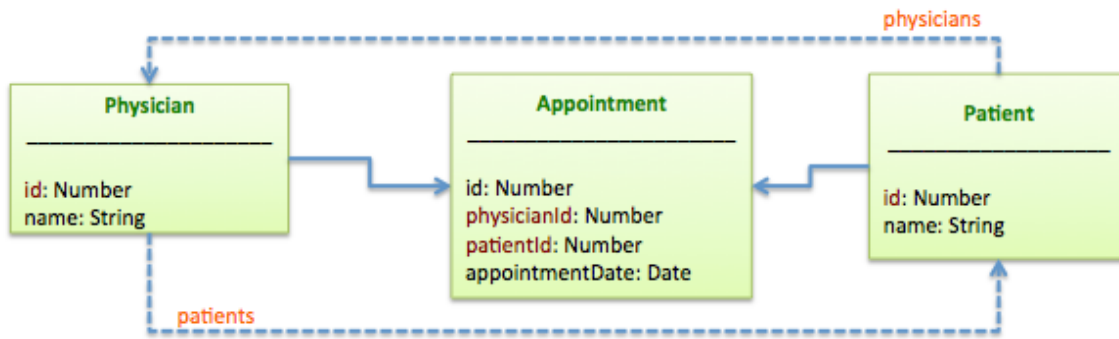
Customer.hasMany(Order, {as: 'orders', foreignKey: 'customerId'});
```

Scope methods created on the base model by `hasMany` allows to build, create and query instances of other class. For example:

```
customer.orders(filter, callback); // Find orders for the customer
customer.orders.build(data); // Build a new order
customer.orders.create(data, callback); // Create a new order for the customer
customer.orders.destroyAll(callback); // Remove all orders for the customer
customer.orders.findById(orderId, callback); // Find an order by id
customer.orders.destroy(orderId, callback); // Delete an order by id
```

hasMany through

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, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:



```

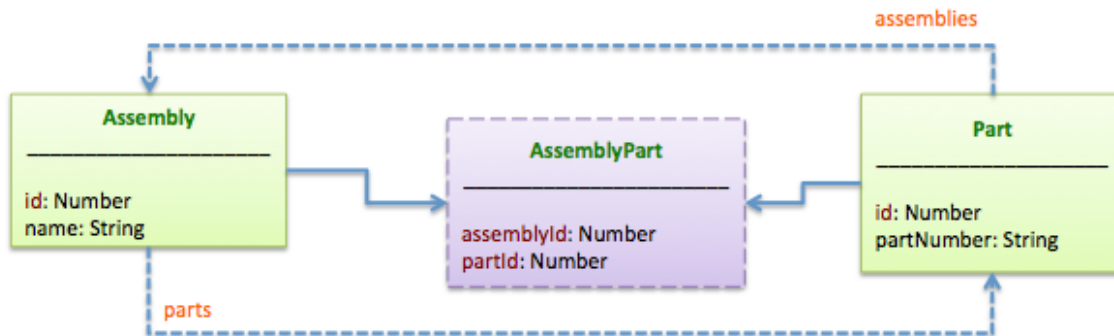
var Physician = ds.createModel('Physician', {name: String});
var Patient = ds.createModel('Patient', {name: String});
var Appointment = ds.createModel('Appointment', {
  physicianId: Number,
  patientId: Number,
  appointmentDate: Date
});

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
  
```

hasAndBelongsToMany

A `hasAndBelongsToMany` relation creates a direct many-to-many connection with another model, with no intervening model. For example, if your application includes users and groups, with each group having many users and each user appearing in many groups, you could declare the models this way,



```
User.hasAndBelongsToMany('groups', {model: Group, foreignKey: 'groupId'});
user.groups(callback); // get groups of the user
user.groups.create(data, callback); // create a new group and connect it with the user
user.groups.add(group, callback); // connect an existing group with the user
user.groups.remove(group, callback); // remove the user from the group
```

Built-in models

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 token model** - identify users by creating access tokens when they connect to the Loopback app.
- **Access control models** - ACL, 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.

Defining a model with `loopback.createModel()` is really just extending the base `loopback.Model` type using `loopback.Model.extend()`. The bundled models extend from the base `loopback.Model` so you can extend them arbitrarily.

Access control models

- **ACL model**
 - Properties
 - Methods
 - `checkPermission()`
 - `checkAccess()`
 - `checkAccessForToken()`
 - `create()`
- **Role model**
 - Properties
 - Methods
 - `isInRole()`
 - `getRoles()`
 - `registerResolver()`
- **Scope model**
 - Properties
 - Methods
 - `checkPermission()`
- **RoleMapping**
 - Properties

REVIEW COMMENT

Do we need to expose this API documentation externally? Maybe not that useful for external devs; maintenance issues, etc.

Needs review by Raymond - methods have been refactored so check for changes.

Do we need info on Hooks?

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?

Properties

The following table describes the properties of the ACL object:

Property	Type	Description
----------	------	-------------

model	String	Name of the property, method, scope, or relation
property	String	Name of the model property
accessType	String	Type of access: Valid values are: <ul style="list-style-type: none"> • READE • WRITE • EXEC
permission	String	Type of permission granted. Valid values are: <ul style="list-style-type: none"> • 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.
principalType	String	Type of the principal - Application/User/Role
principalId	String	Id of the principal - such as appld, userId or roleId

ACL model definition

```
{
  model: String, // The name of the model
  property: String, // The name of the property, method, scope, or relation
  accessType: String, // Type of access: READ, WRITE, or EXEC
  permission: String, // Type of permission: ALARM, ALLOW, AUDIT, or DENY
  principalType: String, // Type of principal, e.g. application, user, role
  principalId: String
};
```

Methods

The ACL object has several relevant methods:

- `ACL.checkPermission()`
- `ACL.checkAccess()`
- `ACL.checkAccessForToken()`

checkPermission()

The `ACL.checkPermission()` method checks if the given principal is allowed to access the model/property.

```
ACL.checkPermission(principalType, principalId, model, property, accessType, callback);
```

REVIEW COMMENT

Does this function return a value that indicates whether permission to the model is allowed? Or how does it indicate whether access is allowed?

The following table describes the method's parameters:

Parameter	Description	Type
-----------	-------------	------

principalType	Principal type	String
principalId	Principal ID	String
model	Model name	String
property	Property/method/relation name	String
callback	Callback function. Parameters: <ul style="list-style-type: none"> err: Error object or string Object: access permission 	Function

checkAccess()

The `ACL.checkAccess()` method checks if the request has the permission to access.

```
ACL.checkAccess(context, callback);
```

The following table describes the method's parameters:

Parameter	Type	Description
context	Object	See properties below
<ul style="list-style-type: none"> context.model 	String or Model	Model name or class
<ul style="list-style-type: none"> context.id 	*	Model instance ID
<ul style="list-style-type: none"> context.property 	String	Property/method/relation name
<ul style="list-style-type: none"> context.accessType 	String	Access type
callback	Function	Callback function

checkAccessForToken()

The `ACL.checkAccessForToken()` method checks if the given access token can invoke the specified method.

```
ACL.checkAccessForToken(token, model, modelId, method, callback);
```

The following table describes the method's parameters:

Parameter	Type	Description
token	String	Access token
model	String	Model name
modelId	*	Model ID
method	String	Method name
callback	Function	Callback function. Parameters: <ul style="list-style-type: none"> err: Error object or string allowed: Is request allowed (Boolean)

create()

The `create()` method creates a new ACL instance.

REVIEW COMMENT

Is above an accurate and useful description?

```

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) { }
                                                })
            })

```

Role model

Properties

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

The following JSON defines the properties of the role model:

Role model definition

```

{
  id: {type: String, id: true}, // Id
  name: {type: String, required: true}, // The name of a role
  description: String, // Description
  // Timestamps
  created: {type: Date, default: Date},
  modified: {type: Date, default: Date}
};

```

REVIEW COMMENT

Does below table have correct column headings? I wasn't clear what \$owner, etc, actually represent.

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

Methods

The Role object has several relevant methods:

- `Role.isInRole()` - checks if a principal is in a role.
- `Role.getRoles()` - returns the roles for the specified principal.
- `Role.registerResolver()` - adds a custom handler function for roles.

isInRole()

The `isInRole()` method checks if a principal is in the specified role.

```
Role.isInRole(role, context, callback);
```

The following table describes the method's parameters:

Parameter	Type	Description
role	String	Name of the role
context	Object	Context object
callback	Function	callback function

getRoles()

The `getRoles()` method returns the roles for the specified principal.

```
Role.getRoles = function (principalType, principalId, callback);
```

The following table describes the method's parameters:

Parameter	Type	Description
principalType	String	Principal type
principalId	String or Number	Principal ID
callback	Function	callback function that has takes parameters: err and an an array of role IDs

registerResolver()

The `registerResolver()` method adds a custom handler function for roles.

```
Role.registerResolver = function(role, resolver);
```

The following table describes the method's parameters:

Parameter	Type	Description
role		
resolver	Function	Function that determines if a principal is in the role. It has the following signature: function(role, context, callback)

Scope model

Properties

The following table describes the properties of the Scope model:

Property	Type	Description
name	String	Scope name; required
description	String	Description of the scope

The Scope model has ID, name, description properties:

Scope model definition

```
{
  name: {type: String, required: true},
  description: String
};
```

Methods

checkPermission()

The `Scope.checkPermission()` method checks if the given scope is allowed to access the model/property:

```
Scope.checkPermission(scope, model, property, accessType, callback);
```

The following table describes the method's parameters:

Parameter	Type	Description
scope	String	Scope name
model	String	Model name
property	String	Property/method/relation name
accessType	String	Access type
callback	Function	Callback function. Parameters: <ul style="list-style-type: none">err: Error object or stringObject: access permission

RoleMapping

A RoleMapping entry maps one or more principals to one role. A RoleMapping entry belongs to one role, based on the roleId property.

Properties

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

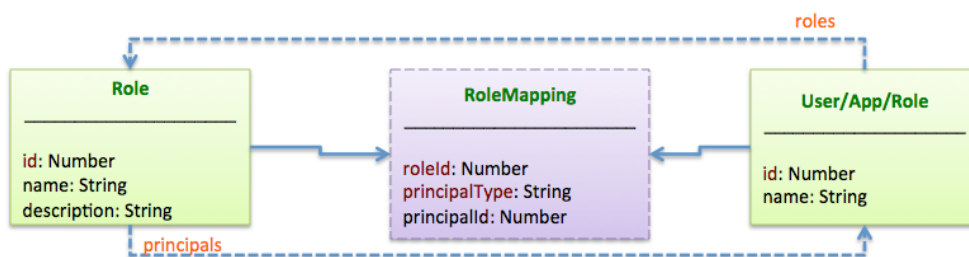
The following JSON defines the properties of the role model:

RoleMapping model definition

```
{
  id: {type: String, id: true}, // Id
  roleId: String, // The role id
  principalType: String, // The principal type, such as user, application, or role
  principalId: String // The principal id
};
```

REVIEW COMMENT

Does RoleMapping have any useful methods?



REVIEW COMMENT

Need to redo diagram to match JSON.

Access token model

Identify users by creating access tokens when they connect to the app.

By default, the `loopback.User` model uses the `loopback.AccessToken` model to persist access tokens. To change this, set the `accessToken` property.

```
// define a custom accessToken model
var MyAccessToken = loopback.AccessToken.extend('MyAccessToken');

// define a custom User model
var User = loopback.User.extend('user');

// use the custom accessToken model
User.accessToken = MyAccessToken;

// attach both accessToken and User to a data source
User.attachTo(loopback.memory());
MyAccessToken.attachTo(loopback.memory());
```

Application model

- Basic properties
 - User properties
 - Authorization properties
- Security keys
- Push notification settings
- Authentication schemes
 - Authentication scheme settings

- **Methods**
 - register()
 - resetKeys()
 - authenticate()

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.

Key	Type	Required?	Description
id			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

User properties

An application has the following properties linking to users:

- owner: The user id of the developer who registers the application
- collaborators: An array of users ids who have permissions to work on this app

Authorization properties

REVIEW COMMENT

Do you need to do anything special to enable these?

An application has the following OAuth 2.0 settings

- url: The application url
- callbackUrls: An array of preregistered callback urls for OAuth 2.0
- permissions: An array of OAuth 2.0 scopes that can be requested by the application

Security keys

The following keys are automatically generated by the application creation process. They can be reset upon request.

- 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 settings

The application can be configured to support multiple methods of push notifications.

```

pushSettings: {
  apns: { certData: config.apnsCertData,
    keyData: config.apnsKeyData,
    production: false, // Development mode
    pushOptions: { /* Extra options can go here for APN */ },
    feedbackOptions: { batchFeedback: true, interval: 300 }
  },
  gcm: { serverApiKey: config.gcmServerApiKey }
}

```

Authentication schemes

- authenticationEnabled
- anonymousAllowed
- authenticationSchemes

Authentication scheme settings

- scheme: Name of the authentication scheme, such as local, facebook, google, twitter, linkedin, github
- credential: Scheme-specific credentials

Methods

In addition to the CRUD methods, the Application model also has the following methods:

- register()
- resetKeys()
- authenticate()

register()

Use the `register()` method to register a new application, providing the owner user ID, application name, and other properties in the options object; for example:

```

Application.register('joeuser',
  'MyApp1',
  {description: 'My first loopback application'},
  function (err, result) { var app = result; ... });

```

See also: [Application.register](#) in API documentation.

resetKeys()

Use the `resetKeys()` method to reset keys for a given application by ID; for example:

```

Application.resetKeys(appId,
  function (err, result) { var app = result; ... });

```

See also: [Application.resetKeys](#) in API documentation.

authenticate()

Use the `authenticate()` method to authenticate an application by ID with a client key. If successful, the method calls back with the key name in the result argument. Otherwise, the `keyName` is null. For example:

```
Application.authenticate(appId,
                        clientKey,
                        function (err, keyName) { assert.equal(keyName, 'clientKey');
... });
```

See also: [Application.authenticate](#) in API documentation.

Email model

Email model

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',
    secureConnection: 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.

User model

User model

Register and authenticate users of your app locally or against third-party services.

Defining a user model

The following code sample illustrates how to extend a vanilla Loopback model using the built-in user model. You must attach both the `User` and `User.accessToken` models to a data source.

```
// create a data source
var memory = loopback.memory();

// define a User model
var User = loopback.User.extend('user');

// attach to the memory connector
User.attachTo(memory);

// also attach the accessToken model to a data source
User.accessToken.attachTo(memory);

// expose over the app's API
app.model(User);
```



By default the `loopback.User` model uses the `loopback.AccessToken` model to persist accessTokens. You can change this by setting the `accessToken` property.

Creating a new user

Create a user like 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 an access token for a user using the local authentication and authorization strategy.

```
User.login({username: 'foo', password: 'bar'}, function(err, accessToken) {
  console.log(accessToken);
});
```

REST endpoint

```
POST /users/login
```

Parameters

POST payload:

```
{
  "email": "foo@bar.com",
  "password": "bar"
}
```

Return value

```
200 OK
{
  "sid": "1234abcdefg",
  "uid": "123"
}
```

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).

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();
});
```

REST endpoint

POST /users/logout URI

Parameters

POST payload:

```
{
  "sid": "<accessToken id from user login>"
}
```

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.

```
// first setup the mail datasource (see #mail-model for more info)
var mail = loopback.createDataSource({
  connector: loopback.Mail,
  transports: [{
    type: 'smtp',
    host: 'smtp.gmail.com',
    secureConnection: true,
    port: 465,
    auth: {
      user: 'you@gmail.com',
      pass: 'your-password'
    }
  }]
});

User.email.attachTo(mail);
User.requireEmailVerification = true;
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');
});
```

REST endpoint

```
POST /users/reset-password
```

Parameters

POST payload:


```
{
  "email": "foo@bar.com"
}
...
```

Return value

200 OK

You must 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)
    });
  });
});
```

Data sources and connectors

- [Overview](#)
- [LoopBack DataSource object](#)
 - [Creating a DataSource](#)
- [Creating a model](#)
- [More DataSource features](#)
 - [Discovering model definitions from a data source](#)
 - [Synchronizing model definitions against the data source](#)
- [Connectors](#)
 - [Memory connector](#)
 - [Accessing data and services](#)
- [Building your own connectors](#)
 - [Implementing a generic connector](#)
 - [Implementing a CRUD connector](#)

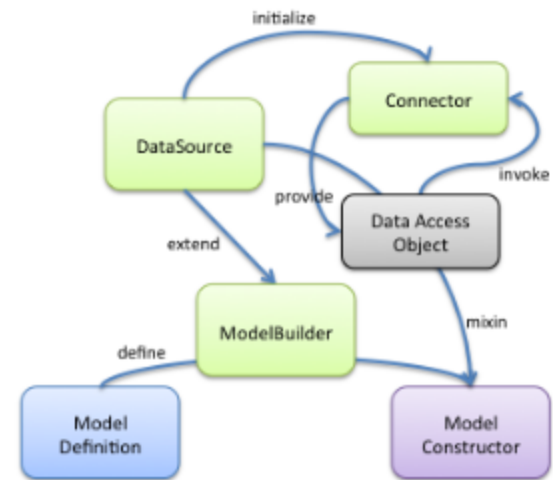
Overview

LoopBack is centered around *models* that represent data and behaviors. Data sources enable exchange of data between models and backend systems such as databases. Data sources 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 which implement the data exchange

logic using database drivers or other client APIs. In general, connectors are not used directly by application code. The `DataSource` class provides APIs to configure the underlying connector and exposes functions via `DataSource` or model classes.

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`. In addition, `DataSource` adds behaviors to model classes by mixing in methods from the `DataAccessObject` into the model class.
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.

LoopBack DataSource object

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

The `DataSource` constructor is in the `loopback-datasource-juggler` module; for example:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
DataSource constructor accepts two arguments: - connector: The name or instance of the
connector module - settings: An object of properties to configure the connector
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 refer to connector's readme file.

Creating a model

`DataSource` extends from `ModelBuilder`, which is a factory for plain model classes that only have properties. `DataSource` connected with specific databases or 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 shares same connector type and one database connection or connection pool. But it's possible to use more than one data source to connect with different databases.

Alternatively, a plain model constructor created from `ModelBuilder` can be attached 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
```

More DataSource features

In addition to data access functions mixed into the model class, `DataSource` also provides APIs to interact with the underlying backend system.

Discovering model definitions from a data source

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);

```

You can also discover and build model classes in one shot:

```

// Start with INVENTORY table and follow the primary/foreign relationships to discover
associated tables
ds.discoverAndBuildModels('INVENTORY', {visited: {}, relations: true}, function (err,
models) {

    // Now we have an object of models keyed by the model name
    // Find the 1st record for Inventory
    models.Inventory.findOne({}, function (err, inv) {
        if(err) {
            console.error(err);
            return;
        }
        console.log("\nInventory: ", inv);

        // Follow the product relation to get information about the product
        inv.product(function (err, prod) {
            console.log("\nProduct: ", prod);
            console.log("\n ----- ");
        });
    });
});

```

In addition to the asynchronous APIs, DataSource also provides the synchronous ones. Please refer to the DataSource API references.

Synchronizing model definitions against the data source

DataSource instance have two methods for updating db structure: `automigrate` and `autoupdate` for relational databases.

The `automigrate` method drop table (if exists) and create it again, `autoupdate` method generates ALTER TABLE query. Both method accepts an optional array of model names and a callback function to be called when migration/update done. If the `models` argument is not present, all models are checked.

In the following example, we create first version of the CustomerTest model, use `automigrate` to create the database table, redefine the model with second version, and use `autoupdate` to alter the database table.

```
// Create the 1st version of 'CustomerTest'
ds.createModel(schema_v1.name, schema_v1.properties, schema_v1.options);

// Create DB table for the model
ds.automigrate(schema_v1.name, function () {

    // Discover the model properties from DB table
    ds.discoverModelProperties('CUSTOMER_TEST', function (err, props) {
        console.log(props);

        // Redefine the 2nd version of 'CustomerTest'
        ds.createModel(schema_v2.name, schema_v2.properties, schema_v2.options);

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

To check if any database changes required use `isActual` method. It accepts and a callback argument, which receive boolean value depending on database state:

- false if db structure outdated
- true when `dataSource` and `db` is in sync

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

Connectors

Create a data source with a specific connector.

```
var memory = loopback.createDataSource({
  connector: loopback.Memory
});
```

The following connectors are available:

Connector	Module
In Memory	Built in to LoopBack
Oracle	http://github.com/strongloop/loopback-connector-oracle
MongoDB	http://github.com/strongloop/loopback-connector-mongodb
MySQL	http://github.com/strongloop/loopback-connector-mysql

REST	http://github.com/strongloop/loopback-connector-rest
Email	Built in to LoopBack

Installing connectors

Include the connector in your package.json dependencies and run `npm install`.

```
{
  "dependencies": {
    "loopback-connector-oracle": "latest"
  }
}
```

Memory connector

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.

Accessing data and services

The connector instance can have an optional property named as `DataAccessObject` that provides static and prototype methods to be mixed into the model constructor. `DataSource` has a built-in `DataAccessObject` to support CRUD operations. The connector can choose to use the CRUD `DataAccessObject` or define its own.

When a method is invoked from the model class or instance, it's delegated to the `DataAccessObject` which is backed by the connector.

For example:

```
User.create() --> dataSource.connector.create() --> Oracle.prototype.create()
```

Building your own connectors

LoopBack connectors provide access to backend systems including databases, REST APIs and other services. Application code does not directly

use a connector. Rather, you create a DataSource to interact with the connector.

For example:

```
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
});
```

Implementing a generic connector

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 property 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) {
};
```

Data Source Juggler

- Overview
 - Installation
- LoopBack Definition Language
- DataSource
 - Creating data sources
- LoopBack connectors

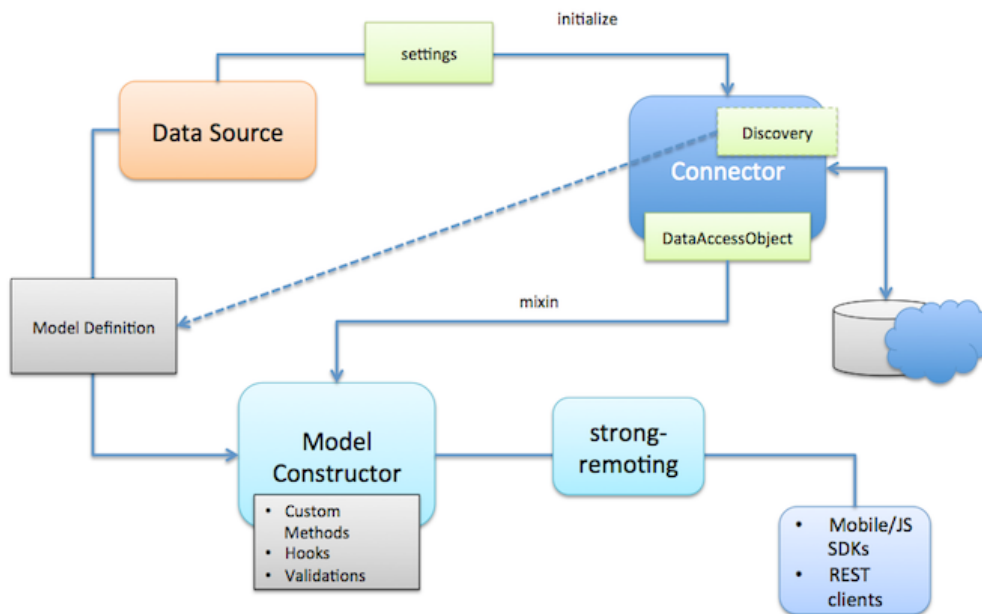
See also the [Datasource Juggler API reference](#).

The LoopBack Data Source Juggler provides a common set of interfaces for interacting with databases, REST APIs, and other data sources. It was initially forked from [JugglingDB](#).

Overview

As illustrated in the following figure, LoopBack Data Source Juggler consists of the following components:

- LoopBack Definition Language
- DataSource
- Connector



Installation

With a typical LoopBack application created with slc, you don't need to do anything additional to install Data Source Juggler. However, if you are creating an applications manually, you can install it the standard way:

```
$ npm install loopback-datasource-juggler
```

Also install the appropriated connector, for example for mongodb:

```
$ npm install loopback-connector-mongodb
```

LoopBack Definition Language

To define models use the `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. Type definition can be function representing type of property (String, Number, Date, Boolean), or object with {type: String|Number|..., index: true|false} format.
- **settings:** Object with model-wide settings such as `tableName`.

For more information, see [LoopBack Definition Language](#).

Examples of model definition:

```
var User = dataSource.define('User', {
  email: String,
  password: String,
  birthDate: Date,
  activated: Boolean
});

var User = dataSource.define('User', {
  email: { type: String, limit: 150, index: true },
  password: { type: String, limit: 50 },
  birthDate: Date,
  registrationDate: {
    type: Date,
    default: function () { return new Date }
  },
  activated: { type: Boolean, default: false }
}, {
  tableName: 'users'
});
```

DataSource

DataSource is a factory for model classes. User 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;
```

DataSource constructor accepts two arguments. First argument is connector. It could be connector name or connector package:

```
var dataSourceByConnectorName = new DataSource('memory');
var dataSourceByConnectorModule = new DataSource(require('redis'));
```

Settings

The second argument is optional settings. 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 refer to connector's README file.

LoopBack connectors

Type	Package Name
MongoDB	loopback-connector-mongodb
Oracle	loopback-connector-oracle
MySQL	loopback-connector-mysql

LoopBack connectors provide access to backend systems including databases, REST APIs and other services. Connectors are not used directly by application code. We create a `DataSource` to interact with the connector.

For example:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var oracleConnector = require('loopback-connector-oracle');

var ds = new DataSource(oracleConnector, {
  host : '127.0.0.1',
  database : 'XE',
  username : 'strongloop',
  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', which will be converted to 'loopback-connector-'
- A local module under `./connectors/` folder

Datasource Juggler API

- Base SQL API
- Data access object API
- Data source API
- Geopoint API (datasource-juggler)
- Model API
- Model builder API
- Relation API
- Validations API

Base SQL API



- `BaseSQL`
- `BaseSQL.prototype.relational`
- `baseSQL.table`
- `baseSQL.column`
- `baseSQL.columnMetadata`
- `baseSQL.propertyName`
- `baseSQL.idColumn`
- `baseSQL.idColumnEscaped`
- `baseSQL.escapeName`
- `baseSQL.tableEscaped`
- `baseSQL.columnEscaped`
- `baseSQL.save`
- `baseSQL.exists`
- `baseSQL.find`
- `baseSQL.delete`
- `baseSQL.deleteAll`

Module: loopback-datasource-juggler

- [baseSQL.count](#)
- [baseSQL.updateAttributes](#)
- [baseSQL.disconnect](#)
- [baseSQL.automigrate](#)
- [baseSQL.dropTable](#)
- [baseSQL.createTable](#)

BaseSQL()

Base class for connectors that are backed by relational databases/SQL

BaseSQL.prototype.relational

Set the relational property to indicate the backend is a relational DB

baseSQL.table(model)

Get the table name for a given model

Arguments

Name	Type	Description
model	String	The model name

baseSQL.column(model, property)

Get the column name for given model property

Arguments

Name	Type	Description
model	String	The model name
property	String	The property name

baseSQL.columnMetadata(model, property)

Get the column name for given model property

Arguments

Name	Type	Description
model	String	The model name
property	String	The property name

baseSQL.propertyName(model, column)

Get the corresponding property name for a given column name

Arguments

Name	Type	Description
model	String	The model name
column	String	The column name

baseSQL.idColumn(model)

Get the id column name

Arguments

Name	Type	Description
model	String	The model name

baseSQL.idColumnEscaped(model)

Get the escaped id column name

Arguments

Name	Type	Description
model	String	The model name

baseSQL.escapeName(name)

Escape the name for the underlying database

Arguments

Name	Type	Description
name	String	The name

baseSQL.tableEscaped(model)

Get the escaped table name

Arguments

Name	Type	Description
model	String	The model name

baseSQL.columnEscaped(model, property)

Get the escaped column name for a given model property

Arguments

Name	Type	Description
model	String	The model name
property	String	The property name

baseSQL.save(model, data, callback)

Save the model instance into the backend store

Arguments

Name	Type	Description
model	String	The model name
data	Object	The model instance data
callback	Function	The callback function

baseSQL.exists(model, id, callback)

Check if a model instance exists for the given id value

Arguments

Name	Type	Description
model	String	The model name
id	*	The id value
callback	Function	The callback function

baseSQL.find(model, id, callback)

Find a model instance by id

Arguments

Name	Type	Description
model	String	The model name
id	*	The id value
callback	Function	The callback function

baseSQL.delete

Delete a model instance by id value

Arguments

Name	Type	Description
------	------	-------------

Name	Type	Description
model	String	The model name
id	*	The id value
callback	Function	The callback function

baseSQL.deleteAll
Delete all model instances

Arguments

Name	Type	Description
model	String	The model name
callback	Function	The callback function

baseSQL.count(model, callback, where)
Count all model instances by the where filter

Arguments

Name	Type	Description
model	String	The model name
callback	Function	The callback function
where	Object	The where clause

baseSQL.updateAttributes(model, id, data, cb)
Update attributes for a given model instance

Arguments

Name	Type	Description
model	String	The model name
id	*	The id value
data	Object	The model data instance containing all properties to be updated
cb	Function	The callback function

baseSQL.disconnect()
Disconnect from the connector
baseSQL.automigrate([models], [cb])
Recreate the tables for the given models

Arguments

Name	Type	Description
[models]	[String] or String	A model name or an array of model names, if not present, apply to all models defined in the connector
[cb]	Function	The callback function

baseSQL.dropTable(model, [cb])
Drop the table for the given model

Arguments

Name	Type	Description
model	String	The model name

[cb]	Function	The callback function
--------	----------	-----------------------

baseSQL.createTable(model, [cb])
 Create the table for the given model

Arguments

Name	Type	Description
model	String	The model name
[cb]	Function	The callback function

Data access object API



- [module.exports](#)
- [util](#)
- [DataAccessObject](#)
- [DataAccessObject.create](#)
- [DataAccessObject.upsert](#)
- [DataAccessObject.findOrCreate](#)
- [DataAccessObject.exists](#)
- [DataAccessObject.findById](#)
- [DataAccessObject.find](#)
- [DataAccessObject.findOne](#)
- [DataAccessObject.count](#)
- [dataAccessObject.save](#)
- [dataAccessObject.updateAttribute](#)
- [dataAccessObject.updateAttributes](#)
- [dataAccessObject.reload](#)
- [defineReadonlyProp](#)
- [DataAccessObject.scope](#)

Module: loopback-datasource-juggler

module.exports
 Module exports class Model
 util
 Module dependencies
 DataAccessObject(data)
 DAO class - base class for all persist objects provides **common API** to access any database connector. This class describes only abstract behavior layer, refer to `lib/connectors/*.js` to learn more about specific connector implementations

DataAccessObject mixes Inclusion classes methods

Arguments

Name	Type	Description
data	Object	<ul style="list-style-type: none"> • initial object data

DataAccessObject.create([optional], obj), with)
 Create new instance of Model class, saved in database

Arguments

Name	Type	Description
[optional]	data	
obj)	callback(err or	
with	called	arguments:

DataAccessObject.upsert
 Update or insert a model instance

Arguments

Name	Type	Description
data	Object	The model instance data

[callback]	Function	The callback function
------------	----------	-----------------------

`DataAccessObject.findOrCreate(query, data, cb)`

Find one record, same as `all`, limited by 1 and return object, not collection, if not found, create using data provided as second argument

Arguments

Name	Type	Description
<code>query</code>	Object	<ul style="list-style-type: none"> search conditions: <code>{where: {test: 'me'}}</code>.
<code>data</code>	Object	<ul style="list-style-type: none"> object to create.
<code>cb</code>	Function	<ul style="list-style-type: none"> callback called with (err, instance)

`DataAccessObject.exists(id, cb)`

Check whether a model instance exists in database

Arguments

Name	Type	Description
<code>id</code>	id	<ul style="list-style-type: none"> identifier of object (primary key value)
<code>cb</code>	Function	<ul style="list-style-type: none"> callback called with (err, exists: Bool)

`DataAccessObject.findById(id, cb)`

Find object by id

Arguments

Name	Type	Description
<code>id</code>	*	<ul style="list-style-type: none"> primary key value
<code>cb</code>	Function	<ul style="list-style-type: none"> callback called with (err, instance)

`DataAccessObject.find(params, callback)`

Find all instances of Model, matched by query make sure you have marked as `index: true` fields for filter or sort

Arguments

Name	Type	Description
<code>params</code>	Object	(optional)
<code>callback</code>	Function	(required) called with arguments:

`DataAccessObject.findOne(params, cb)`

Find one record, same as `all`, limited by 1 and return object, not collection

Arguments

Name	Type	Description
<code>params</code>	Object	<ul style="list-style-type: none"> search conditions: <code>{where: {test: 'me'}}</code>
<code>cb</code>	Function	<ul style="list-style-type: none"> callback called with (err, instance)

`DataAccessObject.count(where, cb)`

Return count of matched records

Arguments

Name	Type	Description
<code>where</code>	Object	<ul style="list-style-type: none"> search conditions (optional)

cb	Function	<ul style="list-style-type: none"> callback, called with (err, count)
----	----------	--

`dataAccessObject.save({validate:, obj})`

Save instance. When instance haven't id, create method called instead. Triggers: validate, save, update | create

Arguments

Name	Type	Description
<code>{validate:}</code>	options	true, throws: false} [optional]
<code>obj)</code>	callback(err or	

`dataAccessObject.updateAttribute(name, value, callback)`

Update single attribute

equals to ``updateAttributes({name: value}, cb)`

Arguments

Name	Type	Description
name	String	<ul style="list-style-type: none"> name of property
value	Mixed	<ul style="list-style-type: none"> value of property
callback	Function	<ul style="list-style-type: none"> callback called with (err, instance)

`dataAccessObject.updateAttributes(data, callback)`

Update set of attributes

this method performs validation before updating

Arguments

Name	Type	Description
data	Object	<ul style="list-style-type: none"> data to update
callback	Function	<ul style="list-style-type: none"> callback called with (err, instance)

`dataAccessObject.reload(callback)`

Reload object from persistence

Arguments

Name	Type	Description
callback	Function	<ul style="list-style-type: none"> called with (err, instance) arguments

`defineReadonlyProp(obj, key, value)`

Define readonly property on object

Arguments

Name	Type	Description
obj	Object	
key	String	
value	Mixed	

`DataAccessObject.scope()`

Define scope

Data source API



- `exports.DataSource`
- `DataSource`

Module: loopback-datasource-juggler

- `dataSource.createModel`
- `dataSource.mixin`
- `dataSource.attach`
- `dataSource.defineProperty`
- `dataSource.automigrate`
- `dataSource.autoupdate`
- `dataSource.discoverModelDefinitions`
- `dataSource.discoverModelDefinitionsSync`
- `dataSource.discoverModelProperties`
- `dataSource.discoverModelPropertiesSync`
- `dataSource.discoverPrimaryKeys`
- `dataSource.discoverPrimaryKeysSync`
- `dataSource.discoverForeignKeys`
- `dataSource.discoverForeignKeysSync`
- `dataSource.discoverExportedForeignKeys`
- `dataSource.discoverExportedForeignKeysSync`
- `dataSource.discoverSchema`
- `dataSource.discoverSchemas`
- `dataSource.discoverSchemasSync`
- `dataSource.discoverAndBuildModels`
- `dataSource.discoverAndBuildModelsSync`
- `dataSource.isActual`
- `dataSource.freeze`
- `dataSource.tableName`
- `dataSource.columnName`
- `dataSource.columnMetadata`
- `dataSource.columnNames`
- `dataSource.idColumnName`
- `dataSource.idName`
- `dataSource.idNames`
- `dataSource.defineForeignKey`
- `dataSource.disconnect`
- `dataSource.enableRemote`
- `dataSource.disableRemote`
- `dataSource.getOperation`
- `dataSource.operations`
- `dataSource.defineOperation`
- `dataSource.isRelational`
- `dataSource.ready`
- `hiddenProperty`
- `defineReadonlyProp`

`exports.DataSource`

Export public API

`DataSource(name, settings)`

`DataSource` - connector-specific classes factory.

All classes in single `dataSource` shares same connector type and one database connection

Arguments

Name	Type	Description
<code>name</code>	String	<ul style="list-style-type: none"> • type of <code>dataSource</code> connector (mysql, mongoose, oracle, redis)
<code>settings</code>	Object	<ul style="list-style-type: none"> • any database-specific settings which we need to

`dataSource.createModel`

Define a model class

Arguments

Name	Type	Description
<code>className</code>	String	
<code>properties</code>	Object	<ul style="list-style-type: none"> • hash of class properties in format
<code>settings</code>	Object	<ul style="list-style-type: none"> • other configuration of class

`dataSource.mixin(ModelCtor)`

Mixin `DataSourceObject` methods.

Arguments

Name	Type	Description
<code>ModelCtor</code>	Function	The model constructor

`dataSource.attach(modelClass)`

Attach an existing model to a data source.

Arguments

Name	Type	Description
<code>modelClass</code>	Function	The model constructor

`dataSource.defineProperty(model, prop, params)`

Define single property named `prop` on `model`

Arguments

Name	Type	Description
<code>model</code>	String	<ul style="list-style-type: none">name of model
<code>prop</code>	String	<ul style="list-style-type: none">name of property
<code>params</code>	Object	<ul style="list-style-type: none">property settings

`dataSource.automigrate(or, [cb])`

Drop each model table and re-create. This method make sense only for sql connectors.

Arguments

Name	Type	Description
<code>or</code>	String	<code>{{String}}</code> Models to be migrated, if not present, apply to all models
<code>[cb]</code>	Function	The callback function

`dataSource.autoupdate(or, [cb])`

Update existing database tables. This method make sense only for sql connectors.

Arguments

Name	Type	Description
<code>or</code>	String	<code>{{String}}</code> Models to be migrated, if not present, apply to all models
<code>[cb]</code>	Function	The callback function

`dataSource.discoverModelDefinitions(options, [cb])`

Discover existing database tables. This method returns an array of model objects, including {type, name, owner}

`options`

```
all: true - Discovering all models, false - Discovering the models owned by the current user
views: true - Including views, false - only tables
limit: The page size
offset: The starting index
```

Arguments

Name	Type	Description
<code>options</code>	Object	The options
<code>[cb]</code>	Function	The callback function

`dataSource.discoverModelDefinitionsSync(options)`
The synchronous version of `discoverModelDefinitions`

Arguments

Name	Type	Description
<code>options</code>	Object	The options

`dataSource.discoverModelProperties(modelName, options, [cb])`
Discover properties for a given model.

property description

```
owner {String} The database owner or schema
tableName {String} The table/view name
columnName {String} The column name
dataType {String} The data type
dataLength {Number} The data length
dataPrecision {Number} The numeric data precision
dataScale {Number} The numeric data scale
nullable {Boolean} If the data can be null
```

options

```
owner/schema The database owner/schema
```

Arguments

Name	Type	Description
<code>modelName</code>	String	The table/view name
<code>options</code>	Object	The options
<code>[cb]</code>	Function	The callback function

`dataSource.discoverModelPropertiesSync(modelName, options)`
The synchronous version of `discoverModelProperties`

Arguments

Name	Type	Description
<code>modelName</code>	String	The table/view name
<code>options</code>	Object	The options

`dataSource.discoverPrimaryKeys(modelName, options, [cb])`
Discover primary keys for a given owner/modelName

Each primary key column description has the following columns:

```
owner {String} => table schema (may be null)
tableName {String} => table name
columnName {String} => column name
keySeq {Number} => sequence number within primary key( a value of 1 represents the first column of the
primary key, a value of 2 would represent the second column within the primary key).
pkName {String} => primary key name (may be null)

The owner, default to current user
```

options

```
owner/schema The database owner/schema
```

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options
[cb]	Function	The callback function

dataSource.discoverPrimaryKeysSync(modelName, options)

The synchronous version of discoverPrimaryKeys

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options

dataSource.discoverForeignKeys(modelName, options, [cb])

Discover foreign keys for a given owner/modelName

foreign key description

```
fkOwner String => foreign key table schema (may be null)
fkName String => foreign key name (may be null)
fkTableName String => foreign key table name
fkColumnName String => foreign key column name
keySeq Number => sequence number within a foreign key( a value of 1 represents the first column of the
foreign key, a value of 2 would represent the second column within the foreign key).
pkOwner String => primary key table schema being imported (may be null)
pkName String => primary key name (may be null)
pkTableName String => primary key table name being imported
pkColumnName String => primary key column name being imported
```

options

```
owner/schema The database owner/schema
```

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options
[cb]	Function	The callback function

dataSource.discoverForeignKeysSync(modelName, options)

The synchronous version of discoverForeignKeys

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options

dataSource.discoverExportedForeignKeys(modelName, options, [cb])

Retrieves a description of the foreign key columns that reference the given table's primary key columns (the foreign keys exported by a table). They are ordered by fkTableOwner, fkTableName, and keySeq.

foreign key description

```

fkOwner {String} => foreign key table schema (may be null)
fkName {String} => foreign key name (may be null)
fkTableName {String} => foreign key table name
fkColumnName {String} => foreign key column name
keySeq {Number} => sequence number within a foreign key( a value of 1 represents the first column of
the foreign key, a value of 2 would represent the second column within the foreign key).
pkOwner {String} => primary key table schema being imported (may be null)
pkName {String} => primary key name (may be null)
pkTableName {String} => primary key table name being imported
pkColumnName {String} => primary key column name being imported

```

options

```
owner/schema The database owner/schema
```

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options
[cb]	Function	The callback function

dataSource.discoverExportedForeignKeysSync(modelName, options)

The synchronous version of discoverExportedForeignKeys

Arguments

Name	Type	Description
modelName	String	The model name
options	Object	The options

dataSource.discoverSchema(modelName, [options], [cb])

Discover one schema from the given model without following the relations

Arguments

Name	Type	Description
modelName	String	The model name
[options]	Object	The options
[cb]	Function	The callback function

dataSource.discoverSchemas(modelName, [options], [cb])

Discover schema from a given modelName/view

options

```

{String} owner/schema - The database owner/schema name
{Boolean} relations - If relations (primary key/foreign key) are navigated
{Boolean} all - If all owners are included
{Boolean} views - If views are included

```

Arguments

Name	Type	Description
modelName	String	The model name
[options]	Object	The options

[cb]	Function	The callback function
------	----------	-----------------------

`dataSource.discoverSchemasSync(modelName, [options])`
 Discover schema from a given table/view synchronously

`options`

```
{String} owner/schema - The database owner/schema name
{Boolean} relations - If relations (primary key/foreign key) are navigated
{Boolean} all - If all owners are included
{Boolean} views - If views are included
```

Arguments

Name	Type	Description
modelName	String	The model name
[options]	Object	The options

`dataSource.discoverAndBuildModels(modelName, [options], [cb])`
 Discover and build models from the given owner/modelName

`options`

```
{String} owner/schema - The database owner/schema name
{Boolean} relations - If relations (primary key/foreign key) are navigated
{Boolean} all - If all owners are included
{Boolean} views - If views are included
```

Arguments

Name	Type	Description
modelName	String	The model name
[options]	Object	The options
[cb]	Function	The callback function

`dataSource.discoverAndBuildModelsSync(modelName, [options])`
 Discover and build models from the given owner/modelName synchronously

`options`

```
{String} owner/schema - The database owner/schema name
{Boolean} relations - If relations (primary key/foreign key) are navigated
{Boolean} all - If all owners are included
{Boolean} views - If views are included
```

Arguments

Name	Type	Description
modelName	String	The model name
[options]	Object	The options

`dataSource.isActual([models])`
 Check whether migrations needed This method make sense only for sql connectors.

Arguments

Name	Type	Description
[models]	String[]	A model name or an array of model names. If not present, apply to all models

dataSource.freeze()
Freeze dataSource. Behavior depends on connector
dataSource.tableName(modelName)
Return table name for specified modelName

Arguments

Name	Type	Description
modelName	String	The model name

dataSource.columnName(modelName, The)
Return column name for specified modelName and propertyName

Arguments

Name	Type	Description
modelName	String	The model name
The	propertyName	property name

dataSource.columnMetadata(modelName, The)
Return column metadata for specified modelName and propertyName

Arguments

Name	Type	Description
modelName	String	The model name
The	propertyName	property name

dataSource.columnNames(modelName)
Return column names for specified modelName

Arguments

Name	Type	Description
modelName	String	The model name

dataSource.idColumnName(modelName)
Find the ID column name

Arguments

Name	Type	Description
modelName	String	The model name

dataSource.idName(modelName)
Find the ID property name

Arguments

Name	Type	Description
modelName	String	The model name

dataSource.idNames(modelName)
Find the ID property names sorted by the index

Arguments

Name	Type	Description
modelName	String	The model name

dataSource.defineForeignKey(className, key, foreignClassName)
Define foreign key to another model

Arguments

Name	Type	Description
className	String	The model name that owns the key
key	String	<ul style="list-style-type: none">name of key field
foreignClassName	String	The foreign model name

dataSource.disconnect([cb])
Close database connection

Arguments

Name	Type	Description
[cb]	Fucntion	The callback function

dataSource.enableRemote(operation)
Enable a data source operation to be remote.

Arguments

Name	Type	Description
operation	String	The operation name

dataSource.disableRemote(operation)
Disable a data source operation to be remote.

Arguments

Name	Type	Description
operation	String	The operation name

dataSource.getOperation(operation)
Get an operation's metadata.

Arguments

Name	Type	Description
operation	String	The operation name

dataSource.operations()
Get all operations.
dataSource.defineOperation(name, options, fn)
Define an operation to the data source

Arguments

Name	Type	Description
name	String	The operation name
options	Object	The options
fn	[Function	The function

dataSource.isRelational()
Check if the backend is a relational DB
dataSource.ready(,)
Check if the data source is ready

Arguments

Name	Type	Description
	obj	

	args	
--	------	--

hiddenProperty(obj, key, value)
 Define a hidden property

Arguments

Name	Type	Description
obj	Object	The property owner
key	String	The property name
value	Mixed	The default value

defineReadonlyProp(obj, key, value)
 Define readonly property on object

Arguments

Name	Type	Description
obj	Object	The property owner
key	String	The property name
value	Mixed	The default value

Geopoint API (datasource-juggler)



- [assert](#)
- [exports.GeoPoint](#)
- [GeoPoint.distanceBetween](#)
- [geoPoint.distanceTo](#)
- [geoPoint.toString](#)

Module: loopback-datasource-juggler

assert
 Dependencies.
 exports.GeoPoint
 Export the `GeoPoint` class.
 GeoPoint.distanceBetween()
 Determine the spherical distance between two geo points.
 geoPoint.distanceTo()
 Determine the spherical distance to the given point.
 geoPoint.toString()
 Simple serialization.

Model API



- [module.exports](#)
- [util](#)
- [ModelBaseClass](#)
- [ModelBaseClass.defineProperty](#)
- [ModelBaseClass.toString](#)
- [modelBaseClass.toObject](#)
- [modelBaseClass.propertyChanged](#)
- [modelBaseClass.reset](#)

Module: loopback-datasource-juggler

module.exports
 Module exports class Model
 util
 Module dependencies
 ModelBaseClass(data)
 Model class - base class for all persist objects provides **common API** to access any database connector. This class describes only abstract behavior layer, refer to `lib/connectors/*.js` to learn more about specific connector implementations

ModelBaseClass mixes Validatable and Hookable classes methods

Arguments

Name	Type	Description
------	------	-------------

data	Object	<ul style="list-style-type: none"> initial object data
------	--------	---

ModelBaseClass.defineProperty(params)
@param {String} prop - property name

Arguments

Name	Type	Description
params	Object	<ul style="list-style-type: none"> various property configuration

ModelBaseClass.toString()
Return string representation of class
modelBaseClass.toObject(onlySchema)
Convert instance to Object

Arguments

Name	Type	Description
onlySchema	Boolean	<ul style="list-style-type: none"> restrict properties to dataSource only, default false

modelBaseClass.propertyChanged(propertyName)
Checks is property changed based on current property and initial value

Arguments

Name	Type	Description
propertyName	String	<ul style="list-style-type: none"> property name

modelBaseClass.reset()
Reset dirty attributes

this method does not perform any database operation it just reset object to it's initial state

Model builder API



Module: loopback-datasource-juggler

- [exports.ModelBuilder](#)
- [ModelBuilder](#)
- [this.models](#)
- [this.definitions](#)
- [modelBuilder.getModel](#)
- [modelBuilder.define](#)
- [ModelClass.registerProperty](#)
- [modelBuilder.defineProperty](#)
- [modelBuilder.extendModel](#)
- [modelBuilder.getSchemaName](#)
- [modelBuilder.resolveType](#)
- [modelBuilder.buildModels](#)
- [modelBuilder.buildModelFromInstance](#)

exports.ModelBuilder
Export public API
ModelBuilder()
ModelBuilder - A builder to define data models
this.models
@property {Object} models Model constructors
this.definitions
@property {Object} definitions Definitions of the models
modelBuilder.getModel(name, forceCreate)
Get a model by name

Arguments

Name	Type	Description
name	String	The model name
forceCreate	Boolean	Indicate if a stub should be created for the

`modelBuilder.define(className, properties, settings)`
Define a model class

Arguments

Name	Type	Description
<code>className</code>	String	
<code>properties</code>	Object	<ul style="list-style-type: none">hash of class properties in format
<code>settings</code>	Object	<ul style="list-style-type: none">other configuration of class

`ModelClass.registerProperty()`
Register a property for the model class

Arguments

Name	Type	Description
	<code>propertyName</code>	

`modelBuilder.defineProperty(model, propertyName, propertyDefinition)`
Define single property named `propertyName` on `model`

Arguments

Name	Type	Description
<code>model</code>	String	<ul style="list-style-type: none">name of model
<code>propertyName</code>	String	<ul style="list-style-type: none">name of property
<code>propertyDefinition</code>	Object	<ul style="list-style-type: none">property settings

`modelBuilder.extendModel(model, props)`
Extend existing model with bunch of properties

Arguments

Name	Type	Description
<code>model</code>	String	<ul style="list-style-type: none">name of model
<code>props</code>	Object	<ul style="list-style-type: none">hash of properties

`modelBuilder.getSchemaName()`
Get the schema name
`modelBuilder.resolveType(type)`
Resolve the type string to be a function, for example, 'String' to String

Arguments

Name	Type	Description
<code>type</code>	String	The type string, such as 'number', 'Number', 'boolean', or 'String'. It's case insensitive

`modelBuilder.buildModels(schemas)`
Build models from schema definitions

`schemas` can be one of the following:

1. An array of named schema definition JSON objects
2. A schema definition JSON object
3. A list of property definitions (anonymous)

Arguments

Name	Type	Description

schemas	*	The schemas
---------	---	-------------

modelBuilder.buildModelFromInstance(name, json, options)
 Introspect the json document to build a corresponding model

Arguments

Name	Type	Description
name	String	The model name
json	Object	The json object
options	[Object	The options

Relation API



- [i8n](#)
- [Relation.hasMany](#)
- [Relation.belongsTo](#)
- [Relation.hasAndBelongsToMany](#)

Module: loopback-datasource-juggler

i8n
 Dependencies
 Relation.hasMany(anotherClass, params)
 Declare hasMany relation

Arguments

Name	Type	Description
anotherClass	Relation	<ul style="list-style-type: none"> • class to has many
params	Object	<ul style="list-style-type: none"> • configuration {as:, foreignKey:}

Relation.belongsTo(anotherClass, params)
 Declare belongsTo relation

Arguments

Name	Type	Description
anotherClass	Class	<ul style="list-style-type: none"> • class to belong
params	Object	<ul style="list-style-type: none"> • configuration {as: 'propertyName', foreignKey: 'keyName'}

Relation.hasAndBelongsToMany()
 Many-to-many relation
 Post.hasAndBelongsToMany('tags'); creates connection model 'PostTag'

Validations API



- [exports.ValidationError](#)
- [exports.Validatable](#)
- [Validatable.validatesPresenceOf](#)
- [Validatable.validatesLengthOf](#)
- [Validatable.validatesNumericalityOf](#)
- [Validatable.validatesInclusionOf](#)
- [Validatable.validatesExclusionOf](#)
- [Validatable.validatesFormatOf](#)
- [Validatable.validate](#)
- [Validatable.validateAsync](#)
- [Validatable.validatesUniquenessOf](#)
- [validatePresence](#)
- [validateLength](#)
- [validateNumericality](#)
- [validateInclusion](#)
- [validateExclusion](#)

Module: loopback-datasource-juggler

- `validateFormat`
- `validateCustom`
- `validateUniqueness`
- `validatable.isValid`
- `blank`

`exports.ValidationError`

Module exports

`exports.Validatable`

Validation mixins for `model.js`

Basically validation configurators is just class methods, which adds validations configs to `AbstractClass._validations`. Each of this validations run when `obj.isValid()` method called.

Each configurator can accept n params (n-1 field names and one config). Config is `{Object}` depends on specific validation, but all of them has one common part: `message` member. It can be just string, when only one situation possible, e.g. `Post.validatesPresenceOf('title', { message: 'can not be blank' })`;

In more complicated cases it can be `{Hash}` of messages (for each case): `User.validatesLengthOf('password', { min: 6, max: 20, message: {min: 'too short', max: 'too long'}})`;

`Validatable.validatesPresenceOf`

Validate presence. This validation fails when validated field is blank.

Default error message "can't be blank"

`Validatable.validatesLengthOf`

Validate length. Three kinds of validations: min, max, is.

Default error messages:

- min: too short
- max: too long
- is: length is wrong

`Validatable.validatesNumericalityOf`

Validate numericality.

`Validatable.validatesInclusionOf`

Validate inclusion in set

`Validatable.validatesExclusionOf`

Validate exclusion

`Validatable.validatesFormatOf`

Validate format

Default error message: is invalid

`Validatable.validate`

Validate using custom validator

Default error message: is invalid

Example:

```
User.validate('name', customValidator, {message: 'Bad name'});
function customValidator(err) {
  if (this.name === 'bad') err();
};
var user = new User({name: 'Peter'});
user.isValid(); // true
user.name = 'bad';
user.isValid(); // false
```

`Validatable.validateAsync`

Validate using custom async validator

Default error message: is invalid

Example:

```

User.validateAsync('name', customValidator, {message: 'Bad name'});
function customValidator(err, done) {
  process.nextTick(function () {
    if (this.name === 'bad') err();
    done();
  });
};
var user = new User({name: 'Peter'});
user.isValid(); // false (because async validation setup)
user.isValid(function (isValid) {
  isValid; // true
})
user.name = 'bad';
user.isValid(); // false
user.isValid(function (isValid) {
  isValid; // false
})

```

Validatable.validatesUniquenessOf
 Validate uniqueness

Default error message: is not unique

validatePresence()
 Presence validator
 validateLength()
 Length validator
 validateNumericality()
 Numericality validator
 validateInclusion()
 Inclusion validator
 validateExclusion()
 Exclusion validator
 validateFormat()
 Format validator
 validateCustom()
 Custom validator
 validateUniqueness()
 Uniqueness validator
 validatable.isValid(callback)

This method performs validation, triggers validation hooks. Before validation `obj.errors` collection cleaned. Each validation can add errors to `obj.errors` collection. If collection is not blank, validation failed.

Arguments

Name	Type	Description
callback	Function	called with (valid)

blank(v)

Return true when v is undefined, blank array, null or empty string otherwise returns false

Arguments

Name	Type	Description
v	Mix	

Oracle connector

- [Prerequisites](#)
- [Connector settings](#)
- [Discovering models](#)
 - [Discover, build, and try models](#)
- [Model definition for Oracle](#)
- [Type mapping](#)
 - [JSON to Oracle Types](#)
 - [Oracle Types to JSON](#)
- [Destroying models](#)

See also the [LoopBack Oracle Connector API reference](#).

- [Auto-migrate / Auto-update](#)
- [Running examples](#)

Prerequisites

See [Installing the Oracle connector](#) for installation instructions.

Connector settings

The connector can be configured using the following settings from the data source.

- host or hostname (default to 'localhost'): The host name or ip address of the Oracle DB server
- port (default 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 (default to 'XE'): The Oracle DB listener name
- debug (default to false)

Discovering models

Oracle data sources enable you to discover model definition information from existing Oracle databases. See the following APIs:

- [dataSource.discoverModelDefinitions\(\)](#)
- [dataSource.discoverSchema\(\)](#)

See also [Oracle discovery API](#).

Discover, build, and try models

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

```
dataSource.discoverAndBuildModels('INVENTORY', { owner: 'STRONGLOOP', visited: {},
associations: true},
  function (err, models) {
    // Show records from the models
    for(var m in models) {
      models[m].all(show);
    };

    // Find one record for inventory
    models.Inventory.findOne({}, function(err, inv) {
      console.log("\nInventory: ", inv);
      // Follow the foreign key to navigate to the product
      inv.product(function(err, prod) {
        console.log("\nProduct: ", prod);
        console.log("\n ----- ");
      });
    });
  });
```

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

- Number
- Boolean
- String
- null
- Object
- undefined
- Date
- Array
- Buffer

JSON to Oracle Types

- String|JSON|Text|default: VARCHAR2, default length is 1024
- Number: NUMBER
- Date: DATE
- Timestamp: TIMESTAMP(3)
- Boolean: CHAR(1)

Oracle Types to JSON

- CHAR(1): Boolean
- CHAR(n), VARCHAR, VARCHAR2, LONG VARCHAR, NCHAR, NVARCHAR2: String
- LONG, BLOB, CLOB, NCLOB: Buffer
- NUMBER, INTEGER, DECIMAL, DOUBLE, FLOAT, BIGINT, SMALLINT, REAL, NUMERIC, BINARY_FLOAT, BINARY_DOUBLE, UROWID, ROWID: Number
- DATE, TIMESTAMP: Date
- default: String

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()`. Only call `Model.automigrate()` on new models as 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 sequence, for example, PRODUCT_ID_TRIGGER

Running examples

- `example/app.js`: Demonstrate asynchronous discovery
- `example/app-sync.js`: Demonstrate synchronous discovery

Installing the Oracle connector

- [Overview](#)
- [Adding dependency](#)
 - [Mac OSX](#)
 - [Linux](#)
- [Standalone use for local tests](#)
- [Production](#)
- [Changes made to your environment](#)
 - [MacOSX](#)
 - [Linux](#)
 - [Windows](#)

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.

Adding dependency

Add [loopback-oracle-installer](#) as a dependency for [loopback-connector-oracle](#). Declare the dependency in your application's `package.json` file as follows:

```
"dependencies": {
  "loopback-oracle-installer":
    "git+ssh://git@github.com:strongloop/loopback-oracle-installer.git",
  ...
},
"config": {
  "oracleUrl":
    "http://7e9918db41dd01dbf98e-ec15952f71452bc0809d79c86f5751b6.r22.cf1.rackcdn.com"
},
```

The `config.oracleUrl` defines the base URL to download the corresponding `node-oracle` bundle for the local environment. You can override `oracleUrl` by using the `LOOPBACK_ORACLE_URL` environment variable.

The bundle file name is `loopback-oracle-<platform>-<arch>-<version>.tar.gz`. The version is the same as shown in `package.json`.

During `npm install` of the `loopback-connector-oracle` module, it will detect the local platform and download the corresponding prebuilt [node-oracle](#) module and Oracle Database Instant Client into the `node_modules` folder as illustrated below.

```
loopback-connector-oracle
+--- node_modules
+-- oracle
+-- instantclient
```

Mac OSX

For MacOSX, the full URL is:

<http://7e9918db41dd01dbf98e-ec15952f71452bc0809d79c86f5751b6.r22.cf1.rackcdn.com/loopback-oracle-MacOSX-x64-0.0.1.tar.gz>

Linux

You must install the `libaio` library on Linux systems.

On Ubuntu/Debian, use this command:

```
$ sudo apt-get install libaio1
```

On Fedora/CentOS/RHEL, use this command

```
$ sudo yum install libaio
```



Make sure `c:\instantclient_12_1\vc10` comes before `c:\instantclient_12_1`.

Standalone use for local tests

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

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>/r
```

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_m
```

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. You must log out and log in again to make it effective.

Oracle connector API



- `oracle`
- `exports.initialize`
- `Oracle`
- `oracle.connect`
- `oracle.executeSQL`
- `oracle.executeSQLSync`
- `oracle.query`
- `oracle.querySync`
- `oracle.count`
- `oracle.destroyAll`
- `oracle.create`
- `oracle.updateOrCreate`
- `oracle.save`
- `oracle.all`
- `oracle.exists`
- `oracle.find`

Module: loopback-connector-oracle

- [oracle.discoverModelDefinitions](#)
- [oracle.discoverModelDefinitionsSync](#)
- [oracle.discoverModelProperties](#)
- [oracle.discoverModelPropertiesSync](#)
- [oracle.discoverPrimaryKeys](#)
- [oracle.discoverPrimaryKeysSync](#)
- [oracle.discoverForeignKeys](#)
- [oracle.discoverForeignKeysSync](#)
- [oracle.discoverExportedForeignKeys](#)
- [oracle.discoverExportedForeignKeysSync](#)
- [oracle.autoupdate](#)
- [oracle.isActual](#)
- [oracle.alterTable](#)
- [oracle.dropTable](#)
- [oracle.createTable](#)
- [oracle.disconnect](#)

oracle

Oracle connector for LoopBack

exports.initialize(dataSource, [callback])

@module loopback-connector-oracle

Initialize the Oracle connector against the given data source

Arguments

Name	Type	Description
dataSource	DataSource	The loopback-datasource-juggler dataSource
[callback]	Function	The callback function

Oracle(oracle, settings)

Oracle connector constructor

settings is an object with the following possible properties:

- {String} hostname - The host name or ip address of the Oracle DB server
- {Number} port - The port number of the Oracle DB Server
- {String} user - The user name
- {String} password - The password
- {String} database - The database name (TNS listener name)
- {Boolean|Number} debug - The flag to control if debug messages will be printed out

Arguments

Name	Type	Description
oracle	Oracle	Oracle node.js binding
settings	Object	An object for the data source settings

oracle.connect([callback])

Connect to Oracle

Arguments

Name	Type	Description
[callback]	Function	The callback after the connection is established

oracle.executeSQL(sql, params, [callback])

Execute the sql statement

Arguments

Name	Type	Description
sql	String	The SQL statement
params	String[]	The parameter values for the SQL statement
[callback]	Function	The callback after the SQL statement is executed

oracle.executeSQLSync(sql, params)

Execute the sql statement synchronously

Arguments

Name	Type	Description
sql	String	The SQL statement
params	String[]	An array of parameters

oracle.query(sql, params, [callback])

Execute a sql statement with the given parameters

Arguments

Name	Type	Description
sql	String	The SQL statement
params	String[]	An array of parameter values
[callback]	Function	The callback function

oracle.querySync(sql, params)

Run a query synchronously

Arguments

Name	Type	Description
sql	String	The SQL statement
params	String[]	An array of parameter values

oracle.count(model, [callback], filter)

Count the number of instances for the given model

Arguments

Name	Type	Description
model	String	The model name
[callback]	Function	The callback function
filter	Object	The filter for where

oracle.destroyAll(model, [where], [callback])

Delete instances for the given model

Arguments

Name	Type	Description
------	------	-------------

Name	Type	Description
model	String	The model name
[where]	Object	The filter for where
[callback]	Function	The callback function

oracle.create(model, data, [callback])

Create the data model in Oracle

Arguments

Name	Type	Description
model	String	The model name
data	Object	The model instance data
[callback]	Function	The callback function

oracle.updateOrCreate(model, data, [callback])

Update if the model instance exists with the same id or create a new instance

Arguments

Name	Type	Description
model	String	The model name
data	Object	The model instance data
[callback]	Function	The callback function

oracle.save(model, data, [callback])

Save the model instance to Oracle DB

Arguments

Name	Type	Description
model	String	The model name
data	Object	The model instance data
[callback]	Function	The callback function

oracle.all(model, filter, [callback])

Find matching model instances by the filter

Arguments

Name	Type	Description
model	String	The model name
filter	Object	The filter
[callback]	Function	The callback function

oracle.exists(model, id, [callback])

Check if a model instance exists by id

Arguments

Name	Type	Description
model	String	The model name
id	*	The id value
[callback]	Function	The callback function

oracle.find(model, id, [callback])

Find a model instance by id

Arguments

Name	Type	Description
model	String	The model name
id	*	The id value
[callback]	Function	The callback function

oracle.discoverModelDefinitions(options, [cb])

Discover model definitions

Arguments

Name	Type	Description
options	Object	Options for discovery
[cb]	Function	The callback function

oracle.discoverModelDefinitionsSync(options)

Discover the tables/views synchronously

Arguments

Name	Type	Description
options	Object	The options for discovery

oracle.discoverModelProperties(table, options, [cb])

Discover model properties from a table

Arguments

Name	Type	Description
table	String	The table name
options	Object	The options for discovery
[cb]	Function	The callback function

oracle.discoverModelPropertiesSync(table, options)

Discover model properties from a table synchronously

Arguments

Name	Type	Description
table	String	The table name

options	Object	The options for discovery
---------	--------	---------------------------

oracle.discoverPrimaryKeys(table, options, [cb])

Discover primary keys for a given table

Arguments

Name	Type	Description
table	String	The table name
options	Object	The options for discovery
[cb]	Function	The callback function

oracle.discoverPrimaryKeysSync(table, options)

Discover primary keys synchronously for a given table

Arguments

Name	Type	Description
table	String	
options	Object	

oracle.discoverForeignKeys(table, options, [cb])

Discover foreign keys for a given table

Arguments

Name	Type	Description
table	String	The table name
options	Object	The options for discovery
[cb]	Function	The callback function

oracle.discoverForeignKeysSync(table, options)

Discover foreign keys synchronously for a given table

Arguments

Name	Type	Description
table	String	The table name
options	Object	The options for discovery

oracle.discoverExportedForeignKeys(table, options, [cb])

Discover foreign keys that reference to the primary key of this table

Arguments

Name	Type	Description
table	String	The table name
options	Object	The options for discovery
[cb]	Function	The callback function

oracle.discoverExportedForeignKeysSync(owner, options)

Discover foreign keys synchronously for a given table

Arguments

Name	Type	Description
owner	String	The DB owner/schema name
options	Object	The options for discovery

oracle.autoupdate([models], [cb])

Perform autoupdate for the given models

Arguments

Name	Type	Description
[models]	String[]	A model name or an array of model names. If not present, apply to all models
[cb]	Function	The callback function

oracle.isActual([models], [cb])

Check if the models exist

Arguments

Name	Type	Description
[models]	String[]	A model name or an array of model names. If not present, apply to all models
[cb]	Function	The callback function

oracle.alterTable(model, actualFields, [cb])

Alter the table for the given model

Arguments

Name	Type	Description
model	String	The model name
actualFields	Object[]	Actual columns in the table
[cb]	Function	The callback function

oracle.dropTable(model, [cb])

Drop a table for the given model

Arguments

Name	Type	Description
model	String	The model name
[cb]	Function	The callback function

oracle.createTable(model, [cb])

Create a table for the given model

Arguments

Name	Type	Description
model	String	The model name
[cb]	Function	The callback function

oracle.disconnect([cb])

Disconnect from Oracle

Arguments

Name	Type	Description
[cb]	Function	The callback function

Oracle discovery API

Methods for discovery

[View docs for loopback-connector-oracle in GitHub](#)

NOTE: All the following methods are asynchronous.

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:

- all: {Boolean} To include tables/views from all schemas/owners
- owner/schema: {String} The schema/owner name
- views: {Boolean} Whether to 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' }
```

discoverModelProperties(table, options, cb)

Parameter	Description
table: {String}	The name of a table or view
options	owner/schema: {String} The schema/owner name
cb	Get a list of model property definitions; see example below.

```
{ 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(table, options, cb)

Parameter	Description
table: {String}	Name of a table or view
options	owner/schema: {String} The schema/owner name
cb	Get a list of primary key definitions; see example below.

```
{ 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(table, options, cb)

Parameter	Description
table: {String}	Name of a table or view
options	owner/schema: {String} The schema/owner name
cb	Get a list of foreign key definitions; see example below.

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

discoverExportedForeignKeys(table, options, cb)

Parameter	Description
table: {String}	The name of a table or view
options	owner/schema: {String} The schema/owner name
cb	Get a list of foreign key definitions that reference the primary key of the given table; see example below.

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

Synchronous discovery methods

- Oracle.prototype.discoverModelDefinitionsSync = function (options)
- Oracle.prototype.discoverModelPropertiesSync = function (table, options)
- Oracle.prototype.discoverPrimaryKeysSync = function (table, options)

- `Oracle.prototype.discoverForeignKeysSync= function(table, options)`
- `Oracle.prototype.discoverExportedForeignKeysSync= function(table, options)`

MySQL connector

- [Installation](#)
- [Create data source](#)
- [Data type mappings](#)
 - [JSON to MySQL types](#)
 - [MySQL to JSON types](#)
- [Using the dataType field/column option with MySQL](#)
 - [Floating-point types](#)
 - [Fixed-point exact value types](#)
 - [Other types](#)
 - [Asynchronous discovery methods](#)
 - [discoverModelDefinitions](#)
 - [discoverModelProperties](#)
 - [discoverPrimaryKeys](#)
 - [discoverForeignKeys](#)
 - [discoverExportedForeignKeys](#)
 - [Discover/build/try the models](#)
 - [Build a LDL schema by discovery](#)

See also the [LoopBack MySQL Connector API reference](#).

Installation

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

Create data source

To use the MySQL connector you need `loopback-datasource-juggler`.

1. Setup dependencies in `package.json`:

```
{
  ...
  "dependencies": {
    "loopback-datasource-juggler": "latest",
    "loopback-connector-mysql": "latest"
  },
  ...
}
```

2. Create the data source in your application with code such as this:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var dataSource = new DataSource('mysql', {
  host: 'localhost',
  port: 3306,
  database: 'mydb',
  username: 'myuser',
  password: 'mypass'
});
```

You can optionally 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.

Data type mappings

loopback-connector-mysql uses the following rules to map between JSON types and MySQL data types.

JSON to MySQL types

JSON Data Type	MySQL Data Type
String/JSON	VARCHAR
Text	TEXT
Number	INT
Date	DATETIME
Boolean	TINYINT(1)
Point/GeoPoint	POINT
Enum	ENUM

MySQL to JSON types

MySQL Data Type	JSON Data Type
CHAR	String
CHAR(1)	Boolean
VARCHAR TINYTEXT MEDIUMTEXT LONGTEXT TEXT ENUM SET	String
TINYBLOB MEDIUMBLOB LONGBLOB BLOB BINARY VARBINARY BIT	Binary
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"
  }
}

```

loopback-connector-mysql also supports using the `dataType` column/property attribute to specify what MySQL column type is used 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 }}
```

Discovering models

MySQL data sources allow you to discover model definition information from existing mysql databases. See the following APIs:

- `dataSource.discoverModelDefinitions([owner], fn)`
- `dataSource.discoverSchema([owner], name, fn)`

Asynchronous discovery methods

discoverModelDefinitions

```
MySQL.prototype.discoverModelDefinitions = function (options, cb)
```

options:

- `all`: {Boolean} To include tables/views from all schemas/owners
- `owner/schema`: {String} The schema/owner name
- `views`: {Boolean} To include views

cb:

- Get a list of table/view names, for example:

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

discoverModelProperties

```
MySQL.prototype.discoverModelProperties = function (table, options, cb)
```

- `table`: {String} The name of a table or view
- options:

- owner/schema: {String} The schema/owner name
- cb:
- Get a list of model property definitions, for example:

```
{ 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

```
MySQL.prototype.discoverPrimaryKeys= function(table, options, cb)
```

- table: {String} The name of a table or view
- options:
- owner/schema: {String} The schema/owner name
- cb:
- Get a list of primary key definitions, for example:

```
{ 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

```
MySQL.prototype.discoverForeignKeys= function(table, options, cb)
```

- table: {String} The name of a table or view
- options:
- owner/schema: {String} The schema/owner name
- cb:
- Get a list of foreign key definitions, for example:

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

discoverExportedForeignKeys

```
MySQL.prototype.discoverExportedForeignKeys= function(table, options, cb)
```

- table: {String} The name of a table or view
- options:
- owner/schema: {String} The schema/owner name
- cb:
- Get a list of foreign key definitions that reference the primary key of the given table, for example:
- ```
{ fkName: 'PRODUCT_FK',

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

### ***Discover/build/try the models***

#### **Build a LDL schema by discovery**

Data sources backed by the MySQL connector can discover LDL models from the database using the `discoverSchema` API. For example,

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

Here is the sample result. Please note there are 'mysql' properties in addition to the regular LDL model options and properties. The 'mysql' objects contain the MySQL specific mappings.

```
{
 "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"
 }
}
```

```

 }
 }
}

```

We can also discover and build model classes in one shot. The following example uses `discoverAndBuildModels` to discover, build and try the models:

```

dataSource.discoverAndBuildModels('INVENTORY', { owner: 'STRONGLOOP', visited: {},
associations: true},
 function (err, models) {
 // Show records from the models
 for(var m in models) {
 models[m].all(show);
 };

 // Find one record for inventory
 models.Inventory.findOne({}, function(err, inv) {
 console.log("\nInventory: ", inv);
 // Follow the foreign key to navigate to the product
 inv.product(function(err, prod) {
 console.log("\nProduct: ", prod);
 console.log("\n ----- ");
 });
 });
 });
}

```

### MySQL connector API



Module: loopback-connector-mysql

- `exports.initialize`
- `MySQL`
- `mysql.query`
- `mysql.create`
- `mysql.updateOrCreate`
- `mysql.all`
- `mysql.destroyAll`
- `mysql.autoupdate`
- `mysql.isActual`
- `fixedPointOptions`
- `mysql.disconnect`

**exports.initialize(dataSource, [callback])**

@module loopback-connector-mysql

Initialize the MySQL connector against the given data source

#### Arguments

| Name       | Type       | Description                                |
|------------|------------|--------------------------------------------|
| dataSource | DataSource | The loopback-datasource-juggler dataSource |
| [callback] | Function   | The callback function                      |

### MySQL(client)

@constructor Constructor for MySQL connector

## Arguments

| Name   | Type   | Description                  |
|--------|--------|------------------------------|
| client | Object | The node-mysql client object |

## mysql.query(sql, [callback])

Execute the sql statement

## Arguments

| Name       | Type     | Description                                      |
|------------|----------|--------------------------------------------------|
| sql        | String   | The SQL statement                                |
| [callback] | Function | The callback after the SQL statement is executed |

## mysql.create(model, data, [callback])

Create the data model in MySQL

## Arguments

| Name       | Type     | Description             |
|------------|----------|-------------------------|
| model      | String   | The model name          |
| data       | Object   | The model instance data |
| [callback] | Function | The callback function   |

## mysql.updateOrCreate(model, data, [callback])

Update if the model instance exists with the same id or create a new instance

## Arguments

| Name       | Type     | Description             |
|------------|----------|-------------------------|
| model      | String   | The model name          |
| data       | Object   | The model instance data |
| [callback] | Function | The callback function   |

## mysql.all(model, filter, [callback])

Find matching model instances by the filter

## Arguments

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| filter     | Object   | The filter            |
| [callback] | Function | The callback function |

## mysql.destroyAll(model, [where], [callback])

Delete instances for the given model

## Arguments

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

|            |          |                       |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| [where]    | Object   | The filter for where  |
| [callback] | Function | The callback function |

#### mySQL.autoupdate([models], [cb])

Perform autoupdate for the given models

##### Arguments

| Name     | Type     | Description                                                                  |
|----------|----------|------------------------------------------------------------------------------|
| [models] | String[] | A model name or an array of model names. If not present, apply to all models |
| [cb]     | Function | The callback function                                                        |

#### mySQL.isActual([models], [cb])

Check if the models exist

##### Arguments

| Name     | Type     | Description                                                                  |
|----------|----------|------------------------------------------------------------------------------|
| [models] | String[] | A model name or an array of model names. If not present, apply to all models |
| [cb]     | Function | The callback function                                                        |

#### fixedPointOptions()

declaration which would default to DECIMAL(10,0). Instead defaulting to (9,2).

#### mySQL.disconnect()

Disconnect from MySQL

### MySQL discovery API



- [mySQL.discoverModelDefinitions](#)
- [mySQL.discoverModelProperties](#)
- [mySQL.discoverPrimaryKeys](#)
- [mySQL.discoverForeignKeys](#)
- [mySQL.discoverExportedForeignKeys](#)

Module: loopback-connector-mysql

#### mySQL.discoverModelDefinitions(options, [cb])

Discover model definitions

##### Arguments

| Name    | Type     | Description           |
|---------|----------|-----------------------|
| options | Object   | Options for discovery |
| [cb]    | Function | The callback function |

#### mySQL.discoverModelProperties(table, options, [cb])

Discover model properties from a table

##### Arguments

| Name | Type | Description |
|------|------|-------------|
|      |      |             |

|         |          |                           |
|---------|----------|---------------------------|
| table   | String   | The table name            |
| options | Object   | The options for discovery |
| [cb]    | Function | The callback function     |

#### **mysql.discoverPrimaryKeys(table, options, [cb])**

Discover primary keys for a given table

##### **Arguments**

| Name    | Type     | Description               |
|---------|----------|---------------------------|
| table   | String   | The table name            |
| options | Object   | The options for discovery |
| [cb]    | Function | The callback function     |

#### **mysql.discoverForeignKeys(table, options, [cb])**

Discover foreign keys for a given table

##### **Arguments**

| Name    | Type     | Description               |
|---------|----------|---------------------------|
| table   | String   | The table name            |
| options | Object   | The options for discovery |
| [cb]    | Function | The callback function     |

#### **mysql.discoverExportedForeignKeys(table, options, [cb])**

Discover foreign keys that reference to the primary key of this table

##### **Arguments**

| Name    | Type     | Description               |
|---------|----------|---------------------------|
| table   | String   | The table name            |
| options | Object   | The options for discovery |
| [cb]    | Function | The callback function     |

## **MongoDB connector**

- [Using the MongoDB connector](#)
- [Customizing MongoDB configuration for tests/examples](#)

See also the [LoopBack MongoDB Connector API reference](#).

### **Using the MongoDB connector**

To use the MongoDB connector, you need `loopback-datasource-juggler@1.0.x`.

1. Setup dependencies in `package.json`:



```
{
 ...
 "dependencies": {
 "loopback-datasource-juggler": "1.0.x",
 "loopback-connector-mongodb": "1.0.x"
 },
 ...
}
```

2. Use:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var ds = new DataSource('mongodb');
...
```

### Customizing MongoDB configuration for tests/examples

By default, examples and tests from this module assume there is a MongoDB server instance running on localhost at port 27017.

To customize the settings, you can drop in a `.loopbackrc` file to the root directory of the project or the home folder.

The `.loopbackrc` file should be in JSON format, for example:

```
{
 "dev": {
 "mongodb": {
 "host": "127.0.0.1",
 "database": "test",
 "username": "youruser",
 "password": "yourpass",
 "port": 27017
 }
 },
 "test": {
 "mongodb": {
 "host": "127.0.0.1",
 "database": "test",
 "username": "youruser",
 "password": "yourpass",
 "port": 27017
 }
 }
}
```



Username and password are only required if the MongoDB server has authentication enabled.

### MongoDB connector API



- `exports.initialize`
- `MongoDB`
- `mongodb.connect`
- `mongodb.collection`
- `mongodb.create`
- `mongodb.save`

Module: `loopback-connector-mongodb`

- [mongodb.exists](#)
- [mongodb.find](#)
- [mongodb.updateOrCreate](#)
- [mongodb.destroy](#)
- [mongodb.all](#)
- [mongodb.destroyAll](#)
- [mongodb.count](#)
- [mongodb.updateAttributes](#)
- [mongodb.disconnect](#)

#### **exports.initialize(dataSource, [callback])**

Initialize the MongoDB connector for the given data source

##### **Arguments**

| Name       | Type       | Description              |
|------------|------------|--------------------------|
| dataSource | DataSource | The data source instance |
| [callback] | Function   | The callback function    |

#### **MongoDB(settings, dataSource)**

The constructor for MongoDB connector

##### **Arguments**

| Name       | Type       | Description              |
|------------|------------|--------------------------|
| settings   | Object     | The settings object      |
| dataSource | DataSource | The data source instance |

#### **mongodb.connect([callback], )**

Connect to MongoDB

##### **Arguments**

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| [callback] | Function | The callback function |
| undefined  | callback |                       |

##### **Callback**

| Name | Type  | Description         |
|------|-------|---------------------|
| err  | Error | The error object    |
| db   | Db    | The mongo DB object |

#### **mongodb.collection(name)**

Access a MongoDB collection by name

##### **Arguments**

| Name | Type   | Description         |
|------|--------|---------------------|
| name | String | The collection name |

#### **mongodb.create(model, data, [callback])**

Create a new model instance for the given data

##### **Arguments**

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| data       | Object   | The model data        |
| [callback] | Function | The callback function |

**mongodb.save(model, data, [callback])**

Save the model instance for the given data

#### Arguments

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| data       | Object   | The model data        |
| [callback] | Function | The callback function |

**mongodb.exists(model, id, [callback])**

Check if a model instance exists by id

#### Arguments

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| id         | *        | The id value          |
| [callback] | Function | The callback function |

**mongodb.find(model, id, [callback])**

Find a model instance by id

#### Arguments

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| id         | *        | The id value          |
| [callback] | Function | The callback function |

**mongodb.updateOrCreate(model, data, [callback])**

Update if the model instance exists with the same id or create a new instance

#### Arguments

| Name       | Type     | Description             |
|------------|----------|-------------------------|
| model      | String   | The model name          |
| data       | Object   | The model instance data |
| [callback] | Function | The callback function   |

**mongodb.destroy(model, id, The)**

Delete a model instance by id

#### Arguments

| Name  | Type       | Description       |
|-------|------------|-------------------|
| model | String     | The model name    |
| id    | *          | The id value      |
| The   | [callback] | callback function |

#### **mongodb.all(model, filter, [callback])**

Find matching model instances by the filter

##### **Arguments**

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| filter     | Object   | The filter            |
| [callback] | Function | The callback function |

#### **mongodb.destroyAll(model, [where], [callback])**

Delete all instances for the given model

##### **Arguments**

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| [where]    | Object   | The filter for where  |
| [callback] | Function | The callback function |

#### **mongodb.count(model, [callback], filter)**

Count the number of instances for the given model

##### **Arguments**

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| [callback] | Function | The callback function |
| filter     | Object   | The filter for where  |

#### **mongodb.updateAttributes(model, data, [callback])**

Update properties for the model instance data

##### **Arguments**

| Name       | Type     | Description           |
|------------|----------|-----------------------|
| model      | String   | The model name        |
| data       | Object   | The model data        |
| [callback] | Function | The callback function |

#### **mongodb.disconnect()**

Disconnect from MongoDB

## REST connector

- [Overview](#)
- [Resource CRUD](#)
- [Defining a custom method using REST template](#)

See also the [LoopBack REST Connector API reference](#).

### Overview

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 CRUD
- Defining a custom method using REST template

### Resource CRUD

If the REST APIs supports 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

Below is a simple 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 REST template

Imagine that you use 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



- [exports.initialize](#)
- [RestConnector](#)
- [restConnector.define](#)
- [restConnector.installPostProcessor](#)
- [restConnector.preProcess](#)
- [restConnector.postProcess](#)
- [restConnector.getResource](#)
- [restConnector.create](#)
- [restConnector.updateOrCreate](#)
- [restConnector.responseHandler](#)
- [restConnector.save](#)
- [restConnector.exists](#)
- [restConnector.find](#)
- [restConnector.destroy](#)
- [restConnector.all](#)
- [restConnector.destroyAll](#)
- [restConnector.count](#)
- [restConnector.updateAttributes](#)

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, debug)**

The RestConnector constructor

#### **Arguments**

| Name    | Type    | Description    |
|---------|---------|----------------|
| baseUrl | string  | The base URL   |
| debug   | boolean | The debug flag |

### **restConnector.define(descr)**

Hook for defining a model by the data source

#### **Arguments**

| Name  | Type   | Description           |
|-------|--------|-----------------------|
| descr | object | The model description |

### **restConnector.installPostProcessor(descr)**

Install the post processor

#### **Arguments**

| Name  | Type   | Description           |
|-------|--------|-----------------------|
| descr | object | The model description |

### **restConnector.preProcess(data)**

Pre-process the request data

#### **Arguments**

| Name | Type | Description      |
|------|------|------------------|
| data | *    | The request data |

### **restConnector.postProcess(model, data, many)**

Post-process the response data

#### **Arguments**

| Name  | Type    | Description       |
|-------|---------|-------------------|
| model | string  | The model name    |
| data  | *       | 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 |

**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 |

**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         | *        | 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         | *        | 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         | *        | 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         | *        | The id value            |
| data       | object   | The model instance data |
| [callback] | function | The callback function   |

## **REST resource API**



- [RestResource](#)
- [restResource.debug](#)
- [wrap](#)
- [restResource.create](#)
- [restResource.update](#)
- [restResource.delete](#)
- [restResource.deleteAll](#)
- [restResource.find](#)
- [restResource.all](#)

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          |

#### **restResource.debug(enabled)**

Enable/disable debug

##### **Arguments**

| Name    | Type    | Description |
|---------|---------|-------------|
| enabled | boolean |             |

#### **wrap(cb)**

Wrap the callback so that it takes (err, result, response)

##### **Arguments**

| Name | Type     | Description           |
|------|----------|-----------------------|
| cb   | function | The callback function |

**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     | *        | 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     | *        | The id value          |
| [ cb ] | function | The callback function |

**restResource.deleteAll(id, [cb])**

Map the delete operation to POST `/{model}`

**Arguments**

| Name   | Type     | Description           |
|--------|----------|-----------------------|
| id     | *        | 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     | *        | 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

- [format](#)
- [RequestBuilder](#)
- [isObject](#)
- [requestBuilder.debug](#)
- [requestBuilder.attach](#)
- [requestBuilder.redirects](#)
- [requestBuilder.url](#)
- [requestBuilder.method](#)
- [requestBuilder.timeout](#)
- [requestBuilder.header](#)
- [requestBuilder.type](#)
- [requestBuilder.query](#)
- [requestBuilder.body](#)
- [requestBuilder.buffer](#)
- [requestBuilder.timeout](#)
- [requestBuilder.responsePath](#)
- [requestBuilder.parse](#)
- [requestBuilder.auth](#)
- [requestBuilder.toJSON](#)
- [RequestBuilder.compile](#)
- [requestBuilder.build](#)
- [requestBuilder.operation](#)
- [requestBuilder.invoke](#)
- [RequestBuilder.resource](#)
- [RequestBuilder.request](#)

### format

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 |

### isObject(obj)

Check if obj is an object.

#### Arguments

| Name | Type   | Description |
|------|--------|-------------|
| obj  | object |             |

### requestBuilder.debug(enabled)

Turn on/off debug

#### Arguments

| Name    | Type    | Description                                           |
|---------|---------|-------------------------------------------------------|
| enabled | boolean | Set it true to enable debug or false to disable debug |

### **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 |

### **requestBuilder.method(method)**

Configure the HTTP method

#### **Arguments**

| Name   | Type   | Description     |
|--------|--------|-----------------|
| method | string | The HTTP method |

### **requestBuilder.timeout(timeout)**

Configure the timeout

#### **Arguments**

| Name    | Type   | Description             |
|---------|--------|-------------------------|
| timeout | number | The timeout value in ms |

### **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 |

### **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   |
|-------------------|--------|---------------|
| <code>user</code> | string | The user name |
| <code>pass</code> | string | The password  |

**requestBuilder.toJSON()**

Serialize the RequestBuilder to a JSON object

**RequestBuilder.compile**

Load the REST request from a JSON object

**Arguments**

| Name             | Type   | Description             |
|------------------|--------|-------------------------|
| <code>req</code> | object | The request json object |

**requestBuilder.build(options)**

Build the request by expanding the templated properties with the named values from options

**Arguments**

| Name                 | Type   | Description |
|----------------------|--------|-------------|
| <code>options</code> | object |             |

**requestBuilder.operation(parameterNames)**

Map the request builder to a function

**Arguments**

| Name                        | Type     | Description                                                                                               |
|-----------------------------|----------|-----------------------------------------------------------------------------------------------------------|
| <code>parameterNames</code> | [string] | The parameter names that define the order of args. It be an array of strings or multiple string arguments |

**requestBuilder.invoke(parameters, cb)**

Invoke a REST API with the provided parameter values in the parameters object

**Arguments**

| Name                    | Type     | Description                                         |
|-------------------------|----------|-----------------------------------------------------|
| <code>parameters</code> | object   | An object that provide {name: value} for parameters |
| <code>cb</code>         | 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          |

### **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 |

## **Memory connector**

### **Overview**

The built-in memory connector allows 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 production.

The memory connector supports

- Standard [query and 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 API](#) for more information on geo-filtering.

### **Using the memory connector**

Creating a data source using the memory connector is very simple.

```

/ use the built in memory function
// to create a memory data source
var memory = loopback.memory();

// or create it using the standard
// data source creation api
var memory = loopback.createDataSource({
 connector: loopback.Memory
});

// create a model using the
// memory data source
var properties = {
 name: String,
 price: Number
};

var Product = memory.createModel('product', properties);

Product.create([
 {name: 'apple', price: 0.79},
 {name: 'pear', price: 1.29},
 {name: 'orange', price: 0.59},
], count);

function count() {
 Product.count(console.log); // 3
}

```

## Authentication and authorization

- [Overview](#)
- [Use cases](#)
- [Authentication and authorization flow](#)
- [Creating and using access tokens](#)

See also the [Access control API](#) reference.

### Overview

Most applications need to control who (or what) can access data. Typically, this involves things like 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.

### Use cases

- Authenticate client applications, users, and/or devices to identify who (which principals) are calling the APIs
  - Personalization (use the established principals to personalize the response)
  - Protection (check the established principals against access control lists to protect sensitive information)
- Authorize access to protected resources against those who makes the request

### Authentication and authorization flow

| Step | Input | Action | Output | Model |
|------|-------|--------|--------|-------|
|------|-------|--------|--------|-------|

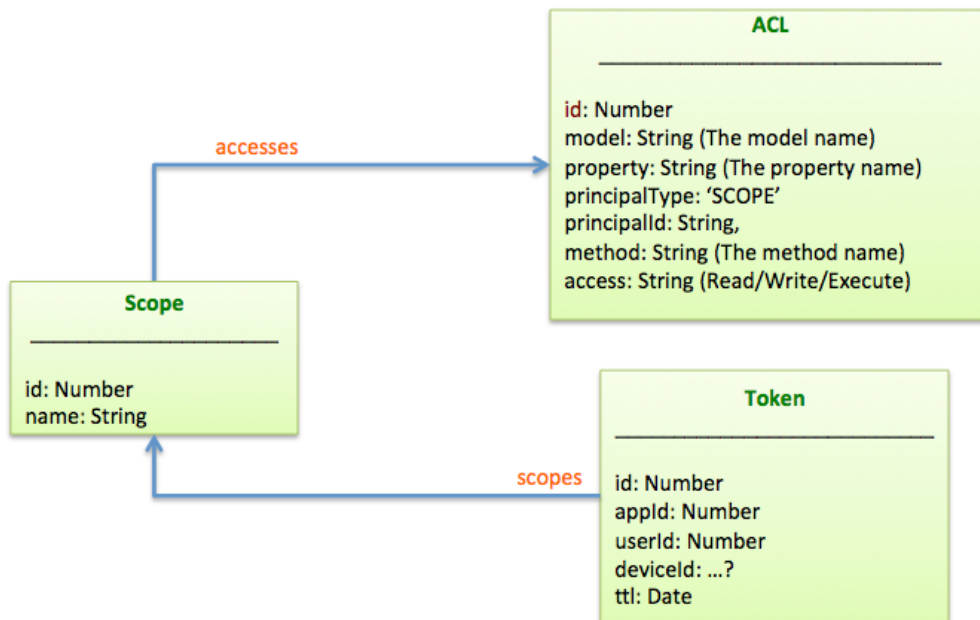
|                                                                              |                                                                                                                   |                                                                                                                                                                                                                                                                                                                                           |                                                                                                    |                                                         |
|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| 1. Sign up a new application                                                 | <ul style="list-style-type: none"> <li>Name</li> <li>Description</li> <li>URL</li> </ul>                          | <ul style="list-style-type: none"> <li>Generate application keys</li> <li>Create a new record for the application</li> </ul>                                                                                                                                                                                                              | <ul style="list-style-type: none"> <li>ID</li> <li>Key</li> </ul>                                  | Application                                             |
| 2. Register a new user                                                       | <ul style="list-style-type: none"> <li>Name</li> <li>Email</li> <li>Password</li> </ul>                           | <ul style="list-style-type: none"> <li>Create a new record for the user</li> </ul>                                                                                                                                                                                                                                                        | <ul style="list-style-type: none"> <li>ID</li> </ul>                                               | User                                                    |
| 3. Request an access token from the client application on behalf of the user | <ul style="list-style-type: none"> <li>App credential</li> <li>User credential</li> <li>List of scopes</li> </ul> | <ul style="list-style-type: none"> <li>Authenticate the principals with credentials</li> <li>Check the existence of scopes by name</li> <li>User authorized the application</li> <li>Store the user authorizations: userId, appId, scope, TTL (time to live or expiration time)</li> <li>Generate an access token and store it</li> </ul> | <ul style="list-style-type: none"> <li>Access token</li> </ul>                                     | Application<br>User<br>Scope<br>AccessToken<br>UserGran |
| 4. Invoke an API using the access token                                      | <ul style="list-style-type: none"> <li>API request</li> <li>Access token</li> </ul>                               | <ul style="list-style-type: none"> <li>Find/validate the token</li> <li>Find/validate required scopes for the protected resource</li> <li>Establish the subject from the token</li> <li>Find ACLs for the target model</li> <li>Find roles for the given userId</li> <li>Check if the appId/userId/roles pass the ACL</li> </ul>          | <ul style="list-style-type: none"> <li>API response if allowed</li> <li>Error if denied</li> </ul> | AccessToken<br>Scope<br>Model<br>Role<br>ACL            |

## Creating and using access tokens

To access protected server models from client applications through REST APIs, the request must bear an access token to represent the permissions granted by resource owners. Instead of passing credentials back and forth between the client and server, LoopBack provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (such as a username and password pair). This way, LoopBack allows client applications to access server-side models on behalf of a resource owner (such as a different client or an end-user).

The flow involves a few steps:

1. The client application asks LoopBack server for an access token for accessing protected resources. The requested permissions are expressed as a list of scopes, such as 'show-email' and 'read-posts'.
2. Resource owners grant or deny permissions to client applications. If granted, continue with step 3. Otherwise, an error will be sent to the client.
3. LoopBack generates an access token, store the associated metadata, and respond to the client application with the token string.
4. Now the client application sends a request to access the model with the access token.



## Creating and authenticating users

- Creating users
- Logging in and authenticating users
- Making authenticated requests with access tokens
- Logging out users and deleting access tokens

### Creating users

Create a user like any other model 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 milliseconds. 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": "GOkZRwgZ6lq0XXVxvxlB8TS1D6lrG7Vb9V8YwRDfy3YGAN7TM7EnxWHqdbIZfheZ",
 "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}
});
```

## Controlling data access

- [Overview](#)
- [Enabling access control](#)
- [General process](#)
  - [Defining principals and roles](#)
  - [Determining which models require access control](#)
  - [Defining access control](#)
    - [Static permissions](#)
    - [Dynamic roles](#)
    - [Enforcing access control](#)

### Overview

LoopBack apps access data through models (see [Working with models and data sources](#)), 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. In LoopBack you do this with *access control lists* (ACLs).

For any given request, the application decides to allow or deny the access. The main actors are:

- **Principal:** The source of a request: an application, a user, a device, or some combination. Each principal has an ID or name that is unique within its type.
  - **Role:** a group of principals; this enables an application to assign permissions to a group of principals altogether.
- **Protected Resource:** A model, a model instance, or a model property, method, or relation. Protected resources can have principals as owners.
- **Permission:** The outcome of the access control, one of: ALLOW, ALARM, AUDIT, or DENY.
  - **Scope:** a group of permissions for protected resources, so that resource owners can grant access to client applications.

An application uses ACLs to determine if a principal can read, write, or execute a given protected resource.

For example, an online car rental application might have the following requirements:

- Allow anyone to find/browse car inventory. No login required.
- Require a user to authenticate (login) to see/update/cancel their reservations.
- Allow a user to see (but not update or cancel) friends' reservations.
- Allow the car dealer to see reservations made in its system.
- Allow the car dealer to update its inventory.

## Enabling access control

If you created your app with the `slc lb` command, then you don't need to do anything to enable access control. Otherwise, see [Enabling access control manually](#).

## General process

The general process to implement access control for an application is:

1. **Define principals and roles.** Depending on your application, principals might be users, devices, or combinations thereof. Typically, you'll define groups of principals as roles: for example, you might create roles for anonymous users, authorized users, and administrators.
2. **Determine which models require access control.**
3. **Define access controls** for the desired models. See [Defining access control](#).
4. **Add access control enforcement** to the models. See [Enforcing access control](#).

## Defining principals and roles

There are different types of principals that LoopBack can support, for example, users, client applications, auth scopes, and roles. Each of the principal types has its own model to create new instances.

- User
- Application
- Scope
- Role

When you sign up a user, register a client application, define an auth scope, or define a role, you add principals to your LoopBack application.

Role is a special type of principal. It provides a logical way of grouping users/clients with common access privileges to your models. The role-based access control is more coarse-grained than ACLs directly linked to users.

The Role model has APIs to define roles and add other principals to a role.

### Adding a user to a role

```
// Create a new user
User.create({name: 'John', email: 'x@y.com', password: 'foobar'}, function (err, user)
{
 // Create a role named 'userRole'
 Role.create({name: 'userRole'}, function (err, role) {
 // Add user 'John' to 'userRole'
 role.principals.create({principalType: RoleMapping.USER, principalId: user.id},
function (err, p) {
 // Find all roles
 Role.find(function (err, roles) {

 ...

 });
 // List all principals in the role
 role.principals(function (err, principals) {
 ...
 });
 // List all users in the role
 role.users(function (err, users) {

 ...

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



### Adding a role to a role

```
// Create a role named 'user'
Role.create({name: 'user'}, function (err, userRole) {
 // Create a role named 'admin'
 Role.create({name: 'admin'}, function (err, adminRole) {
 // Add 'admin' role to 'user' role
 userRole.principals.create({principalType: RoleMapping.ROLE, principalId:
adminRole.id},
 function (err, mapping) {
 // Find all roles
 Role.find(function (err, roles) {
 ...
 });
 // Find role mappings
 RoleMapping.find(function (err, mappings) {
 ...
 });
 // List principals in the 'user' role
 userRole.principals(function (err, principals) {
 ...
 });
 // List all child roles in the 'user' role
 userRole.roles(function (err, roles) {
 ...
 });
 });
 });
});
```

#### REVIEW COMMENT

TBD

### ***Determining which models require access control***

Your app requirements will determine which models require access control.

### ***Defining access control***

The easiest way to define access control for an app is with the `slc lb` command. This enables you to create a static definition before runtime.

The `slc lb` subcommand provides options for defining ACLs. The general syntax is:

```
slc lb acl [--model modelName | all-models]
[--read | --write | --execute | --all]
[--everyone | --owner | --related | --authenticated | --unauthenticated]
[--method methodName / --property propertyName]
[--alarm | --allow | --deny | --audit]
```

For example, the first command below disables all access to all models. Other permissions will override this. The second command allows authors to edit posts.

The third and fourth commands allow only owners to access objects

```
$ slc lb acl --deny --all-models --everyone --all
$ slc lb acl --model post --allow --owner --all
$ slc lb acl --deny --all-models --everyone --all
$ slc lb acl --allow --all-models --owner --all
```

You can also define an ACL when you create a new data source with the `DataSource.create()` method. See [Data Source.createModel\(\)](#).

### Static permissions

Allow / Deny

#### REVIEW COMMENT

Explain how execute is use and when it applies.

Dynamic permissions: alarm and audit.

#### REVIEW COMMENT

Explain what these do.

### Dynamic roles

- Everyone
- Owner
- Related
- Authenticated
- Unauthenticated

## Enforcing access control

Once you've defined access control to your models, use a LoopBack remote hook to enforce the access controls using methods of the `Role`, `ACL`, and `Scope` objects.

For more information on remote hooks, see [Remote methods and hooks](#).

The `Role` object has several relevant methods:

- `Role.isInRole()` - checks if a principal is in a role.
- `Role.getRoles()` - returns the roles for the specified principal.
- `Role.registerResolver()` - adds a custom handler function for roles.

For more information, see [Access control API reference](#).

The `ACL` object has several relevant methods:

- `ACL.checkPermission()` - checks if the given principal is allowed to access the model/property.
- `ACL.checkAccess()` - checks if the request has the permission to access.
- `ACL.checkAccessForToken()` - checks if the given access token can invoke the specified method.

For more information, see [Access control API reference](#).

## Access control example app



This is a work in progress.



This example assumes you have already gone through the steps in [Getting started](#).

The [loopback-example-access-control](#) app provides a basic example of LoopBack access control.

## Building the access control example app

1. Create the application using the `slc` command line tool.

```
$ mkdir -p access-control/client
$ cd access-control
$ slc lb project server
```

2. Define a `Bank` model to store a set of `Banks` in the database.

```
$ cd server
$ slc lb model bank
```

3. Define an `Account` model to store user's bank accounts.

```
$ slc lb model account
```

4. Define a `Transaction` model to store user transactions.

```
$ slc lb model transaction
```

5. Setup the relationships between banks / accounts / users and transactions.

See the `models.json` file for the relations. Below is an example.

```
...

"user": {
 "options": {
 "base": "User",
 "relations": {
 "accessTokens": {
 "model": "accessToken",
 "type": "hasMany",
 "foreignKey": "userId"
 },
 "account": {
 "model": "account",
 "type": "belongsTo"
 },
 "transactions": {
 "model": "transaction",
 "type": "hasMany"
 }
 }
 },
},

...
```

6. Secure all the APIs.

```
$ slc lb acl --all-models --deny --everyone
```

## 7. Open up specific APIs

```
$ slc lb acl --allow --everyone --read --model bank
$ slc lb acl --allow --everyone --call create --model user
$ slc lb acl --allow --owner --all --model user
$ slc lb acl --allow --owner --read --model account
$ slc lb acl --allow --owner --write --model account
```

## Registering a User

**REVIEW COMMENT**  
Add curl example?

The following code is in `access-control/server/models/user.js`.

```
var app = require('../app');
var User = app.models.User;

User.beforeRemote('create', function(ctx, next, method) {
 var user = new User(ctx.instance);
 user.isValidAccount(next);
});

var accountError = new Error('account does not exist or was not specified');
accountError.statusCode = 422;

User.prototype.isValidAccount = function(cb) {
 if(this.accountId) {
 this.account(function(err, account) {
 if(err) {
 return cb(err);
 } else if(account) {
 return cb();
 } else {
 return cb(accountError);
 }
 });
 } else {
 return cb(accountError);
 }
}
```

## Logging in

**REVIEW COMMENT**  
Add curl example?

## Advanced topics

- [Enabling access control manually](#)
- [Defining access control at runtime](#)
  - [Using DataSource createModel\(\) method](#)
  - [Using the ACL create\(\) method](#)
- [Architecture](#)

### Enabling access control manually

If you didn't create your app with the `slc lb` command, then you must manually enable access control: Call the `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. See [Access control API](#) for more information.

```
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) {...});

```

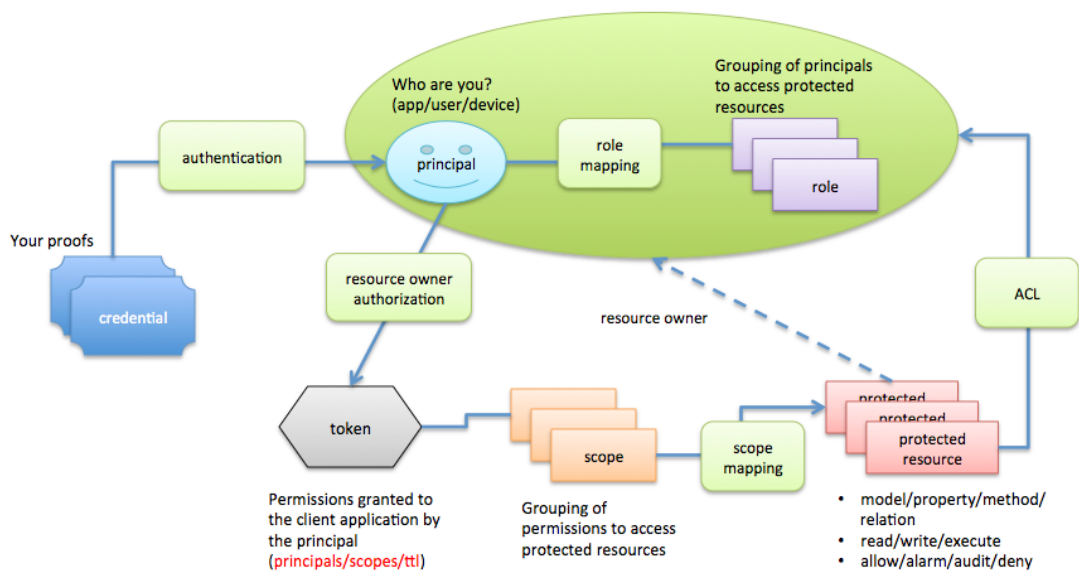
See [Access control models](#) for more information.

#### REVIEW COMMENT

Can we provide a short example here to complement what's in API reference?

## Architecture

The following diagram illustrates the architecture of the LoopBack access control system.



Please see the authentication and authorization flow for more details.

## Creating push notifications

- [Overview](#)
- [Set up push notifications in LoopBack application](#)
  - [Create a push model](#)
  - [Configure the application with push settings](#)
    - [Register a mobile application](#)
    - [Register a mobile device](#)
  - [Send push notifications](#)
    - [Send out the push notification immediately](#)
    - [Schedule the push notification request](#)
  - [Error handling](#)
- [Architecture](#)

See also:

- [API reference](#)
- [Example server application](#)
- [Example iOS app](#)
- [Example Android app](#)

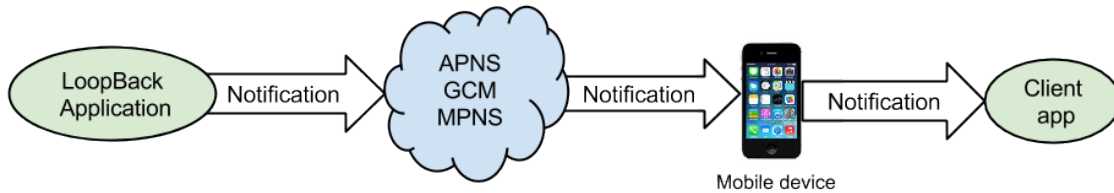
## Overview

Push notifications enable applications (known as *providers* in push parlance) to send information to mobile applications 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 or APNs
- **Android** - Google Cloud Messaging or 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 such as APNS, GCM, and MPNS.
- Push model to provide high-level APIs for device-independent push notifications.

## Set up push notifications in LoopBack application

To send push notifications, you'll need set up your LoopBack application, then make corresponding changes in the iOS and Android client apps.

### Create a push model

To support 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-push-notification
var PushModel = require('loopback-push-notification')(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.

```

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'
);
}

```

For information on getting the API keys, see:

- [Android - Obtain your Google Cloud Messaging credentials.](#)
- [iOS - Apple Push Certificates Portal.](#)

### ***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":
"756244503c9f95b49d7ff82120dc193cale3a7cb56f60c2ef2a19241e8f33305",
 "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()`: Select a list of devices by the 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

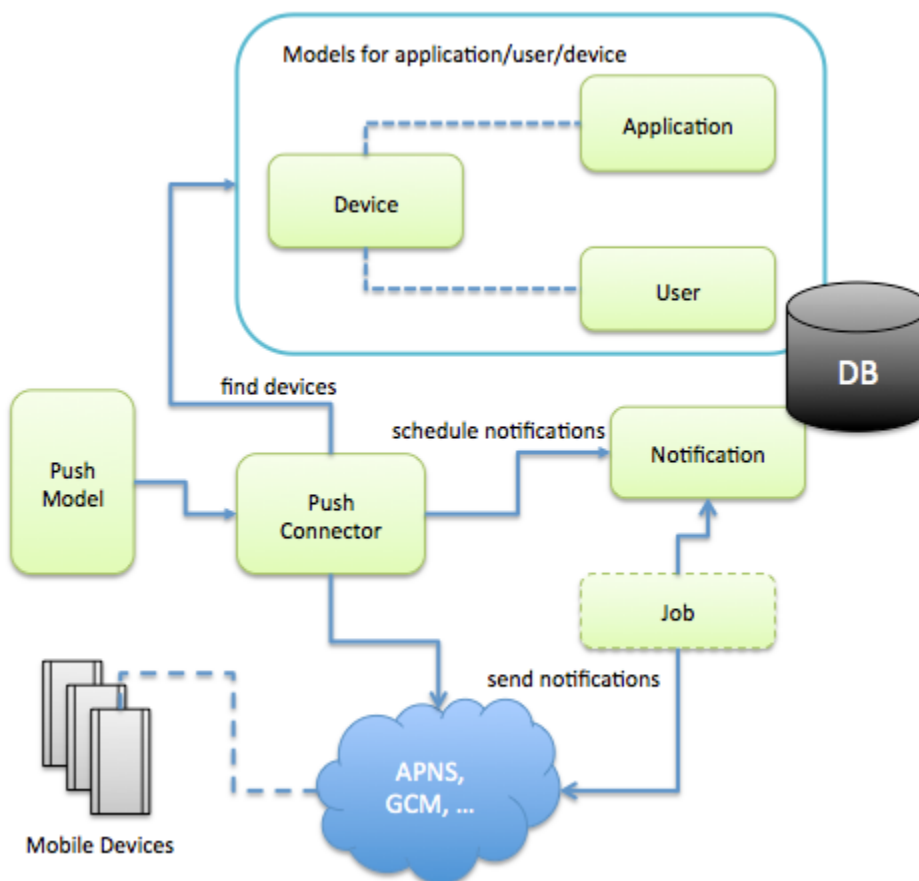
LoopBack has two mechanisms for reporting push notification errors:

- 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 Android apps

 For a complete working example Android app, see [LoopBack push notification Android sample app](#)..

- Overview
- Get your Google Cloud Messaging credentials
- Configure GCM push settings in your server Application
- Prepare your Android project
- Check for Google Play Services APK
- Create LocalInstallation
- Register with GCM if needed
- Register with LoopBack server

- [Handle received notifications](#)
- [Troubleshooting](#)

## Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application. See [Creating 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.

## 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. Follow these steps:

1. [Create a Google API Project](#)
2. [Enable the GCM service](#)
3. [Obtain the Android API key](#)
4. Create a new server API key that will be used by the LoopBack server:
  - a. In the sidebar on the left, select **APIs & auth > Credentials**.
  - b. Click **Create new key**.
  - c. Select **Server key**.
  - d. Leave the list of allowed IP addresses empty for now.
  - e. Click **Create**.
  - f. Copy down the API key. Later you will use this as SENDER\_ID in the Android app.

## 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"
 }
}
```

where `server-api-key` is the API key you obtained in the previous section (step 2).

## Prepare your 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 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 the device to make sure it 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 (error != null) {
 Log.e(TAG, "GCM Registration failed.", error);
 } else {
 saveInstallation(installation);
 }
 }
 }.execute(null, null, null);
}

```

## Register with LoopBack server

Once we have all Installation properties set, we 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 details of that are handled by the LoopBack Android SDK, your code just have 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 notifications for iOS apps

 For a complete working example iOS app, see [LoopBack push notification iOS sample app](#).

- [Overview](#)
- [Configure APN push settings in your server application](#)
- [Add LoopBack iOS SDK as a framework](#)
- [Initialize LBRESTAdapter](#)
- [Register the device](#)
- [Handle received notifications](#)

## Overview

This article provides information on creating iOS apps that can get push notifications from a LoopBack application. See [Creating 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.

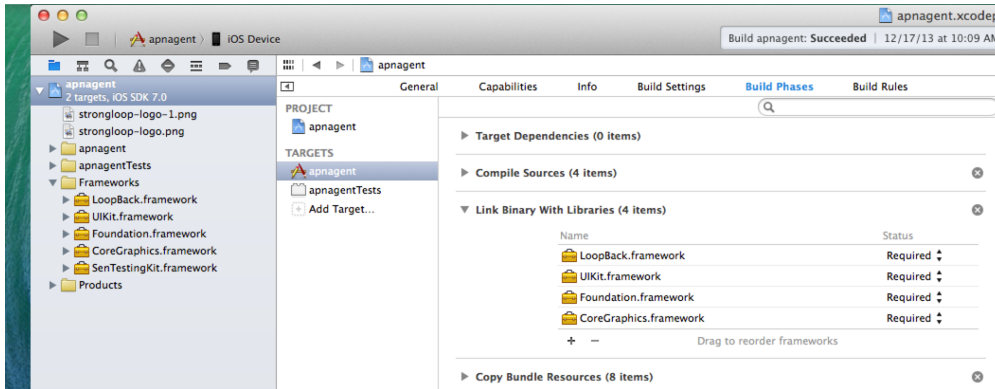
For general information on the Apple push notifications, see [Apple iOS Local and Push Notification Programming Guide](#). For additional useful information, see [Delivering iOS Push Notifications with Node.js](#).

## 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 notification API

- [Installation API](#)
- [Notification API](#)
- [PushManager API](#)

### Installation API



- [Installation](#)
- [Installation.findByApp](#)
- [Installation.findByUser](#)
- [Installation.findBySubscriptions](#)

Module: loopback-push-notification

#### **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-push-notification

## Notification

Notification Model

See the official documentation for more details on provider-specific properties.

[Android - GCM](#)

[iOS - APN](#)

## PushManager API



- [PushManager](#)
- [PushManager.providers](#)
- [pushManager.configureProvider](#)
- [pushManager.configureApplication](#)
- [pushManager.notifyById](#)
- [pushManager.notifyByQuery](#)
- [pushManager.notify](#)

Module: loopback-push-notification

### ***PushManager(settings)***

@class

#### Arguments

| Name     | Type   | Description       |
|----------|--------|-------------------|
| settings | Object | The push settings |

### ***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 |

### ***pushManager.configureApplication(appId)***

Lookup or set up push notification service for the given appId

#### Arguments

| Name  | Type   | Description        |
|-------|--------|--------------------|
| appId | String | The application id |

### ***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.  |
| 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=) |                                        |

## LoopBack Definition Language

- [Describing a simple model](#)
- [Advanced LDL features](#)
  - [Extending a model](#)
  - [Mixing in model definitions](#)

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 the `slc lb model` command to define a model. For more information, see [Creating a model using slc](#).

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

Note: The type name is case-insensitive; so for example you can use either "Number" or "number." See [LDL data 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.

Now we have the definition of a model, how do we use it in LoopBack Node.js code? It's easy, LoopBack will build a JavaScript constructor (or class) for you.

## Advanced LDL features

### 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);
```

## LDL data types

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                                  | { "firstName": "John",<br>"lastName": "Smith", "age": 25 } |
| Array    | JSON array                                   | [ "one", 2, true ]                                         |
| Date     | JavaScript <a href="#">Date object</a>       | new Date("December 17, 2003<br>03:24:00");                 |
| Buffer   | Node.js <a href="#">Buffer object</a>        | new Buffer(42);                                            |
| GeoPoint | LoopBack <a href="#">GeoPoint API object</a> | new GeoPoint({lat: 10.32424, lng:<br>5.84978});            |

### **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'.

## Model definition reference

- Overview
- Options
  - Relations
  - ACLs
  - Models
- Properties
  - Property options
    - Constraint options
    - Conversion and formatting options
    - Mapping options
    - Data source options
    - Model ID
- Extending a model

### Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

### Overview

You can define a model in `models.json` and you can programmatically define and extend models using the `Model` class. The following reference applies to both cases.

Below is a simple example of a `models.json` file generated by running the `slc lb project` command. The "user" key is the model name and its value (JSON) is the model definition.

#### models.json

```

{
 "user": {
 "options": { ... /* see "options" below */ ... },
 "properties": { ... /* See "properties" below */ ... }
 "dataSource": "db",
 "public": true
 }
}

```

| Key | Type | Description |
|-----|------|-------------|
|-----|------|-------------|

|            |         |                                                                                                                                                                                                                                                                                                                                                  |
|------------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| options    | Object  | <p>JSON object that specifies model options. See <a href="#">options</a> below.</p> <p>Passed to the Datasource Juggler when the model is defined.</p>                                                                                                                                                                                           |
| properties | Object  | <p>JSON object that specifies the properties in the model. See <a href="#">properties</a> below.</p>                                                                                                                                                                                                                                             |
| dataSource | String  | <p>Name of a datasource definition. The <code>Model</code> will be attached (using <code>Model.attachTo()</code>) to the given datasource.</p>                                                                                                                                                                                                   |
| public     | Boolean | <p>If <code>true</code>, the model is attached to the app and is made available over REST. If <code>false</code>, model is not available over REST.</p> <div style="border: 1px dashed black; background-color: yellow; padding: 5px; margin-top: 10px;"> <p><b>REVIEW COMMENT</b><br/>Is model attached to the app if this is false?</p> </div> |

## Options

The `options` key specifies model options, as illustrated in the example JSON below.

```
"options": {
 "base": "User",
 "relations": { ... /* See "relations" below */ ... },
 "acls": [... /* See "acls" below */]
}
```

| Key       | Type                | Description                                                                                                                         |
|-----------|---------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| base      | String (model name) | Base <code>Model</code> class being extended.                                                                                       |
| relations | Object              | Object containing relation names and relation definitions. See <a href="#">relations</a> below.                                     |
| acls      | Array               | Set of <code>ACL</code> specifications that describes access control for the model. See <a href="#">acl</a> below.                  |
| models    | Object              | A set of models the model depends on. Fulfills dependencies to other models when defining a new model. See <a href="#">models</a> . |

## 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. One of: <ul style="list-style-type: none"> <li>hasMany</li> <li>belongsTo</li> <li>hasAndBelongsToMany</li> </ul> |
| foreignKey | String | Optional foreign key used to find related model instances.                                                                                 |

## ACLs

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 |
|-----|------|-------------|
|-----|------|-------------|



|               |        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| permission    | String | Type of permission granted. Required.<br><br>One of: <ul style="list-style-type: none"> <li>• <b>ALARM</b> - Generate an alarm, in a system dependent way, the access specified in the permissions component of the ACL entry.</li> <li>• <b>ALLOW</b> - Explicitly grants access to the resource.</li> <li>• <b>AUDIT</b> - Log, in a system dependent way, the access specified in the permissions component of the ACL entry.</li> <li>• <b>DENY</b> - Explicitly denies access to the resource.</li> </ul>                                                                                                                                                                            |
| principalType | String | Type of the principal. Required.<br><br>One of: <ul style="list-style-type: none"> <li>• Application</li> <li>• User</li> <li>• Role</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| principalId   | String | Principal identifier. Required.<br><br>Examples: <ul style="list-style-type: none"> <li>• A user ID (String number any)</li> <li>• A dynamic role <ul style="list-style-type: none"> <li>• \$everyone - Everyone</li> <li>• \$owner - Owner of the Object</li> <li>• \$related - Any user with a relationship to the Object</li> <li>• \$authenticated - Authenticated user</li> <li>• \$unauthenticated - Unauthenticated user</li> </ul> </li> <li>• A static role name</li> </ul> <div style="border: 1px dashed black; padding: 5px; margin-top: 10px;"> <b>REVIEW COMMENT</b><br/> Are \$everyone, etc, string literals or identifiers? What is a static role name? A String? </div> |
| accessType    | String | The type of access to apply. One of: <ul style="list-style-type: none"> <li>• READ</li> <li>• WRITE</li> <li>• EXEC</li> <li>• ALL (default)</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| property      |        | Specifies a property/method/relation on a given model. It further constrains where the ACL applies.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## Models

### REVIEW COMMENT

Can we also provide a JSON example?

A set of models the model depends on. Fulfills dependencies to other models when defining a new model.

## Example

```

var Foo = Model.extend('Foo');

var options = {
 models: {
 MyFoo: 'Foo'
 }
};

var Bar = Model.extend('Bar', {}, options);

assert.equal(Bar.MyFoo, Foo);

```

## Properties

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

```

"properties": {
 "firstName": {"type": "String", "required": "true"}
},

```

| Key      | Type    | Description                                                                                                                                                            |
|----------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| type     | String  | Property type. Required. Can be any type described in <a href="#">LDL data types</a> .                                                                                 |
| required | Boolean | Whether a value for the property is required.<br><div><b>REVIEW COMMENT</b><br/>Is this a required property? i.e. if not specified, does it default to "false"?2</div> |
| doc      | String  | Documentation for the property. Optional.                                                                                                                              |
| options  | Object  | Property options. See <a href="#">Property options</a> below.                                                                                                          |

### Property options

A property definition can also have options defined by a JSON object, for example:

### Data-source specific mapping example

```
{
 "name": "Location",
 "options": {
 "idInjection": false,
 "oracle": {
 "schema": "BLACKPOOL",
 "table": "LOCATION"
 }
 },
 ...
}
```

The following table describes property options.

| Option                        | Type              | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------------|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| strict                        | Boolean           | <ul style="list-style-type: none"><li>• <code>true</code>: Only properties defined in the model are accepted. Use this mode if you want to make sure only predefined properties are accepted.</li><li>• <code>false</code>: 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.</li><li>• <code>undefined</code>: Default to false unless the data source is backed by a relational database such as Oracle or MySQL.</li></ul> |
| id                            | Boolean or Number | Whether the property is a unique identifier. See <a href="#">Model ID</a> below.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| idInjection                   | Boolean           | Whether to automatically add an id property to the model: <ul style="list-style-type: none"><li>• <code>true</code>: id property is added to the model automatically</li><li>• <code>false</code>: id property is not added to the model</li></ul>                                                                                                                                                                                                                                                                                                                                                                                |
| plural                        | String            | Plural form of the model name. If not present, the plural is derived from the model name following English conventions.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Data source specific mappings | Object            | Connector-specific options to customize the mapping between the model and the connector. See the above example of corresponding schema/table names for Oracle                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

```
"id": {"type": "number", "id": true, "doc": "User ID"}

"firstName": {"type": "string", "required": true, "oracle": {"column": "FIRST_NAME",
"type": "VARCHAR(32)"}}
```

Note the shorthand "id": "number" for "id": {"type": "number"}.

### Constraint options

Constraints are modeled as options too.

| 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 |
| min/max  | <div>REVIEW COMMENT<br/>How is this expressed?</div> | Property minimum and maximum values.                  |
| length   | Number                                               | Maximum allowed length of a string.                   |

### Conversion and formatting options

Format conversions can also be declared as options, as described in the following table:

| Key       | Type | Description                        |
|-----------|------|------------------------------------|
| trim      |      | Trim the string                    |
| lowercase |      | Convert the string to be lowercase |
| uppercase |      | Convert the string to be uppercase |
| format    |      | Format a Date                      |

**REVIEW COMMENT**  
Are these all Booleans? Or...?

### Mapping options

You can also specify data-source specific mappings with property options. For example, to map a property a column in an Oracle database table, use the following syntax:

```
"oracle": {"column": "FIRST_NAME", "type": "VARCHAR", "length": 32}
```

### Data source options

| Key       | Type   | Description                  |
|-----------|--------|------------------------------|
| connector | String | oracle, mysql, mongodb       |
| host      | String | Database server host name    |
| port      | Number | Database server port number. |
| user      | String | Database server username.    |

|                          |        |                                                    |
|--------------------------|--------|----------------------------------------------------|
| password                 | String | Database server password.                          |
| URL                      | String | Alternative to host + port + user + password.      |
| Cookiesecret             |        | required by Express cookie parser to sign cookies. |
| Arbitrary key/value pair | Any    |                                                    |

### Model ID

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

By default, if no ID properties are defined and the `idInjection` of the model options is `false`, LDL automatically adds an `id` property to the model as follows:

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

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.

LDL 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.

### 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);

```

## LoopBack API

- [Access context API](#)
- [AccessToken API](#)
- [Application API](#)
- [ACL API](#)
- [App API](#)
- [Data Source](#)
- [DataSource model API](#)
- [Email API](#)
- [GeoPoint API](#)
- [LoopBack object API](#)
- [Model](#)
- [Role API](#)
- [Token API](#)
- [User API](#)

### Access context API



- Class: `AccessContext`
  - `accessContext.addPrincipal`
  - `accessContext.getUserId`
  - `accessContext.getAppId`
  - `accessContext.isAuthenticated`
  - `accessContext.debug`
- Class: `Principal`
  - `principal.equals`

Module: loopback

- [Class: AccessRequest](#)
  - [accessRequest.isWildcard](#)

***accessContext = new AccessContext(context)***

Access context represents the context for a request to access protected resources

#### Arguments

| Name    | Type   | Description        |
|---------|--------|--------------------|
| context | Object | The context object |

***accessContext.addPrincipal(principalType, principalId, [principalName])***

Add a principal to the context

#### Arguments

| Name            | Type   | Description        |
|-----------------|--------|--------------------|
| principalType   | String | The principal type |
| principalId     | *      | The principal id   |
| [principalName] | String | The principal name |

***accessContext.getUserId()***

Get the user id

***accessContext.getAppId()***

Get the application id

***accessContext.isAuthenticated()***

Check if the access context has authenticated principals

***accessContext.debug()***

Print debug info for access context.

***principal = new Principal(type, id, [name])***

This class represents the abstract notion of a principal, which can be used to represent any entity, such as an individual, a corporation, and a login id

#### Arguments

| Name   | Type   | Description        |
|--------|--------|--------------------|
| type   | String | The principal type |
| id     | *      | The principl id    |
| [name] | String | The principal name |

***principal.equals(The)***

Compare if two principals are equal

#### Arguments

| Name | Type | Description     |
|------|------|-----------------|
| The  | p    | other principal |

***accessRequest = new AccessRequest(model, property, accessType, permission)***

A request to access protected resources

#### Arguments

| Name       | Type   | Description     |
|------------|--------|-----------------|
| model      | String | The model name  |
| property   | String |                 |
| accessType | String | The access type |
| permission | String | The permission  |

#### `accessRequest.isWildcard()`

Is the request a wildcard

## AccessToken API



- Class: `AccessToken`
  - `AccessToken.ANONYMOUS`
  - `AccessToken.createAccessTokenId`
  - `AccessToken.findForRequest`
  - `accessToken.validate`

Module: loopback

#### ***accessToken = new AccessToken***

Token based authentication and access control.

#### **AccessToken.ANONYMOUS**

Anonymous Token

```
assert(AccessToken.ANONYMOUS.id === '$anonymous');
```

#### **AccessToken.createAccessTokenId(callback)**

Create a cryptographically random access token id.

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

#### callback

| Name  | Type   | Description |
|-------|--------|-------------|
| err   | Error  |             |
| token | String |             |

#### **AccessToken.findForRequest(req, [options], callback)**

Find a token for the given `ServerRequest`.

#### Arguments

| Name      | Type                       | Description                   |
|-----------|----------------------------|-------------------------------|
| req       | <code>ServerRequest</code> |                               |
| [options] | Object                     | Options for finding the token |
| callback  | Function                   |                               |



## callback

| Name  | Type        | Description |
|-------|-------------|-------------|
| err   | Error       |             |
| token | AccessToken |             |

### accessToken.validate(callback)

Validate the token.

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

## callback

| Name    | Type    | Description |
|---------|---------|-------------|
| err     | Error   |             |
| isValid | Boolean |             |

## Application API



- ApplicationSchema
- crypto
- Class: Application
  - Application.register
  - application.resetKeys
  - Application.resetKeys
  - Application.authenticate

Module: loopback

### ApplicationSchema

Data model for Application

### crypto

Application management functions

### *application = new Application*

Manage client applications and organize their users.

### Application.register(Owner's, Name, Other, Callback)

Register a new application

#### Arguments

| Name     | Type    | Description        |
|----------|---------|--------------------|
| Owner's  | owner   | user id            |
| Name     | name    | of the application |
| Other    | options | options            |
| Callback | cb      | function           |

### application.resetKeys(callback)

Reset keys for the application instance

#### Arguments

|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |

#### Application.resetKeys(appId, callback)

Reset keys for a given application by the appId

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| appId    | Any      |             |
| callback | Function |             |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |

#### Application.authenticate(appId, key, callback)

Authenticate the application id and key.

matched will be one of

- clientKey
- javascriptKey
- restApiKey
- windowsKey
- masterKey

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| appId    | Any      |             |
| key      | String   |             |
| callback | Function |             |

#### callback

| Name    | Type   | Description                                                          |
|---------|--------|----------------------------------------------------------------------|
| err     | Error  |                                                                      |
| matched | String | <ul style="list-style-type: none"> <li>• The matching key</li> </ul> |

## ACL API



- ACLSchema
- Class: ACL
  - ACL.getMatchingScore
  - ACL.checkPermission
  - ACL.checkAccess
  - ACL.checkAccessForToken
- Class: Scope
  - Scope.checkPermission

Module: loopback

## ACLSchema

System grants permissions to principals (users/applications, can be grouped into roles).

Protected resource: the model data and operations (model/property/method/relation/...)

For a given principal, such as client application and/or user, is it allowed to access (read/write/execute) the protected resource?

## ACL

A Model for access control meta data.

### ACL.getMatchingScore(rule, req)

Calculate the matching score for the given rule and request

#### Arguments

| Name | Type          | Description   |
|------|---------------|---------------|
| rule | ACL           | The ACL entry |
| req  | AccessRequest | The request   |

### ACL.checkPermission(principalType, principalId, model, property, accessType, callback, )

Check if the given principal is allowed to access the model/property

#### Arguments

| Name          | Type     | Description                       |
|---------------|----------|-----------------------------------|
| principalType | String   | The principal type                |
| principalId   | String   | The principal id                  |
| model         | String   | The model name                    |
| property      | String   | The property/method/relation name |
| accessType    | String   | The access type                   |
| callback      | Function | The callback function             |
| undefined     | callback |                                   |

#### Callback

| Name   | Type            | Description           |
|--------|-----------------|-----------------------|
| err    | String or Error | The error object      |
| result | AccessRequest   | The access permission |

### ACL.checkAccess(context, callback)

Check if the request has the permission to access

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| context  | Object   |             |
| callback | Function |             |

### ACL.checkAccessForToken(token, model, modelId, method, callback)

Check if the given access token can invoke the method

#### Arguments

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

| Name     | Type        | Description      |
|----------|-------------|------------------|
| token    | AccessToken | The access token |
| model    | String      | The model name   |
| modelId  | *           | The model id     |
| method   | String      | The method name  |
| callback | Function    |                  |

#### callback

| Name    | Type            | Description            |
|---------|-----------------|------------------------|
| err     | String or Error | The error object       |
| allowed | Boolean         | is the request allowed |

### **scope = new Scope**

Resource owner grants/delegates permissions to client applications

For a protected resource, does the client application have the authorization from the resource owner (user or system)?

Scope has many resource access entries

### **Scope.checkPermission(scope, model, property, accessType, callback)**

Check if the given scope is allowed to access the model/property

#### Arguments

| Name       | Type     | Description                       |
|------------|----------|-----------------------------------|
| scope      | String   | The scope name                    |
| model      | String   | The model name                    |
| property   | String   | The property/method/relation name |
| accessType | String   | The access type                   |
| callback   | Function |                                   |

#### callback

| Name   | Type            | Description           |
|--------|-----------------|-----------------------|
| err    | String or Error | The error object      |
| result | AccessRequest   | The access permission |

## App API



- Class: App
  - app.remotes
  - app.model
  - app.models
  - app.dataSource
  - app.remoteObjects
  - app.remotes
  - app.docs
  - app.dataSources
  - app.enableAuth
  - app.boot
  - app.installMiddleware
  - app.listen

Module: loopback

## ***var app = loopback()***

### **LoopBackApplication**

The `App` object represents a Loopback application.

The `App` object extends [Express](#) and supports [Express / Connect middleware](#). See [Express documentation](#) for details.

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

app.get('/', function(req, res){
 res.send('hello world');
});

app.listen(3000);
```

### **app.remotes()**

Lazily load a set of [remote objects](#).

**NOTE:** Calling `app.remotes()` multiple times will only ever return a single set of remote objects.

### **app.model(modelName, config)**

Define and attach a model to the app. The `Model` will be available on the `app.models` object.

```
var Widget = app.model('Widget', {dataSource: 'db'});
Widget.create({name: 'pencil'});
app.models.Widget.find(function(err, widgets) {
 console.log(widgets[0]); // => {name: 'pencil'}
});
```

### **Arguments**

| Name      | Type   | Description                     |
|-----------|--------|---------------------------------|
| modelName | String | The name of the model to define |
| config    | Object | The model's configuration       |

### **config**

| Name         | Type   | Description                                                                                        |
|--------------|--------|----------------------------------------------------------------------------------------------------|
| dataSource   | String | The <code>DataSource</code> to attach the model to                                                 |
| [options]    | Object | an object containing <code>Model</code> options                                                    |
| [properties] | Object | object defining the <code>Model</code> properties in <a href="#">Loop Back Definition Language</a> |

### **app.models()**

Get the models exported by the app. Only models defined using `app.model()` will show up in this list.

There are two ways how to access models.

#### **1. A list of all models**

Call `app.models()` to get a list of all models.

```
var models = app.models();

models.forEach(function (Model) {
 console.log(Model.modelName); // color
});
```

## 2. By model name

`app.model` has properties for all defined models.

In the following example the `Product` and `CustomerReceipt` models are accessed using the `models` object.

```
var loopback = require('loopback');
var app = loopback();
app.boot({
 dataSources: {
 db: {connector: 'memory'}
 }
});

app.model('product', {dataSource: 'db'});
app.model('customer-receipt', {dataSource: 'db'});

// available based on the given name
var Product = app.models.Product;

// also available as camelCase
var product = app.models.product;

// multi-word models are available as pascal cased
var CustomerReceipt = app.models.CustomerReceipt;

// also available as camelCase
var customerReceipt = app.models.customerReceipt;
```

### **`app.dataSource(name, config)`**

Define a DataSource.

#### **Arguments**

| Name   | Type   | Description            |
|--------|--------|------------------------|
| name   | String | The data source name   |
| config | Object | The data source config |

### **`app.remoteObjects()`**

Get all remote objects.

### **`app.remotes()`**

Get the apps set of remote objects.

### **`app.docs()`**

Enable swagger REST API documentation.

*Note: This method is deprecated, use the extension [loopback-explorer](#) instead.*

#### **Options**

- `basePath` The basepath for your API - eg. `'http://localhost:3000'`.

#### **Example**

```
// enable docs
app.docs({basePath: 'http://localhost:3000'});
```

Run your app then navigate to [the API explorer](#). Enter your API basepath to view your generated docs.

### **`app.dataSources`**

An object to store `dataSource` instances.

### **app.enableAuth()**

Enable app wide authentication.

### **app.boot([options])**

Initialize an application from an options object or a set of JSON and JavaScript files.

#### **What happens during an app *boot*?**

1. **DataSources** are created from an `options.dataSources` object or `datasources.json` in the current directory
2. **Models** are created from an `options.models` object or `models.json` in the current directory
3. Any JavaScript files in the `./models` directory are loaded with `require()`.
4. Any JavaScript files in the `./boot` directory are loaded with `require()`.

#### **Options**

- `cwd` - *optional* - the directory to use when loading JSON and JavaScript files
- `models` - *optional* - an object containing `Model` definitions
- `dataSources` - *optional* - an object containing `DataSource` definitions

**NOTE:** *mixing `app.boot()` and `app.model(name, config)` in multiple files may result in models being **undefined** due to race conditions. To avoid this when using `app.boot()` make sure all models are passed as part of the `models` definition*

#### **Model Definitions**

The following is an example of an object containing two `Model` definitions: "location" and "inventory".

```

{
 "dealership": {
 // a reference, by name, to a dataSource definition
 "dataSource": "my-db",
 // the options passed to Model.extend(name, properties, options)
 "options": {
 "relations": {
 "cars": {
 "type": "hasMany",
 "model": "Car",
 "foreignKey": "dealerId"
 }
 }
 },
 // the properties passed to Model.extend(name, properties, options)
 "properties": {
 "id": { "id": true },
 "name": "String",
 "zip": "Number",
 "address": "String"
 }
 },
 "car": {
 "dataSource": "my-db"
 "properties": {
 "id": {
 "type": "String",
 "required": true,
 "id": true
 },
 "make": {
 "type": "String",
 "required": true
 },
 "model": {
 "type": "String",
 "required": true
 }
 }
 }
}

```

### Model definition properties

- `dataSource` - **required** - a string containing the name of the data source definition to attach the Model to
- `options` - *optional* - an object containing Model options
- `properties` *optional* - an object defining the Model properties in [LoopBack Definition Language](#)

### DataSource definition properties

- `connector` - **required** - the name of the [connector](#)

### `app.installMiddleware()`

Install all express middleware required by LoopBack.

It is possible to inject your own middleware by listening on one of the following events:

- `middleware:preprocessors` is emitted after all other request-preprocessing middleware was installed, but before any request-handling middleware is configured.

Usage:

```

app.once('middleware:preprocessors', function() {
 app.use(loopback.limit('5.5mb'))
});

```

- `middleware:handlers` is emitted when it's time to add your custom request-handling middleware. Note that you should not install any express routes at this point (express routes are discussed later).



Usage:

```
app.once('middleware:handlers', function() {
 app.use('/admin', adminExpressApp);
 app.use('/custom', function(req, res, next) {
 res.send(200, { url: req.url });
 });
});
```

- `middleware:error-loggers` is emitted at the end, before the loopback error handling middleware is installed. This is the point where you can install your own middleware to log errors.

Notes:

- The middleware function must take four parameters, otherwise it won't be called by express.
- It should also call `next(err)` to let the loopback error handler convert the error to an HTTP error response.

Usage:

```
var bunyan = require('bunyan');
var log = bunyan.createLogger({name: "myapp"});
app.once('middleware:error-loggers', function() {
 app.use(function(err, req, res, next) {
 log.error(err);
 next(err);
 });
});
```

Express routes should be added after `installMiddleware` was called. This way the express router middleware is injected at the right place in the middleware chain. If you add an express route before calling this function, bad things will happen: Express will automatically add the router middleware and since we haven't added request-preprocessing middleware like cookie & body parser yet, your route handlers will receive raw unprocessed requests.

This is the correct order in which to call `app` methods:

```
app.boot(__dirname); // optional

app.installMiddleware();

// [register your express routes here]

app.listen();
```

### **app.listen(cb)**

Listen for connections and update the configured port.

When there are no parameters or there is only one callback parameter, the server will listen on `app.get('host')` and `app.get('port')`.

```
// listen on host/port configured in app config
app.listen();
```

Otherwise all arguments are forwarded to `http.Server.listen`.

```
// listen on the specified port and all hosts, ignore app config
app.listen(80);
```

The function also installs a listening callback that calls `app.set('port')` with the value returned by `server.address().port`. This way the `port` param contains always the real port number, even when `listen` was called with port number 0.

### **Arguments**

| Name | Type      | Description                                            |
|------|-----------|--------------------------------------------------------|
| cb   | Function= | If specified, the callback will be added as a listener |

# Data Source

## Data Source object

[View docs for loopback in GitHub](#)

LoopBack models can manipulate data via the DataSource object. Attaching a DataSource to a Model adds instance methods and static methods to the Model; some of the added methods may be remote methods.

Define a data source for persisting models.

```
var oracle = loopback.createDataSource({
 connector: 'oracle',
 host: '111.22.333.44',
 database: 'MYDB',
 username: 'username',
 password: 'password'
});
```

## Methods

### dataSource.createModel(name, properties, options)

Define a model and attach it to a DataSource.

```
var Color = oracle.createModel('color', {name: String});
```

You can define an ACL when you create a new data source with the DataSource.create() method. For example:

```
var Customer = ds.createModel('Customer', {
 name: {
 type: String,
 acls: [
 {principalType: ACL.USER, principalId: 'u001', accessType: ACL.WRITE, permission: ACL.DENY},
 {principalType: ACL.USER, principalId: 'u001', accessType: ACL.ALL, permission: ACL.ALLOW}
]
 }, {
 acls: [
 {principalType: ACL.USER, principalId: 'u001', accessType: ACL.ALL, permission: ACL.ALLOW}
]
 }
});
```

### dataSource.discoverModelDefinitions([username], fn)

Discover a set of model definitions (table or collection names) based on tables or collections in a data source.

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

### dataSource.discoverSchema([owner], name, fn)

Discover the schema of a specific table or collection.

**Example schema from oracle connector:**

```
{
 "name": "Product",
 "options": {
```

```
"idInjection": false,
"oracle": {
 "schema": "BLACKPOOL",
 "table": "PRODUCT"
}
},
"properties": {
 "id": {
 "type": "String",
 "required": true,
 "length": 20,
 "id": 1,
 "oracle": {
 "columnName": "ID",
 "dataType": "VARCHAR2",
 "dataLength": 20,
 "nullable": "N"
 }
 },
 "name": {
 "type": "String",
 "required": false,
 "length": 64,
 "oracle": {
 "columnName": "NAME",
 "dataType": "VARCHAR2",
 "dataLength": 64,
 "nullable": "Y"
 }
 },
 "audibleRange": {
 "type": "Number",
 "required": false,
 "length": 22,
 "oracle": {
 "columnName": "AUDIBLE_RANGE",
 "dataType": "NUMBER",
 "dataLength": 22,
 "nullable": "Y"
 }
 },
 "effectiveRange": {
 "type": "Number",
 "required": false,
 "length": 22,
 "oracle": {
 "columnName": "EFFECTIVE_RANGE",
 "dataType": "NUMBER",
 "dataLength": 22,
 "nullable": "Y"
 }
 },
 "rounds": {
 "type": "Number",
 "required": false,
 "length": 22,
 "oracle": {
 "columnName": "ROUNDS",
 "dataType": "NUMBER",
 "dataLength": 22,
 "nullable": "Y"
 }
 },
 "extras": {
 "type": "String",
 "required": false,
 "length": 64,
 "oracle": {
 "columnName": "EXTRAS",
 "dataType": "VARCHAR2",
 "dataLength": 64,
 "nullable": "Y"
 }
 }
}
```

```

 }
 },
 "fireModes": {
 "type": "String",
 "required": false,
 "length": 64,
 "oracle": {
 "columnName": "FIRE_MODES",
 "dataType": "VARCHAR2",
 "dataLength": 64,
 "nullable": "Y"
 }
 }
}
}
}

```

#### **dataSource.enableRemote(operation)**

Enable remote access to a data source operation. Each [connector](#) has its own set of set remotely enabled and disabled operations. You can always list these by calling `dataSource.operations()`.

#### **dataSource.disableRemote(operation)**

Disable remote access to a data source operation. Each [connector](#) has its own set of set enabled and disabled operations. You can always list these by calling `dataSource.operations()`.

```

// all rest data source operations are
// disabled by default
var oracle = loopback.createDataSource({
 connector: require('loopback-connector-oracle'),
 host: '...',
 ...
});

// or only disable it as a remote method
oracle.disableRemote('destroyAll');

```

#### **Notes:**

- Disabled operations will not be added to attached models.
- Disabling the remoting for a method only affects client access (it will still be available from server models).
- Data sources must enable / disable operations before attaching or creating models.

#### **dataSource.operations()**

List the enabled and disabled operations.

```
console.log(oracle.operations());
```

#### **Output:**

```

{
 find: {
 remoteEnabled: true,
 accepts: [...],
 returns: [...],
 enabled: true
 },
 save: {
 remoteEnabled: true,
 prototype: true,
 accepts: [...],
 returns: [...],
 enabled: true
 },
 ...
}

```

## DataSource model API



- Class: [Model](#)
  - [Model.checkAccess](#)

Module: loopback

### ***model = new Model***

The built in loopback.Model.

#### Arguments

| Name | Type   | Description |
|------|--------|-------------|
| data | Object |             |

### **Model.checkAccess(token, modelId, method, The, callback)**

Check if the given access token can invoke the method

#### Arguments

| Name     | Type        | Description       |
|----------|-------------|-------------------|
| token    | AccessToken | The access token  |
| modelId  | *           | The model id      |
| method   | String      | The method name   |
| The      | callback    | callback function |
| callback | Function    |                   |

#### callback

| Name    | Type            | Description            |
|---------|-----------------|------------------------|
| err     | String or Error | The error object       |
| allowed | Boolean         | is the request allowed |

## Email API



- Class: [Email](#)
  - [email.send](#)

Module: loopback

### ***email = new Email***

The Email Model.

#### Properties

- to - { **String** } required
- from - { **String** } required
- subject - { **String** } required
- text - { **String** }
- html - { **String** }

### **email.send(options, callback)**

Send an email with the given `options`.

Example Options:

```
{
 from: "Fred Foo <foo@blurdybloop.com>", // sender address
 to: "bar@blurdybloop.com, baz@blurdybloop.com", // list of receivers
 subject: "Hello ", // Subject line
 text: "Hello world ", // plaintext body
 html: "Hello world " // html body
}
```

See <https://github.com/andris9/Nodemailer> for other supported options.

#### Arguments

| Name     | Type     | Description                                           |
|----------|----------|-------------------------------------------------------|
| options  | Object   |                                                       |
| callback | Function | Called after the e-mail is sent or the sending failed |

## GeoPoint API

### GeoPoint object

[View docs for loopback in GitHub](#)

The GeoPoint object represents a physical location.

Use the GeoPoint class.

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

Embed a latitude / longitude point in a [Model](#).

```
var CoffeeShop = loopback.createModel('coffee-shop', {
 location: 'GeoPoint'
});
```

You can query LoopBack models with a GeoPoint property and an attached data source using geo-spatial filters and sorting. For example, the following code finds the three nearest coffee shops.

```
CoffeeShop.attachTo(oracle);
var here = new GeoPoint({lat: 10.32424, lng: 5.84978});
CoffeeShop.find({where: {location: {near: here}}, limit:3}, function(err, nearbyShops) {
 console.info(nearbyShops); // [CoffeeShop, ...]
});
```

### Distance Types

**Note:** all distance methods use miles by default.

- miles
- radians
- kilometers
- meters
- miles
- feet
- degrees

### Methods

#### geoPoint.distanceTo(geoPoint, options)

Get the distance to another GeoPoint; for example:

```
var here = new GeoPoint({lat: 10, lng: 10});
var there = new GeoPoint({lat: 5, lng: 5});
console.log(here.distanceTo(there, {type: 'miles'})); // 438
```

### **GeoPoint.distanceBetween(a, b, options)**

Get the distance between two points; for example:

```
GeoPoint.distanceBetween(here, there, {type: 'miles'}) // 438
```

## **Properties**

### **geoPoint.lat**

The latitude point in degrees. Range: -90 to 90.

### **geoPoint.lng**

The longitude point in degrees. Range: -180 to 180.

## **LoopBack object API**



- [loopback](#)
- [loopback.version](#)
- [loopback.mime](#)
- [createApplication](#)
- [loopback.createDataSource](#)
- [loopback.createModel](#)
- [loopback.remoteMethod](#)
- [loopback.template](#)
- [loopback.memory](#)
- [loopback.getModel](#)
- [loopback.getModelByType](#)
- [loopback.setDefaultDataSourceForType](#)
- [loopback.getDefaultDataSourceForType](#)
- [loopback.autoAttach](#)
- [loopback.Model](#)

Module: loopback

### **loopback**

`loopback` is the main entry for LoopBack core module. It provides static methods to create models and data sources. The module itself is a function that creates loopback app. For example,

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

### **loopback.version**

Framework version.

### **loopback.mime**

Expose mime.

### **createApplication()**

Create an loopback application.

### **loopback.createDataSource(name, options)**

Create a data source with passing the provided options to the connector.

### **Arguments**

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

|         |        |            |
|---------|--------|------------|
| name    | String | (optional) |
| options | Object |            |

#### **loopback.createModel(name, properties, options)**

Create a named vanilla JavaScript class constructor with an attached set of properties and options.

##### **Arguments**

| Name       | Type   | Description                                                      |
|------------|--------|------------------------------------------------------------------|
| name       | String | <ul style="list-style-type: none"> <li>must be unique</li> </ul> |
| properties | Object |                                                                  |
| options    | Object | (optional)                                                       |

#### **loopback.remoteMethod(fn, options)**

Add a remote method to a model.

##### **Arguments**

| Name    | Type     | Description |
|---------|----------|-------------|
| fn      | Function |             |
| options | Object   | (optional)  |

#### **loopback.template(path)**

Create a template helper.

```
var render = loopback.template('foo.ejs');
var html = render({foo: 'bar'});
```

##### **Arguments**

| Name | Type   | Description                |
|------|--------|----------------------------|
| path | String | Path to the template file. |

#### **loopback.memory([name])**

Get an in-memory data source. Use one if it already exists.

##### **Arguments**

| Name   | Type   | Description                                                          |
|--------|--------|----------------------------------------------------------------------|
| [name] | String | The name of the data source. If not provided, the 'default' is used. |

#### **loopback.getModel(modelName)**

Look up a model class by name from all models created by loopback.createModel()

##### **Arguments**

| Name      | Type   | Description    |
|-----------|--------|----------------|
| modelName | String | The model name |

#### **loopback.getModelByType(The)**



Look up a model class by the base model class. The method can be used by LoopBack to find configured models in models.json over the base model.

Arguments

| Name | Type  | Description      |
|------|-------|------------------|
| The  | Model | base model class |

loopback.setDefaultDataSourceForType(type, dataSource)

Set the default dataSource for a given type.

Arguments

| Name       | Type                 | Description                          |
|------------|----------------------|--------------------------------------|
| type       | String               | The datasource type                  |
| dataSource | Object or DataSource | The data source settings or instance |

loopback.getDefaultDataSourceForType(type)

Get the default dataSource for a given type.

Arguments

| Name | Type   | Description         |
|------|--------|---------------------|
| type | String | The datasource type |


loopback.autoAttach()

Attach any model that does not have a dataSource to the default dataSource for the type the Model requests

loopback.Model

Built in models / services

Model

 You can also expose a model's instance and static methods to clients; see [Remote methods and hooks](#).

Model object

View docs for loopback in GitHub

A Loopback Model is a vanilla JavaScript class constructor with an attached set of properties and options. A Model instance is created by passing a data object containing properties to the Model constructor. A Model constructor will clean the object passed to it and only set the values matching the properties you define.

Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

```
// valid color
var Color = loopback.createModel('color', {name: String});
var red = new Color({name: 'red'});
console.log(red.name); // red

// invalid color
var foo = new Color({bar: 'bat baz'});
console.log(foo.bar); // undefined
```

Properties

A model defines a list of property names, types and other validation metadata. A DataSource uses this definition to validate a Model during

operations such as `save()`.

## Options

Some [DataSources](#) may support additional `Model` options.

Define A Loopbackmodel.

```
var User = loopback.createModel('user', {
 first: String,
 last: String,
 age: Number
});
```

## Methods

### **Model.attachTo(dataSource)**

Attach a model to a [DataSource](#). Attaching a [DataSource](#) updates the model with additional methods and behaviors.

```
var oracle = loopback.createDataSource({
 connector: require('loopback-connector-oracle'),
 host: '111.22.333.44',
 database: 'MYDB',
 username: 'username',
 password: 'password'
});

User.attachTo(oracle);
```

**Note:** until a model is attached to a data source it will **not** have any **attached methods**.

## Properties

### **Model.properties**

An object containing a normalized set of properties supplied to `loopback.createModel(name, properties)`.

Example:

```
var props = {
 a: String,
 b: {type: 'Number'},
 c: {type: 'String', min: 10, max: 100},
 d: Date,
 e: loopback.GeoPoint
};

var MyModel = loopback.createModel('foo', props);

console.log(MyModel.properties);
```

Outputs:

```
{
 "a": {type: String},
 "b": {type: Number},
 "c": {
 "type": String,
 "min": 10,
 "max": 100
 },
 "d": {type: Date},
 "e": {type: GeoPoint},
 "id": {
 "id": 1
 }
}
```

## CRUD and Query Mixins

Mixins are added by attaching a vanilla model to a [data source](#) with a [connector](#). Each [connector](#) enables its own set of operations that are mixed into a Model as methods. To see available methods for a data source call `dataSource.operations()`.

Log the available methods for a memory data source.

```
var ops = loopback
 .createDataSource({connector: loopback.Memory})
 .operations();

console.log(Object.keys(ops));
```

Outputs:

```
['create',
 'updateOrCreate',
 'upsert',
 'findOrCreate',
 'exists',
 'findById',
 'find',
 'all',
 'findOne',
 'destroyAll',
 'deleteAll',
 'count',
 'include',
 'relationNameFor',
 'hasMany',
 'belongsToMany',
 'hasAndBelongsToMany',
 'save',
 'isNewRecord',
 'destroy',
 'delete',
 'updateAttribute',
 'updateAttributes',
 'reload']
```

Here is the definition of the `count()` operation.

```
{
 accepts: [{ arg: 'where', type: 'object' }],
 http: { verb: 'get', path: '/count' },
 remoteEnabled: true,
 name: 'count'
}
```

## Static Methods

**Note:** These are the default mixin methods for a `Model` attached to a data source. See the specific connector for additional API documentation.

#### **Model.create(data, [callback])**

Create an instance of `Model` with given data and save to the attached data source. Callback is optional.

```
User.create({first: 'Joe', last: 'Bob'}, function(err, user) {
 console.log(user instanceof User); // true
});
```

**Note:** You must include a callback and use the created model provided in the callback if your code depends on your model being saved or having an id.

#### **Model.count([query], callback)**

Query count of `Model` instances in data source. Optional query param allows to count filtered set of `Model` instances.

```
User.count({approved: true}, function(err, count) {
 console.log(count); // 2081
});
```

#### **Model.find(filter, callback)**

Find all instances of `Model`, matched by query. Fields used for filter and sort should be declared with `{index: true}` in model definition.

##### **filter**

- **where** `Object` { key: val, key2: {gt: 'val2'}} The search criteria
- Format: {key: val} or {key: {op: val}}
- Operations:
  - gt: >
  - gte: >=
  - lt: <
  - lte: <=
  - between
  - inq: IN
  - nin: NOT IN
  - neq: !=
  - like: LIKE
  - nlike: NOT LIKE
- **include** `String`, `Object` or `Array` Allows you to load relations of several objects and optimize numbers of requests.
  - Format:
  - 'posts': Load posts
  - ['posts', 'passports']: Load posts and passports
  - {'owner': 'posts'}: Load owner and owner's posts
  - {'owner': ['posts', 'passports']}: Load owner, owner's posts, and owner's passports
  - {'owner': [{posts: 'images', 'passports'}]}: Load owner, owner's posts, owner's posts' images, and owner's passports
- **order** `String` The sorting order
- Format: 'key1 ASC, key2 DESC'
- **limit** `Number` The maximum number of instances to be returned
- **skip** `Number` Skip the number of instances
- **offset** `Number` Alias for skip
- **fields** `Object` | `Array` | `String` The included/excluded fields
- ['foo'] or 'foo' - include only the foo property
- ['foo', 'bar'] - include the foo and bar properties
- {foo: true} - include only foo
- {bat: false} - include all properties, exclude bat

Find the second page of 10 users over age 21 in descending order excluding the password property.

```
User.find({
 where: {
 age: {gt: 21}},
 order: 'age DESC',
 limit: 10,
 skip: 10,
 fields: {password: false}
},
console.log
);
```

**Note:** See the specific connector's [docs](#) for more info.

#### **Model.destroyAll([where], callback)**

Delete all Model instances from data source. **Note:** destroyAll method does not perform destroy hooks.

```
Product.destroyAll({price: {gt: 99}}, function(err) {
 // removed matching products
});
```

**\*\*NOTE:** *where* is optional and a where object... do NOT pass a filter object

#### **Model.findById(id, callback)**

Find instance by id.

```
User.findById(23, function(err, user) {
 console.info(user.id); // 23
});
```

#### **Model.findOne(where, callback)**

Find a single instance that matches the given where expression.

```
User.findOne({id: 23}, function(err, user) {
 console.info(user.id); // 23
});
```

#### **Model.upsert(data, callback)**

Update when record with id=data.id found, insert otherwise. **Note:** no setters, validations or hooks applied when using upsert.

#### **Custom static methods**

Define a static model method.

```

User.login = function (username, password, fn) {
 var passwordHash = hashPassword(password);
 this.findOne({username: username}, function (err, user) {
 var failErr = new Error('login failed');

 if(err) {
 fn(err);
 } else if(!user) {
 fn(failErr);
 } else if(user.password !== passwordHash) {
 MyAccessTokenModel.create({userId: user.id}, function (err, accessToken) {
 fn(null, accessToken.id);
 });
 } else {
 fn(failErr);
 }
 });
};
}

```

Setup the static model method to be exposed to clients as a [remote method](#).

```

loopback.remoteMethod(
 User.login,
 {
 accepts: [
 {arg: 'username', type: 'string', required: true},
 {arg: 'password', type: 'string', required: true}
],
 returns: {arg: 'sessionId', type: 'any'},
 http: {path: '/sign-in'}
 }
);

```

### ***Instance methods***

**Note:** These are the default mixin methods for a `Model` attached to a data source. See the specific connector for additional API documentation.

#### **model.save([options], [callback])**

Save an instance of a `Model` to the attached data source.

```

var joe = new User({first: 'Joe', last: 'Bob'});
joe.save(function(err, user) {
 if(user.errors) {
 console.log(user.errors);
 } else {
 console.log(user.id);
 }
});

```

#### **model.updateAttributes(data, [callback])**

Save specified attributes to the attached data source.

```

user.updateAttributes({
 first: 'updatedFirst',
 name: 'updatedLast'
}, fn);

```

#### **model.destroy([callback])**

Remove a model from the attached data source.

```
model.destroy(function(err) {
 // model instance destroyed
});
```

### Custom instance methods

Define an instance method.

```
User.prototype.logout = function (fn) {
 MySessionModel.destroyAll({userId: this.id}, fn);
}
```

Define a remote model instance method.

```
loopback.remoteMethod(User.prototype.logout)
```

## Relationships

### Model.hasMany(Model, options)

Define a “one to many” relationship.

```
// by referencing model
Book.hasMany(Chapter);
// specify the name
Book.hasMany('chapters', {model: Chapter});
```

Query and create the related models.

```
Book.create(function(err, book) {
 // create a chapter instance
 // ready to be saved in the data source
 var chapter = book.chapters.build({name: 'Chapter 1'});

 // save the new chapter
 chapter.save();

 // you can also call the Chapter.create method with
 // the `chapters` property which will build a chapter
 // instance and save the it in the data source
 book.chapters.create({name: 'Chapter 2'}, function(err, savedChapter) {
 // this callback is optional
 });

 // query chapters for the book using the
 book.chapters(function(err, chapters) {
 // all chapters with bookId = book.id
 console.log(chapters);
 });

 book.chapters({where: {name: 'test'}, function(err, chapters) {
 // all chapters with bookId = book.id and name = 'test'
 console.log(chapters);
 });
});
```

### Model.belongsTo(Model, options)

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 users and posts, and each post can be written by exactly one user.

```
Post.belongsTo(User, {as: 'author', foreignKey: 'userId'});
```

The code above basically says Post has a reference called author to User using the `userId` property of Post as the foreign key. Now we can access the author in one of the following styles:

```
post.author(callback); // Get the User object for the post author asynchronously
post.author(); // Get the User object for the post author synchronously
post.author(user) // Set the author to be the given user
```

#### **Model.hasAndBelongsToMany(Model, options)**

A `hasAndBelongsToMany` relation creates a direct many-to-many connection with another model, with no intervening model. For example, if your application includes users and groups, with each group having many users and each user appearing in many groups, you could declare the models this way,

```
User.hasAndBelongsToMany('groups', {model: Group, foreignKey: 'groupId'});
user.groups(callback); // get groups of the user
user.groups.create(data, callback); // create a new group and connect it with the user
user.groups.add(group, callback); // connect an existing group with the user
user.groups.remove(group, callback); // remove the user from the group
```

### **Validations**

#### **Model.validatesFormatOf(property, options)**

Require a model to include a property that matches the given format.

```
User.validatesFormat('name', {with: /\w+/});
```

#### **Model.validatesPresenceOf(properties...)**

Require a model to include a property to be considered valid.

```
User.validatesPresenceOf('first', 'last', 'age');
```

#### **Model.validatesLengthOf(property, options)**

Require a property length to be within a specified range.

```
User.validatesLengthOf('password', {min: 5, message: {min: 'Password is too short'}});
```

#### **Model.validatesInclusionOf(property, options)**

Require a value for property to be in the specified array.

```
User.validatesInclusionOf('gender', {in: ['male', 'female']});
```

#### **Model.validatesExclusionOf(property, options)**

Require a value for property to not exist in the specified array.

```
User.validatesExclusionOf('domain', {in: ['www', 'billing', 'admin']});
```

#### **Model.validatesNumericalityOf(property, options)**



Require a value for property to be a specific type of Number.

```
User.validatesNumericalityOf('age', {int: true});
```

#### **Model.validatesUniquenessOf(property, options)**

Ensure the value for property is unique in the collection of models.

```
User.validatesUniquenessOf('email', {message: 'email is not unique'});
```

**Note:** not available for all [connectors](#).

Currently supported in these connectors:

- [In Memory](#)
- [Oracle](#)
- [MongoDB](#)

#### **myModel.isValid()**

Validate the model instance.

```
user.isValid(function (valid) {
 if (!valid) {
 console.log(user.errors);
 // => hash of errors
 // => {
 // => username: [errmsg, errmsg, ...],
 // => email: ...
 // => }
 }
});
```

#### **loopback.ValidationError**

ValidationError is raised when the application attempts to save an invalid model instance.

Example:

```
{
 "name": "ValidationError",
 "status": 422,
 "message": "The Model instance is not valid. \n\n 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"
]
 }
 },
}
```

You might run into situations where you need to raise a validation error yourself, for example in a “before” hook or a custom model method.

```
MyModel.prototype.preflight = function(changes, callback) {
 // Update properties, do not save to db
 for (var key in changes) {
 model[key] = changes[key];
 }


 if (model.isValid()) {
 return callback(null, { success: true });
 }

 // This line shows how to create a ValidationError
 err = new ValidationError(model);
 callback(err);
}
```

**Shared methods**

Any static or instance method can be decorated as `shared`. These methods are exposed over the provided transport (eg. `loopback.rest`).

**Remote methods and hooks**

 For an example app that illustrates defining and using remote methods, see <https://github.com/ritch/loopback-example-remote-methods>.

**Remote methods and hooks**

[View docs for loopback in GitHub](#)

You can expose a Model's instance and static methods to clients. A remote method must accept a callback with the conventional `fn(err, result, ...)` signature.

**Static Methods**

**`loopback.remoteMethod(fn, [options])`**

Expose a remote method.

```
Product.stats = function(fn) {
 var statsResult = {
 totalPurchased: 123456
 };
 var err = null;

 // callback with an error and the result
 fn(err, statsResult);
}

loopback.remoteMethod(
 Product.stats,
 {
 returns: {arg: 'stats', type: 'object'},
 http: {path: '/info', verb: 'get'}
 }
);
```

**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 description. The callback argument is assumed; do not specify. |
|         |           |                                                                                                                      |

|             |    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| returns     | No | Describes the remote method's callback arguments; See Argument description. The <code>err</code> argument is assumed; do not specify.                                                                                                                                                                                                                                                                                                                                                      |
| http        | No | HTTP routing information: <b>http.path</b> : path (relative to the model) at which the method is exposed. May be a path fragment (for example, <code>/:myArg</code> ) that will be populated by an arg of the same name in the <code>accepts</code> description. For example, the <code>stats</code> method above will be at the whole path <code>/products/stats</code> . <b>http.verb</b> : HTTP method (verb) from which the method is available (one of: get, post, put, del, or all). |
| description | No | A text description of the method. This is used by API documentation generators like Swagger.                                                                                                                                                                                                                                                                                                                                                                                               |

### Argument description

The arguments description defines either a single argument as an object or an ordered set of arguments as an array. Each individual argument has keys for:

- `arg`: argument name
- `type`: argument datatype; must be a [loopback type](#).
- `required`: Boolean value indicating if argument is required.
- `root`: 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`: For input arguments: a function or an object describing mapping from HTTP request to the argument value, as explained 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

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                     | The value is looked up using <code>req.param</code> , which searches route arguments, the request body and the query string. |
| query                    | An alias for form (see above).                                                                                               |
| path                     | An alias for form (see above).                                                                                               |
| 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' } }
```

The use the second way to specify HTTP mapping for input parameters, specify a custom mapping function that looks like this:

```
{
 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.

### Remote hooks

Run a function before or after a remote method is called by a client.

```
// *.save == prototype.save
User.beforeRemote('*.save', function(ctx, user, next) {
 if(ctx.user) {
 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();
});
```

Remote hooks also support wildcards. 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

Remote hooks are provided with a Context `ctx` object which contains transport specific data (eg. for http: `req` and `res`). The `ctx` object also has a set of consistent apis across transports.

### *ctx.user*

A `Model` representing the user calling the method remotely. **Note:** this is undefined if the remote method is not invoked by a logged in user.

### *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.

## REST

When `loopback.rest` is used the following additional `ctx` properties are available.  
`ctx.req`

The express `ServerRequest` object. [See full documentation.](#)  
`ctx.res`

The express `ServerResponse` object. [See full documentation.](#)

## Role API



- [RoleMappingSchema](#)
- [RoleMapping](#)
- [roleMapping.application](#)
- [roleMapping.user](#)
- [roleMapping.childRole](#)
- [Class: Role](#)
  - [Role.registerResolver](#)
  - [Role.isOwner](#)
  - [Role.isAuthenticated](#)
  - [Role.isInRole](#)
  - [Role.getRoles](#)

Module: loopback

### **RoleMappingSchema**

Map principals to roles

### **RoleMapping**

Map Roles to

### **roleMapping.application(callback)**

Get the application principal

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

#### callback

| Name        | Type        | Description |
|-------------|-------------|-------------|
| err         | Error       |             |
| application | Application |             |

#### roleMapping.user(callback)

Get the user principal

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |
| user | User  |             |

#### roleMapping.childRole(callback)

Get the child role principal

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| callback | Function |             |

#### callback

| Name      | Type  | Description |
|-----------|-------|-------------|
| err       | Error |             |
| childUser | User  |             |

#### *role = new Role*

The Role Model

#### Role.registerResolver(, The)

Add custom handler for roles

#### Arguments

| Name | Type     | Description                                             |
|------|----------|---------------------------------------------------------|
|      | role     |                                                         |
| The  | resolver | resolver function decides if a principal is in the role |

#### Role.isOwner(modelClass, modelId, userId, callback)

Check if a given userId is the owner the model instance

#### Arguments

| Name       | Type     | Description     |
|------------|----------|-----------------|
| modelClass | Function | The model class |
| modelId    | *        | The model id    |
| userId     | *)       | The user id     |
| callback   | Function |                 |

#### Role.isAuthenticated(context, callback)

Check if the user id is authenticated

#### Arguments

| Name     | Type     | Description          |
|----------|----------|----------------------|
| context  | Object   | The security context |
| callback | Function |                      |

#### callback

| Name            | Type    | Description |
|-----------------|---------|-------------|
| err             | Error   |             |
| isAuthenticated | Boolean |             |

#### Role.isInRole(role, context, callback)

Check if a given principal is in the role

#### Arguments

| Name     | Type     | Description        |
|----------|----------|--------------------|
| role     | String   | The role name      |
| context  | Object   | The context object |
| callback | Function |                    |

#### callback

| Name     | Type    | Description |
|----------|---------|-------------|
| err      | Error   |             |
| isInRole | Boolean |             |

#### Role.getRoles(context, callback, callback)

List roles for a given principal

#### Arguments

| Name     | Type     | Description          |
|----------|----------|----------------------|
| context  | Object   | The security context |
| callback | Function |                      |
| callback | Function |                      |

#### callback

| Name | Type     | Description       |
|------|----------|-------------------|
|      | err      |                   |
| An   | String[] | array of role ids |

## Token API



- [loopback.token](#)

Module: loopback

### loopback.token(options)

#### Options

- **cookies** - An Array of cookie names
- **headers** - An Array of header names
- **params** - An Array of param names
- **model** - Specify an AccessToken class to use

Each array is used to add additional keys to find an accessToken for a request.

The following example illustrates how to check for an accessToken in a custom cookie, query string parameter and header called foo-auth.

```
app.use(loopback.token({
 cookies: ['foo-auth'],
 headers: ['foo-auth', 'X-Foo-Auth'],
 cookies: ['foo-auth', 'foo_auth']
}));
```

#### Defaults

By default the following names will be checked. These names are appended to any optional names. They will always be checked, but any names specified will be checked first.

- **access\_token**
- **X-Access-Token**
- **authorization**
- **access\_token**

**NOTE:** The `loopback.token()` middleware will only check for [signed cookies](#).

## User API



- [Model](#)
- [properties](#)
- **Class: User**
  - [User.login](#)
  - [User.logout](#)
  - [user.hasPassword](#)
  - [user.verify](#)
  - [User.confirm](#)
  - [User.resetPassword](#)

Module: loopback

#### Model

Module Dependencies.

#### properties

Default User properties.

#### *user = new User*

Extends from the built in `loopback.Model` type.

Default User ACLs.

- DENY EVERYONE \*
- ALLOW EVERYONE create



- ALLOW OWNER removeById
- ALLOW EVERYONE login
- ALLOW EVERYONE logout
- ALLOW EVERYONE findById
- ALLOW OWNER updateAttributes

### User.login(credentials, callback)

Login a user by with the given credentials.

```
User.login({username: 'foo', password: 'bar'}, function (err, token) {
 console.log(token.id);
});
```

#### Arguments

| Name        | Type     | Description |
|-------------|----------|-------------|
| credentials | Object   |             |
| callback    | Function |             |

#### callback

| Name  | Type        | Description |
|-------|-------------|-------------|
| err   | Error       |             |
| token | AccessToken |             |

### User.logout(accessTokenID, callback)

Logout a user with the given accessToken id.

```
User.logout('asd0a9f8dsj9s0s3223mk', function (err) {
 console.log(err || 'Logged out');
});
```

#### Arguments

| Name          | Type     | Description |
|---------------|----------|-------------|
| accessTokenID | String   |             |
| callback      | Function |             |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |

### user.hasPassword(password)

Compare the given password with the users hashed password.

#### Arguments

| Name     | Type   | Description             |
|----------|--------|-------------------------|
| password | String | The plain text password |

### user.verify(options)

Verify a user's identity by sending them a confirmation email.

```

var options = {
 type: 'email',
 to: user.email,
 template: 'verify.ejs',
 redirect: '/'
};

user.verify(options, next);

```

#### Arguments

| Name    | Type   | Description |
|---------|--------|-------------|
| options | Object |             |

#### User.confirm(userId, token, redirect, callback)

Confirm the user's identity.

#### Arguments

| Name     | Type     | Description                                |
|----------|----------|--------------------------------------------|
| userId   | Any      |                                            |
| token    | String   | The validation token                       |
| redirect | String   | URL to redirect the user to once confirmed |
| callback | Function |                                            |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |

#### User.resetPassword(options, callback)

Create a short lived access token for temporary login. Allows users to change passwords if forgotten.

#### Arguments

| Name     | Type     | Description |
|----------|----------|-------------|
| options  | Object   |             |
| callback | Function |             |

#### options

| Name  | Type   | Description              |
|-------|--------|--------------------------|
| email | String | The user's email address |

#### callback

| Name | Type  | Description |
|------|-------|-------------|
| err  | Error |             |

## REST API

- [Overview](#)
- [Configuration](#)
- [Request format](#)

For more information on using the LoopBack REST API, see:

- JSON query string encoding
- Response format

- Exposing models over a REST API
- REST connector

## Overview

LoopBack provides a number of [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                                                                                     |
|----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>user[name][first]=John<br/>&amp;user[email]=callback@strongloop.com</code>                         | <pre>{ user:   { name: { first: 'John' },     email: 'callback@strongloop.com' } }</pre> |
| <code>user[names][]=John<br/>&amp;user[names][]=Mary<br/>&amp;user[email]=callback@strongloop.com</code> | <pre>{ user:   { names: ['John', 'Mary'],     email: 'callback@strongloop.com' } }</pre> |
| <code>items=a<br/>&amp;items=b</code>                                                                    | <pre>{ items: ['a', 'b'] }</pre>                                                         |

For more examples, see [node-querystring](#)

## Response format

The response format for all requests is a JSON object or array if present. 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: <ul style="list-style-type: none"> <li>• where</li> <li>• include</li> <li>• order</li> <li>• limit</li> <li>• skip / offset</li> <li>• fields</li> </ul> |
| /accessTokens                    | PUT       | Deny               | Update / insert access token instance and persist to data source. Inherited from generic model API .                | JSON object (in request body)                                                                                                                                                                      |
| /accessTokens/ <i>id</i>         | GET       | Deny               | Find access token by ID: Return data for the specified access token instance ID. Inherited from generic model API . | <i>id</i> , the access token instance ID (in URI path)                                                                                                                                             |
| /accessTokens/ <i>id</i>         | PUT       | Deny               | Update attributes for specified access token ID and persist. Inherited from generic model API .                     | Query parameters: <ul style="list-style-type: none"> <li>• data - An object containing property name/value pairs</li> <li>• <i>id</i> - The model id</li> </ul>                                    |
| /accessTokens/ <i>id</i>         | DELETE    | Deny               | Delete access token with specified instance ID. Inherited from generic model API .                                  | <i>id</i> , access token ID (in URI path)                                                                                                                                                          |
| /accessTokens/ <i>id</i> /exists | GET       | Deny               | Check instance existence : Return true if specified access token ID exists. Inherited from generic model API .      | URI path: <ul style="list-style-type: none"> <li>• <i>id</i> - Model instance ID</li> </ul>                                                                                                        |
| /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                                                                                                                                                                                          |
|-----------------|-----------|--------------------|----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /acIs           | POST      | Allow              | Add ACL instance and persist to data source. Inherited from generic <a href="#">model API</a> .                            | JSON object (in request body)                                                                                                                                                                      |
| /acIs           | GET       | Deny               | Find all instances of ACLs that match specified filter. Inherited from generic <a href="#">model API</a> .                 | One or more filters in query parameters: <ul style="list-style-type: none"> <li>• where</li> <li>• include</li> <li>• order</li> <li>• limit</li> <li>• skip / offset</li> <li>• fields</li> </ul> |
| /acIs           | PUT       | Deny               | Update / insert ACL instance and persist to data source. Inherited from generic <a href="#">model API</a> .                | JSON object (in request body)                                                                                                                                                                      |
| /acIs/id        | GET       | Deny               | Find ACLs by ID: Return data for the specified acIs instance ID. Inherited from generic <a href="#">model API</a> .        | id, the ACL instance ID (in URI path)                                                                                                                                                              |
| /acIs/id        | PUT       | Deny               | Update attributes for specified acIs ID and persist. Inherited from generic <a href="#">model API</a> .                    | Query parameters: <ul style="list-style-type: none"> <li>• data - An object containing property name/value pairs</li> <li>• id - The model id</li> </ul>                                           |
| /acIs/id        | DELETE    | Deny               | Delete ACLs with specified instance ID. Inherited from generic <a href="#">model API</a> .                                 | id, acIs ID (in URI path)                                                                                                                                                                          |
| /acIs/id/exists | GET       | Deny               | Check instance existence : Return true if specified acIs ID exists. Inherited from generic <a href="#">model API</a> .     | URI path: <ul style="list-style-type: none"> <li>• id - Model instance ID</li> </ul>                                                                                                               |
| /acIs/count     | GET       | Deny               | Return the number of ACL instances that matches specified where clause. Inherited from generic <a href="#">model API</a> . | Where filter specified in query parameter                                                                                                                                                          |
| /acIs/findOne   | GET       | Deny               | Find first ACL instance that matches specified filter. Inherited from generic <a href="#">model API</a> .                  | Same as <a href="#">Find matching instances</a> .                                                                                                                                                  |

## 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: <ul style="list-style-type: none"><li>• where</li><li>• include</li><li>• order</li><li>• limit</li><li>• skip / offset</li><li>• fields</li></ul> |
| /applications                    | PUT       | Deny               | Update / insert application instance and persist to data source. Inherited from generic model API .                 | JSON object (in request body)                                                                                                                                                               |
| /applications/ <i>id</i>         | GET       | Deny               | Find applications by ID: Return data for the specified applications instance ID. Inherited from generic model API . | <i>id</i> , the application instance ID (in URI path)                                                                                                                                       |
| /applications/ <i>id</i>         | PUT       | Deny               | Update attributes for specified applications ID and persist. Inherited from generic model API .                     | Query parameters: <ul style="list-style-type: none"><li>• data - An object containing property name/value pairs</li><li>• <i>id</i> - The model id</li></ul>                                |
| /applications/ <i>id</i>         | DELETE    | Deny               | Delete application with specified instance ID. Inherited from generic model API .                                   | <i>id</i> , application ID (in URI path)                                                                                                                                                    |
| /applications/ <i>id</i> /exists | GET       | Deny               | Check instance existence : Return true if specified application ID exists. Inherited from generic model API .       | URI path: <ul style="list-style-type: none"><li>• <i>id</i> - Model instance ID</li></ul>                                                                                                   |
| /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

### REVIEW COMMENT

Waitin for resolution of issue #131 - Email.send is not a shared method

- 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.<br><br>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       |                    | <a href="#">Find installations by app ID</a> | Query parameters: <ul style="list-style-type: none"> <li>deviceType</li> <li>appID</li> <li>appVersion</li> </ul> |

|                          |        |  |                                                                                                                               |                                                                                                                                                                                        |
|--------------------------|--------|--|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /Installations/byUser    | GET    |  | Find installations by user ID                                                                                                 | Query parameters: <ul style="list-style-type: none"> <li>deviceType</li> <li>userId</li> </ul>                                                                                         |
| /Installations           | POST   |  | Add installation instance and persist to data source. Inherited from generic <a href="#">model API</a> .                      | JSON object (in request body)                                                                                                                                                          |
| /Installations           | GET    |  | Find all instances of installations that match specified filter. Inherited from generic <a href="#">model API</a> .           | One or more filters in query parameters: <ul style="list-style-type: none"> <li>where</li> <li>include</li> <li>order</li> <li>limit</li> <li>skip / offset</li> <li>fields</li> </ul> |
| /Installations           | PUT    |  | Update / insert installation instance and persist to data source. Inherited from generic <a href="#">model API</a> .          | JSON object (in request body)                                                                                                                                                          |
| /Installations/id        | GET    |  | Find installation by ID: Return data for the specified instance ID. Inherited from generic <a href="#">model API</a> .        | id, the installation ID (in URI path)                                                                                                                                                  |
| /Installations/id        | PUT    |  | Update installation attributes for specified installation ID and persist. Inherited from generic <a href="#">model API</a> .  | Query parameters: <ul style="list-style-type: none"> <li>data An object containing property name/value pairs</li> <li>id - The installation ID</li> </ul>                              |
| /Installations/id        | DELETE |  | Delete installation with specified instance ID. Inherited from generic <a href="#">model API</a> .                            | id, installation ID (in URI path)                                                                                                                                                      |
| /Installations/count     | GET    |  | Return number of installation instances that match specified where clause. Inherited from generic <a href="#">model API</a> . | Query parameter: "where" filter.                                                                                                                                                       |
| /Installations/id/exists | GET    |  | Check instance existence : Return true if specified user ID exists. Inherited from generic <a href="#">model API</a> .        | URI path: id installation ID                                                                                                                                                           |
| /Installations/findOne   | GET    |  | Find first installation instance that matches specified filter. Inherited from generic <a href="#">model API</a> .            | Same as <a href="#">Find matching instances</a> .                                                                                                                                      |

## 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": "756244503c9f95b49d7ff82120dc193cale3a7cb56f60c2ef2a19241e8f33305",
 "subscriptions": [],
 "created": "2014-01-09T23:18:57.194Z",
 "modified": "2014-01-09T23:18:57.194Z"
 },
 ...
]
```

### Errors

#### REVIEW COMMENT

What are error codes?

### 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

Arguments 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": "756244503c9f95b49d7ff82120dc193cae3a7cb56f60c2ef2a19241e8f33305",
 "subscriptions": [],
 "created": "2014-01-09T23:18:57.194Z",
 "modified": "2014-01-09T23:18:57.194Z"
 }
]
```

## Errors

### REVIEW COMMENT

What are error codes?

## 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 reference](#).

- [Add model instance](#)
- [Update / insert instance](#)
- [Check instance existence](#)
- [Find instance by ID](#)
- [Find matching instances](#)
- [Find first instance](#)
- [Delete model instance](#)
- [Get instance count](#)
- [Update model instance attributes](#)
- [Get associated model instances](#)

By default, LoopBack uses `/api` as the URI root for the REST API. You can change this in the `apiPath` variable in the application `app.js` file.

### Quick reference

| URI Pattern              | HTTP Verb | Default Permission                                                                                                | Action                                                                                                | Arguments                                                                                                                                                                                          |
|--------------------------|-----------|-------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>/model-name</code> | POST      | Allow<br>For more information on using the LoopBack REST API, see <a href="#">Exposing models over a REST API</a> | <a href="#">Add model instance for model <code>model-name</code> and persist to data source.</a>      | JSON object (in request body)                                                                                                                                                                      |
| <code>/model-name</code> | GET       | Allow                                                                                                             | <a href="#">Find matching instances of model <code>model-name</code> that match specified filter.</a> | One or more filters in query parameters: <ul style="list-style-type: none"> <li>• where</li> <li>• include</li> <li>• order</li> <li>• limit</li> <li>• skip / offset</li> <li>• fields</li> </ul> |
| <code>/model-name</code> | PUT       | Allow                                                                                                             | <a href="#">Update / insert instance of model <code>model-name</code> and persist to data source.</a> | JSON object (in request body)                                                                                                                                                                      |

### Related articles

- [Creating a LoopBack model](#)
- [Working with models and data sources](#)
- [Creating model relations](#)
- [Model definition reference](#)
- [Model API reference](#)
- [Model REST API](#)

|                                               |        |       |                                                                                                                                                 |                                                                                                                                                                         |
|-----------------------------------------------|--------|-------|-------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>/model-name/id/exists</i>                  | GET    | Allow | <a href="#">Check instance existence</a> : Return true if specified model instance ID exists for <i>model-name</i> .                            | URI path: <ul style="list-style-type: none"> <li><i>model-name</i> - Model name</li> <li><i>id</i> - Model instance ID</li> </ul>                                       |
| <i>/model-name/count</i>                      | GET    | Allow | <a href="#">Get instance count</a> for model <i>model-name</i> that matches specified where clause.                                             | Where filter specified in query parameter                                                                                                                               |
| <i>/model-name/findOne</i>                    | GET    | Allow | <a href="#">Find first instance</a> of model <i>model-name</i> that matches specified filter.                                                   | Same as <a href="#">Find matching instances</a> .                                                                                                                       |
| <i>/model-name/id</i>                         | GET    | Allow | <a href="#">Find instance by ID</a> : Return data for the specified model instance ID for model <i>model-name</i> .                             | <i>id</i> , the model instance ID (in URI path)                                                                                                                         |
| <i>/model-name/id</i>                         | PUT    | Allow | <a href="#">Update model instance attributes</a> for specified model instance and persist.                                                      | Query parameters: <ul style="list-style-type: none"> <li><i>data</i> An object containing property name/value pairs</li> <li><i>id</i> The model id</li> </ul>          |
| <i>/model-name/id</i>                         | DELETE | Allow | <a href="#">Delete model instance</a> for model <i>model-name</i> with specified instance ID.                                                   | Model instance ID, <i>instance-id</i> (in URI path)                                                                                                                     |
| <i>/model1-name/id</i><br><i>/model2-name</i> | GET    | Allow | <a href="#">Get associated model instances</a> for related models <i>model1</i> and <i>model2</i> and return instances of the associated model. | URI path: <ul style="list-style-type: none"> <li><i>model1-name</i> - model1 name</li> <li><i>id</i> - Instance ID</li> <li><i>model2-name</i> - model2 name</li> </ul> |

## 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/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

### REVIEW COMMENT

What's the difference between updating an instance and inserting a new one? Does it detect if there is an existing matching instance and update it if there is, otherwise add a new one? Or...?

Update an existing model instance or insert a new one into the data source

```
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/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/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/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

### REVIEW COMMENT

Is this actually the instance ID?

## Example

Request:

```
curl http://localhost:3000/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, as described in the following table.

| Filter type | Type                     | Description                                                                                                              |
|-------------|--------------------------|--------------------------------------------------------------------------------------------------------------------------|
| where       | Object                   | Search criteria. Format: {key: val} or {key: {op: val}} For list of valid operations, see Operations, below.             |
| include     | String, Object, or Array | Allows you to load relations of several objects and optimize numbers of requests. For format, see Include format, below. |
| order       | String                   | Sort order. Format: 'key1 ASC, key2 DESC', where ASC specifies ascending and DESC specifies descending order.            |

|               |                          |                                                                    |
|---------------|--------------------------|--------------------------------------------------------------------|
| limit         | Number                   | Maximum number of instances to return.                             |
| skip (offset) | Number                   | Skip the specified number of instances. Use offset as alternative. |
| fields        | Object, Array, or String | The included/excluded fields. For format, see fields below.        |

#### Operations available in where filter:

- gt: >
- gte: >=
- lt: <
- lte: <=
- between
- inq: IN
- nin: NOT IN
- neq: !=
- like: LIKE
- nlike: NOT LIKE

#### Include format:

- 'posts': Load posts
- ['posts', 'passports']: Load posts and passports
- {'owner': 'posts'}: Load owner and owner's posts
- {'owner': ['posts', 'passports']}: Load owner, owner's posts, and owner's passports
- {'owner': [{'posts': 'images'}, 'passports']}: Load owner, owner's posts, owner's posts' images, and owner's passports

#### Fields format:

- ['foo'] or 'foo' - include only the foo property
- ['foo', 'bar'] - include the foo and bar properties
- {foo: true} - include only foo
- {bat: false} - include all properties, exclude bat

For example,

- /weapons: Weapons
- /weapons?filter[limit]=2&filter[offset]=5: Paginated Weapons
- /weapons?filter[where][name]=M1911: Weapons with name M1911
- /weapons?filter[where][audibleRange][lt]=10: Weapons with audioRange < 10
- /weapons?filter[fields][name]=1&filter[fields][effectiveRange]=1: Only name and effective ranges
- /weapons?filter[where][effectiveRange][gt]=900&filter[limit]=3: The top 3 weapons with a range over 900 meters
- /weapons?filter[order]=audibleRange%20DESC&filter[limit]=3: The loudest 3 weapons
- /locations: Locations
- /locations?filter[where][geo][near]=153.536,-28.1&filter[limit]=3: The 3 closest locations to a given geo point

### Example

Request:

Find without filter:

```
curl http://localhost:3000/locations
```

Find with a filter:

```
curl http://localhost:3000/locations?filter%5Blimit%5D=2
```



For curl, [ needs to be encoded as %5B, and ] as %5D.

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

- **filter** - Filter that defines where, order, fields, skip, and limit. It's same as find's filter argument. See [Find matching instances](#) for more details.

## Example

Request:

```
curl http://localhost:3000/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/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

### REVIEW COMMENT

Assume there can be multiple properties to match in where clause?

### Example

Request - count without "where" filter

```
curl http://localhost:3000/locations/count
```

Request - count with a "where" filter

```
curl http://localhost:3000/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/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.

## Get associated model instances

Follow the relations from one model to another one to get instances of the associated model.

```
GET /model1Name/instanceID/model2Name
```

## Arguments

- *instanceID* - ID of instance in model1.
- *model1Name* - name of first model.
- *model2Name* - name of second related model.

## Example

Request:

```
curl http://localhost:3000/locations/88/inventory
```

Response:

```
[
 {
 "productId": "2",
 "locationId": "88",
 "available": 10,
 "total": 10
 },
 {
 "productId": "3",
 "locationId": "88",
 "available": 1,
 "total": 1
 }
]
```

## 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:<br>deviceQuery<br><br>Request body:<br>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: <ul style="list-style-type: none"> <li>• where</li> <li>• include</li> <li>• order</li> <li>• limit</li> <li>• skip / offset</li> <li>• fields</li> </ul> |
| /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: <ul style="list-style-type: none"> <li>• data - An object containing property name/value pairs</li> <li>• id - The model id</li> </ul>                                           |
| /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: <ul style="list-style-type: none"> <li>• id - Model instance ID</li> </ul>                                                                                                               |
| /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](#)

### 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. Inherited from generic <a href="#">model API</a> .            | One or more filters in query parameters: <ul style="list-style-type: none"> <li>• where</li> <li>• include</li> <li>• order</li> <li>• limit</li> <li>• skip / offset</li> <li>• fields</li> </ul> |
| /users                 | PUT    | Deny  | Update / insert user instance and persist to data source. Inherited from generic <a href="#">model API</a> .           | JSON object (in request body)                                                                                                                                                                      |
| /users/id              | GET    | Deny  | Find user by ID: Return data for the specified user ID. Inherited from generic <a href="#">model API</a> .             | id, the user ID (in URI path)                                                                                                                                                                      |
| /users/id              | PUT    | Deny  | Update user attributes for specified user ID and persist. Inherited from generic <a href="#">model API</a> .           | Query parameters: <ul style="list-style-type: none"> <li>• data An object containing property name/value pairs</li> <li>• id The model id</li> </ul>                                               |
| /users/id              | DELETE | Deny  | Delete user with specified instance ID. Inherited from generic <a href="#">model API</a> .                             | id, user ID (in URI path)                                                                                                                                                                          |
| /users/accessToken     | POST   | Deny  |                                                                                                                        |                                                                                                                                                                                                    |
| /users/id/accessTokens | GET    | Deny  | Returns access token for specified user ID.                                                                            | <ul style="list-style-type: none"> <li>• id, user ID, in URI path</li> <li>• where in query parameters</li> </ul>                                                                                  |
| /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: <ul style="list-style-type: none"> <li>• uid</li> <li>• token</li> <li>• redirect</li> </ul>                                                                                     |
| /users/count           | GET    | Deny  | Return number of user instances that match specified where clause. Inherited from generic <a href="#">model API</a> .  | Where filter specified in query parameter                                                                                                                                                          |
| /users/id/exists       | GET    | Deny  | Check instance existence : Return true if specified user ID exists. Inherited from generic <a href="#">model API</a> . | URI path: <ul style="list-style-type: none"> <li>• users - Model name</li> <li>• id - Model instance ID</li> </ul>                                                                                 |
| /users/findOne         | GET    | Deny  | Find first user instance that matches specified filter. Inherited from generic <a href="#">model API</a> .             | Same as <a href="#">Find matching instances</a> .                                                                                                                                                  |
| /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.                                                                                                                                                                         |

## 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.

## Client SDKs

LoopBack provides software development kits (SDKs) for:

- **iOS** - See [iOS SDK version 1.0](#). See also [LoopBack iOS API reference](#).
- **Android** - See [Android SDK](#). For API reference documentation, see [LoopBack Android API reference](#).
- Mobile web clients - *not yet released*.

Downloads:

- [Download Android SDK](#)
- [Download iOS SDK](#)

# Android SDK

- Overview
- Getting Started with the guide app
  - Prerequisites
  - Running the LoopBack server application
  - Downloading LoopBack Android guide app
  - Running the guide app
    - Troubleshooting
- Creating and working with your own app
  - Eclipse ADT setup
  - Android Studio setup
  - Working with the SDK
- Creating your own LoopBack model
  - Prerequisites
  - Define model class and properties
  - Define model repository
  - Add a little glue
  - Create and modify widgets

See also the [Android SDK API reference](#).

## Overview

The Android SDK provides simple API calls that enable your Android app to access a [LoopBack](#) server application. It enables you to interact with your models and data sources in a comfortable, first-class, native manner instead of using the clunky `AsyncHttpClient`, `JSONObject`, and similar interfaces.

- [Download Android SDK](#)

## Getting Started with the guide app

The easiest way to get started with the LoopBack Android SDK is with the LoopBack Android guide app. The guide app comes ready to compile with Android Studio and each tab in the app will guide you through the SDK features available to mobile apps.

## Prerequisites

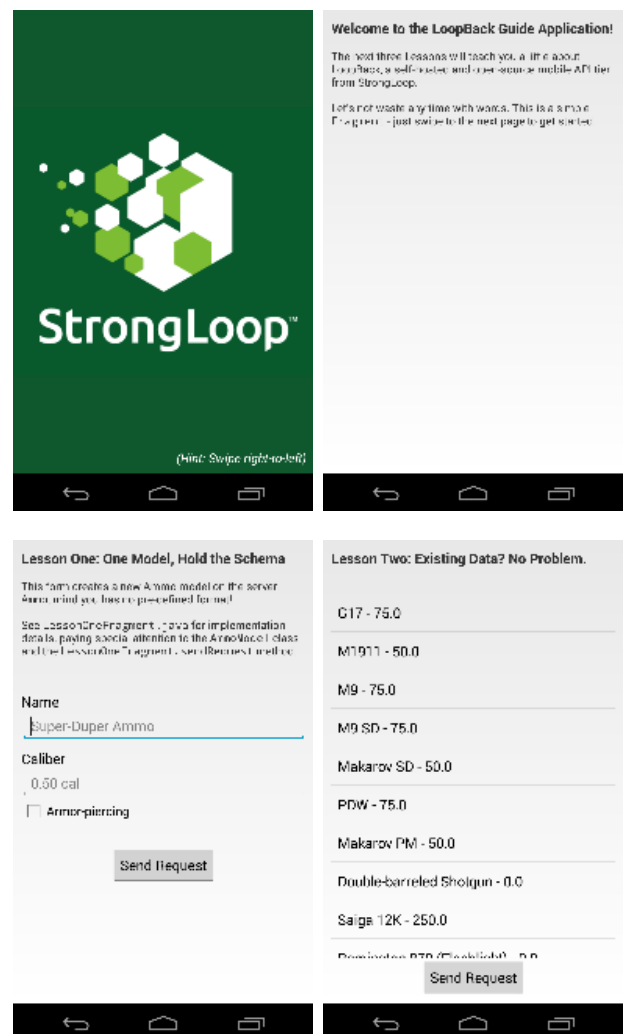
If you haven't already created your application backend, see the [LoopBack Quick Start](#). The Android guide app will connect to the this backend sample app.

Before you start, make sure you've installed the [Eclipse Android Development Tools](#) (ADT).

Now make sure you have the necessary SDK tools installed.

1. In ADT, choose **Window > Android SDK Manager**.
2. 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.
3. To run the LoopBack Android guide application (see below), also install **Extras > Google Play Services**.

Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager**. See [AVD Manager](#) for more information.



The guide application uses Google Maps Android API to render a map. As of November 2013, Google Maps are not supported by the Android emulator, so the application uses a list view of geo-locations. To see the Google Maps display, run the guide app on a real Android device instead of a virtual device



## Running the LoopBack server application

Start the StrongLoop Suite sample backend application. In the directory where you installed StrongLoop Suite, enter these commands:

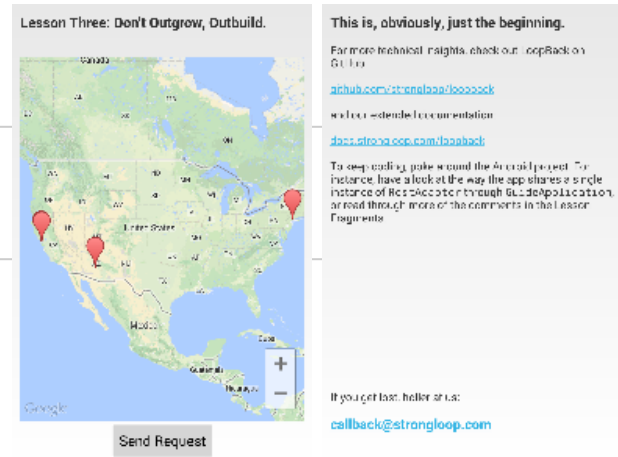
```
$ cd strongloop/samples/sls-sample-app
$ slc run app
```

## Downloading LoopBack Android guide app

To get the LoopBack Android guide application, you will need either the `git` command-line tool or a GitHub account.

To use `git`, enter this command:

```
$ git clone git@github.com:strongloop/loopback-android-getting-started.git
```



Alternatively, if you have a GitHub account, you can clone or download the repository as a zip file from <https://github.com/strongloop/loopback-android-getting-started>.

## Running the guide app

Follow these steps to run the LoopBack Android guide app:

1. Open ADT Eclipse.
2. Import the Loopback Guide Application to your workspace:
  - a. Choose **File > Import**.
  - b. Choose **Android > Existing Android Code into Workspace**.
  - c. Click **Next**.
  - d. Browse to the `loopback-android-getting-started/LoopbackGuideApplication` directory.
  - 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 Guide Application.
  - a. In the Package Explorer frame, select `LoopbackGuideApplication`
  - b. Choose **File > Project Properties**
  - c. Select **Android**
  - d. In the Library frame, click on **Add...** and select `google-play-services_lib`
5. Obtain an API key for Google Maps Android API v2 per [Getting Started instructions](#) and enter it into `AndroidManifest.xml`.
6. Click the green **Run** button in the toolbar to run the application. Each tab (fragment) shows a different way to interact with the LoopBack server. Look at source code of fragments to see implementation details.

It takes some time for the app to initialize: Eventually, you'll see an Android virtual device window. Click the LoopBack app icon in the home screen to view the LoopBack Android guide app.

## Troubleshooting

**Problem:** Build fails with the message `Unable to resolve target 'android-18'`.

**Resolution:** You need to install Android 4.3 (API 18) SDK. See [Prerequisites](#) for instructions on how to install SDK components.

If you don't want to install an older SDK and want to use the most recent one (for example, Android 4.4 API 19), follow these steps:

1. Close Eclipse ADT.
2. Edit the file `project.properties` in the `loopback-android-getting-started` directory and change the `target` property to the API version you have installed. For example: `target=android-19`.
3. Open Eclipse ADT again. The project should build correctly now.

## Creating and working with your own app

If you are creating a new Android application or want to integrate an existing application with LoopBack, then follow the steps in this section.

### Eclipse ADT setup

Follow these steps to add LoopBack SDK to your Eclipse project:

1. Download [LoopBack Android SDK for Eclipse](#).
2. Extract the content of the downloaded zip file and copy the contents of the `libs` folder into the `libs` folder of your Eclipse ADT project.

### Android Studio setup

1. Edit your `build.gradle` file.
2. Make sure you have `mavenCentral()` among the configured repositories:

```
repositories {
 mavenCentral()
}
```

3. Add `com.strongloop:loopback-android:1.+` to your compile dependencies:

```
dependencies {
 compile 'com.strongloop:loopback-android:1.+'
}
```

## Working with the SDK

For the complete API documentation, see [LoopBack Android API](#).

1. You need an adapter to tell the SDK where to find the server:

```
RestAdapter adapter = new RestAdapter("http://example.com");
```

This `RestAdapter` provides the starting point for all client interactions with the running server.

2. Once you have access to `adapter` (for the sake of example, assume the `Adapter` is available through our `Fragment` subclass), you can create basic `Model` and `ModelRepository` objects. Assuming you've previously created a [LoopBack model](#) named "product":

```
ModelRepository productRepository = adapter.createRepository("product");
Model pen = productRepository.createModel(ImmutableMap.of("name", "Awesome Pen")
);
```

All the normal `Model` and `ModelRepository` methods (for example, `create`, `destroy`, `findById`) are now available through `productRepository` and `pen`!

3. You can now start working with your model. Check out the [LoopBack Android API docs](#) or create more Models with the LoopBack CLI or Node API.

## Creating your own LoopBack model

Creating a subclass of `Model` enables your class to get the benefits of a Java class; for example, compile-time type checking.

### Prerequisites

- **Knowledge of Java and Android app development**
- **LoopBack Android SDK** - You should have set this up when you followed one of the preceding sections.
- **Schema** - Explaining the type of data to store and why is outside the scope of this guide, since it is tightly coupled to your application's needs.

### ***Define model class and properties***

As with any Java class, the first step is to build the interface. If you leave custom behavior for later, then it's just a few property declarations and you're ready for the implementation. In this simple example, each widget has a way to be identified and a price.

```
import java.math.BigDecimal;
import com.strongloop.android.loopback.Model;

/**
 * A widget for sale.
 */
public class Widget extends Model {

 private String name;
 private BigDecimal price;

 public void setName(String name) {
 this.name = name;
 }

 public String getName() {
 return name;
 }

 public void setPrice(BigDecimal price) {
 this.price = price;
 }

 public BigDecimal getPrice() {
 return price;
 }
}
```

### ***Define model repository***

The `ModelRepository` is the LoopBack Android SDK's placeholder for what in Node is a JavaScript prototype representing a specific "type" of Model on the server. In our example, this is the model exposed as "widget" (or similar) on the server:

```
var Widget = app.model('widget', {
 dataSource: "db",
 properties: {
 name: String,
 price: Number
 }
});
```

Because of this the class name ('widget', above) needs to match the name that model was given on the server. If you don't have a model, [see the LoopBack documentation](#) for more information. The model *must* exist (even if the schema is empty) before it can be interacted with.

Use this to make creating Models easier. Match the name or create your own.

Since `ModelRepository` provides a basic implementation, you need only override its constructor to provide the appropriate name.

```
public class WidgetRepository extends ModelRepository<Widget> {
 public WidgetRepository() {
 super("widget", Widget.class);
 }
}
```

### ***Add a little glue***

Just as in using the guide app, you need an `RestAdapter` instance to connect to the server:

```
RestAdapter adapter = new RestAdapter("http://myserver:3000");
```

**Remember:** Replace `"http://myserver:3000"` with the complete URL to your server.

Once you have that adapter, you can create our `Repository` instance.

```
WidgetRepository repository = adapter.createRepository(WidgetRepository.class);
```

### ***Create and modify widgets***

Now you have a `WidgetRepository` instance, you can:

Create a `Widget`:

```
Widget pencil = repository.createModel(ImmutableMap.of("name", "Pencil"));
pencil.price = new BigDecimal("1.50");
```

Save the `Widget`:

```
pencil.save(new Model.Callback() {
 @Override
 public void onSuccess() {
 // Pencil now exists on the server!
 }

 @Override
 public void onError(Throwable t) {
 // save failed, handle the error
 }
});
```

Find another `Widget`:

```
repository.findById(2, new ModelRepository.FindCallback<Widget>() {
 @Override
 public void onSuccess(Widget widget) {
 // found!
 }

 public void onError(Throwable t) {
 // handle the error
 }
});
```

Remove a Widget:

```
pencil.destroy(new Model.Callback() {
 @Override
 public void onSuccess() {
 // No more pencil. Long live Pen!
 }

 @Override
 public void onError(Throwable t) {
 // handle the error
 }
});
```

## iOS SDK



This article documents the latest version of the iOS SDK.

- [Overview](#)
  - [Prerequisites](#)
- [The LoopBack iOS guide app](#)
- [Getting Started with the iOS SDK](#)
- [Creating a sub-class of LBModel](#)
  - [Model interface and properties](#)
  - [Model implementation](#)
  - [Repository interface](#)
  - [Repository implementation](#)
  - [A little glue](#)
  - [Using the repository instance](#)

See also the [LoopBack iOS SDK API reference](#).

### Overview

The LoopBack iOS SDK eliminates the need to use the clunky `NSURLRequest` and similar interfaces to interact with a LoopBack-based backend. Instead, interact with your models and data sources in a comfortable, first-class, native manner.

- [Download iOS SDK](#)

### Prerequisites

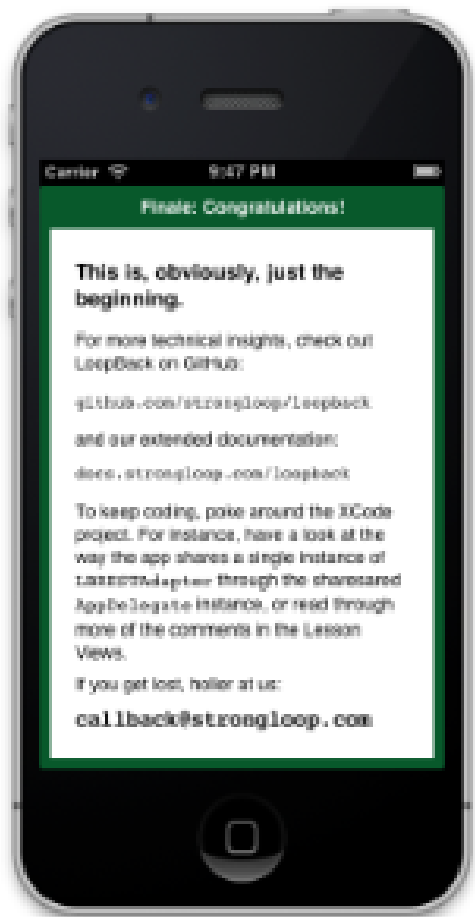
- Knowledge of Objective-C and iOS App Development
- Mac OSX with [Xcode](#) 4.6 or higher
- For on-device testing, an iOS device with iOS 5 or higher
- A LoopBack-powered server application.

- App schema. Explaining the type of data to store and why it is outside the scope of this guide, being tightly coupled to your application's needs.

### The LoopBack iOS guide app

The easiest way to get started with the LoopBack iOS SDK is with the LoopBack iOS guide app. The guide app comes ready to compile with XCode, and each tab in the app guides you through the features available to mobile apps through the SDK. Here are some representative screenshots:





From your usual projects directory:

1. Download the LoopBack guide application to your local machine from [GitHub](#):

```
$ git clone git@github.com:strongloop/loopback-ios-getting-started.git
```

2. Open the Xcode project downloaded as a part of the Guide Application's Git repository.

```
$ cd loopback-ios-getting-started\LoopBackGuideApplication
$ open LoopBackGuideApplication.xcodeproj
```

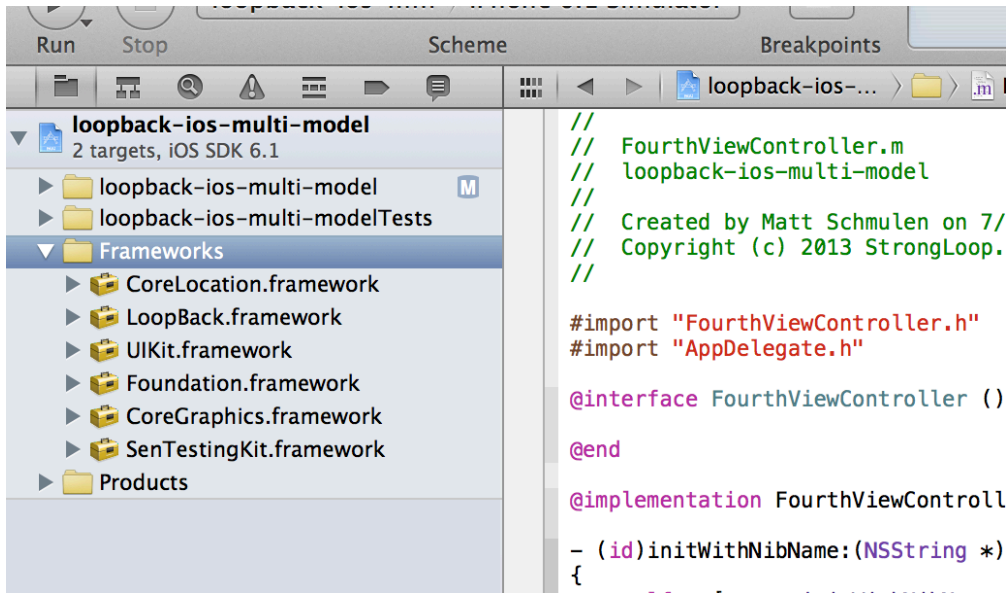
3. Run the application from Xcode (Command+R by default) and follow the instructions on each tab. Popup dialogs in the application will ask you to uncomment various code blocks in each ViewController illustrating how to use the LoopBack SDK to interact with models stored on the server.

## Getting Started with the iOS SDK

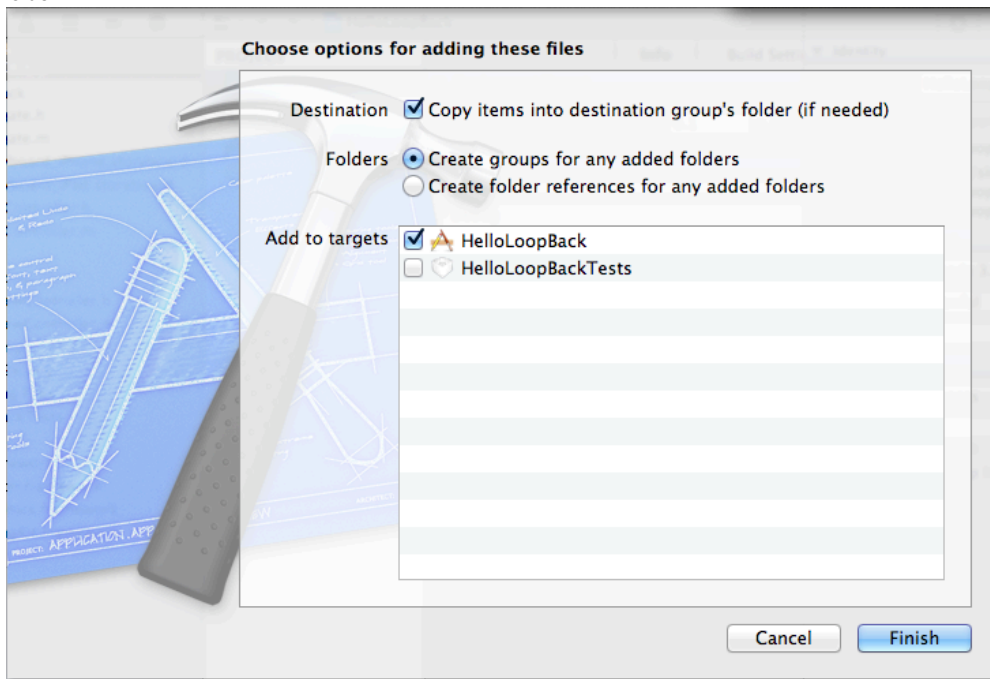
If you are creating a new iOS application or want to integrate an existing application with LoopBack, use the LoopBack SDK directly (LoopBack.framework), independent of the guide application.

Follow these steps:

1. Open the Xcode project you want to use with LoopBack, or create a new one.
2. Drag the entire LoopBack.framework folder from the new Finder window into your Xcode project.



Important: Make sure to select "Copy items into destination group's folder". This places a copy of the SDK within your application's project folder.

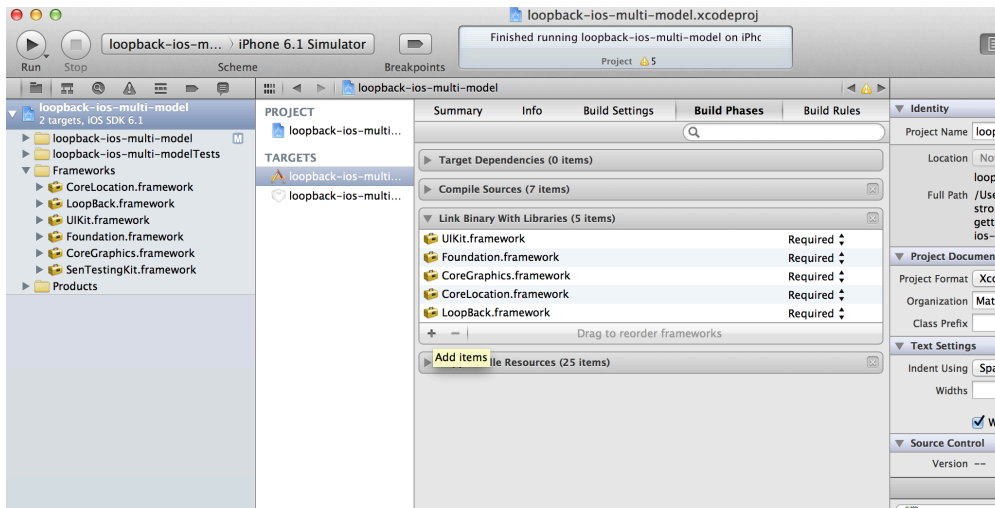


3. Verify LoopBack is included in the list of iOS Frameworks to link against your binary. In your Project settings, check the 'Link with Binaries' section under the 'Build Phases' tab. If it's missing, add it directly by clicking the '+' button and selecting LoopBack.framework.



If **LoopBack.framework** isn't displayed in the list, try the previous step again; Xcode didn't create the copy it was supposed to create.





4. Import the LoopBack.h header into your application just as you would Foundation/Foundation.h. Type this line:

```
#import <LoopBack/LoopBack.h>
```

5. 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 *productRepository = [adapter
repositoryWithModelName:@"products"];
LBModel *pen = [Product modelWithDictionary:@{ "name": "Awesome Pen" }];
```

All the normal LBModel and LBModelRepository methods (for example, create, destroy, and findById) are now available through Product and pen!

6. Go forth and develop! Check out the [API docs](#) or create more Models with the LoopBack [CLI](#) or [Node API](#).

## Creating a sub-class of LBModel

Creating a subclass of LBModel enables you to get the benefits of an Objective-C class (for example, compile-time type checking).

### Model interface and properties

As with any Objective-C class, the first step is to build your interface. If we leave any custom behavior for later, then it's just a few @property declarations and we're ready for the implementation.

```

/** * A widget for sale. */
@interface Widget : LBModel // This is a subclass, after all.

// Being for sale, each widget has a way to be identified and an amount of
// currency to be exchanged for it. Identifying the currency to be exchanged is
// left as an uninteresting exercise for any financial programmers reading this.
@property (nonatomic, copy) NSString *name;
@property (nonatomic) NSNumber *price;

@end

```

## Model implementation

Since we've left custom behavior for later, just leave this here.

```

@implementation Widget
@end

```

## Repository interface

The `LBModelRepository` is the LoopBack iOS SDK's placeholder for what in Node is a JavaScript prototype representing a specific "type" of Model on the server. In our example, this would be the model exposed as "widget" (or similar) on the server:

```

var Widget = app.model('widget', {
 dataSource: "db",
 properties: {
 name: String,
 price: Number
 }
});

```

Because of this the repository class name ('widget', above) needs to match the name that model was given on the server.



If you haven't created a model yet, see [Working with models and data sources](#). The model *must* exist (even if the schema is empty) before your app can interact with it.

Use this to make creating Models easier. Match the name or create your own.

Since `LBModelRepository` provides a basic implementation, we only need to override its constructor to provide the appropriate name.

```

@interface WidgetRepository : LBModelRepository

+ (instancetype)repository;

@end

```

## Repository implementation

Remember to use the right name:

```
@implementation WidgetRepository

+ (instancetype)repository {
 return [self repositoryWithClassName:@"widget"];
}

@end
```

## A little glue

Just as you did in Getting started, you'll need an `LBRESTAdapter` instance to connect to our server:

```
LBRESTAdapter *adapter = [LBRESTAdapter adapterWithURL:[NSURL
 URLWithString:@"http://myserver:3000"]];
```

Remember: Replace `"http://myserver:3000"` with the complete URL to your server.

Once you have that adapter, you can create a repository instance.

```
WidgetRepository *repository = (WidgetRepository *)[adapter
 repositoryWithModelClass:[WidgetRepository class]];
```

## Using the repository instance

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!
)
failure:^(NSError *error) {
 NSLog("An error occurred: %@", error);
}];
```