LoopBack - Core Documentation

LoopBack

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

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

Read LoopBack core concepts to learn about key concepts you need to understand to use LoopBack.

Follow Getting started with LoopBack for an introduction to some of LoopBack's key features.

Check out the LoopBack Developer Forum on Google Groups, a place where developers can ask questions and discuss how they are using LoopBack.

LoopBack documentation is available in other languages! See Docs in other languages.

The LoopBack framework

The LoopBack framework is a set of Node.js modules that you can use independently or together to quickly build applications that expose REST APIs.

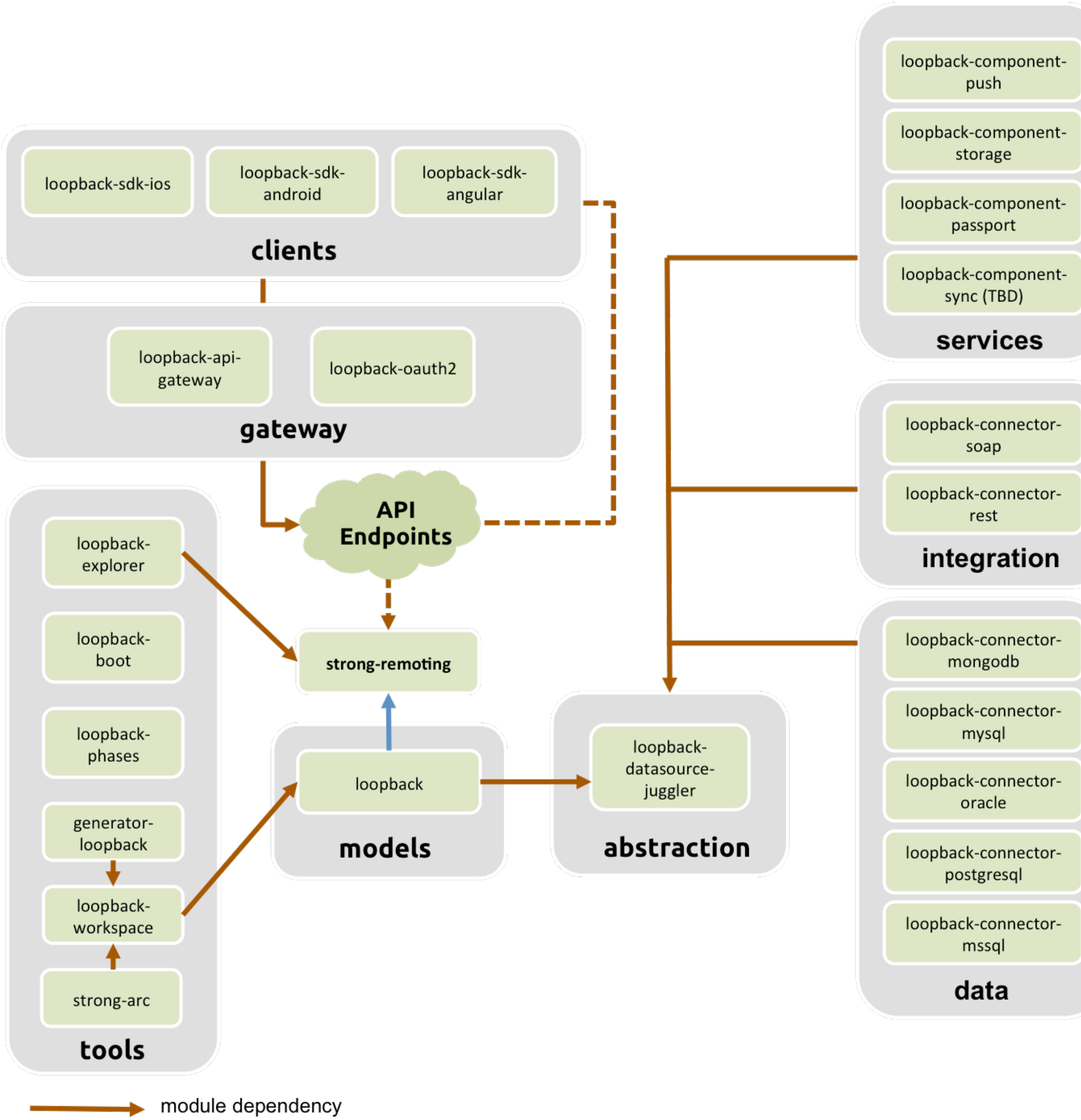
An application interacts with data sources through the LoopBack model API, available locally within Node.js, remotely over REST, and via native client APIs for iOS, Android, and HTML5. Using these APIs, apps can query databases, store data, upload files, send emails, create push notifications, register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

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

New to Node.js? Read Getting Started with Node.js for:

- PHP Developers
- Rails Developers
- Java Developers

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



LoopBack framework modules

Category	Description	Use to...	Modules
Models	Model and API server	Quickly and dynamically mock up models and expose them as APIs without worrying about persisting.	loopback
Abstraction	Model data abstraction to physical persistence	Connect to multiple data sources or services and get back an abstracted model with CRUD capabilities independent on how it is physically stored.	loopback-datasource-juggler
Initialization	Application initialization	Configure data-sources, custom models, configure models and attach them to data sources; Configure application settings and run custom boot scripts.	loopback-boot

Sequencing	Middleware execution	Configure middleware to be executed at various points during application lifecycle.	loopback-phase
Data	RDBMS and noSQL physical data sources	Enable connections to RDBMS, noSQL data sources and get back an abstracted model.	loopback-connector-mongodb loopback-connector-mysql loopback-connector-postgresql loopback-connector-msssql loopback-connector-oracle
Integration	General system connectors	Connect to an existing system that expose APIs through common enterprise and web interfaces	loopback-connector-rest loopback-connector-soap
Tooling	CLI and graphical tools	Yeoman generator used by `slc loopback` command; StrongLoop Arc graphical tool.	generator-loopback strong-arc
Services	Prebuilt services	Integrate with prebuilt services for common use cases to be utilized within LoopBack applications packaged into components.	loopback-component-push loopback-component-storage loopback-component-passport loopback-component-sync (in development)
Gateway	API gateway	Secure your APIs and inject quality of service aspects to the invocation and response workflow.	strong-gateway loopback-component-oauth2
Clients	Client SDKs	Develop your client app using native platform objects (iOS, Android, AngularJS) that interact with LoopBack APIs via REST.	loopback-sdk-ios loopback-sdk-android loopback-sdk-angular

Installing StrongLoop

Quickstart: If you've already installed Node, you're familiar with installing npm modules globally, and you want to start using LoopBack quickly, then just enter:

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

Then, check out Getting started with LoopBack.

Here's what `npm install -g strongloop` does:

- The LoopBack framework, including loopback, loopback-datasource-juggler modules, and numerous other related StrongLoop modules, along with modules that they require.
- The StrongLoop command-line tool, `slc`, for creating LoopBack applications and for running and managing Node applications.
- StrongLoop Arc, the unified graphical tool suite for the API lifecycle, including tools for building, profiling and monitoring Node apps.
- LoopBack Angular command line tools. See AngularJS JavaScript SDK for details.
- Various other tools, Including Yeoman, the LoopBack Yeoman generators to create and scaffold LoopBack applications, and Grunt, the JavaScript task runner.

If you want to profile your application or monitor application metrics, **determine whether you need to install compiler tools**.

Then, **follow the instructions for your operating system**:

- MacOS
- Windows
- Linux

Updating to the latest version

- Basic update
- Performing a clean re-installation

- Updating your Node.js installation

For application dependencies, npm will automatically update packages that your application requires, based on the information in the `package.json` file. For more information on `package.json`, see the npm documentation.

See Security advisories for important upgrade information required to address security issues.

Basic update

The current version of strongloop is .

Update your installation with this command:

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

Performing a clean re-installation

If you installed StrongLoop after 6 Aug 2014, and need to perform a clean reinstallation, follow these steps:

```
$ npm uninstall -g strongloop
$ npm cache clear
$ npm install -g strongloop
```

If you originally installed StrongLoop on or before 6 Aug 2014, follow the instructions in Uninstalling StrongLoop installation from before 6 Aug 2014.

If you have a LoopBack 1.x app

You must make changes to your old app to run it with LoopBack 2.0.

Follow the instructions in Migrating apps to version 2.0 to move your application to version 2.

Updating your Node.js installation

To update your version of Node, simply reinstall Node as you did previously. See nodejs.org for details.

LoopBack core concepts

Read this first to understand how LoopBack works. Then follow Getting started with LoopBack for a basic introduction to creating a LoopBack application.

- Models
- Application logic
- Data sources and connectors
- LoopBack components
- Development tools
- Examples

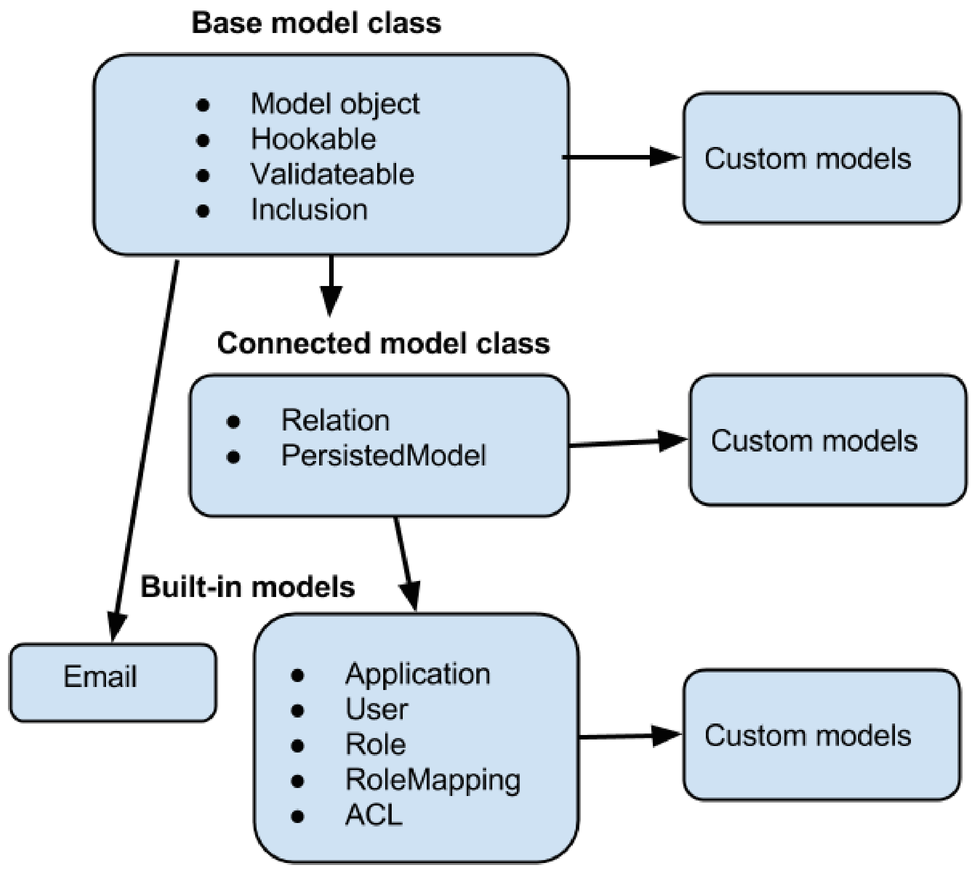
Model inheritance

Models

Models are at the heart of LoopBack, and represent back-end data sources such as databases or other back end services (REST, SOAP, and so on). LoopBack models are JavaScript objects with both Node and REST APIs.

A key powerful feature of LoopBack is that when you define a model it automatically comes with a predefined REST API with a full set of create, read, update, and delete (CRUD) operations.

The Basic model object has methods for adding hooks and for validating data. Other model objects all "inherit from" it. Models have an inheritance hierarchy, as shown at right: When you attach a model to a persistent data source it becomes a connected model with CRUD operations; LoopBack's built-in models inherit from it.



Built-in models

Every LoopBack application has a set of predefined built-in models such as User, Role, and Application, so you don't have to create these common models from scratch.

Custom models

You can define your own custom models specific to your application. You can make your custom models extend built-in models to build on the predefined functionality of User, Application, and other built-in models.

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

- With the model generator, `slc loopback:model`.
- From an existing relational database using *model discovery*. Then you can keep your model synchronized with the database using LoopBack's schema / model synchronization API.
- By instance introspection for free-form data in NoSQL databases or REST APIs.

All three of these methods create a Model definition JSON file that defines your model in LoopBack, by convention in a LoopBack project's `common/models` directory; for example, `common/models/Account.json`.

You can also create and customize models programmatically using the LoopBack API, or by manually editing the Model definition JSON file. In most cases, you shouldn't need to use those techniques to create models, but generally will to customize models for your use.

> The Model definition JSON file includes an `idInjection` property that indicates whether LoopBack automatically adds a unique `id` property to a model. For a model connected to a database, the id property corresponds to the primary key. See ID properties for more information.

Model relations

You can express relationships between models, such as BelongsTo, HasMany, and HasAndBelongsToMany.

Model CRUD operations

When you connect a model to a persistent data source such as a database, it becomes a *connected model* with a full set of create, read, update, and delete (CRUD) operations from the PersistedModel class:

Operation	REST	LoopBack model method (Node API)*	Corresponding SQL Operation
Create	PUT */modelName* POST */modelName*	`create()`*	INSERT
Read (Retrieve)	GET /modelName?filter=...	`find()`*	SELECT
Update (Modify)	POST */modelName* PUT /modelName	`updateAll()`*	UPDATE
Delete (Destroy)	DELETE */modelName/modelID*	`destroyById()`*	DELETE

*Methods listed are just prominent examples; other methods may provide similar functionality; for example: `findById()`, `findOne()`, and `findOrCreate()`. See PersistedModel API documentation for more information.

Application logic

You can add custom application logic in several ways; you can:

- Add application logic to models through remote methods (custom REST endpoints), remote hooks that are triggered by remote methods, and operation hooks that are triggered by model CRUD methods.
- Add boot scripts that run when the application starts.
- Define custom middleware, similar to Express middleware.

You can add code to Validate data before saving it to the model and back-end data store.

Middleware phases

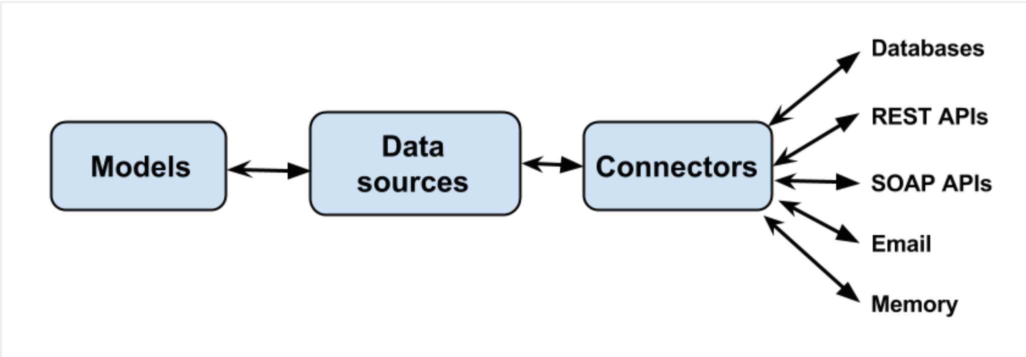
Middleware refers to functions executed when HTTP requests are made to REST endpoints. Since LoopBack is based on Express, LoopBack middleware is the same as Express middleware. However, LoopBack adds the concept of *phases*, to clearly define the order in which middleware is called. Using phases helps to avoid ordering issues that can occur with standard Express middleware.

See Defining middleware for more information.

Data sources and connectors

LoopBack generalizes backend services such as databases, REST APIs, SOAP web services, and storage services as *data sources*.

Data sources are backed by *connectors* that then communicate directly with the database or other back-end service. Applications don't use connectors directly, rather they go through data sources using the DataSource and PersistedModel APIs.



LoopBack components

LoopBack components provide additional "plug-in" functionality:

- Push notifications - enables sending information to mobile apps for immediate display in a "badge," alert, or pop-up message on the mobile device.
- Storage service - enables uploading and downloading files to and from cloud storage providers (Amazon, Rackspace, Openstack, and Azure) as well as the server file system.
- Third-party login - integrates Passport and enables user login (and account linking) using third-party credentials from Facebook, Google, Twitter, Github, or any system that supports OAuth, OAuth 2, or OpenID.
- Synchronization - enables mobile applications to operate offline and then synchronize data with the server application when reconnected.
- OAuth 2.0 - enables LoopBack applications to function as oAuth 2.0 providers to authenticate and authorize client applications and users to access protected API endpoints.

Development tools

LoopBack provides two primary application development tools:

- `slc loopback`, a command line tool for creating and modifying LoopBack applications.
- StrongLoop Arc, a graphical tool for developing, deploying, and monitoring LoopBack applications.

The `slc loopback` command-line tool guides you through the application development process with interactive prompts:

1. Start with the application generator to initially create and scaffold the basic structure of the application: **`slc loopback`**.
2. Add models (and model properties) using the model generator: **`slc loopback:model`**.
   If you need to add properties to existing models, use the property generator, **`slc loopback:property`**.
3. Add data sources using the data source generator, **`slc loopback:datasource`**.
4. Add relationships between models with the relation generator, **`slc loopback:relation`**.

Examples

StrongLoop provides numerous LoopBack example applications. The table below lists some of the key examples. Refer to https://github.com/strongloop/loopback-example for a complete list.

Name	Description
loopback-getting-started	The basics of LoopBack. Follow along in Getting started with LoopBack to build the example.
loopback-getting-started-intermediate	Full-stack example that builds on `loopback-getting-started` to demonstrate intermediate level features of LoopBack. Follow instructions in Getting started part II to build the example.
loopback-example-mongodb	LoopBack with MongoDB.
loopback-example-mssql	LoopBack with Microsoft SQL Server.
loopback-example-mysql	LoopBack with MySQL.
loopback-example-oracle	LoopBack with Oracle.
loopback-example-postgresql	LoopBack with PostgreSQL.
loopback-example-relations	Model relations and filtering via REST
loopback-example-app-logic	How to add your own logic to a LoopBack app
loopback-example-access-control	Controlling access to your API endpoints

Routing

Overview

LoopBack is built on Express and implements Express' routing system. However, basic Express routing is limited only to a small portion of the functionality of LoopBack. A large part of LoopBack's features are implemented using its more detailed extension to the routing system. Understanding this system will help you understand LoopBack better and develop better LoopBack apps.

Summary of Express routing

For those not familiar with routing in Express, here are some key points:

- Routing refers to the rules for capturing requests to the server, and the subsequent passing through and handling of requests through a chain of middleware functions.
- A middleware function accepts three objects, the request object (`req`), the response object (`res`), and the next middleware in the chain (`next`); in that order.
- You load middleware either using `app.use()` or by assigning it as the callback function of a route definition.
- Multiple middleware can be matched to handle the requests to a route, these matched middleware make up the middleware chain for the request. The request will pass through each middleware in the order they were loaded, unless one of the middleware in the chain terminates the propagation.

- Any middleware in the chain may terminate the request propagation by sending a response back to the client.
- A middleware can send the response to the request using one of the response methods in the response object or pass on the request to the next middleware by calling `next()`.
- If a middleware sends the server response, conventionally the request does not propagate further in the middleware chain. Any call to `next()` will likely result in an error.
- A middleware function can also take four arguments. In this case, it is an error handling middleware. The parameters to the function in their order are: the error object (err), the request object (`req`), the response object (`res`), and the next middleware in the chain (`next`).

For more details about routing in Express, see http://expressjs.com/guide/routing.html.

LoopBack routing

LoopBack implements the middleware pattern of request routing, therefore the concept of middleware and routing is the same as in Express. However, the Express aspect of routing is contained within the modified middleware chain generated by LoopBack. The extended routing details in LoopBack is best understood by understanding the LoopBack middleware chain.

LoopBack middleware chain

The LoopBack middleware chain is composed of middleware added to the chain in the following order, and the request passes through them in the given order.

> Except 14, 15, and 16, the listed items refer to LoopBack middleware phases.

1. initial:before
2. initial
3. initial:after
4. session:before
5. session
6. session:after
7. auth:before
8. auth
9. auth:after
10. parse:before
11. parse
12. parse:after
13. routes:before
14. Express middleware
15. Components
16. Boot scripts
17. routes
18. routes:after
19. files:before
20. files
21. files:after
22. final:before
23. final
24. final:after

A middleware loaded earlier in the chain gets the prior opportunity to handle the request. If it happens to send a response or fail to call `next()`, the request will terminate at it, and not propagate any further.

As evident from the list above, LoopBack takes control of the loading order of the middleware in the app, and prioritizes its relevant middleware over those loaded using Express' interface `app.use()`, components, or boot scripts.

> If you add middleware on the `route` or `route:after` phase, it will not execute *after* the route is matched. Instead, it will be ignored because the route was already matched.

The middleware to be loaded during the middleware phases are configured in the middleware.json file of the app. The order of the phases in the file are semantic, and cannot be listed randomly.

LoopBack also supports custom phases. Custom phases can be defined in any position of the middleware chain, and maybe be used to prioritize over LoopBack's built-in middleware phases.

Loading middleware

LoopBack middleware are declaratively loaded using the middleware.json file, or imperatively using the app.middleware() method.

Express middleware can be loaded in the `server.js` file using `app.use()` or a route definition.

LoopBack components can load middleware using the reference to the LoopBack application instance.

Boot scripts can load middleware using the reference to the LoopBack application instance.

LoopBack FAQ

General questions

StrongLoop supports the following operating systems:

- Red Hat Enterprise Linux (RHEL)/CentOS 6.3 (RPM)
- RHEL 6 and RHEL 7 on IBM z systems
- Debian/Ubuntu 12.10 (DEB)
- SUSE Linux Enterprise Server 11 and 12 on IBM z systems (with Node 0.12.7 and 4.2.2)
- Mac OS X Mountain Lion 10.8 (PKG)
- Microsoft Windows 8, 2008 (MSI)
  NOTE: Node does not support using Cygwin. Instead use Windows Command Prompt for command-line tools.

Is LoopBack free? How much does it cost?

There are free and paid versions of LoopBack. See http://strongloop.com/node-js/subscription-plans/ for more information.

LoopBack uses a dual license model: you may use it under the terms of the open source MIT license, or under the commercial StrongLoop License. See the license file for links to both licenses.

Is there a developer forum or mailing list?

Yes! The LoopBack Google Group is a place for devlopers to ask questions and discuss LoopBack and how they are using it. Check it out!

There is also a LoopBack Gitter channel for realtime discussions with fellow LoopBack developers.

StrongLoop also publishes a blog with topics relevant to LoopBack; see Blog posts for a list of the latest posts.

What client SDKs does LoopBack have?

LoopBack has three client SDKs for accessing the REST API services generated by the LoopBack framework:

- iOS SDK (Objective C) for iPhone and iPad apps. See iOS SDK for more information.
- Android SDK (Java) for Android apps. See Android SDK for more information.
- AngularJS (JavaScript) for HTML5 front-ends. See AngularJS JavaScript SDK for more information.

Which data connectors does LoopBack have?

LoopBack provides numerous connectors to access enterprise and other backend data systems.

Database connectors:

- Cloudant connector
- DB2 connector
- Memory connector
- MongoDB connector

- MySQL connector
- Oracle connector
- PostgreSQL connector
- Redis connector
- SQL Server connector

Other connectors:

- Email connector
- Push connector
- Remote connector
- REST connector
- SOAP connector
- Storage connector

Additionally, there are community connectors created by developers in the LoopBack open source community.

Why do curl requests to my LoopBack app fail?

If the URL loads fine in a browser, but when you make a `curl` request to your app you get the error:

```
curl: (7) Failed to connect to localhost port 3000: Connection refused
```

The cause is likely to be because of incompatible IP versions between your app and `curl`.

On Mac OS 10.10 (Yosemite), `curl` uses IP v6 by default.

LoopBack, by default uses IP v4, and `curl` might be using IP v6. If you see IP v6 entries in your hosts file (::1 localhost, fe80::1%lo0 localhost), it is likely that `curl` is making requests using IP v6. To make request using IP v4, specify the `--ipv4` option in your curl request as shown below.

```
$ curl http://localhost:3000 --ipv4
```

You can make your LoopBack app use IP v6 by specifying an IP v6 address as shown below:

```
app.start = function() {
  // start the web server
  return app.listen(3000, '::1',function() {
    app.emit('started');
    console.log('Web server listening at: %s', app.get('url'));
  });
};
```

If you are trying query filters with curl, use the `-g` or `--globoff` option to use brackets `[` and `]` in request URLs.

Detailed questions

Once you start working with LoopBack, you may have more detailed questions. Some of the most common are collected here, along with brief answers and links to the documentation for more information.

How do you perform a GET request to a remote server?

First, you have to configure a data source using the REST connector. In the datasources.json file that configures the data source, you can define operations against the REST API using the `operations` property.

For a short example, see loopback-example-rest-connector.

Can an application respond with XML instead of JSON?

Yes: in in `server/config.json` set the `remoting.rest.xml` property to `true`. See config.json for more information.

How do you send email from an application?

In brief:

1. Configure a datasource to use the email connector.
2. Map the built-in `Email` model to the the email datasource.
3. Send an email using the configured model with `Email.send()`.

See loopback-example-app-logic for a short example.

How do you use static middleware?

Static middleware enables an application to serve static content such as HTML, CSS, images, and client JavaScript files. To add it:

1. Remove the contents of the default `"routes"` property in `middleware.json`.
2. Add the following to the `"files"` property in `middleware.json`: to serve static content from the project's `/client` directory.

```
"loopback#static": {
  "params": "$!../client"
}
```

   Of course, change the value to use a different directory to contain static content.

See Defining middleware for more information, and loopback-example-middleware for a short example.

What kind of hooks do models support?

LoopBack models support:

- Operation hooks that execute when the model performs CRUD (create, read, update, and delete) operations.
- Remote hooks that execute before or after a remote method is called.

User management questions

See Managing users for more information and loopback-example-user-management for relevant code examples.

Note

- You must configure LoopBack to send email for email-related features.
- If you're using Gmail, simply replace user and pass with your own credentials.

How do you register a new user?

1. Create a form to gather sign up information.
2. Create a remote hook to send a verification email.

Notes:

- Upon execution, `user.verify` sends an email using the provided options.
- The verification email is configured to redirect the user to the `/verified` route in our example. For your app, you should configure the redirect to match your use case.
- The options are self-explanitory except `type`, `template` and `user`.
  - `type` - value must be `email`.
  - `template` - the path to the template to use for the verification email.
  - `user` - when provided, the information in the object will be used to in the verification link email.

How do you send an email verification for a new user registration?

See step 2 in the previous question.

How do you log in a user?

1. Create a form to accept login credentials.
2. Create an route to handle the login request.

How do you log a user out?

1. Create a logout link with the access token embedded into the URL.
2. Call `User.logout` with the access token.

Notes:

- We use the loopback token middleware to process access tokens. As long as you provide `access_token` in the query string of URL, the access token object will be provided in `req.accessToken` property in your route handler.

How do you perform a password reset for a registered user?

1. Create a form to gather password reset information.
2. Create an endpoint to handle the password reset request. Calling `User.resetPassword` ultimately emits a `resetPasswordRequest` event and creates a temporary access token.
3. Register an event handler for the `resetPasswordRequest` that sends an email to the registered user. In our example, we provide a URL that redirects the user to a password reset page authenticated with a temporary access token.
4. Create a password reset form for the user to enter and confirm their new password.
5. Create an endpoint to process the password reset.

Note: For the `resetPasswordRequest` handler callback, you are provided with an `info` object which contains information related to the user that is requesting the password reset.

Troubleshooting

Error message: loopback deprecated Routes "/methods" and "/models" are considered dangerous and should not be used

If you see this error message, you need to update your version of LoopBack. This issue is fixed in LoopBack 2.14+. See LoopBack PR #1135.

If you need to use an older version of LoopBack, or want to enable these routes, you can set a property in config.json to do so.

StrongLoop Labs

StrongLoop Labs projects provide early access to advanced or experimental functionality. In general, these projects may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports these projects: Paying customers can open issues using the StrongLoop customer support system (Zendesk), and community users can report bugs on GitHub.

These are the StrongLoop Labs projects:

- Android SDK
- iOS SDK
- LoopBack in the client
- Push notifications
- Redis connector
- Synchronization

Using the application object

Describes the LoopBack application object in detail, which will help developers understand the underlying system better and write better LoopBack components.

Security advisories

At StrongLoop, we take security very seriously. We make every effort to ensure the security of StrongLoop-supported modules, and to fix any vulnerabilities or provide workarounds as soon as possible.

These are important advisories about known security issues:

Some advisories may require action on your part, for example to upgrade StrongLoop packages.

How to report a security issue

If you think you have discovered a new security issue with any StrongLoop package, please do not report it on GitHub. Instead, send an email to callback@us.ibm.com with a full description and steps to reproduce.

Security advisory 01-12-2016

- **Security risk**: Medium critical
- **Vulnerability**: Hijacking Local Accounts

Description

If a user creates a local account, then tries to log in with a third-party account (for example, Facebook, Twitter, Google, etc.) that uses the same email address, then the local account gets hijacked and taken over by the third-party account.

Reported by

GitHub account jamesjjk

Versions affected

loopback-component-passport 1.6.0 and earlier.

Solution

Version 2.0.0 of loopback-component-passport fixes this issue.

Ensure that your application's `package.json` has the following line:

```
"dependencies": {
    ...
    "loopback-component-passport": "^1.6.0",

    ...
  },
```

Then upgrade your project dependencies to use the latest version :

```
$ cd <app-root>
$ npm update
```

Be sure that you have updated to loopback-component-passport **version 2.0.0** or higher.

Security advisory 06-04-2015

LoopBack and HTTP parameter pollution

HTTP parameter pollution is a known vulnerability of Express applications. LoopBack addressed the problem in the following module versions; to avoid this vulnerability, make sure your application uses these:

- `loopback` version 2.17.1 or newer.
- `strong-remoting` version 2.16.3 or newer.
- `loopback-datasource-juggler` version 2.26.1 or newer.

If you are using vanilla Express, then add hpp middleware to your application to protect against the vulnerability.

What is HTTP parameter pollution?

Let's start with a quick quiz. Consider the following HTTP request:

```
GET /search?firstname=John&firstname=Jane
```

Now, what is the value of `req.query.firstname`?

The correct answer is that `req.query.firstname` is set to an array of two items, `['John', 'Jane']`, because Express populates HTTP request parameters with an array when the parameter name is repeated multiple times.

While this is a useful feature that many modules depend on (including LoopBack), it allows an attacker to intentionally pollute request parameters that are not supposed to be an array and bypass input validation or even cause a denial of service.

For example, if the record above were a POST request the updated database record might end up as follows:

```
{
 "firstname": ["John", "Jane"],
 "lastname": "Smith"
}
```

If the handler expects the `firstname` parameter to be a string and calls one of the string prototype methods on the parameter, then the application may crash on an unhandled error:

```
TypeError: Object John,Jane has no method 'trim'
(...)
```

Consequences for LoopBack applications

Fortunately, LoopBack has an extra layer of abstraction on top of Express implemented by the `strong-remoting` module. This module understands the types of arguments that are passed to request handlers and thus it can effectively prevent parameter pollution.

Unfortunately the code that handles parameter type conversions was not implemented correctly in respect to this particular edge case. For example, the following code snippet allows the attacker to crash your server process:

```
Car.greet = function(whom, cb) {
 process.nextTick(function() {
   cb(null, 'Hello ' + whom.toUpperCase());
 });
};
Car.remoteMethod('greet', {
 isStatic: true,
 accepts: { arg: 'whom', type: 'string', required: true },
 returns: { arg: 'message', type: 'string' },
 http: { verb: 'GET' }
});
```

The request GET /cars/greet?whom=Jane&whom=John triggers unhandled exception in the server:

```
cb(null, 'Hello ' + whom.toUpperCase());
                        ^
TypeError: undefined is not a function
```

The vulnerability was fixed by strong-remoting#207. The fix revealed a bug in remoting metadata provided by LoopBack's built-in User model, and we addressed the problem in loopback#1332.

Beyond HTTP requests

There is one more place where LoopBack deals with request parameters: model properties. When setting model properties from request data, loopback-datasource-juggler (LoopBack's ORM framework) coerces data types to ensure values match the type provided by the model definition.

For example, one can send a string value "123" for a number property and loopback-datasource-juggler will automatically convert the string to the number 123.

Here is an overview of conversion result for array values:

- For String properties, an array value specified in the request is converted into a single comma-delimited string. For example, the request `{name: ['a','b']}` is converted to model data `{name: ['a,b']}`.
- For Number properties, when the request specifies an array value, then the following rule is applied:
  - An empty array is converted to 0. For example, `{count: []}` is converted to `{count: 0]}`.
  - An array of with a single numeric element is converted to a number. For example, `{count: [18]}` is converted to `{count: 18}`.
  - An other array values are converted to NaN. Note that NaN (not a number) values are later serialized as null in the JSON response body.
    For example, `{count: [18, 19]}` is converted to `{count: NaN}` and produces `{count: null}` in the server response.
- For Boolean properties, array values are converted to true. For example, `{isChecked: [1,2,3]}` is converted to `{isChecked: true}`.
- For Date properties, an array value is converted to a comma-delimited string first and then it's parsed as a date string. (This typically produces an "Invalid Date" value). For example, `{when: [2015,04,02]}` is converted to 2015-04-02T00:00:00.000.
- For Object or Any properties, the original array value is used.
- For properties of array type, for example ['number'] (array of numbers) or [string] (array of strings), each array item is converted using the appropriate rule from the rules above. Examples:
  - `{strings: [['a','b'], 'c']}` is converted to `{strings: ['a,b', 'c']}`.
  - `{numbers: [[1,2], 3]}` is converted to `{numbers: [NaN, 3]}`.

Although all looks good on the first sight, I found a flaw in the implementation of the validation rule "required" for numbers, where NaN value was considered as "truthy" and thus passed the validation. This can be exploited by passing an array value for a number property, for example:

```
> POST /api/records
{
  "count": [1,2,3]
}
< 200 OK
{
  "count": null
}
```

The issue was fixed by loopback-datasource-juggler#568.

Security advisory 01-09-2015

LoopBack connectors SQL injection vulnerability

If you installed LoopBack connectors for PostgreSQL, Microsoft SQL Server, Oracle, or MySQL prior to **9 Jan 2015** you need update the affected packages.

- **Date**: 09 Jan 2015
- **Security risk**: Highly critical
- **Vulnerability**: SQL Injection

Description

LoopBack allows you to define model properties (including id) as number types. A vulnerability in the implementations of relational database connectors allows an attacker to send specially crafted requests (SQL statements as the value of numbers) resulting in arbitrary SQL execution. This vulnerability can be exploited by anonymous users.

Reported by

David Kirchner

Versions affected

- loopback-connector-postgresql prior to 1.3.0
- loopback-connector-mssql prior to 1.3.0
- loopback-connector-oracle prior to 1.5.0
- loopback-connector-mysql prior to 1.5.0 (The SQL injection is not possible but invalid numbers are treated as NaN).

Solution

Please upgrade your project dependencies to use the latest versions of connectors and run **npm update**:

- loopback-connector-postgresql@1.3.0
- loopback-connector-mssql@1.3.0
- loopback-connector-oracle@1.5.0
- loopback-connector-mysql@1.5.0

Before running `npm update`, check your application's `package.json` to ensure that it specifies the correct version, for example:

```
"loopback-connector-oracle": "^1.5.0"
```

How to report security vulnerabilities?

Please send us an e-mail at **callback@strongloop.com**.

Getting started with LoopBack

Prerequisites: Before following this tutorial:

- Install StrongLoop software.
- Read LoopBack core concepts.

This tutorial will walk through the initial steps to create a basic LoopBack application.

The application you'll create is in the loopback-getting-started GitHub repository. To make it easy for you to pick up the tutorial at any point, there are tags for each step of the tutorial.

You can run through the steps in order to create the app and get a sense for some of the things LoopBack can do, or just skip to the one that interests you:

- Create a simple API
- Use API Explorer
- Connect your API to a data source
- Extend your API
- Add a static web page
- Add a custom Express route
- Run in a cluster
- Next steps

New to Node?

Read Getting Started with Node.js for...

- PHP Developers
- Rails Developers
- Java Developers

Create a simple API

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

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

Use the LoopBack command-line tool, `slc loopback`, to quickly create a LoopBack application, models, and data sources.

Use the LoopBack command-line tool, `slc loopback`, to create and *scaffold* applications. Scaffolding simply means generating the basic code for your application to save you time. You can then extend and modify the code as desired for your specific needs.

Before following this tutorial, make sure you have the latest version of StrongLoop with:

`$ npm install -g strongloop`

Make sure you run this command recently (within 24 hours) to get the latest updates. We're always improving LoopBack!

Create new application

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

```
$ slc loopback
```

The LoopBack generator will greet you with some friendly ASCII art and prompt you for the name of the application.

Enter `loopback-getting-started`. Then the generator will prompt you for the name of the directory to contain the project; press Enter to accept the default (the same as the application name):

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

You can use a different name for the application, but if you do, be sure to substitute your name for "loopback-getting-started" throughout the rest of this tutorial.

The generator will scaffold the application including:

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

Create models

Now that you've scaffolded the initial project, you're going to create create a *CoffeeShop* model that will automatically have REST API endpoints.

Go into your new application directory, then run the LoopBack model generator:

```
$ cd loopback-getting-started
$ slc loopback:model
```

The generator will prompt for a model name. Enter **CoffeeShop**:

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

It will ask if you want to attach the model to any data sources that have already been defined.

At this point, only the default in-memory data source is available. Press **Enter** to select it:

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

Then the generator will prompt you for the base class to use for the model. Since you will eventually connect this model to a persistent data source in a database, press down-arrow to choose **PersistedModel**, then press **Enter**:

```
[?] Select model's base class: (Use arrow keys)
  Model
 PersistedModel
  ACL
  AccessToken
  Application
  Change
  Checkpoint
```

PersistedModel is the base object for all models connected to a persistent data source such as a database. See LoopBack core concepts for an overview of the model inheritance hierarchy.

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

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

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

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

Press **Enter** to accept the default plural form (CoffeeShops):

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

Next, you'll be asked whether you want to create the model on the server only or in the `/common` directory, where it can potentially be used by both server and client LoopBack APIs. Keep, the default, common, even though in this application you'll only be working with server-side models:

```
? Common model or server only?
 common
  server
```

Every model has properties. Right now, you're going to define one property, "name," for the CoffeeShop model.

Select **`string`** as the property type (press **Enter**, since string is the default choice):

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

Each property can be optional or required. Enter **y** to make `name` required:

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

Then, you'll be prompted to add another property. Follow the prompts to add a required property named "city."

```
Let's add another CoffeeShop property.
? Property name: city
? Property type: string
? Required? Yes
```

End the model creation process by pressing **Enter** when prompted for the name of the next property.

The model generator will create two files in the application's `common/models` directory that define the model: `coffee-shop.json` and `coffee-shop.js`.

> ⚠ The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

Check out the project structure

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

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

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

> ⓘ All your model JSON and JavaScript files go in the `/common/models` directory.

File or directory	Description	How to access in code
Top-level application directory		

`/node-modules` directory	Contains Node packages as specified as dependencies in `package.json`. Update with `npm install`.	N/A
`package.json`	Standard npm package specification. See package.json.	N/A
`README.md`	Stub file for internal documentation.	N/A
/server directory - Node application files		
`/boot` directory	Add scripts to perform initialization and setup. See boot scripts.	Scripts are automatically executed in alphabetical order.
`component-config.json`	Specifies LoopBack components to load.	
`config.json`	Application settings. See config.json.	`app.get('setting-name')`
`datasources.json`	Data source configuration file. See datasources.json. For an example, see Create new data source.	`app.datasources['datasource-name']`
`middleware.json`	Middleware definition file. For more information, see Defining middleware.	N/A
`middleware.production.json`	Middleware definition file with production configuration. See Preparing for deployment.	
`model-config.json`	Model configuration file. See model-config.json. For more information, see Connecting models to data sources.	N/A
`server.js`	Main application program file.	N/A
/client directory - Client application files		
README.md	LoopBack generators create empty `README.md` file.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
/common directory - shared application files		
`/models` directory	Custom model files: • Model definition JSON files, by convention named *`model-name`*`.json`; for example `customer.json`. • Custom model scripts by convention named *`model-name`*`.js`; for example, `customer.js`. For more information, see Model definition JSON file and Customizing models. See note below.	Node: `myModel = app.models.myModelName`

> The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

For all the details of the canonical LoopBack application structure, see Project layout reference.

Run the application

Start the application:

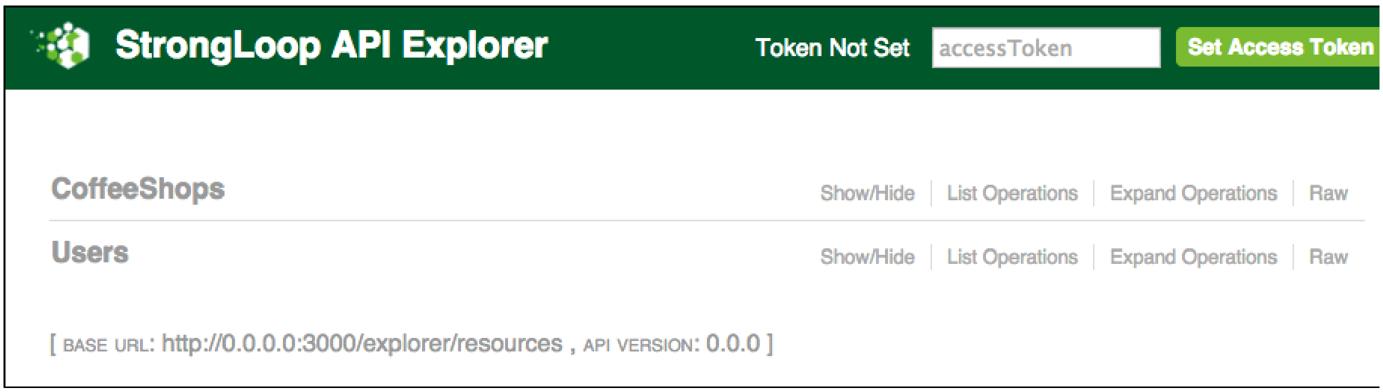
```
$ node .
...
Browse your REST API at http://0.0.0.0:3000/explorer
Web server listening at: http://0.0.0.0:3000/
```

> Running your app with the `node` command is appropriate when you're developing on your local machine. Once you're ready to prepare for moving to production, you can run it with `slc start` to run it under control of StrongLoop Process Manager, that provides options for clustering, logging, monitoring, and much more. See Operating Node applications for more information on the power of the `slc` command-line tool.

Open your browser to http://0.0.0.0:3000/ (on some systems, you may need to use http://localhost:3000 instead). You'll see the default application response that displays some JSON with some status information; for example:

`{"started":"2014-11-20T21:59:47.155Z","uptime":42.054}`

Now open your browser to http://0.0.0.0:3000/explorer or http://localhost:3000/explorer. You'll see the StrongLoop API Explorer:



Through a set of simple steps using LoopBack, you've created a CoffeeShop model, specified its properties and then exposed it through REST.

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

Use API Explorer

> **Prerequisite**: Install StrongLoop software as described in Installing StrongLoop.
>
> **Recommended**: Read LoopBack core concepts.

- Run API Explorer
- About LoopBack built-in models
- Exploring the CoffeeShop model

> LoopBack applications come with a built-in API Explorer you can use to test REST API operations during development.

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

> If you followed Create a simple API, keep that app running and skip down to Run API Explorer.
>
> If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

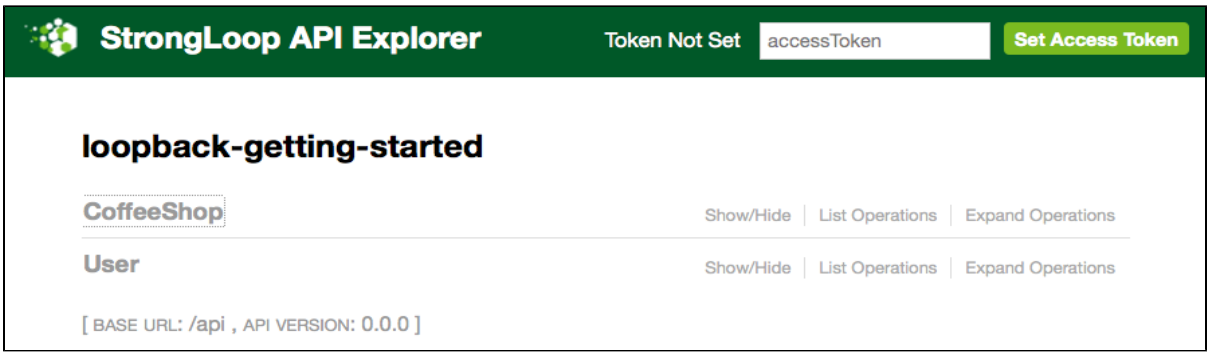
```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step1
$ npm install
```

Run API Explorer

Run the application:

```
$ node .
```

Now go to http://localhost:3000/explorer. You'll see the StrongLoop API Explorer showing the two models this application has: **Users** and **Coffee Shops:**



In addition to the CoffeeShop model that you defined, by default Loopback generates the User model and its endpoints for every application.

About LoopBack built-in models

Actually, LoopBack creates several other built-in models for common use cases:

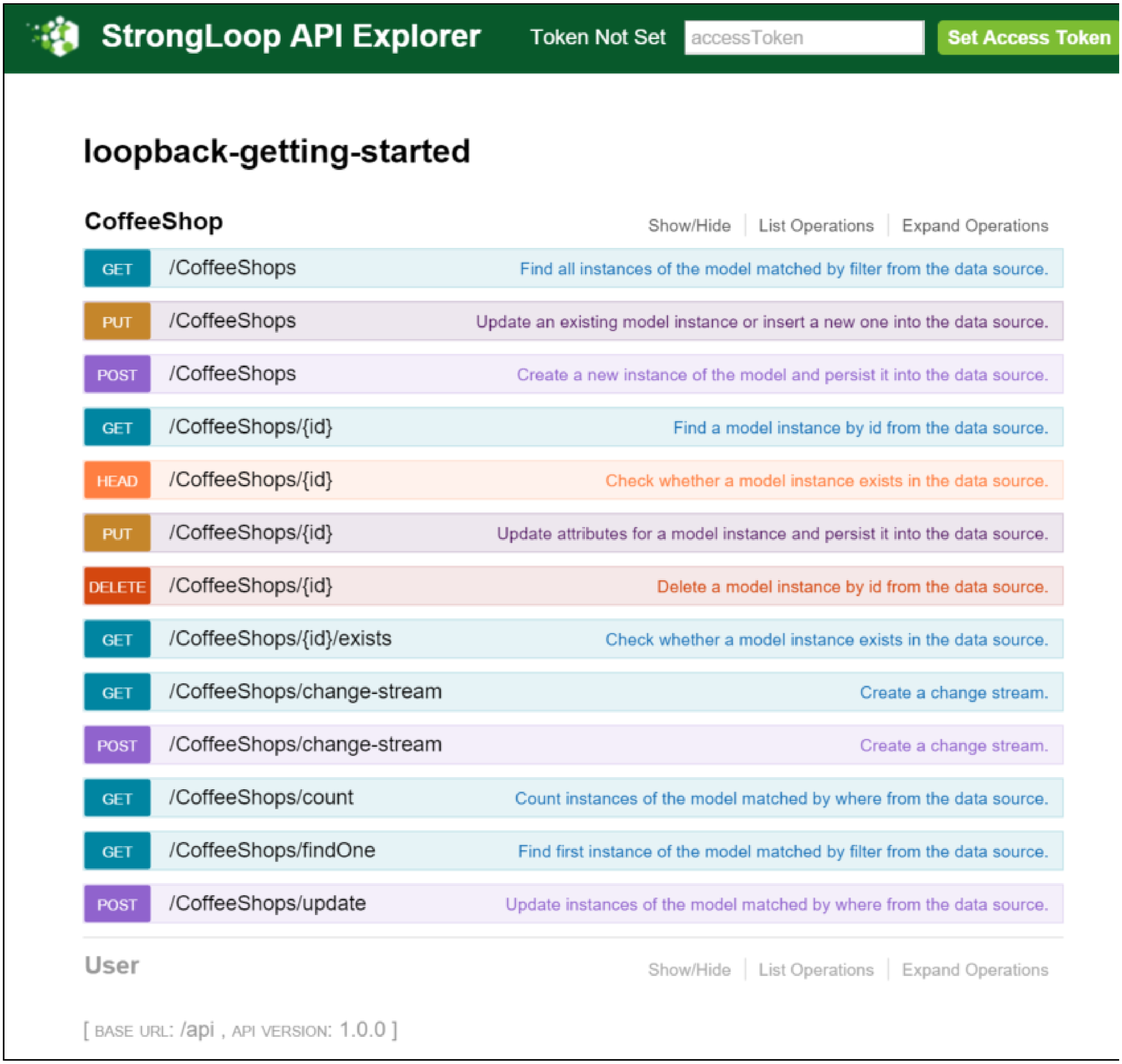
- **Application model** - contains metadata for a client application that has its own identity and associated configuration with the LoopBack server.
- **User model** - register and authenticate users of your app locally or against third-party services.
- **Access control models** - ACL, AccessToken, Scope, Role, and RoleMapping models for controlling access to applications, resources, and methods.
- **Email model** - send emails to your app users using SMTP or third-party services.

The built-in models (except for Email) extend PersistedModel, so they automatically have a full complement of create, update, and delete (CRUD) operations.

> By default, only the User model is exposed over REST. To expose the other models, change the model's `public` property to true in `server/model-config.json`. See Exposing models for more information. **Use caution**: exposing some of these models over public API may be a security risk.

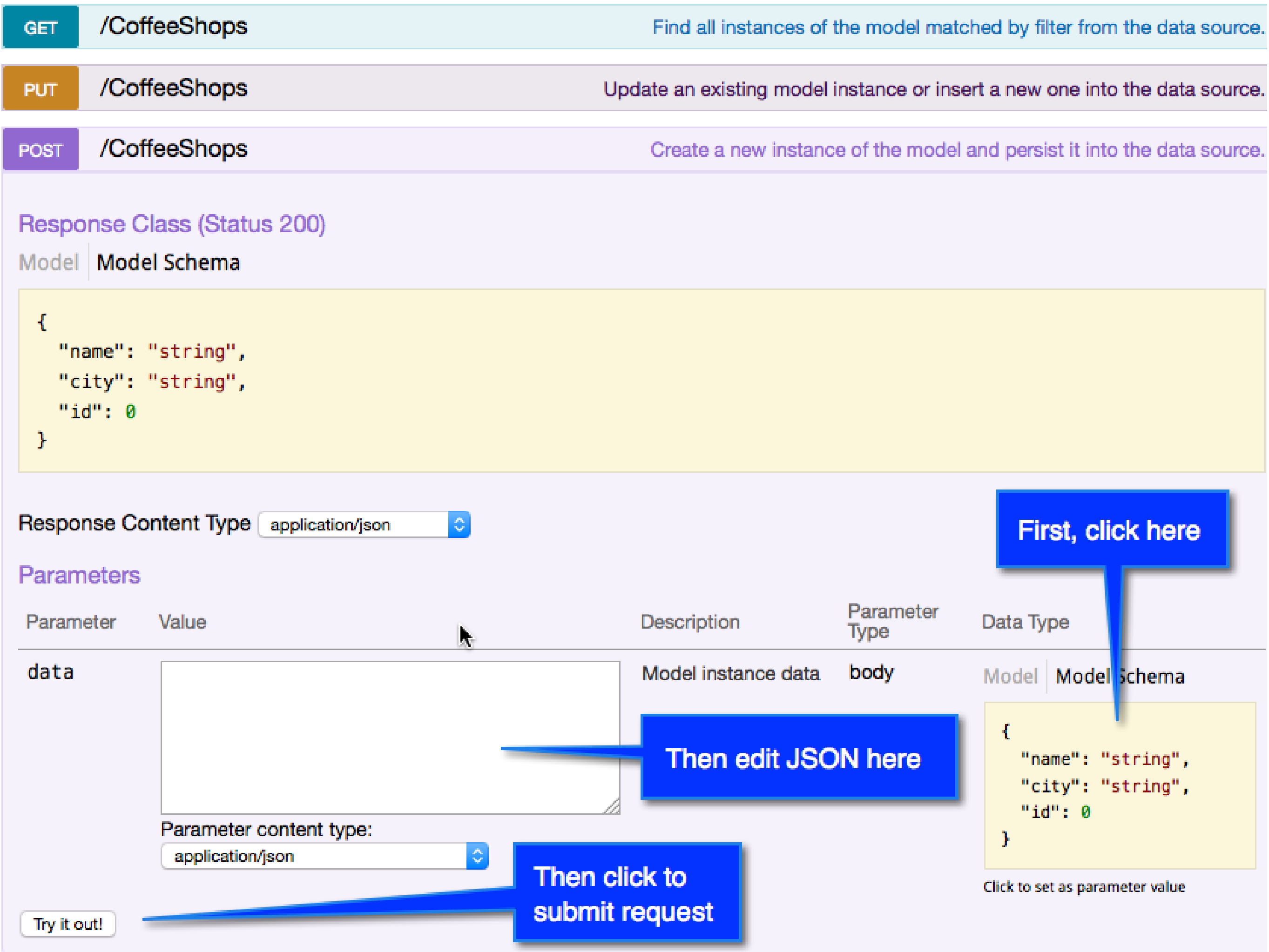
Exploring the CoffeeShop model

Right now, you're going to "drill down" on the CoffeeShop model. Click on **CoffeeShops** to show all its API endpoints:



Scan down the rows of the API endpoints: you can see that they cover all the normal create, read, update, and delete (CRUD) operations, and then some.

Click on the first row, **POST /CoffeeShops Create a new instance of the model and persist it into the data source** to expand that operation:



Follow the instructions in the diagram above.

Click in Model Schema to get a JSON "data template" that you can edit in the **data** field.

Add some text for the `name` property. You don't have to put anything for the `id` property, because LoopBack will automatically manage it to ensure there is always a unique ID for each model instance.

```
{
  "name": "My Coffee Shop",
  "id": 0
}
```

Then click the **Try it out!** button.

You'll see information on the REST request submitted and the application's response (for example):

Request URL

```
http://0.0.0.0:3000/api/CoffeeShops
```

Response Body

```
{
  "name": "Verve Coffee",
  "city": "Santa Cruz",
  "id": 5
}
```

Response Code

```
200
```

Response Headers

```
{
  "x-powered-by": "Express",
  "vary": "Origin, Accept-Encoding",
  "access-control-allow-credentials": "true",
  "content-type": "application/json; charset=utf-8",
  "content-length": "50",
  "etag": "W/\"32-G3mJV1irbSl9w4U6v7VQfw\"",
  "date": "Wed, 20 Jan 2016 21:19:10 GMT",
  "connection": "keep-alive"
}
```

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

Now click on **GET /CoffeeShops** to expand that endpoint. Click **Try it out!** to retrieve the data you entered for the CoffeeShop model. You should see the record you created using the POST API.

If you are so inclined, try some other requests: You can enter more complicated queries using the **filter** field to specify a Where filter, Limit filter, and other kinds of filters on the query. See Querying data for more information.

API Explorer automatically adds "filter" to the query string, but you must enter Stringified JSON in the **filter** field. Also make sure that the quotes you use are proper straight quotes ("), not curved or typographic quotation marks (“ or ”). These can often be hard to distinguish visually.

You may have noticed the **accessToken** field and **Set Access Token** button at the top right of the API Explorer window. Use these to authenticate a user and "login" to an app so you can perform actions that require authentication. For more information, see Introduction to User model authentication.

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

Connect your API to a data source

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

- Add a data source
- Install MySQL connector
- Configure data source
- Connect CoffeeShop model to MySQL
- Add some test data and view it

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

You're going to take the app from the previous section and connect it to MySQL.

> If you followed the previous steps in the tutorial, go to Add a data source.
>
> If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step1
$ npm install
```

Add a data source

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

```
$ slc loopback:datasource
```

The generator will prompt you to name the data source:

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

Enter **mysqlDs** and hit **Enter**.

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

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

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

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

datasources.json

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

Install MySQL connector

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

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

Configure data source

If you have a MySQL database server that you can use, please use it. Create a new database called "getting_started." If you wish, you can use a different database name. Just make sure the `mysqlDs.database` property in `datasources.json` matches it (see below).

If not, you can use the StrongLoop MySQL server running on demo.strongloop.com. However, be aware that it is a shared resource. There is a small chance that two users may run the script that creates sample data (see Add some test data and view it, below) at the same time and may run into race condition. For this reason, we recommend you use your own MySQL server if you have one.

Next, you need configure the data source to use the desired MySQL server.

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

`"connector": "mysql"`

add `host`, `port`, `database`, `username`, and `password` properties.

To use the StrongLoop MySQL server: running on demo.strongloop.com, then enter the values shown below.

To use your own MySQL server: enter the hostname, port number, and login credentials for your server.

/server/datasources.json

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

Connect CoffeeShop model to MySQL

Now you've created a MySQL data source and you have a CoffeeShop model; you just need to connect them. LoopBack applications use the model-config.json file to link models to data sources. Edit `/server/model-config.json` and look for the CoffeeShop entry:

/server/model-config.json

```
...
  "CoffeeShop": {
    "dataSource": "db",
    "public": true
  }
  ...
```

Change the `dataSource` property from `db` to `mysqlDs`. This attaches the CoffeeShop model to the MySQL datasource you just created and configured.

Add some test data and view it

Now you have a CoffeeShop model in LoopBack, how do you create the corresponding table in MySQL database?

You could try executing some SQL statements directly...but LoopBack provides a Node API to do it for you automatically using a process called *auto-migration*. For more information, see Creating a database schema from models.

The `loopback-getting-started` module contains the `create-sample-models.js` script to demonstrate auto-migration. If you've been following along from the beginning (and didn't just clone this module), then you'll need to copy it from below or from GitHub . Put it in the application's `/server/boot` directory so it will get executed when the application starts.

> The auto-migration script below is an example of a *boot script* that LoopBack executes when an application initially starts up. Use boot scripts for initialization and to perform any other logic your application needs to perform when it starts. See Defining boot scripts for more information.

/server/boot/create-sample-models.js

```
module.exports = function(app) {
  app.dataSources.mysqlDs.automigrate('CoffeeShop', function(err) {
    if (err) throw err;

    app.models.CoffeeShop.create([
      {name: 'Bel Cafe', city: 'Vancouver'},
      {name: 'Three Bees Coffee House', city: 'San Mateo'},
      {name: 'Caffe Artigiano', city: 'Vancouver'},
    ], function(err, coffeeShops) {
      if (err) throw err;

      console.log('Models created: \n', coffeeShops);
    });
  });
};
```

This will save some test data to the data source.

> The boot script containing the automigration command will run *each time* you run your application. Since `automigrate()` first drops tables before trying to create new ones, it won't create duplicate tables.

Now run the application:

```
$ node .
```

In the console, you'll see this:

```
...
Browse your REST API at http://0.0.0.0:3000/explorer
Web server listening at: http://0.0.0.0:3000/
Models created: [ { name: 'Bel Cafe',
    city: 'Vancouver',
    id: 1 },
  { name: 'Three Bees Coffee House',
    city: 'San Mateo',
    id: 3 },
  { name: 'Caffe Artigiano',
    city: 'Vancouver',
    id: 2 } ]
```

You can also use the API Explorer:

1. Browse to http://0.0.0.0:3000/explorer/ (you may need to use http://localhost:3000/explorer, depending on your browser and OS).
2. Click **GET /CoffeeShops Find all instance of the model matched by filter...**
3. Click **Try it out!**
4. You'll see the data for the three coffee shops created in the above script.

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

Extend your API

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

In LoopBack, a Node function attached to a custom REST endpoint is called a *remote method*.

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

If you followed the previous steps in the tutorial, skip down to Add a remote method

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step2
$ npm install
```

Add a remote method

Follow these steps:

1. Look in your application's `/common/models` directory. You'll notice there are `coffee-shop.js` and `coffee-shop.json` files there.

   The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

2. Open `coffee-shop.js` in your favorite editor. By default, it contains an empty function:

   ```
   module.exports = function(CoffeeShop) {
   };
   ```

3. Add the following code to this function to extend the model's behavior with a remote method, so it looks as shown here:

```
module.exports = function(CoffeeShop) {
  CoffeeShop.status = function(cb) {
    var currentDate = new Date();
    var currentHour = currentDate.getHours();
    var OPEN_HOUR = 6;
    var CLOSE_HOUR = 20;
    console.log('Current hour is ' + currentHour);
    var response;
    if (currentHour > OPEN_HOUR && currentHour < CLOSE_HOUR) {
      response = 'We are open for business.';
    } else {
      response = 'Sorry, we are closed. Open daily from 6am to 8pm.';
    }
    cb(null, response);
  };
  CoffeeShop.remoteMethod(
    'status',
    {
      http: {path: '/status', verb: 'get'},
      returns: {arg: 'status', type: 'string'}
    }
  );
};
```

This defines a simple remote method called "status" that takes no arguments, and checks the time and returns a JSON status message that says either "Open for business" or "Sorry we are closed", depending on the current time.

Of course, in practice you can do much more interesting and complex things with remote methods such as manipulating input data before persisting it to a database. You can also change the route where you call the remote method, and define complex arguments and return values. See Remote methods for all the details.

4. Save the file.

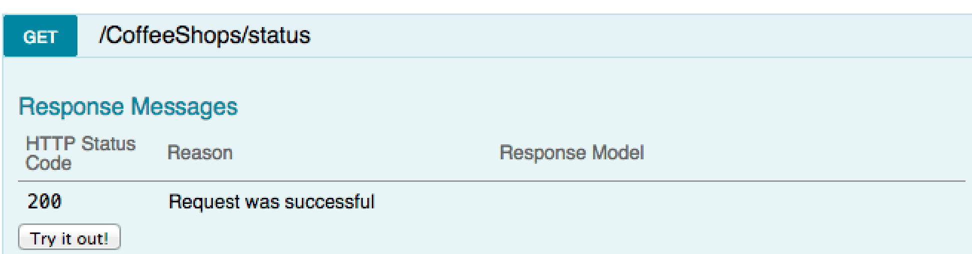
> If you don't want to expose a remote method to everyone, it's easy to constrain access to it using access control lists (ACLs). See Adding ACLs to remote methods.

Try the remote method

1. Back in the application root directory, run the app:

```
$ node .
```

2. Go to http://localhost:3000/explorer to see API Explorer. Then click on CoffeeShops and you'll see there is a new REST endpoint, `GET/CoffeeShop/status` that calls the remote method.



3. Click **Try it Out!**
You'll see the result of calling your remote method :

```
{
  "status": "Open for business."
}
```

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

For more information, see Remote methods.

Running CRUD methods in a remote method

The `status` remote method is trivial, but a remote method can also access any of the standard model CRUD methods to perform data processes and validation. Here is a simple example (this is not in the loopback-getting-started repository):

```
module.exports = function(CoffeeShop) {
...
  CoffeeShop.getName = function(shopId, cb) {
    CoffeeShop.findById( shopId, function (err, instance) {
        response = "Name of coffee shop is " + instance.name;
        cb(null, response);
        console.log(response);
    });
  }
...
  CoffeeShop.remoteMethod (
        'getName',
        {
          http: {path: '/getname', verb: 'get'},
          accepts: {arg: 'id', type: 'number', http: { source: 'query' } },
          returns: {arg: 'name', type: 'string'}
        }
    );
}
```

Then, if you access the remote method at, for example:

```
http://0.0.0.0:3000/api/CoffeeShops/getname?id=1
```

You'll get the response:

```
{
  "name": "Name of coffee shop is Bel Cafe"
}
```

Next: In Add a static web page, you'll add Express middleware to serve static client assets such as HTML/CSS, images, and JavaScript.

Add a static web page

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

LoopBack leverages Express middleware to make it easy to serve up static content such as web pages.

If you followed the previous steps in the tutorial, skip down to Introduction to middleware.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step3
$ npm install
```

Introduction to middleware

LoopBack is built on Express, one of the most popular Node application frameworks. The top-level LoopBack `app` object inherits all the methods and properties of the Express app object. See Working with LoopBack objects.

Before continuing, you need to understand a basic concept that LoopBack inherits from Express: middleware.

Middleware is simply a JavaScript function with access to the request object (`req`) representing the HTTP request, the response object (`res`) repr esenting the HTTP response, and the next middleware in line in the request-response cycle of an Express application, commonly denoted by a variable named `next`. Middleware can:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware in the stack.

LoopBack middleware is exactly like Express middleware, except that LoopBack adds the concept of *phases*, that enables you to easily set the order in which middleware is called. This avoids one of the tricky aspects of Express: making sure middleware gets executed when it should be.

When you create an application with slc loopback, it creates a `server/middleware.json` file that specifies what middleware is executed in which phase. Registering new middleware is as simple as editing this JSON file. Expand this code to see what it looks like:

server/middleware.json Expand source

```
{
  "initial:before": {
    "loopback#favicon": {}
  },
  "initial": {
    "compression": {},
    "cors": {
      "params": {
        "origin": true,
        "credentials": true,
        "maxAge": 86400
      }
    }
  },
  "session": {
  },
  "auth": {
  },
  "parse": {
  },
  "routes": {
    "loopback#rest": {
      "paths": ["${restApiRoot}"]
    }
  },
  "files": {
  },
  "final": {
    "loopback#urlNotFound": {}
  },
  "final:after": {
    "loopback#errorHandler": {}
  }
}
```

Each of the top-level keys in `middleware.json` defines a middleware phase: `initial`, `session`, `auth`, and so on, ending with `final`. There are also modifiers to register middleware `before` and `after` a given phase. There's a bit more to it, but that covers the basics. See Defining middleware for all the details.

Change or modify the default root route handler

Applications typically need to serve static content such as HTML and CSS files, client JavaScript files, images, and so on. It's very easy to do this with the default scaffolded LoopBack application. You're going to configure the application to serve any files in the `/client` directory as static assets.

First, you have to disable the default route handler for the root URL. Remember back in Create a simple API (you have been following along, haven't you?) when you loaded the application root URL, http://localhost:3000/, you saw the application respond with a simple status message such as this:

```
{"started":"2014-11-20T21:59:47.155Z","uptime":42.054}
```

This happens because by default the scaffolded application has a boot script named `root.js` that sets up route-handling middleware for the root route ("/"):

server/boot/root.js

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

This code says that for any GET request to the root URI ("/"), the application will return the results of `loopback.status()`.

To make your application serve static content you need to disable this script. **Either delete it or just rename it** to something without a `.js` endin g (that ensures the application won't execute it).

Define static middleware

Next, you need to define static middleware to serve files in the `/client` directory.

Edit `server/middleware.json`. Look for the "files" entry:

server/middleware.json

```
...
  "files": {
  },
...
```

Add the following:

server/middleware.json

```
...
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  },
...
```

These lines define *static middleware* that makes the application serve files in the `/client` directory as static content. The `$!` characters indicate that the path is relative to the location of `middleware.json`.

Add an HTML file

Now, the application will serve any files you put in the `/client` directory as static (client-side) content. So, to see it in action, add an HTML file to `/client`. For example, add a file named `index.html` with this content:

/client/index.html

```
<head><title>LoopBack</title></head>
<body>
    <h1>LoopBack Rocks!</h1>
    <p>Hello World... </p>
</body>
```

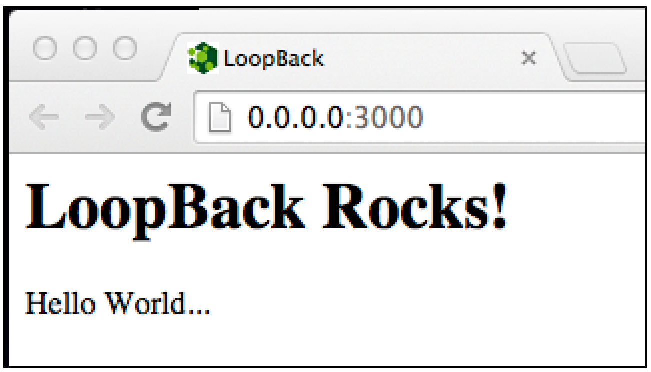
Of course, you can add any static HTML you like–this is just an example.

Run it....!

Now run the application again:

```
$ node .
```

When you load http://0.0.0.0:3000/ now instead of the status JSON, you'll see this:



Next: In Add a custom Express route, you'll add a simple route handler in the same way you would in an Express application.

Add a custom Express route

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

- Introducing boot scripts
- Add a new boot script
- Run the boot script

Because LoopBack is built on Express, you can add custom routes just as you do in Express.

In this part of the tutorial, you're going to add a new custom route.

If you followed the previous steps in the tutorial, skip down to Introducing boot scripts.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step4
$ npm install
```

Introducing boot scripts

When a LoopBack application starts (or "bootstraps"), it runs the scripts in the `/server/boot` directory, known as *boot scripts*. By default, LoopBack loads boot scripts in alphabetical order.

The standard scaffolded LoopBack application created by the application generator contains the following standard boot scripts (in `/server/boot`) that perform basic initialization:

- `authentication.js` - Enables authentication for the application by calling `app.enableAuth()`.
- `root.js` - Defines a root route to `/` that returns server status using `loopback.status()` middleware. You already encountered this in the previous step, when you renamed this file so your app could serve static content.

For more information on boot scripts, see Defining boot scripts.

Add a new boot script

For example, add a new boot script named `routes.js` in `/server/boot` directory, with this code:

/server/boot/routes.js

```
module.exports = function(app) {
  // Install a "/ping" route that returns "pong"
  app.get('/ping', function(req, res) {
    res.send('pong');
  });
}
```

As an aside, you could have just as well used Express router middleware instead, like this:

/server/boot/routes.js

```
module.exports = function(app) {
  var router = app.loopback.Router();
  router.get('/ping', function(req, res) {
    res.send('pongaroo');
  });
  app.use(router);
}
```

In fact you can also add routes right in `server.js` using the Express API. For example, add this call to `app.use()` just before the call to `app.start()`:

server/server.js

```
...
app.use('/express-status', function(req, res, next) {
  res.json({ running: true });
});

// start the server if `$ node server.js`
if (require.main === module) {
  app.start();
}
```

The point is that a LoopBack application can easily do all the things that an Express application can. If you're familiar with Express, this will make LoopBack easier to learn and use.

Run the boot script

Now, run the application again:

```
$ node .
```

Load http://0.0.0.0:3000/ping. You'll see "pong" as the response.

Next: Check out Run in a cluster to see how to run the application in a multi-process cluster.

Run in a cluster

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

- Start the application
- Display the application's status
- Change the cluster size

You can run your application using `slc start` to use StrongLoop Process Manager features such as clustering, profiling, monitoring, and more.

In this part of the tutorial, you're going to run the application in StrongLoop Process Manager (StrongLoop PM). You'll also change the cluster size and display the application's status.

If you followed the previous steps in the tutorial, skip down to Start the application.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
$ git checkout step4
$ npm install
```

Start the application

If you've been following the tutorial, up until now, you've been using `node .` to run the application. Running your app with the `node` command is appropriate when you're in development mode. Once you're ready to prepare for deployment, use `slc start` to run it under control of StrongLoop PM. See Operating Node applications for more information on the power of StrongLoop Process Manager.

In the root directory of your application, enter this command to start a local instance of StrongLoop PM and run the application under its control:

```
$ slc start
Process Manager is attempting to run app `.`.
  To confirm it is started: slc ctl status loopback-getting-started
  To view the last logs: slc ctl log-dump loopback-getting-started
  To see more options: slc ctl -h
  To see metrics, the profilers and other diagnostic features run: slc arc
```

Display the application's status

Now get a quick status overview with all worker PIDs, cluster IDs, other key information. Enter this command:

```
$ slc ctl
Service ID: 1
Service Name: loopback-getting-started
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.0.32      1.5.3            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.73137  73137   0
    1.1.73144  73144   1     0.0.0.0:3001
    1.1.73145  73145   2     0.0.0.0:3001
    1.1.73146  73146   3     0.0.0.0:3001
    1.1.73147  73147   4     0.0.0.0:3001
```

Change the cluster size

By default the Process Manager runs one process per CPU. So, on a four-core system it runs the app in a four-process cluster. You can easily change the cluster size.

Use the `slc ctl set-size` command to change the cluster size. Then display status again, and you'll see only two worker processes running:

```
$ slc ctl set-size loopback-getting-started 2
$ slc ctl
Service ID: 1
Service Name: loopback-getting-started
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.0.32      1.5.3            2
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.73137  73137   0
    1.1.73144  73144   1     0.0.0.0:3001
    1.1.73145  73145   2     0.0.0.0:3001
```

Next: Check out Next steps for information on what to read next.

Next steps

Getting Started part II

If you want to continue on the tutorial track, continue on to Getting started part II. It covers:

- Using multiple data sources in an single application.
- Relations between models.
- Remote hooks.
- Using access control lists to protect REST endpoints.
- User registration and authentication
- Using the AngularJS SDK

Learn more

There's so much more to learn about LoopBack!

LoopBack documentation includes lots of tutorials and examples; also read the StrongLoop blog for great tips and information.

StrongLoop Doc Tips

Use the left navigation tree to browse and find doc articles; Click ⊞ to expand subjects under a topic.

Use the search field at upper right to search the documentation.

Learn more about:

- **Creating an application**
- **Managing users**
  - Registering users
  - Logging in users
  - Partitioning users with realms
- **Authentication, authorization, and permissions**
- **Defining models**
  - Creating models
  - Customizing models
  - Attaching models to data sources
  - Exposing models over REST
  - Validating model data
  - Creating model relations
- **Working with data**
- **Adding application logic**
  - Adding logic to models
  - Defining boot scripts
  - Defining mixins
  - Defining middleware
  - Working with LoopBack objects
  - Using current context
  - Events

Check out the LoopBack Developer Forum on Google Groups, a place where devlopers can ask questions, discuss LoopBack, and how they are using it.

Getting started part II

This tutorial picks up where Getting started with LoopBack ended, and it assumes you understand the basic concepts and tasks introduced in the first tutorial.

This tutorial covers:

- Using multiple data sources in an single application.
- Relations between models.

- Remote hooks.
- Using access control lists to protect REST endpoints.
- User registration and authentication
- Using the AngularJS SDK

> **Next**: Read Introducing the Coffee Shop Reviews app to understand the example application that you'll create when you follow the tutorial.

Introducing the Coffee Shop Reviews app

> **Prerequisites**:
>
> - Install StrongLoop software as described in Installing StrongLoop
> - Follow Getting started with LoopBack.
>
> **Recommended**: Read LoopBack core concepts.

> "Coffee Shop Reviews" is a complete small application that illustrates many of LoopBack's basic features and how they work together.

Overview of the application

Coffee Shop Reviews is a website that you can use to post reviews on coffee shops, like Yelp for coffee shops.

The app will persist data to two different datasources: it will store reviewer data in a MySQL database and coffee shop and review data in a MongoDB database.

This application has three models:

- CoffeeShop (defined in Getting started with LoopBack)
- Review
- Reviewer

They are related as follows:

- A CoffeeShop has many reviews
- A CoffeeShop has many reviewers
- A review belongs to a CoffeeShop
- A review belongs to a reviewer
- A reviewer has many reviews

In general, users can create, edit, delete, and read reviews of coffee shops, with the following basic rules and permissions implemented through ACLs:

- Anyone can read reviews, but you must be logged in to create, edit, or delete them.
- Anyone can register as a user; then log in and log out.
- Logged-in users can create new reviews, and edit or delete their own reviews; however they cannot modify the originally chosen CoffeeShop.

Get the repo

First, clone the repository.

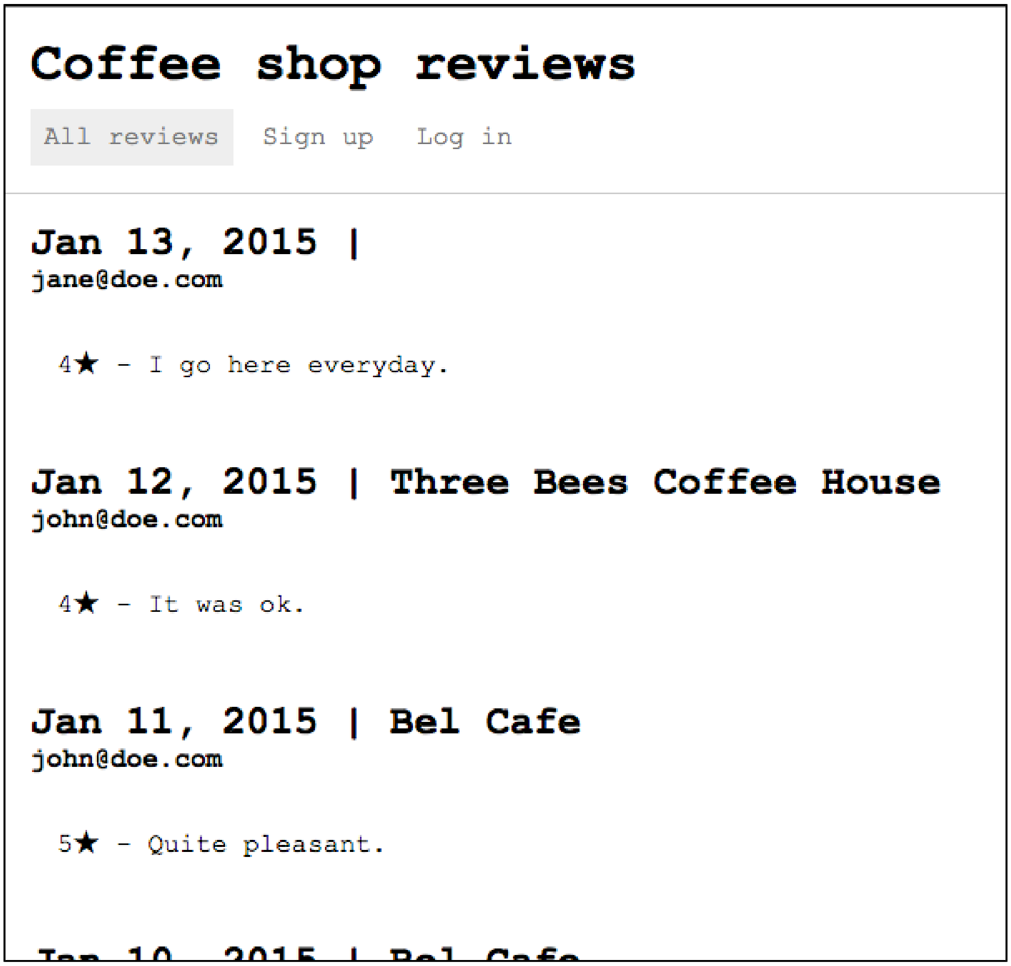
```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
```

Run the application

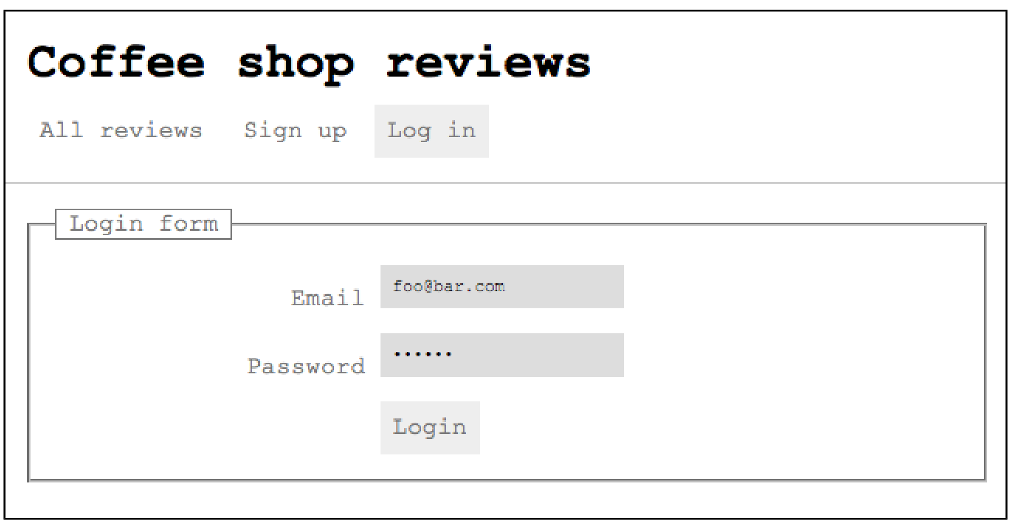
To better understand what's going on, run the Coffee Shop Reviews application:

```
$ cd loopback-getting-started-intermediate
$ npm install
...
$ node .
...
Browse your REST API at http://0.0.0.0:3000/explorer
Web server listening at: http://0.0.0.0:3000/
> models created sucessfully
```

Now load http://0.0.0.0:3000/ in your browser. You'll see the application home page:



Click on **Log in**. You'll see the login page:



Click **Login** to login with the provided email and password. Notice if you change the email or password, you can't login (but the app doesn't display an error – that's left as an exercise for the reader).

After logging in, you'll see the "Add review" page by default:

Click **Add Review** to create a new review, **My Reviews** to view only your reviews, or **All Reviews** to view all reviews again. You can only review the three predefined coffee shops; the application does not provide the ability to add a new coffee shop as an administrator yet.

Create your own app

To understand all the features of the Coffee Shop Reviews app, you're going to recreate it from scratch. The starting point is the app you created in Getting started with LoopBack.

So, if you followed that tutorial, simply change to that directory:

```
$ cd <my-getting-started-app>
```

If you didn't follow that tutorial, you can just clone the repository:

```
$ git clone https://github.com/strongloop/loopback-getting-started.git
$ cd loopback-getting-started
```

Next: Continue to Create new data source to add a new data source that the application will use.

Create new data source

Prerequisites:

- Install StrongLoop software as described in Installing StrongLoop
- Follow Getting started with LoopBack.

Recommended: Read LoopBack core concepts.

- Add a data source
- Install MongoDB connector
- Configure data source

You can easily connect a LoopBack application to multiple different data sources.

Add a data source

You're going to add a MongoDB data source in addition to the MySQL data source created in Connect your API to a data source.

```
$ slc loopback:datasource
```

When prompted, respond as follows:

```
? Enter the data-source name: mongoDs
? Select the connector for mongoDs: MongoDB (supported by StrongLoop)
```

Install MongoDB connector

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

Configure data source

If you have a MongoDB database server that you can use, please do so. Create a new database called "getting_started_intermediate". If you wish, you can use a different database name. Just make sure the `mongoDs.database` property in `datasources.json` matches it (see below).

If not, you can use the StrongLoop MongoDB server running on demo.strongloop.com. However, be aware that it is a shared resource. There is a small chance that two users may perform operations that conflict. For this reason, we recommend you use your own MongoDB server if you have one.

Edit `datasources.json` to configure the data source so that it connects to the StrongLoop demo MongoDB server. Add the following JSON after the two existing data source definitions (for "db" and "mysqlDs"):

server/datasources.json

```
...
"mongoDs": {
    "name": "mongoDs",
    "connector": "mongodb",
    "host": "demo.strongloop.com",
    "port": 27017,
    "database": "getting_started_intermediate",
    "username": "demo",
    "password": "L00pBack"
  }
```

Next: Continue to Create new models.

Create new models

Prerequisites:

- Install StrongLoop software as described in Installing StrongLoop
- Follow Getting started with LoopBack.

Recommended: Read LoopBack core concepts.

- Define the Review model
- Define the Reviewer model
- Update boot script to add data

Creating models with `slc loopback` is quick and easy.

Recall in Create a simple API step of Getting started you created a CoffeeShop model.

Now you're going to create two new models, Review and Reviewer, with the `slc loopback` model generator.

> If you followed the previous step in the tutorial, go to Define the Review model.
>
> If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step1
$ npm install
```

Define the Review model

Enter:

```
$ slc loopback:model
```

When prompted, enter or select the following:

- **Model name**: Review
- **Data source**: mongoDs (mongodb)
- **Base class**: Use the down-arrow key to select **PersistedModel**.
- **Expose Reviewer via the REST API?** Press RETURN to accept the default, Yes.
- **Custom plural form (used to build REST URL)**: Press RETURN to accept the default, Yes.
- **Common model or server only**: Press RETURN to accept the default, common model.

Then, follow the prompts to add these properties:

Property name	Property type	Required?
date	date	y
rating	number	n
comments	string	y

To exit the model generator, press RETURN when prompted for property name.

Define the Reviewer model

Enter:

```
$ slc loopback:model
```

When prompted, enter or select the following:

- **Model name**: Reviewer
- **Data source**: mongoDs (mongodb)
- **Base class**: Use the down-arrow key to select **User**.
- **Expose Reviewer via the REST API?** Press RETURN to accept the default, Yes.
- **Custom plural form (used to build REST URL)**: Press RETURN to accept the default, Yes.

Don't add any properties, since they are all inherited from the base User model.

To exit the model generator, press RETURN when prompted for property name.

Update boot script to add data

Recall back in part I of Getting started, you added a boot script to create a database table from the model (via auto-migration) and add some data to the database.

Now that you have some new models and a new data source, you need to update this script so it will create data structures in MongoDB and insert data via the new models.

Copy and paste the code below into `server/boot/create-sample-models.js`, replacing the existing code.

Then run

```
$ npm install --save async
```

This boot script has several functions:

- `createCoffeeShops()` creates a MySQL table for the CoffeeShop model and adds data to the table. This is what the `create-sample-models.js` script from Getting started did.
- `createReviewers()` creates the Reviewer data structure in MongoDB using auto-migration and adds data to it.
- `createReviews()` creates the Reviews data structure in MongoDB using auto-migration and adds data to it.

See Creating a database schema from models for more information on auto-migration.

server/boot/create-sample-models.js

```
var async = require('async');
module.exports = function(app) {
  //data sources
  var mongoDs = app.dataSources.mongoDs;
  var mysqlDs = app.dataSources.mysqlDs;
  //create all models
  async.parallel({
    reviewers: async.apply(createReviewers),
    coffeeShops: async.apply(createCoffeeShops),
  }, function(err, results) {
    if (err) throw err;
    createReviews(results.reviewers, results.coffeeShops, function(err) {
      console.log('> models created sucessfully');
    });
  });
  //create reviewers
  function createReviewers(cb) {
    mongoDs.automigrate('Reviewer', function(err) {
      if (err) return cb(err);
      var Reviewer = app.models.Reviewer;
      Reviewer.create([
        {email: 'foo@bar.com', password: 'foobar'},
        {email: 'john@doe.com', password: 'johndoe'},
        {email: 'jane@doe.com', password: 'janedoe'}
      ], cb);
    });
  }
  //create coffee shops
  function createCoffeeShops(cb) {
    mysqlDs.automigrate('CoffeeShop', function(err) {
      if (err) return cb(err);
      var CoffeeShop = app.models.CoffeeShop;
      CoffeeShop.create([
        {name: 'Bel Cafe', city: 'Vancouver'},
        {name: 'Three Bees Coffee House', city: 'San Mateo'},
```

```
      {name: 'Caffe Artigiano', city: 'Vancouver'},
    ], cb);
  });
}
//create reviews
function createReviews(reviewers, coffeeShops, cb) {
  mongoDs.automigrate('Review', function(err) {
    if (err) return cb(err);
    var Review = app.models.Review;
    var DAY_IN_MILLISECONDS = 1000 * 60 * 60 * 24;
    Review.create([
      {
        date: Date.now() - (DAY_IN_MILLISECONDS * 4),
        rating: 5,
        comments: 'A very good coffee shop.',
        publisherId: reviewers[0].id,
        coffeeShopId: coffeeShops[0].id,
      },
      {
        date: Date.now() - (DAY_IN_MILLISECONDS * 3),
        rating: 5,
        comments: 'Quite pleasant.',
        publisherId: reviewers[1].id,
        coffeeShopId: coffeeShops[0].id,
      },
      {
        date: Date.now() - (DAY_IN_MILLISECONDS * 2),
        rating: 4,
        comments: 'It was ok.',
        publisherId: reviewers[1].id,
        coffeeShopId: coffeeShops[1].id,
      },
      {
        date: Date.now() - (DAY_IN_MILLISECONDS),
        rating: 4,
        comments: 'I go here everyday.',
        publisherId: reviewers[2].id,
        coffeeShopId: coffeeShops[2].id,
      }
    ], cb);
```

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

Next: Define model relations

Define model relations

Prerequisites:

- Install StrongLoop software as described in Installing StrongLoop
- Follow Getting started with LoopBack.

Recommended: Read LoopBack core concepts.

Individual models are easy to understand and work with. But in reality, models are often connected or related.   For applications with multiple models, you typically need to define *relations* between models.

Relations among models enable you to query related models and perform corresponding validations.

- Introducing model relations
- Define relations
- Review the model JSON files

If you followed the previous step in the tutorial, go to Introducing model relations.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step2
$ npm install
```

Introducing model relations

LoopBack supports many different kinds of model relations, including: BelongsTo, HasMany, HasManyThrough, and HasAndBelongsToMany, among others. For more information, see Creating model relations.

In the Coffee Shop Reviews app, the models are related as follows:

- A coffee shop has many reviews.
- A coffee shop has many reviewers.
- A review belongs to a coffee shop.
- A review belongs to a reviewer.
- A reviewer has many reviews.

Define relations

Now, you're going to define these relationships between the models. In all there are five relations. Once again, you'll use `slc loopback`, but this time you'll use the `relation` sub-command (relation generator). For each relation, enter:

```
$ slc loopback:relation
```

The tool will prompt you to provide the information required to define the relation, as summarized below.

A coffee shop has many reviews; No through model and no foreign key.

```
? Select the model to create the relationship from: CoffeeShop
? Relation type: has many
? Choose a model to create a relationship with: Review
? Enter the property name for the relation: reviews
? Optionally enter a custom foreign key:
? Require a through model? No
```

A coffee shop has many reviewers; No through model and no foreign key.

```
? Select the model to create the relationship from: CoffeeShop
? Relation type: has many
? Choose a model to create a relationship with: Reviewer
? Enter the property name for the relation: reviewers
? Optionally enter a custom foreign key:
? Require a through model? No
```

A review belongs to a coffee shop; No foreign key.

```
? Select the model to create the relationship from: Review
? Relation type: belongs to
? Choose a model to create a relationship with: CoffeeShop
? Enter the property name for the relation: coffeeShop
? Optionally enter a custom foreign key:
```

A review belongs to a reviewer; foreign key is `publisherId`.

```
? Select the model to create the relationship from: Review
? Relation type: belongs to
? Choose a model to create a relationship with: Reviewer
? Enter the property name for the relation: reviewer
? Optionally enter a custom foreign key: publisherId
```

A reviewer has many reviews; foreign key is `publisherId`.

```
? Select the model to create the relationship from: Reviewer
? Relation type: has many
? Choose a model to create a relationship with: Review
? Enter the property name for the relation: reviews
? Optionally enter a custom foreign key: publisherId
? Require a through model? No
```

Review the model JSON files

Now, look at `common/models/review.json`. You should see this:

common/models/review.json

```
...
"relations": {
    "coffeeShop": {
      "type": "belongsTo",
      "model": "CoffeeShop",
      "foreignKey": ""
    },
    "reviewer": {
      "type": "belongsTo",
      "model": "Reviewer",
      "foreignKey": "publisherId"
    }
  },
...
```

Likewise, `common/models/reviewer.json` should have this:

common/models/reviewer.json

```
...
"relations": {
    "reviews": {
      "type": "hasMany",
      "model": "Review",
      "foreignKey": "publisherId"
    }
  },
...
```

And `common/models/coffee-shop.json` should have this:

common/models/coffee-shop.json

```
...
 "relations": {
    "reviews": {
      "type": "hasMany",
      "model": "Review",
      "foreignKey": ""
    },
    "reviewers": {
      "type": "hasMany",
      "model": "Reviewer",
      "foreignKey": ""
    }
  },
...
```

Next: Continue to Define access controls.

Define access controls

Prerequisites:

- Install StrongLoop software as described in Installing StrongLoop
- Follow Getting started with LoopBack.

Recommended: Read LoopBack core concepts.

Access controls determine which users are allowed to read and write model data or execute methods on the models

If you followed the previous step in the tutorial, go to Introducing access controls.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step3
$ npm install
```

Introducing access controls

LoopBack applications access data through models, so controlling access to data means putting restrictions on models; that is, specifying who or what can read and write the data or execute methods on the models.   LoopBack access controls are determined by *access control lists* or ACLs. For more information, see Controlling data access.

You're going to set up access control for the Review model.

The access controls should enforce the following rules:

- Anyone can read reviews, but you must be logged in to create, edit, or delete them.
- Anyone can register as a user; then log in and log out.
- Logged-in users can create new reviews, and edit or delete their own reviews; however they cannot modify the coffee shop for a review.

Define access controls

Once again, you'll use `slc loopback`, but this time you'll use the `acl` sub-command; for each ACL, enter:

```
$ slc loopback:acl
```

The tool will prompt you to provide the required information, as summarized below.

Deny everyone all endpoints. This is often the starting point when defining ACLs, because then you can selectively allow access for specific actions.

```
? Select the model to apply the ACL entry to: Review
? Select the ACL scope: All methods and properties
? Select the access type: All (match all types)
? Select the role: All users
? Select the permission to apply: Explicitly deny access
```

Now allow everyone to read reviews.

```
? Select the model to apply the ACL entry to: Review
? Select the ACL scope: All methods and properties
? Select the access type: Read
? Select the role: All users
? Select the permission to apply: Explicitly grant access
```

Allow authenticated users to write a review; that is, if you're logged in, you can add a review.

```
? Select the model to apply the ACL entry to: Review
? Select the ACL scope: A single method
? Enter the method name: create
? Select the role: Any authenticated user
? Select the permission to apply: Explicitly grant access
```

Now, **enable the author of a review (its "owner") to make any changes to it**.

```
$ slc loopback:acl
? Select the model to apply the ACL entry to: Review
? Select the ACL scope: All methods and properties
? Select the access type: Write
? Select the role: The user owning the object
? Select the permission to apply: Explicitly grant access
```

Review the review.json file

When you're done, the ACL section in `common/models/review.json` should look like this:

```
...
 "acls": [
    {
      "accessType": "*",
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    },
    {
      "accessType": "READ",
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW"
    },
    {
      "accessType": "EXECUTE",
      "principalType": "ROLE",
      "principalId": "$authenticated",
      "permission": "ALLOW",
      "property": "create"
    },
    {
      "accessType": "WRITE",
      "principalType": "ROLE",
      "principalId": "$owner",
      "permission": "ALLOW"
    }
  ],
...
```

Next: Continue to Define a remote hook.

Define a remote hook

Prerequisites:

- Install StrongLoop software as described in Installing StrongLoop
- Follow Getting started with LoopBack.

Recommended: Read LoopBack core concepts.

- Introducing remote hooks
- Create the remote hook

A *remote hook* is a function that's executed before or after a remote method.

If you followed the previous step in the tutorial, go to Introducing remote hooks.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step4
$ npm install
```

Introducing remote hooks

A remote hook is simply a function that gets executed before or after a remote method (either a custom remote method or a built-in CRUD method).   In this example, you're going to define a remote hook that is called whenever the `create()` method is called on the Review model; that is, when a new review is created.

You can define two kinds of remote hooks:

- `beforeRemote()` runs before the remote method.
- `afterRemote()` runs after the remote method.

In both cases, you provide two arguments: a string that matches the remote method you want to which you want to "hook" your function, and a callback function. Much of the power of remote hooks is that the string can include wildcards, so it is triggered by any matching method.

> LoopBack also provides operation hooks, functions that are executed before or after models perform backend operations such as creating, saving, and updating model data, regardless of how those operations are invoked. In contrast, a remote hook is called only when the exact method you specify is invoked.

Create the remote hook

Here, you're going to define a remote hook on the review model, specifically `Review.beforeRemote`.

`Modify common/models/review.js`, and add the following code:

common/models/review.js

```
module.exports = function(Review) {
  Review.beforeRemote('create', function(context, user, next) {
    var req = context.req;
    req.body.date = Date.now();
    req.body.publisherId = req.accessToken.userId;
    next();
  });
};
```

This function is called before a new instance of the Review model is created. The code:

- Inserts the `publisherId` using the access token attached to the request.
- Sets the date of the review instance to the current date.

> **Next**: Continue to Create AngularJS client .

Create AngularJS client

> **Prerequisite**: Install StrongLoop software as described in Installing StrongLoop.
>
> **Recommended**: Read LoopBack core concepts.

> To follow this step, you should have a basic understanding of AngularJS.

The LoopBack AngularJS SDK automatically creates a client JavaScript API that enables you to make AngularJS calls to your LoopBack models.

If you followed the previous step in the tutorial, go to Introducing the AngularJS SDK.

If you're just jumping in, follow the steps below to catch up...

Get the app (in the state following the last article plus all the client files) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step6
$ npm install
```

Introducing the AngularJS SDK

AngularJS is an open-source JavaScript model–view–controller (MVC) framework for browser-based applications. LoopBack provides an AngularJS JavaScript SDK to facilitate creating AngularJS clients for your LoopBack API server-side apps. The SDK is installed when you install StrongLoop.

The SDK provides auto-generated AngularJS services, compatible with `ngResource.$resource`, that provide client-side representation of the models and remote methods in the LoopBack server application. The SDK also includes some command-line tools, including `lb-ng` that generates Angular $resource services for your LoopBack application, creating in effect a dynamic client that automatically includes client-side APIs to access your LoopBack models and methods. You don't have to manually write any static code.

For more information, see AngularJS JavaScript SDK.

Generate lb-services.js

To generate the Angular services for a LoopBack application, use the AngularJS SDK `lb-ng` command-line tool. First, create the `client/js/services` directory, if you don't already have it (by using the `mkdir` command, for example), then in the project root directory, enter the `lb-ng` command as follows:

```
$ mkdir -p client/js/services
$ lb-ng server/server.js client/js/services/lb-services.js
```

This command creates `client/js/services/lb-services.js`.

Copy the other client files

The `lb-ng` tool does the "heavy lifting" of creating the client JavaScript API that works with your LoopBack back-end. However, you still need to create the HTML/CSS and client JavaScript code that actually calls into this AngularJS API and defines the client-side functionality and appearance of your app. In general, creating this part of the app is entirely up to you. This tutorial includes an example of such a client implementation that you can use to understand the process.

If you've been following the entire tutorial (and didn't jump in and clone the project mid-way through), then you'll need to clone it now to get the client files required for this step. Then copy the `client` sub-directory to your project directory:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cp -r loopback-getting-started-intermediate/client <your-app-dir>/client
```

Now let's take a look at what you now have in the `client` directory:

- `index.html`
- **css** - stylesheets
  - `style.css`
- **js** - application JavaScript files
  - `app.js`
  - **controllers** - AngularJS controllers
    - `auth.js`
    - `review.js`
  - **services** - AngularJS services
    - `auth.js`
    - `lb-services.js`
- **vendor** - AngularJS libraries (dependencies)
  - `angular-resource.js`
  - `angular-ui-router.js`
  - `angular.js`
- **views** - HTML view files
  - `all-reviews.html`
  - `forbidden.html`
  - `my-reviews.html`
  - `sign-up-form.html`
  - `login.html`
  - `review-form.html`
  - `sign-up-success.html`

Each file and directory is briefly described below

index.html

The `index.html` file is the only file in the top level of the `/client` directory, and defines the application's main landing page. Open it in your editor:

client/index.html

Expand source

```
<!DOCTYPE html>
<html lang="en" ng-app="app">
  <head>
    <meta charset="utf-8">
    <title>loopback-getting-started-intermediate</title>
    <link href="css/style.css" rel="stylesheet">
  </head>
  <body>
    <header>
      <h1>Coffee shop reviews</h1>
      <h2 ng-show="currentUser">Hello {{currentUser.email}}</h2>
      <nav>
        <ul>
          <li>
            <a ui-sref="all-reviews" ui-sref-active="active">All reviews</a>
          </li>
          <li ng-hide="currentUser">
            <a ui-sref="sign-up" ui-sref-active="active">Sign up</a>
          </li>
          <li ng-show="currentUser">
            <a ui-sref="my-reviews" ui-sref-active="active">My Reviews</a>
          </li>
          <li ng-show="currentUser">
            <a ui-sref="add-review" ui-sref-active="active">Add Review</a>
          </li>
          <li ng-hide="currentUser">
            <a ui-sref="login" ui-sref-active="active">Log in</a>
          </li>
          <li ng-show="currentUser">
            <a ui-sref="logout" ui-sref-active="active">Log out</a>
          </li>
        </ul>
      </nav>
    </header>
    <main ui-view></main>
    <script src="vendor/angular.js"></script>
    <script src="vendor/angular-resource.js"></script>
    <script src="vendor/angular-ui-router.js"></script>
    <script src="js/app.js"></script>
    <script src="js/services/lb-services.js"></script>
    <script src="js/controllers/auth.js"></script>
    <script src="js/controllers/review.js"></script>
    <script src="js/services/auth.js"></script>
  </body>
</html>
```

Perusing the file, you can see the references to the stylesheet in the `/css` directory and client JavaScript files in the `/vendor` and `/js` directories.

Main client JavaScript files (app.js)

The `js/app.js` file defines application configurations.

client/js/app.js

Expand source

```
angular
  .module('app', [
    'ui.router',
    'lbServices'
  ])
  .config(['$stateProvider', '$urlRouterProvider', function($stateProvider,
      $urlRouterProvider) {
    $stateProvider
      .state('add-review', {
        url: '/add-review',
        templateUrl: 'views/review-form.html',
        controller: 'AddReviewController',
        authenticate: true
      })
      .state('all-reviews', {
        url: '/all-reviews',
        templateUrl: 'views/all-reviews.html',
        controller: 'AllReviewsController'
      })
      .state('edit-review', {
        url: '/edit-review/:id',
        templateUrl: 'views/review-form.html',
        controller: 'EditReviewController',
        authenticate: true
      })
      .state('delete-review', {
        url: '/delete-review/:id',
        controller: 'DeleteReviewController',
        authenticate: true
      })
      .state('forbidden', {
        url: '/forbidden',
        templateUrl: 'views/forbidden.html',
      })
      .state('login', {
        url: '/login',
        templateUrl: 'views/login.html',
        controller: 'AuthLoginController'
      })
      .state('logout', {
        url: '/logout',
        controller: 'AuthLogoutController'
      })
      .state('my-reviews', {
        url: '/my-reviews',
        templateUrl: 'views/my-reviews.html',
        controller: 'MyReviewsController',
        authenticate: true
      })
      .state('sign-up', {
        url: '/sign-up',
        templateUrl: 'views/sign-up-form.html',
        controller: 'SignUpController',
      })
      .state('sign-up-success', {
        url: '/sign-up/success',
        templateUrl: 'views/sign-up-success.html'
```

```
    });
  $urlRouterProvider.otherwise('all-reviews');
}])
.run(['$rootScope', '$state', function($rootScope, $state) {
  $rootScope.$on('$stateChangeStart', function(event, next) {
    // redirect to login page if not logged in
    if (next.authenticate && !$rootScope.currentUser) {
      event.preventDefault(); //prevent current page from loading
      $state.go('forbidden');
```

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

Lines 2 - 4 include dependencies `app`, `ui.router`, and `lbServices`. The latter is the AngularJS services library you generated previously using `lb-ng`.

Lines 61 - 66 define an interceptor that triggers when a state change happens: If the user is not logged in, then redirect to the forbidden page.

The other lines define application states. *States* determine which pages appears when the user navigates, changes URLs, or clicks on a link. Any state for which `authenticate` is `true` requires you to log in first. If you navigate directly to one of these URLs, you will see a forbidden access page (state = `forbidden`, url = `/forbidden`. Each call to `state()` specifies the template to use for the state, the controller to use, and whether authentication is required.

The following table summarizes the states, and how the correspond to controllers, templates, and URLs.

State	URL	Description	Controller	View / Template	Must be logged in?
'add-review'	/add-review	Add a new coffee shop review.	AddReviewController	review-form.html	Yes
'all-reviews'	/all-reviews	List all reviews.	AllReviewsController	all-reviews.html	No
'edit-review'	/edit-review/:id	Edit selected review.	EditReviewController	review-form.html	Yes
'delete-review'	/delete-review/:id	Delete selected review.	DeleteReviewController	None	Yes
'forbidden'	/forbidden	Forbidden URL error. • Notifies user they can't perform the action. • Displays link to login page.	EditReviewController	forbidden.html	No
'login'	/login	Login Redirects to add-review page upon successfully login	AuthLoginController	login.html	No
'logout'	/logout	Logout • Notifies user they've logged out. • Display link to to the all-reviews page.	AuthLogoutController	None	No
'my-reviews'	/my-reviews	List only reviews of the logged-in user.	MyReviewsController	my-reviews.html	Yes
'sign-up'	/sign-up	Sign up for account.	SignUpController	sign-up-form.html	No
'sign-up-success'	/sign-up/success	Successful sign-up. Display link to /all-reviews page.	None	sign-up-success.html	No

Controllers

In Angular, a *controller* is a JavaScript constructor function that is used to augment the Angular Scope.

When a controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified constr uctor function. A new child scope will be available as an injectable parameter to the controller's constructor function as `$scope`. For more information on controllers, see Understanding Controllers (AngularJS documentation).

The `client/js/controllers` directory contains two files that define controllers: `auth.js` and `review.js`.

The controller in `auth.js` handles user registration, login, and logout. When the user is logged in, a `currentUser` object is set in the root scope. Other parts of the app check the `currentUser` object when performing actions. When logging out, the `currentUser` object is destroyed.

js/controllers/auth.js

Expand source

```
angular
  .module('app')
  .controller('AuthLoginController', ['$scope', 'AuthService', '$state',
      function($scope, AuthService, $state) {
    $scope.user = {
      email: 'foo@bar.com',
      password: 'foobar'
    };
    $scope.login = function() {
      AuthService.login($scope.user.email, $scope.user.password)
        .then(function() {
          $state.go('add-review');
        });
    };
  }])
  .controller('AuthLogoutController', ['$scope', 'AuthService', '$state',
      function($scope, AuthService, $state) {
    AuthService.logout()
      .then(function() {
        $state.go('all-reviews');
      });
  }])
  .controller('SignUpController', ['$scope', 'AuthService', '$state',
      function($scope, AuthService, $state) {
    $scope.user = {
      email: 'baz@qux.com',
      password: 'bazqux'
    };
    $scope.register = function() {
      AuthService.register($scope.user.email, $scope.user.password)
        .then(function() {
          $state.transitionTo('sign-up-success');
        });
    };
  }]);
```

The other file, `review.js`, defines controllers for review actions.

Expand source

```
angular
  .module('app')
  .controller('AllReviewsController', ['$scope', 'Review', function($scope,
      Review) {
    $scope.reviews = Review.find({
      filter: {
        include: [
          'coffeeShop',
          'reviewer'
        ]
      }
    });
  }])
```

```
.controller('AddReviewController', ['$scope', 'CoffeeShop', 'Review',
    '$state', function($scope, CoffeeShop, Review, $state) {
  $scope.action = 'Add';
  $scope.coffeeShops = [];
  $scope.selectedShop;
  $scope.review = {};
  $scope.isDisabled = false;
  CoffeeShop
    .find()
    .$promise
    .then(function(coffeeShops) {
      $scope.coffeeShops = coffeeShops;
      $scope.selectedShop = $scope.selectedShop || coffeeShops[0];
    });
  $scope.submitForm = function() {
    Review
      .create({
        rating: $scope.review.rating,
        comments: $scope.review.comments,
        coffeeShopId: $scope.selectedShop.id
      })
      .$promise
      .then(function() {
        $state.go('all-reviews');
      });
  };
}])
.controller('DeleteReviewController', ['$scope', 'Review', '$state',
    '$stateParams', function($scope, Review, $state, $stateParams) {
  Review
    .deleteById({ id: $stateParams.id })
    .$promise
    .then(function() {
      $state.go('my-reviews');
    });
}])
.controller('EditReviewController', ['$scope', '$q', 'CoffeeShop', 'Review',
    '$stateParams', '$state', function($scope, $q, CoffeeShop, Review,
    $stateParams, $state) {
  $scope.action = 'Edit';
  $scope.coffeeShops = [];
  $scope.selectedShop;
  $scope.review = {};
  $scope.isDisabled = true;
  $q
    .all([
      CoffeeShop.find().$promise,
      Review.findById({ id: $stateParams.id }).$promise
    ])
    .then(function(data) {
      var coffeeShops = $scope.coffeeShops = data[0];
      $scope.review = data[1];
      $scope.selectedShop;
      var selectedShopIndex = coffeeShops
        .map(function(coffeeShop) {
          return coffeeShop.id;
        })
        .indexOf($scope.review.coffeeShopId);
      $scope.selectedShop = coffeeShops[selectedShopIndex];
```

```
    });
  $scope.submitForm = function() {
    $scope.review.coffeeShopId = $scope.selectedShop.id;
    $scope.review
      .$save()
      .then(function(review) {
        $state.go('all-reviews');
      });
  };
}])
.controller('MyReviewsController', ['$scope', 'Review', '$rootScope',
    function($scope, Review, $rootScope) {
  $scope.reviews = Review.find({
    filter: {
      where: {
        publisherId: $rootScope.currentUser.id
      },
      include: [
        'coffeeShop',
        'reviewer'
      ]
```

Copyright 2016 StrongLoop, an IBM company

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

The following table describes the controllers defined in `review.js`.

Controller	Description
AllReviewsController	Performs a `Review.find()` to fetch reviews. Uses an include filter to add coffeeShop and review models. This is possible due the relations previously defined.
AddReviewController	Coffee shops are populated from the server when the page first loads via CoffeeShop.find() down menu. When the form is submitted, we create a review and change to the `all-reviews` page when the promise resolves.
DeleteReviewController	There is no view corresponding to this state when triggered; the corresponding review is deleted by ID. The ID is in the URL.
EditReviewController	Similar to AddReviewController when the page is first loaded. The app performs two requests at the same time using $q to get the required models. With these models, it then populates the dropdown menu with the available coffee shops. Once the app has displayed the coffee shops in the dropdown, it selects the coffee shop previously chosen in the original review. Then the app sets `coffeeShopId` to the selected coffee shop.
MyReviewController	Similar to AllReviewsController, this controller uses a "where" filter to restrict the result set based on the publisherId, where publisherId is set from the currently logged-in user. It then uses an include filter to include coffeeShop and reviewer models.

Services

Angular *services* are substitutable objects that you connect together using dependency injection (DI). You can use services to organize and share code across your app.

The `js/services` directory contains two AngularJS services libraries: `auth.js` and `lb-services.js`.

You generated the lb`-services.js` previously, and it's described in Generate lb-services.js.

The other file, `auth.js`, provides a simple interface for low-level authentication mechanisms. It uses the `Reviewer` model (that extends the base `User` model) and defines the following services:

- `login`: logs a user inLoopback automatically manages the authentication token is stored in browser HTML5 localstorage.
- `logout`: logs a user out. Stores the token in browser HTML5 localstorage.
- `register`: registers a new user with the provided email and password, the mininum requirements for creating a new user in LoopBack.

js/services/auth.js

Expand source

```
angular
  .module('app')
  .factory('AuthService', ['Reviewer', '$q', '$rootScope', function(User, $q,
      $rootScope) {
    function login(email, password) {
      return User
        .login({email: email, password: password})
        .$promise
        .then(function(response) {
          $rootScope.currentUser = {
            id: response.user.id,
            tokenId: response.id,
            email: email
          };
        });
    }
    function logout() {
      return User
       .logout()
       .$promise
       .then(function() {
         $rootScope.currentUser = null;
       });
    }
    function register(email, password) {
      return User
        .create({
         email: email,
         password: password
       })
       .$promise;
    }
    return {
      login: login,
      logout: logout,
      register: register
    };
  }]);
```

Views

The `client/views` directory contains seven "partial" view templates loaded by `client/index.html` using the ngView directive A "partial" is a segment of a template in its own HTML file.

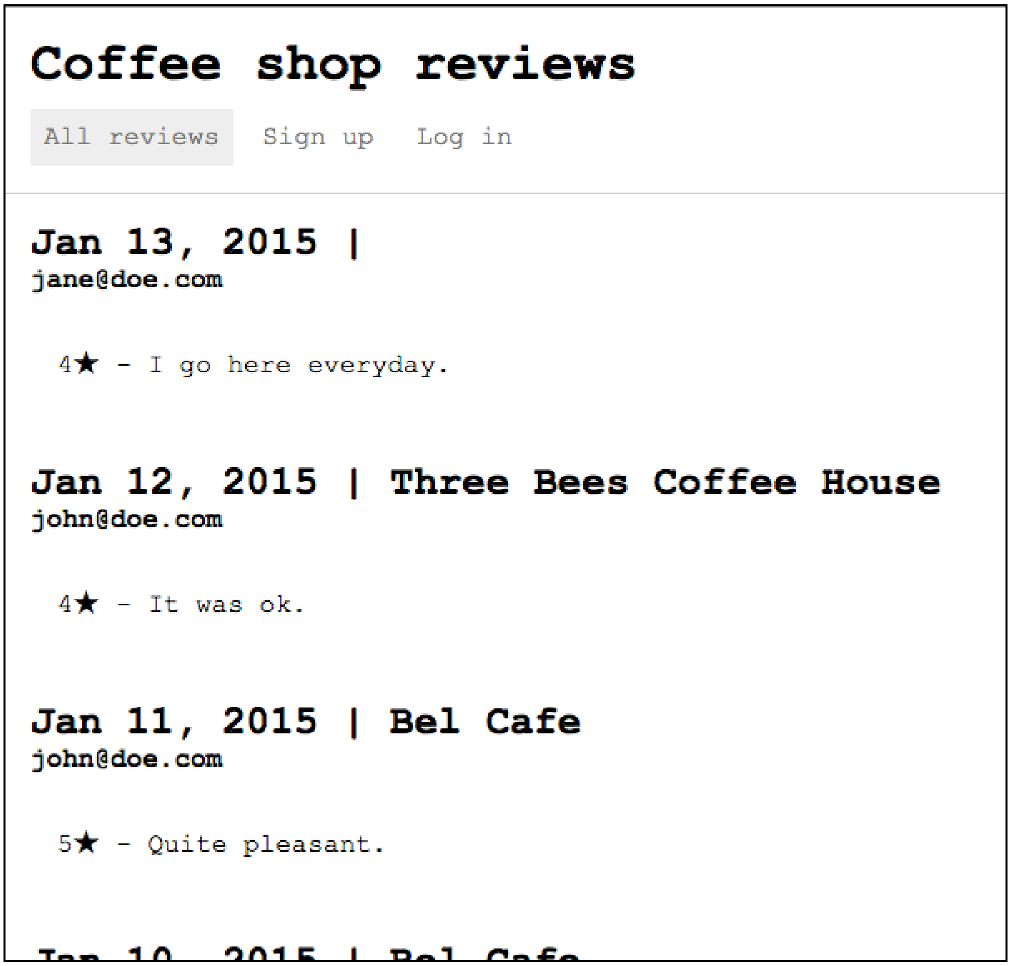
The table above describes how the views correspond to states and controllers.

Run the application

Now you can run the Coffee Shop Reviews application:

```
$ node .
...
Browse your REST API at http://0.0.0.0:3000/explorer
Web server listening at: http://0.0.0.0:3000/
> models created sucessfully
```

Now load http://0.0.0.0:3000/ in your browser. You should see the application home page:



You should be able to run the application through its paces, as described in Introducing the Coffee Shop Reviews app.

Next: Go to Deploying your application for an example of deploying your application to production using StrongLoop Process Manager.

Deploying your application

Prerequisite: Install StrongLoop software as described in Installing StrongLoop.

Recommended: Read LoopBack core concepts.

- Running StrongLoop Process Manager
- Building and deploying with Arc

StrongLoop Process Manager manages Node applications, providing automatic restart, cluster control, profiling, monitoring and many other features.

If you followed the previous step in the tutorial, go to Run ning StrongLoop Process Manager

If you're just jumping in, follow the steps below to catch up...

Get the application (in the state following the last article plus all the client files) from GitHub and install all its dependencies:

```
$ git clone https://github.com/strongloop/loopback-getting-started-intermediate.git
$ cd loopback-getting-started-intermediate
$ git checkout step7
$ npm install
```

If you've followed the tutorial this far, you have an idea of how easy it is to create and modify an application. Now, you are ready to build and deploy it with StrongLoop Process Manager.

You'll see how to use both here.

Running StrongLoop Process Manager

In practice, when you're ready to deploy to production, you would install and run StrongLoop PM on your production host, then use Arc to deploy your app there. For more information, see Setting up a production host

For the purposes of this tutorial, you'll just run StrongLoop PM locally, then deploy your app to `localhost`. It all works basically the same, but there are few differences when you're actually ready to go to production. You might want to do this, for example, to test your application's behavior under clustered mode for scaling.

Run StrongLoop PM with the following command. By default, StrongLoop PM listens on port 8701. You can change this default with the `-l` option. See slc pm for more information.

```
$ slc pm
slc pm: StrongLoop PM v3.1.9 (API v5.0.3) listening on port `8701`
slc pm: control listening on path `/loopback-getting-started-intermediate/pmctl`
slc pm: listen on 8701, work base is
`/loopback-getting-started-intermediate/.strong-pm`
```

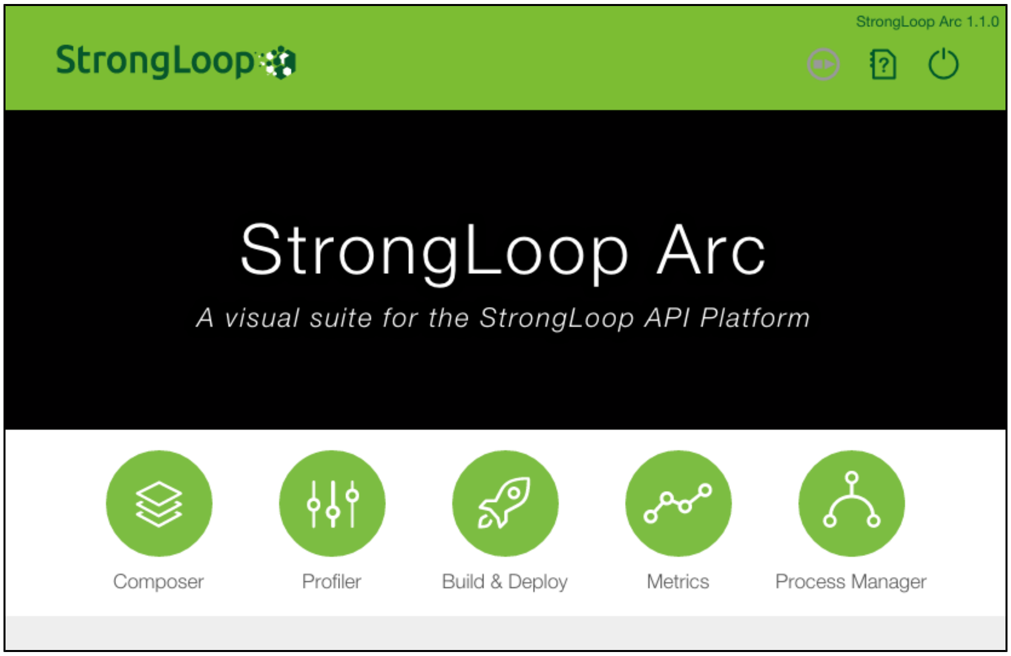
Building and deploying with Arc

Next, make sure you are in `loopback-getting-started-intermediate` directory, then start the StrongLoop Arc with the `slc arc` command:

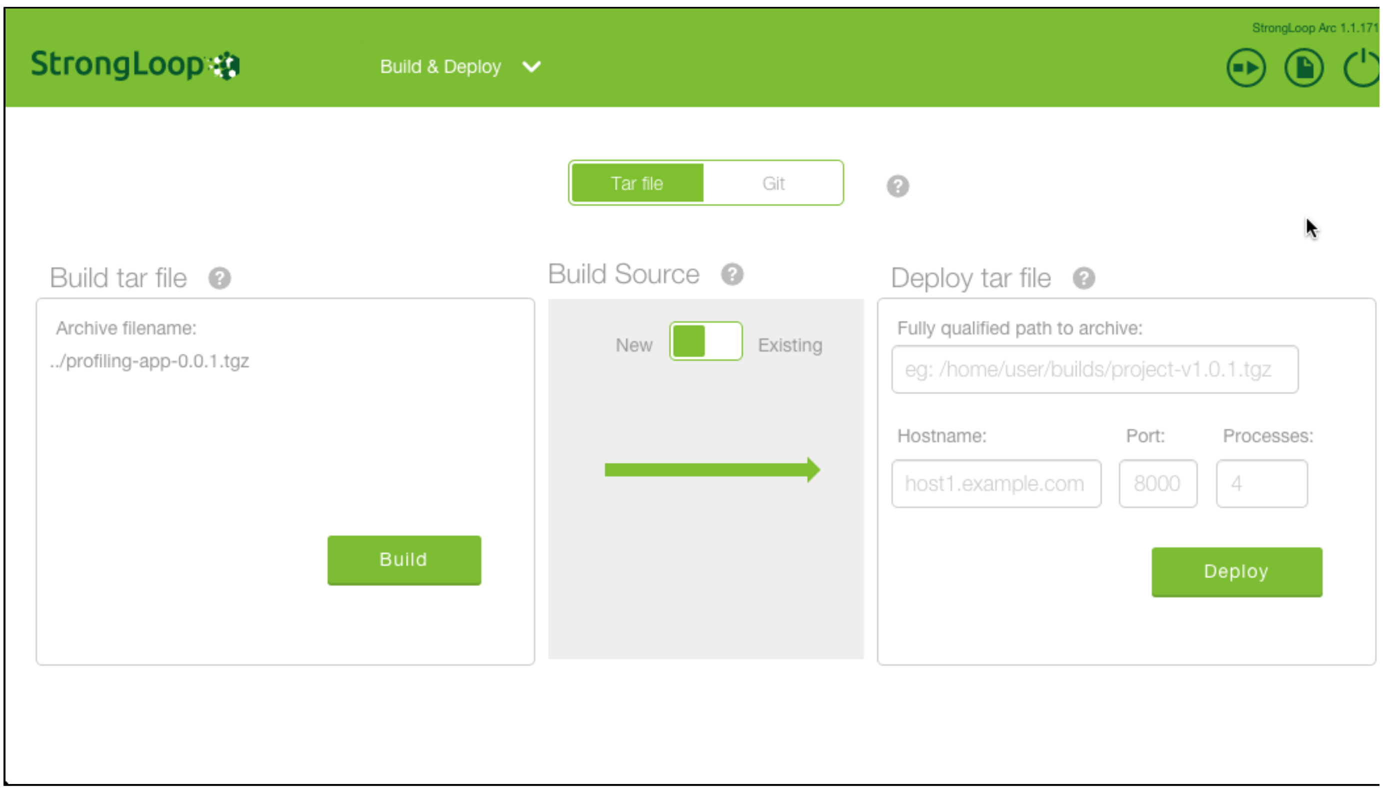
```
$ cd loopback-getting-started-intermediate
$ slc arc
```

StrongLoop Arc will open in your default browser, and you'll see the login page. Arc is free to use, but you must log in. If you haven't registered for an account, click **Register** on the bottom right of the **Sign In** page to do so. See Using Arc for details.

Once you log in, you'll see the StrongLoop Arc launchpad:



Then, click **Build & Deploy** to display the Arc Build and Deploy module:



Arc provides two ways to build and deploy an application: using a tar file and using a Git archive. You're going to use a tar file, so leave the default **Tar file** selected.

1. Click **Build** to build a tar file for the app. You'll see a progress spinner, and then the message "Successfully built using tar file" when it completes. Arc creates a tar file in the parent directory of your working directory. It will display the name of the .tgz file it creates, in this case: `../loopback-getting-started-intermediate-0.0.1.tgz`.
2. Then, under **Deploy tar file**, enter the following:
   - **Fully qualified path to archive**: the path to the tar file just created, for example: `/Users/rand/loopback-getting-started-intermediate-0.0.1.tgz`.
   - **Hostname**: localhost (for real production deployment, you'd use the remote host name)
   - **Port**: 8701
   - **Processes**: Enter a number less than or equal to the number of processors on the system to which you're deploying; for example, 2.

3. Click **Deploy**.
   You'll see a progress spinner, then when the deployment completes, the message "Successfully deployed using tar file".

You can then load the application home page at http://localhost:3001/ and the API explorer at http://localhost:3001/explorer/.

The app is now deployed to the local Process Manager. If you had set up StrongLoop PM on a remote production host, you'd have a production deployment with clustering and automatic restart.

Next: Go to Learn more for some tips on what to read next.

Learn more

There's so much more to learn about LoopBack!

LoopBack documentation includes lots of tutorials and examples; also read the StrongLoop blog for great tips and information.

StrongLoop Doc Tips

Use the left navigation tree to browse and find doc articles; Click [+] to expand subjects under a topic.

Use the search field at upper right to search the documentation.

Learn more about:

- **Creating an application**
- **Managing users**
  - Registering users
  - Logging in users
  - Partitioning users with realms
- **Authentication, authorization, and permissions**
- **Defining models**
  - Creating models
  - Customizing models
  - Attaching models to data sources
  - Exposing models over REST
  - Validating model data
  - Creating model relations
- **Working with data**
- **Adding application logic**
  - Adding logic to models
  - Defining boot scripts
  - Defining mixins
  - Defining middleware
  - Working with LoopBack objects
  - Using current context
  - Events
- **AngularJS client**
  - Angular example app
  - AngularJS Grunt plugin
  - Generating Angular API docs

Check out the LoopBack Developer Forum on Google Groups, a place where devlopers can ask questions, discuss LoopBack, and how they are using it.

Creating an application

Prerequisites

- Install StrongLoop software.
- Read LoopBack core concepts first.
- Follow Getting started with LoopBack for a basic introduction to LoopBack.

- Creating a new application
  - Standard project layout

Related articles:

- Main application script (server.js)

See also:

- Creating models
- Using built-in models

Creating a new application

As you saw in Getting Started > Create a simple API, the easiest way to create an application is to use `slc loopback`, the application generator.

> It is possible to create a LoopBack application by coding it from scratch, but `slc loopback` does all the "heavy lifting" to create the basic scaffolding of the standard project layout. You can then customize the application to suit your needs. When you create your application this way, you can continue to use `slc loopback` to add models, data sources, and so on.
>
> In general, the documentation assumes you've created your application using `slc loopback`.

Once you create your application, you may want to configure it, for example: Turn off stack traces, disable API Explorer, and retrieve the values of environment variables. See Environment-specific configuration for more information.

Standard project layout

The application generator creates an application with the standard project layout. To summarize:

- `server` directory
  - `server.js` - Main application script; see below.
  - `config.json` - Global application settings, such as the REST API root, host name and port to use, and so on. See config.json.
  - `model-config.json` - Binds models to data sources and specifies whether a model is exposed over REST, among other things. See model-config.json.
  - `datasources.json` - Data source configuration file. See datasources.json.
- `client` directory (empty except for a README stub)
- `common/models` directory - created when you create a model with the Model generator, `slc loopback:model`.
  - A JSON file and a JavaScript file for each model (for example, `my-model.json` and `my-model.js`).

Main application script (server.js)

This is the main application script in the standard scaffolded application, as created by `slc loopback`.

1 - 3: Require LoopBack modules and set up standard objects `loopback`, `app`, and `boot`.

4: Start the web server.

7: Emit the 'started' event.

10 - 13: Start API Explorer.

18: Initialize (boot) the application.

```
var loopback = require('loopback');
var boot = require('loopback-boot');
var app = module.exports = loopback();
app.start = function() {
  // start the web server
  return app.listen(function() {
    app.emit('started');
    var baseUrl = app.get('url').replace(/\/$/, '');
    console.log('Web server listening at: %s', baseUrl);
    if (app.get('loopback-component-explorer')) {
      var explorerPath =
app.get('loopback-component-explorer').mountPath;
      console.log('Browse your REST API at %s%s',
baseUrl, explorerPath);
    }
  });
};
// Bootstrap the application, configure models,
datasources and middleware.
// Sub-apps like REST API are mounted via boot scripts.
boot(app, __dirname, function(err) {
  if (err) throw err;
  // start the server if `$ node server.js`
  if (require.main === module)
    app.start();
});
```

Using LoopBack tools

LoopBack provides two powerful tools for creating and working with applications:

- The command-line tool slc loopback.
- StrongLoop Arc, a graphical tool.

The slc loopback command-line tool

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

The `slc loopback` command provides an Application generator to create a new LoopBack application and a number of sub-generators to scaffold an application, as described in the following table. The commands are listed roughly in the order that you would use them.

Command	**See**	**Description**
`slc loopback`	Application generator	Create a new LoopBack application.
`slc loopback:datasource`	Data source generator	Add a new data source to a LoopBack application
`slc loopback:model`	Model generator	Add a new model to a LoopBack application.
`slc loopback:property`	Property generator	Add a new property to an existing model.

`slc loopback:acl`	ACL generator	Add a new access control list (ACL) entry to the LoopBack application.
`slc loopback:relation`	Relation generator	Add a new model relationship.
`slc loopback:middleware`	Middleware generator	Add a new middleware configuration.
`slc loopback:boot-script`	Boot script generator	Add a new boot scripts.
`slc loopback:export-api-def`	API definition generator	Export Swagger API definition.
`slc loopback:swagger`	Swagger generator	Generates a fully-functional application that provides the APIs conforming to the Swagger 2.0 specification.

The `slc` command has many additional sub-commands not specific to LoopBack for building, deploying, and managing Node applications. See Operating Node applications for more information and Command-line reference for the command reference.

StrongLoop Arc

StrongLoop Arc is a graphical tool for building, deploying, and monitoring LoopBack applications.

Arc Composer enables you to:

- **Create and modify models**. See Creating and editing models for more information.
- **Create and modify data sources**. See Creating and editing data sources for more information.
- **Discover models from data sources** that support the discovery API. See Discovering models from a database for more information.
- **Automatically create database schemas** based on your application models (auto-migration). See Migrating a model for more information.

For more information, see Composing APIs.

Profiler module enables you to generate and view:

- Application CPU profiles (per process).
- Application heap snapshots (per process), to help diagnose memory leaks.

For more information, see Profiling with Arc.

Click **Profile Settings (full)** to set up Smart Profiling. See Smart profiling with Arc for more information.

Build & Deploy module enables you to build, package, and deploy your Node application to a local or remote system.

For more information, see Building and deploying with Arc.

Metrics module enables you to gather and view performance metrics on your application (per process).

For more information, see Viewing metrics with Arc.

Process Manager module enables you to manage an application running in clustered mode across multiple server hosts.

For more information, see Connecting to Process Manager from Arc.

Tracing is available as a private beta feature.

If you are interested, please contact **sales@strongloop.com** for instructions to enable it.

The StrongLoop Arc **Tracing** module enables you to analyze performance and execution of Node applications to discover bottlenecks and trace code execution paths. You can display up to five hours of data to discover how applications perform over time.

You can drill down into specific function calls and execution paths for HTTP and database requests, to see how and where your application is spending time. Tracing provides powerful "flame graph" visualization of an application's function call stack and the corresponding execution times to help you track down where the application spends its time.

For more information, see Tracing.

Environment-specific configuration

- Overview

Overview

LoopBack applications have the following types of configuration files:

- **Application-wide configuration files**, by default `server/config.json`. You can also use `server/config.local.js` to set values that you can't with simple JSON.
- **Data source configuration files**, by default `server/datasources.json`. You can also use `server/datasources.local.js` to set values that you can't with simple JSON.
- **Application-level configuration of Models**, by default `server/model-config.json`.
- **Middleware configuration files**, by default `server/middleware.json`.
- **Configuration files for LoopBack components**, by default `server/component-config.json.`

LoopBack will always load the following configuration files, if they exist:

- `server/config.json`.
- `server/config.local.json` *or* `server/config.local.js`.
- `server/datasources.json`
- `server/datasources.local.json` *or* `server/datasources.local.js`
- `server/model-config.json`
- `server/model-config.local.json` *or* `server/model-config.local.js`
- `server/middleware.json`
- `server/middleware.local.json` *or* `server/middleware.local.js`
- `server/component-config.json`
- `server/component-config.local.json` *or* `server/component-config.local.js`

Additionally, when the NODE_ENV environment variable is set, LoopBack will load configuration from:

- `server/config.{env}.json/js`
- `server/datasources.{env}.json/js`
- `server/model-config.{env}.json/js`
- `server/middleware.{env}.json/js`
- `server/component-config.{env}.json/js`

where `{env}` is the value of NODE_ENV (typically "development," "staging," or "production"). This enables you to set up configurations for specific environments (for example, development, staging, and production).

A LoopBack application can load multiple configuration files, which can potentially conflict with each other. The value set by the file with the highest priority will always take effect. The priorities are:

1. **Environment-specific configuration**, based on the value of NODE_ENV; for example, `server/config.staging.json`.
2. **Local configuration file**; for example, `server/config.local.json`.
3. **Default configuration file**; for example, `server/config.json`.

Here are some examples of the application configuration files:

- config.json
- config.local.js

Here are some examples of data source configuration files:

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

Here are some examples of the middleware configuration files, the `server/middleware{.env}.json` file:

- middleware.json
- middleware.production.json

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

Application-wide configuration

Define application server-side settings in `server/config.json`.

You can override values that are set in `config.json` in:

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

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

For example:

config.production.js

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

Turning off stack traces

By default, stack traces are returned in JSON responses. To turn disable stack traces in JSON responses:

- Set the NODE_ENV environment variable to "production"
- Include the following in `server/middleware.production.json`:

server/middleware.production.json

```
"final:after": {
    "loopback#errorHandler": {
      "params": {
        "includeStack": false
      }
    }
  }
```

> As of `strongloop` version 5.0.3, the Application generator creates a middleware.production.json file with the above configuration for you, so all you have to do is set the NODE_ENV environment variable.

Disabling API Explorer

LoopBack API Explorer is great when you're developing your application, but for security reasons you may not want to expose it in production.

For an application using loopback-component-explorer, to disable explorer in production:

- Set the NODE_ENV environment variable to "production".
- Then in `server/component-config.production.json`:

server/component-config.production.json

```
{
  "loopback-component-explorer": false
}
```

For an application using the old `loopback-explorer` (prior to version 2.0), disable API Explorer by deleting or renaming `server/boot/explorer.js`.

Customizing REST error handling

You can customize the REST error handler by adding the error handler callback function to `server/config.js` as follows:

server/config.js

```
module.exports = {
  remoting: {
    errorHandler: {
      handler: function(err, req, res, next) {
        // custom error handling logic
        var log = require('debug')('server:rest:errorHandler'); // example
        log(req.method, req.originalUrl, res.statusCode, err);
        next(); // call next() to fall back to the default error handler
      }
    }
  }
};
```

Exclude stack traces from HTTP responses

To exclude error stack traces from HTTP responses (typical in production), set the includeStack option of LoopBack errorHandler middleware to false in middleware.json.

The standard configuration for development is:

server/middleware.json

```
...
  "final:after": {
    "loopback#errorHandler": {}
  }
}
```

For production, exclude stack traces from HTTP responses as follows:

server/middleware.production.json

```
...
  "final:after": {
    "loopback#errorHandler": {
      "params": {
        "includeStack": false
      }
    }
  }
}
```

Data source configuration

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

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

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

Example data sources:

datasources.json

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

datasources.production.json

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

Getting values from environment variables

You can easily set an environment variable when you run an application. The command you use depends on your operating system.

MacOS and Linux

Use this command to set an environment variable and run an application in one command:

```
$ MY_CUSTOM_VAR="some value" node .
```

or in separate commands:

```
$ export MY_CUSTOM_VAR="some value"
$ node .
```

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

Windows

On Windows systems, use these commands:

```
C:\> set MY_CUSTOM_VAR="some value"
C:\> node .
```

Versioning your API

You can easily add versioning to your REST API routes, based on the application "major" version in `package.json`.

Add a file named `config.local.js` in the application's `/server` directory with the following code:

/server/config.local.js

```
var p = require('../package.json');
var version = p.version.split('.').shift();
module.exports = {
  restApiRoot: '/api' + (version > 0 ? '/v' + version : ''),
  host: process.env.HOST || 'localhost',
  port: process.env.PORT || 3000
};
```

This takes the major version number from the `version` property in `package.json` and appends it to the REST API root.   If your app's major version is 0, then the REST API root remains the default `/api`.

So, for example, if `version` in `package.json` is 2.0.1, then the built-in model route exposed by default at:

`GET http://localhost:3000/api/Users`

is now exposed at:

`GET http://localhost:3000/api/v2/Users`

> Changing the API root in this way doesn't affect routes set in request-handling middleware or the route to API Explorer itself, which remains `http://localhost:3000/explorer`.

Standard project structure

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

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

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

> All your model JSON and JavaScript files go in the `/common/models` directory.

File or directory	Description	How to access in code
	Top-level application directory	
`/node-modules` directory	Contains Node packages as specified as dependencies in `package.json`. Update with `npm install`.	N/A
`package.json`	Standard npm package specification. See package.json.	N/A
`README.md`	Stub file for internal documentation.	N/A
	/server directory - Node application files	
`/boot` directory	Add scripts to perform initialization and setup. See boot scripts.	Scripts are automatically executed in alphabetical order.
`component-config.json`	Specifies LoopBack components to load.	
`config.json`	Application settings. See config.json.	`app.get('setting-name')`
`datasources.json`	Data source configuration file. See datasources.json. For an example, see Create new data source.	`app.datasources['datasource-name']`
`middleware.json`	Middleware definition file. For more information, see Defining middleware.	N/A
`middleware.production.json`	Middleware definition file with production configuration. See Preparing for deployment.	
`model-config.json`	Model configuration file. See model-config.json. For more information, see Connecting models to data sources.	N/A
`server.js`	Main application program file.	N/A
	/client directory - Client application files	
README.md	LoopBack generators create empty `README.md` file.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
	/common directory - shared application files	
`/models` directory	Custom model files: • Model definition JSON files, by convention named `*model-name*.json`; for example `customer.json`. • Custom model scripts by convention named `*model-name*.js`; for example, `customer.js`. For more information, see Model definition JSON file and Customizing models. See note below.	Node: `myModel = app.models.myModelName`

> The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

Managing users

Prerequisites

- Install StrongLoop software.
- Read LoopBack core concepts first.
- Follow Getting started with LoopBack for a basic introduction to LoopBack.

Related articles:

See also:

- Authentication, authorization, and permissions
- Third-party login (Passport)
- loopback-example-user-management

Overview

LoopBack's built-in User model provides essential user management features such as:

- Registration and confirmation via email.
- Login and logout.
- Creating an access token.
- Password reset.

IMPORTANT
You must create your own custom model (named something other than "User," for example "Customer" or "Client") that extends the built-in User model rather than use the built-in User model directly. The built-in User model provides a great deal of commonly-used functionality that you can use via your custom model.

Watch this video for an introduction to user management in LoopBack:

Creating and authenticating users

The basic process to create and authenticate users is:

1. Register a new user with the `User.create()` method, inherited from the generic `PersistedModel` object.
   See Registering users for more information.
2. Log in a user by calling `User.login()` to get an access token.
   See Logging in users for more information.
3. Make subsequent API calls using the access token. Provide the access token in the HTTP header or as a query parameter to the REST API call, as shown in Making authenticated requests with access tokens.

Performance tip
To improve performance during login and user creation, try installing native bcrypt.

```
$ npm install --save bcrypt
```

Understanding the built-in User model

By default, a LoopBack application has a built-in User model defined by user.json (this file is part of the LoopBack framework. Don't modify it; rather, follow the procedure in Extending built-in models).

For a basic introduction to how the LoopBack user model performs authentication, see Introduction to User model authentication.

Default access controls

The built-in User model has the following ACL:

```
{
  "name": "User",
  "properties": {
  ...
  "acls": [
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "create"
    },
    {
      "principalType": "ROLE",
      "principalId": "$owner",
      "permission": "ALLOW",
      "property": "deleteById"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "login"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "logout"
    },
    {
      "principalType": "ROLE",
      "principalId": "$owner",
      "permission": "ALLOW",
      "property": "findById"
    },
    {
      "principalType": "ROLE",
      "principalId": "$owner",
      "permission": "ALLOW",
      "property": "updateAttributes"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "confirm"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "resetPassword",
      "accessType": "EXECUTE"
```

```
    }
  ],
  ...
```

The above ACL denies all operations to everyone, then selectively allows:

- Anyone to create a new user (User instance).
- Anyone to log in, log out, confirm their identity, and reset their own password.
- A user to perform deleteById, findById, and updateAttributes on their own User record (instance).

> You cannot directly modify built-in models such as the User model with the ACL generator `slc loopback:acl`.
>
> However, you can create a custom model that extends the built-in User model, then use the ACL generator to define access controls that are added to those of the default User model. For example, you could create a Customer or Client model that extends the built-in User model, and then modify that model's ACL with `slc loopback:acl`.

User realms

See Partitioning users with realms.

Registering users

The LoopBack User model provides methods to register new users and confirm their email addresses. You can also use the loopback-component-passport module to integrate login with Facebook, Google, and other third-party providers.

- Registering users with the LoopBack User model
  - Creating a new user
  - Adding other registration constraints
  - Verifying email addresses
- Registering users through a third-party system

See also:

- Authentication, authorization, and permissions
- Third-party login (Passport)

Registering users with the LoopBack User model

Creating a new user

Create a user (register a user) by adding a model instance, in the same way as for any other model; email and password are the only required properties.

/common/models/user.js

```
module.exports = function(app) {
  var User = app.models.User;
  User.create({email: 'foo@bar.com', password: 'bar'}, function(err, user) {
    console.log(user);
  });
  ...
```

> You can also do this in a boot script.

Over REST, use the `POST /users` endpoint to create a new user instance, for example:

REST

```
curl -X POST -H "Content-Type:application/json" \
-d '{"email": "me@domain.com", "password": "secret"}' \
http://localhost:3000/api/users
```

For more information, see User REST API.

Adding other registration constraints

Typically, you might want to add methods as part of the registration process, for example to see if a given username is available or if an email address is already registered. A good way to do this is to add methods as beforeRemote hooks on the User object. See Remote hooks for more information.

Verifying email addresses

Typically, an application will require users to verify their email addresses 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.

To enforce this constraint, set the `emailVerificationRequired` user model property to true; in `server/model-config.json`; for example:

See also:

More information on Email model:

- Using the Email model
- Email connector

server/model-config.json

```
...
"user": {
    "dataSource": "db",
    "public": true,
    "options": {
      "emailVerificationRequired": true
    }
...
```

Over REST, use the `GET /users/confirm` endpoint to verify a user's email address. For details, see User REST API.

This example creates a remote hook on the User model executed after the `create()` method is called.

> The example below assumes you have setup a `User` model and `Mail` datasource.

/common/models/user.js

```
var config = require('../../server/config.json');
var path = require('path');

module.exports = function(user) {
  //send verification email after registration
  user.afterRemote('create', function(context, user, next) {
    console.log('> user.afterRemote triggered');

    var options = {
      type: 'email',
      to: user.email,
      from: 'noreply@loopback.com',
      subject: 'Thanks for registering.',
      template: path.resolve(__dirname, '../../server/views/verify.ejs'),
      redirect: '/verified',
      user: user
    };

    user.verify(options, function(err, response, next) {
      if (err) return next(err);

      console.log('> verification email sent:', response);

      context.res.render('response', {
        title: 'Signed up successfully',
        content: 'Please check your email and click on the verification link ' +
            'before logging in.',
        redirectTo: '/',
        redirectToLinkText: 'Log in'
      });
    });
  });
...
```

For a complete example, see user.js in loopback-example-user-management.

Naming your model with camel-case `MyUser` will create files in "lisp case" `/common/models/my-user.js` + `/common/models/my-user.json`

Then, in your view file (for example, an EJS template):

verify.ejs

```
This is the html version of your email.
<strong><%= text %></strong>
```

Registering users through a third-party system

Use the LoopBack Passport component (loopback-component-passport) to enable users to register and log in to your application using existing credentials from:

- Facebook

- Google
- Twitter

For more information, see Third-party login (Passport).

Logging in users

See also:

- Authentication, authorization, and permissions
- Third-party login (Passport)

Login with the LoopBack User model

Logging in

Login (authenticate) a user by calling the `User.login()` method and providing an object containing `password` and `email or username` properties as the first parameter. The method returns an access token.

This example creates a route in boot script to handle login request:

/server/boot/routes.js

```
...
  app.post('/login', function(req, res) {
    User.login({
      email: req.body.email,
      password: req.body.password
    }, 'user', function(err, token) {
      if (err) {
        res.render('response', { //render view named 'response.ejs'
          title: 'Login failed',
          content: err,
          redirectTo: '/',
          redirectToLinkText: 'Try again'
        });
        return;
      }

      res.render('home', { //login user and render 'home' view
        email: req.body.email,
        accessToken: token.id
      });
    });
  });
...
```

`User.login()` has an optional second parameter that is a string or an array of strings. Pass in "user" for this parameter to include the user information. For REST apis, using *?include=user*.

For a complete example, see routes.js in loopback-example-user-management.

You may also specify how long the access token is valid by providing a `ttl` (time to live) property with a value in **seconds**. For example:

Boot script

```
var TWO_WEEKS = 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
});
```

If a login attempt is unsuccessful, an error will be returned in the following format.

```
{
  "status": 401,             // or 400 if the credentails object is invalid
  "message": "login failed"  // could also be "realm is required" or "username or
email is required"
}
```

Over REST, use the `POST /users/login` endpoint; for example:

Shell

```
curl -X POST -H "Content-Type:application/json" \
-d '{"email": "me@domain.com", "password": "secret", "ttl": 1209600000}' \
http://localhost:3000/api/users/login
```

The return value is a JSON object with an `id` property that is the access token to be used in subsequent requests; for example:

Shell

```
{
  "id": "GOkZRwgZ61q0XXVxvxlB8TS1D6lrG7Vb9V8YwRDfy3YGAN7TM7EnxWHqdbIZfheZ",
  "ttl": 1209600,
  "created": "2013-12-20T21:10:20.377Z",
  "userId": 1
}
```

See User REST API for more information.

Logging out

Use the `User.logout()` method to log out a user, providing the user's access token as the parameter.

In the example below, a route to handle logout request is created:

/server/boot/routes.js

```
...
  //log a user out
  app.get('/logout', function(req, res, next) {
    if (!req.accessToken) return res.sendStatus(401); //return 401:unauthorized if
accessToken is not present
    User.logout(req.accessToken.id, function(err) {
      if (err) return next(err);
      res.redirect('/'); //on successful logout, redirect
    });
  });
...
```

To logout user from the server side:

```
// logout a user (server side only)
User.findOne({email: 'foo@bar.com'}, function(err, user) {
  user.logout();
});
```

Over REST, use the `POST /users/logout` endpoint, again providing the user's access token in the `sid` property of the POST payload.

To destroy access tokens over REST API, use the `POST /users/logout` endpoint.

Shell

```
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
```

See User REST API for more information.

Login using third-party systems

Instead of using LoopBack's user system, you can integrate with a third-party system that supports OAuth, such as Google, Facebook, or Twitter.

For more information, see Third-party login (Passport).

Resetting a user's password

Use the `User.resetPassword()` method to reset a user's password. This method creates a short-lived access token for temporary login that allows users to change passwords if forgotten.

For example, in routes.js (in loopback-example-user-management) below, a route: `/request-password-reset` is created to handle password reset request:

/server/boot/routes.js

```
//send an email with instructions to reset an existing user's password
  app.post('/request-password-reset', function(req, res, next) {
    User.resetPassword({
      email: req.body.email
    }, function(err) {
      if (err) return res.status(401).send(err);
      res.render('response', {
        title: 'Password reset requested',
        content: 'Check your email for further instructions',
        redirectTo: '/',
        redirectToLinkText: 'Log in'
      });
    });
  });
...
```

You must the handle the `'resetPasswordRequest'` event to send a reset email containing the short-lived access token, generated by `resertPassword()` method, to the correct user (see example below):

> The example below assumes you have setup a `User` model and `Mail` datasource.

/common/models/user.js

```
...
 //send password reset link when password reset requested
  user.on('resetPasswordRequest', function(info) {
    var url = 'http://' + config.host + ':' + config.port + '/reset-password';
    var html = 'Click <a href="' + url + '?access_token=' +
        info.accessToken.id + '">here</a> to reset your password';
    //'here' in above html is linked to :
'http://<host:port>/reset-password?access_token=<short-lived/temporary access token>'
    user.app.models.Email.send({
      to: info.email,
      from: info.email,
      subject: 'Password reset',
      html: html
    }, function(err) {
      if (err) return console.log('> error sending password reset email');
      console.log('> sending password reset email to:', info.email);
    });
  });
...
```

And when the user follows a link to reset password, temporary access token is used to find the user and update password using `updateAttribute()` method as follows:

/server/boot/routes.js

```
...
   User.findById(req.accessToken.userId, function(err, user) {
      if (err) return res.sendStatus(404);
      user.updateAttribute('password', req.body.password, function(err, user) {
      if (err) return res.sendStatus(404);
        console.log('> password reset processed successfully');
        res.render('response', {
          title: 'Password reset success',
          content: 'Your password has been reset successfully',
          redirectTo: '/',
          redirectToLinkText: 'Log in'
        });
      });
    });
...
```

For a complete example, see routes.js in loopback-example-user-management.

Over REST, use the `POST /users/reset` endpoint. It returns `200 OK` for a successful request. See User REST API for more information.

Partitioning users with realms

See also:

- Authentication, authorization, and permissions
- Third-party login (Passport)

By default, the LoopBack User model manages all users in a global namespace. It does not isolate different applications. In some cases, you may want to partition users into multiple namespaces so that different applications have separate users. LoopBack uses *realms* to support:

- Users and applications belonging to a single global realm (or no realm).
- Distributing users and applications to multiple realms. A user or application can belong to only one realm. Each realm can have many users and many applications.
- Each application is a unique realm and each user belongs to an application (via a realm).

Each application or user instance still has a unique ID across realms. When an application/user is signed up, it can be assigned to a realm. The `User.login()` function:

- Honors the realm property from the user credential.
- Allows the realm to be extracted from the prefix of username/email.

Two settings in the User model control the realms:

- `realmRequired` (Boolean): true | false (default)
- `realmDelimiter` (string): If configured, the email or username can be prefixed as *<realm><realmDelimiter><username or email>*, for example, myRealm:john or myRealm:john@sample.com. If not present, no prefix will be checked against username or email.

For example,

server/model-config.json

```
"User": {
  "dataSource": "db",
  "options": {
    "realmRequired": true,
    "realmDelimiter": ":"
  }
},
```

When realms are enabled, you must provide a `realm` property when you call `User.create()`, for example:

```
User.create({
  realm: 'myRealm',
  username: 'john',
  email: 'john@sample.com',
  password: 'my-password'
}, callback);
```

To login a user within a realm, the credentials should include the realm property too.

```
User.login({
  realm: 'myRealm',
  username: 'john',
  password: 'my-password'
}, callback);
```

If the realmDelimiter is configured (for example, to ":"), the login allows the realm to be passed in as prefix to the username or email.

```
User.login({
  username: 'myRealm:john',
  password: 'my-password'
}, callback);
```

Authentication, authorization, and permissions

Prerequisites

- Install StrongLoop software.
- Read LoopBack core concepts first.
- Follow Getting started with LoopBack for a basic introduction to LoopBack.

Related articles:

See also:

- Managing users
- Third-party login (Passport) (Facebook, Google, etc.)
- Access control models
- Tutorial: access control
- Security considerations

Most applications need to control who (or what) can access data or call services. Typically, this involves requiring users to login to access protected data, or requiring authorization tokens for other applications to access protected data.

For a simple example of implementing LoopBack access control, see the GitHub loopback-example-access-control repository.

LoopBack apps access data through models (see Defining models), so controlling access to data means putting restrictions on models; that is, specifying who or what can read/write the data or execute methods on the models.

Access control concepts

LoopBack's access control system is built around a few core concepts.

Term	Description	Responsibility	Example

Principal	An entity that can be identified or authenticated.	Represents identities of a request to protected resources.	▪ A user ▪ An application ▪ A role (please note a role is also a principal)

Role	A group of principals with the same permissions.	Organizes principals into groups so they can be used.	▪ Dynamic role:
			  ▪ $everyone (for all users)
			  ▪ $unauthenticated (unauthenticated users)
			  ▪ $owner (the principal is owner of the model instance)
			▪ Static role: admin (a defined role for administrators)

RoleMapping	Assign principals to roles	Statically assigns principals to roles.	▪ Assign user with id 1 to role 1 ▪ Assign role 'admin' to role 1
ACL	Access control list	Controls if a principal can perform a certain operation against a model.	▪ Deny everyone to access the project model ▪ Allow 'admin' role to execute find() method on the project model

General process

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

1. **Specify user roles**. Define the user roles that your application requires. For example, you might create roles for anonymous users, authorized users, and administrators.
2. **Define access for each role and model method**. For example, you might enable anonymous users to read a list of banks, but not allow them to do anything else.
   LoopBack models have a set of built-in methods, and each method maps to either the READ or WRITE access type. In essence, this step amounts to specifying whether access is allowed for each role and each Model + access type, as illustrated in the example below.
3. **Implement authentication**: in the application, add code to create (register) new users, login users (get and use authentication tokens), and logout users.

Exposing and hiding models, methods, and endpoints

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

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

Hiding methods and REST endpoints

If you don't want to expose certain CRUD operations, you can easily hide them by calling `disableRemoteMethod()` on the model. For example, following the previous example, by convention custom model code would go in the file `common/models/location.js`. You would add the following lines to "hide" one of the predefined remote methods:

common/models/location.js

```
var isStatic = true;
MyModel.disableRemoteMethod('deleteById', isStatic);
```

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

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

common/models/location.js

```
var isStatic = false;
MyModel.disableRemoteMethod('updateAttributes', isStatic);
```

Be sure to call `disableRemoteMethod()` on your own custom model, not one of the built-in models; in the example below, for instance, the calls are `MyUser.disableRemoteMethod()` *not* `User.disableRemoteMethod()`.

Here's an example of hiding all methods of the `MyUser` model, except for `login` and `logout`:

```
MyUser.disableRemoteMethod("create", true);
MyUser.disableRemoteMethod("upsert", true);
MyUser.disableRemoteMethod("updateAll", true);
MyUser.disableRemoteMethod("updateAttributes", false);

MyUser.disableRemoteMethod("find", true);
MyUser.disableRemoteMethod("findById", true);
MyUser.disableRemoteMethod("findOne", true);

MyUser.disableRemoteMethod("deleteById", true);

MyUser.disableRemoteMethod("confirm", true);
MyUser.disableRemoteMethod("count", true);
MyUser.disableRemoteMethod("exists", true);
MyUser.disableRemoteMethod("resetPassword", true);

MyUser.disableRemoteMethod('__count__accessTokens', false);
MyUser.disableRemoteMethod('__create__accessTokens', false);
MyUser.disableRemoteMethod('__delete__accessTokens', false);
MyUser.disableRemoteMethod('__destroyById__accessTokens', false);
MyUser.disableRemoteMethod('__findById__accessTokens', false);
MyUser.disableRemoteMethod('__get__accessTokens', false);
MyUser.disableRemoteMethod('__updateById__accessTokens', false);
```

Read-Only endpoints example

You may want to only expose read-only operations on your model hiding all POST, PUT, DELETE verbs

common/models/model.js

```
Product.disableRemoteMethod('create', true);    // Removes (POST) /products
Product.disableRemoteMethod('upsert', true);    // Removes (PUT) /products
Product.disableRemoteMethod('deleteById', true);   // Removes (DELETE) /products/:id
Product.disableRemoteMethod("updateAll", true);    // Removes (POST) /products/update
Product.disableRemoteMethod("updateAttributes", false);  // Removes (PUT)
/products/:id
Product.disableRemoteMethod('createChangeStream', true); // removes (GET|POST)
/products/change-stream
```

Hiding endpoints for related models

To disable a REST endpoints for related model methods, use disableRemoteMethod().

For more information, see Accessing related models.

For example, if there are post and tag models, where a post hasMany tags, add the following code to `/common/models/post.js` to disable the remote methods for the related model and the corresponding REST endpoints:

common/models/model.js

```
module.exports = function(Post) {
  Post.disableRemoteMethod('__get__tags', false);
  Post.disableRemoteMethod('__create__tags', false);
  Post.disableRemoteMethod('__destroyById__accessTokens', false); // DELETE
  Post.disableRemoteMethod('__updateById__accessTokens', false); // PUT
};
```

Hiding properties

To hide a property of a model exposed over REST, define a hidden property. See Model definition JSON file (Hidden properties).

Introduction to User model authentication

LoopBack provides a full-featured solution for authentication and authorization. Follow the steps here to get an overview of how it works with the built-in User model using StrongLoop API Explorer.

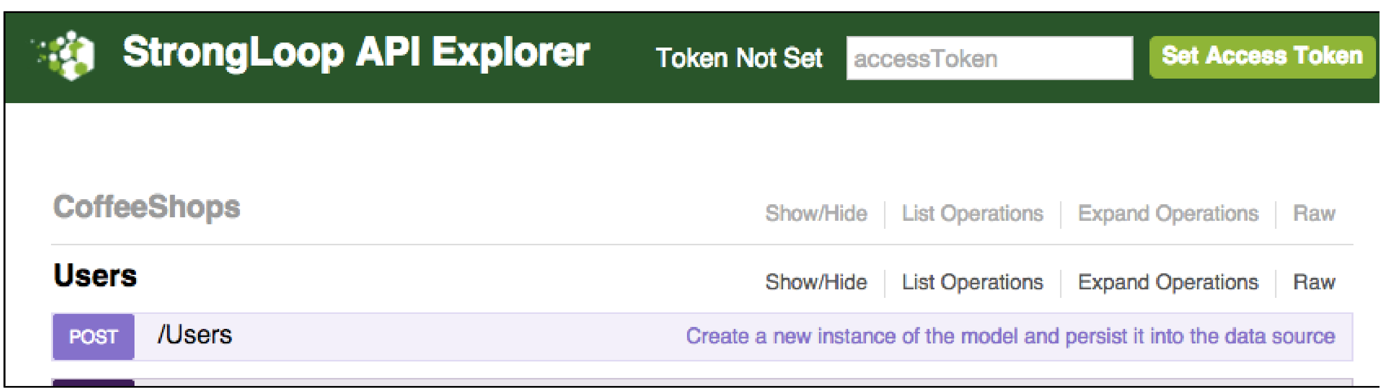
First, if you haven't done so, follow the first steps in Getting started with LoopBack to download the loopback-getting-started application and run it.

Open http://localhost:3000/explorer to view StrongLoop API Explorer. Then:

- Create a new user
- Login as the new user
- Set access token

Create a new user

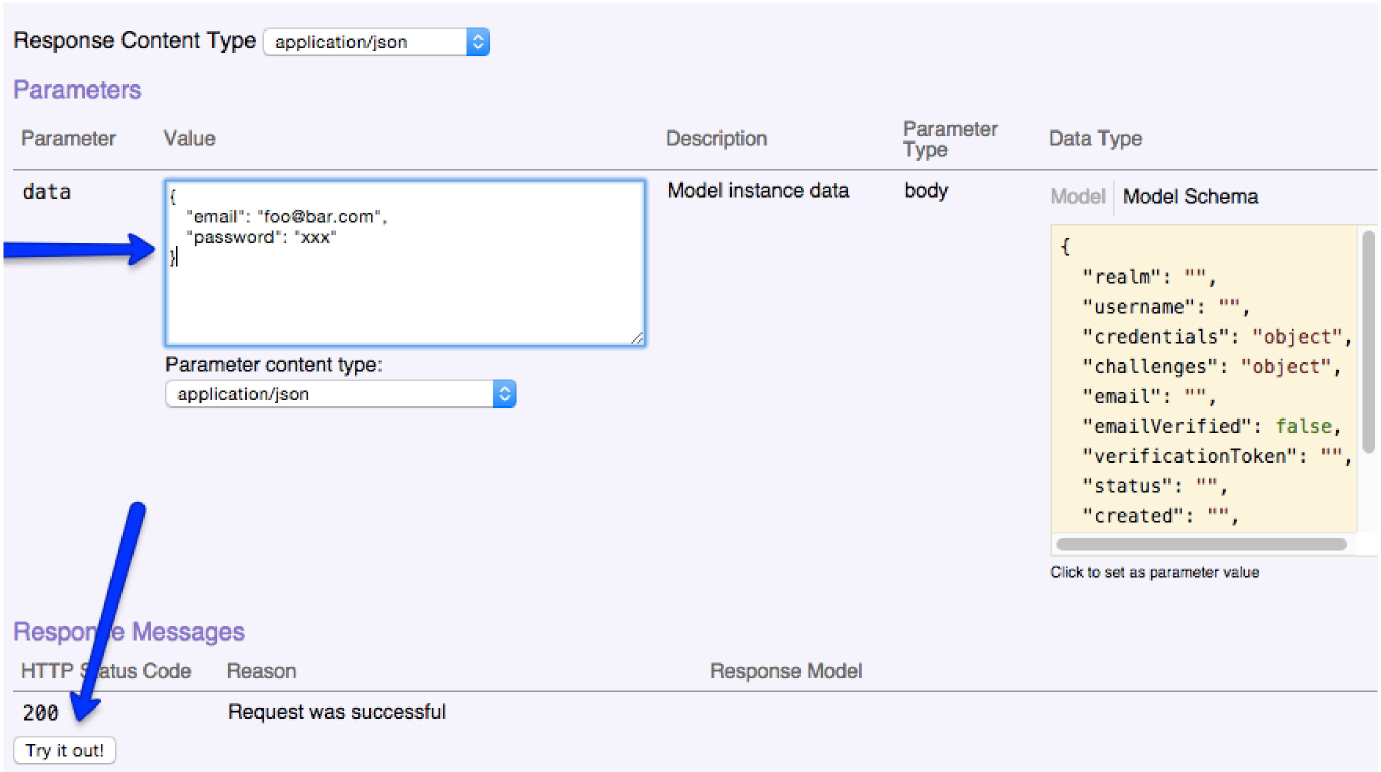
Click on **POST /Users** to create a new user record:



The operation will expand. Under **Parameters**, click on the **data** field and enter a JSON object with email and password properties, for example:

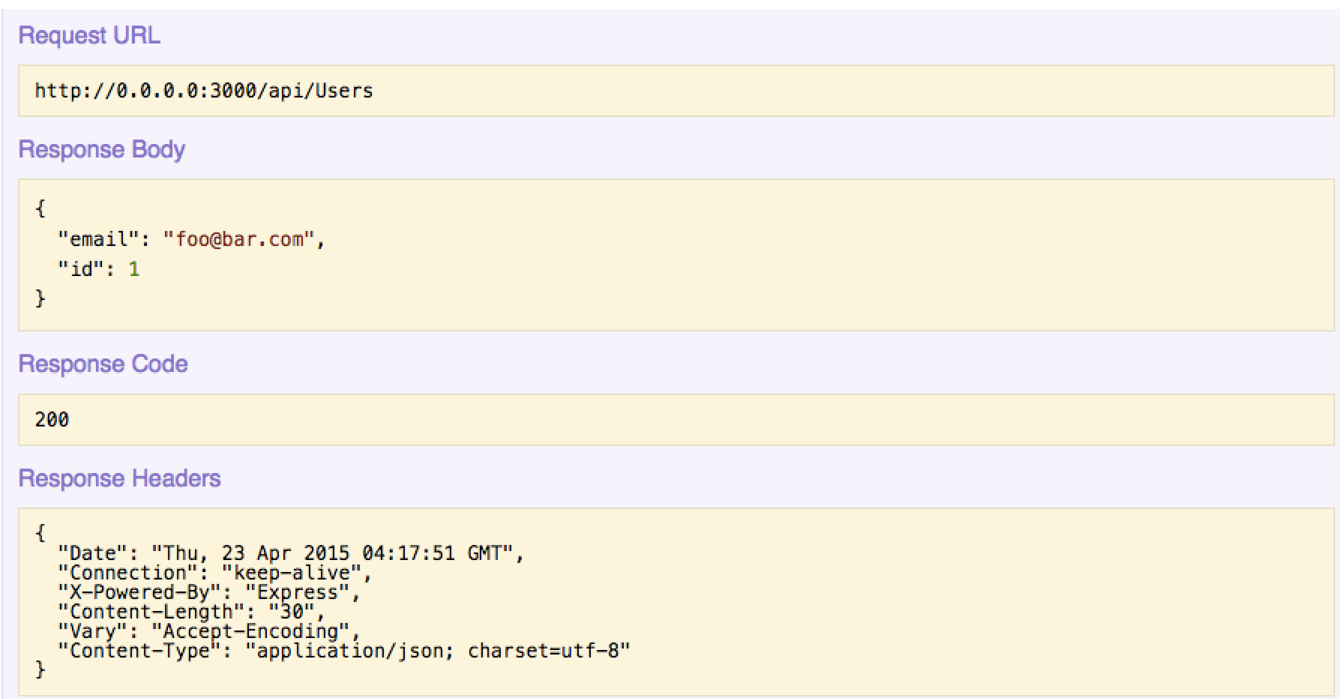
```
{
   "email": "foo@bar.com",
   "password": "xxx"
}
```

The basic User model validates that email has the standard format of an email address, and the password is not empty.



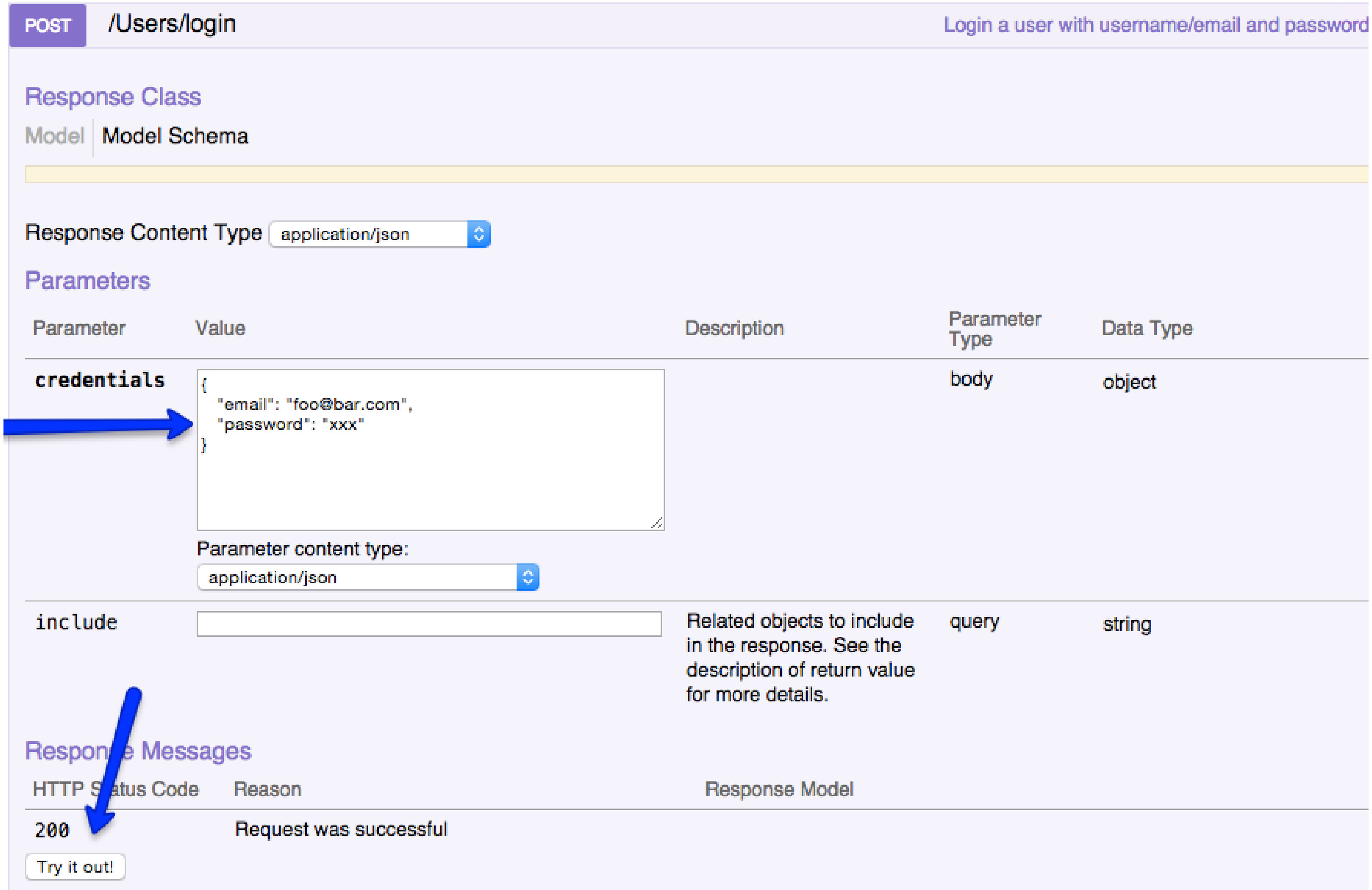
Click **Try it Out!** to submit the request to the REST API.

You'll see a 200 **Response Code**, and the **Response Body** will show the email address and unique ID of the newly-registered user.



Login as the new user

Now click on **POST /Users/login** to login as this new user. Copy and past the same JSON containing the email and password you entered previously to the **credentials** field, then click **Try it Out!**:



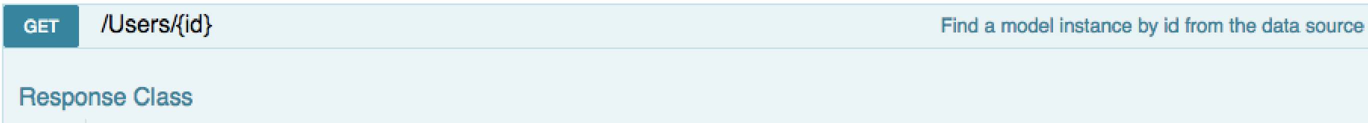
Now the response will contain the authorization token for the user:

Set access token

Select and copy the value of the **id** property in the **Response Body** and paste it into the **accessToken** field in the upper right of the API Explorer window:

After you do this, the message will change to **Token Set**. At this point, the user you created is now logged in and authenticated to the application.

You can now execute certain REST operations; for example, click **GET /Users/{id},** enter 1 in the id field, and click **Try it Out!** to fetch the user model instance data for your own user record:



Certain operations are restricted, even if you are authenticated. For example, you can't view other users' records.

Controlling data access

Related articles:

Specifying user roles

The first step in specifying user roles is to determine what roles your application needs. Most applications will have un-authenticated or anonymous users (those who have not logged in) and authenticated users (those who have logged in). Additionally, many applications will have an administrative role that provides broad access rights. And applications can have any number of additional user roles as appropriate.

For example, the startkicker app consists of four types of users: `guest`, `owner`, `team member` and `administrator`. Each user type has access to various parts of the app based on their role and the access control lists (ACLs) we define.

User access types

LoopBack provides a built-in User model with a corresponding REST API that inherits all the "CRUD" (create, read, update, and delete) methods of the PersistedModel object. Each CRUD method of the LoopBack User model maps to either the READ or WRITE access type, as follows:

READ:

- exists - Boolean method that determines whether a user exists.
- findById - Find a user by ID.
- find - Find all users that match specified conditions.
- findOne - Finds a single user instance that matches specified conditions.
- count - Returns the number of users that match the specified conditions.

WRITE:

- create - create a new user.
- updateAttributes (update) - update a user record.
- upsert (update or insert) - update or insert a new user record.
- destroyById (equivalent to removeById or deleteById) - delete the user with the specified ID.

For other methods, the default access type is EXECUTE; for example, a custom method maps to the EXECUTE access type.

Defining access control

Use the ACL generator to set up access control for an application. Before you do that, though, you must have a clear idea of how you're going to configure access control for your application.

For example, here is how loopback-example-access-control sets up users and their rights:

- Guest - Guest
    - Role = $everyone, $unauthenticated
    - Has access to the "List projects" function, but none of the others
- John - Project owner
    - Role = $everyone, $authenticated, teamMember, $owner
    - Can access all functions except "View all projects"
- Jane - Project team member
    - Role = $everyone, $authenticated, teamMember
    - Can access all functions except "View all projects" and "Withdraw"
- Bob - Administator
    - Role = $everyone, $authenticated, admin
    - Can access all functions except "Withdraw"

Once you've created this kind of specification, you can easily construct `slc loopback:acl` commands to set up access control, as illustrated below.

Using the ACL generator to define access control

The easiest way to define access control for an app is with the ACL generator. This enables you to create a static definition before runtime. The generator prompts you for all the necessary information:

```
$ slc loopback:acl
```

Example

For example, here are the answers to prompts to define ACL entries for the loopback-example-access-control example.

Deny access to all project REST endpoints

- Select the model to apply the ACL entry to: All existing models
- Select the ACL scope: All methods and properties
- Select the access type: All (match all types)
- Select the role: All users
- Select the permission to apply: Explicitly deny access

Allow unrestricted access to GET /api/projects/listProjects

- Select the model to apply the ACL entry to: project
- Select the ACL scope: A single method
- Enter the method name: listProjects
- Select the access type: Execute
- Select the role: All users
- Select the permission to apply: Explicitly grant access

Only allow admin unrestricted access to GET /api/projects

- Select the model to apply the ACL entry to: project
- Select the ACL scope: A single method
- Enter the method name: find
- Select the access type: Read
- Select the role: other
- Enter the role name: admin
- Select the permission to apply: Explicitly grant access

Only allow team members access to GET /api/projects/:id

- Select the model to apply the ACL entry to: project
- Select the ACL scope: A single method
- Enter the method name: findById
- Select the access type: Read
- Select the role: other
- Enter the role name: teamMember
- Select the permission to apply: Explicitly grant access

Allow authenticated users to access POST /api/projects/donate

- Select the model to apply the ACL entry to: project
- Select the ACL scope: A single method
- Enter the method name: donate
- Select the access type: Execute
- Select the role: Any authenticated user
- Select the permission to apply: Explicitly grant access

Allow owners access to POST /api/projects/withdraw

- Select the model to apply the ACL entry to: project
- Select the ACL scope: A single method
- Enter the method name: withdraw
- Select the access type: Execute
- Select the role: The user owning the object
- Select the permission to apply: Explicitly grant access

For more information, see ACL generator.

Applying access control rules

Each incoming request is mapped to an object with three attributes:

- model - The target model name, for example '**order**'
- property - The target method name, for example, '**find'**. You can also specify an array of method names to apply the same constraint to all of them.
- accessType - The access type, '**EXECUTE'**, '**READ'**, and '**WRITE'**

ACL rules are described as an array of objects, each of which consists of attributes listed at Model definition JSON file#ACLs.

1. model
2. property
3. accessType
4. principalType
    a. USER
    b. APP
    c. ROLE
        i. custom roles
        ii. $owner
        iii. $authenticated
        iv. $unauthenticated
        v. $everyone
5. permission
    a. DENY
    b. ALLOW

ACL rule precedence

A single model may have several ACLs applied to it: The ACL of the base model (or models) and that of the model itself, defined in the model definition JSON file. LoopBack determines the ultimate ACL by *adding* all the applicable ACLs with precedence rules for permission and access type to resolve any conflicts.

Permission precedence is applied in this order:

1. DENY
2. ALLOW
3. DEFAULT

So, for example, a DENY rule for a certain operation and user group will take precedence over an ALLOW rule for the same operation and group.

Access type precedence (in order of specificity) is applied in this order:

1. Type (read, write, replicate, update)
2. Method name
3. Wildcard

In general, a more specific rule will take precedence over a more general rule. For example, a rule that denies access to an operation to authenticated users will take precedence over a rule that denies access to all users.

LoopBack sorts multiple rules by the specifics of matching the request against each rule. It calculates the specifics by checking the access request against each ACL rule by the hierarchical order of attributes.

At each level, the matching yields three points:

- 3: exact match
- 2: wildcard match ('*')
- -1: no match

Higher-level matches take precedence over lower-level matches. For example, the exact match at model level will overweigh the wildcard match.

For example, consider the following access request:

```
{
  model: 'order',
  property: 'find',
  accessType: 'EXECUTE'
}
```

Assuming the following ACL rules are defined:

```
[
  // Rule #1
  {
    model: '*',
    property: 'find',
    accessType: 'EXECUTE',
    principalType: 'ROLE',
    principalId: '$authenticated',
    permission: 'ALLOW'
  },
  // Rule #2
  {
    model: 'order',
    property: '*',
    accessType: '*',
    principalType: 'ROLE',
    principalId: '$authenticated',
    permission: 'ALLOW'
  },
  // Rule #3
  {
    model: 'order',
    property: 'find',
    accessType: '*',
    principalType: 'ROLE',
    principalId: '$authenticated',
    permission: 'DENY'
  }
]
```

The order of ACL rules will be #3, #2, #1. As a result, the request will be rejected as the permission set by rule #3 is 'DENY' .

Debugging

Specify a `DEBUG` environment variable with value `loopback:security:*` for the console to log the lookups and checks the server makes as requests come in, useful to understand things from its perspective. Do this in your test environment as there may be quite a lot of output.

Making authenticated requests

- Making authenticated requests with access tokens
- Using current user id as a literal in URLs for REST
- Deleting access tokens

Related articles:

The basic process for an application to create and authenticate users is:

1. Register a new user with the `User.create()` method, inherited from the `PersistedModel` object. See Registering users for details.
2. Call `User.login()` to request an access token from the client application on behalf of the user. See Logging in users for details.
3. Invoke an API using the access token. Provide the access token in the HTTP header or as a query parameter to the REST API call, as illustrated below.

Making authenticated requests with access tokens

Once a user is logged in, LoopBack creates a new AccessToken referencing 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`.

shell

```
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
```

To use cookies for authentication, add the following to server.js (before boot):

/server/server.js

```
app.use(loopback.token({ model: app.models.accessToken }));
```

The Loopback Angular SDK doesn't support using cookies, and expects you to be using an access token returned from `User.login()`.

Using current user id as a literal in URLs for REST

To allow the current logged in user id for REST APIs, configure the token middleware with currentUserLiteral options.

/server/server.js

```
app.use(loopback.token({ model: app.models.accessToken, currentUserLiteral: 'me' }));
```

The *currentUserLiteral* defines a special token that can be used in the URL for REST APIs, for example:

```
curl -X GET http://localhost:3000/api/users/me/orders?access_token=$ACCESS_TOKEN
```

Please note the URL will be rewritten to http://localhost:3000/api/users/<currentLoggedInUserId>/orders?access_token=$ACCESS_TOKEN by LoopBack.

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.

/server/boot/script.js

```
var USER_ID = 1;
var ACCESS_TOKEN = '6Nb2ti5QEXIoDBS5FQGWIz4poRFiBCMMYJbYXSGHWuulOuy0GTEuGx2VCEVvbpBK';
// remove just the token
var token = new AccessToken({id: ACCESS_TOKEN});
token.destroy();
// remove all user tokens
AccessToken.destroyAll({
  where: {userId: USER_ID}
});
```

Defining and using roles

LoopBack enables you to define both static and dynamic roles. Static roles are stored in a data source and are mapped to users. In contrast, dynamic roles aren't assigned to users and are determined during access.

Related articles:

- Static roles
- Dynamic roles

Static roles

Here is an example defining a new static role and assigning a user to that role.

/server/boot/script.js

```
User.create([
    {username: 'John', email: 'john@doe.com', password: 'opensesame'},
    {username: 'Jane', email: 'jane@doe.com', password: 'opensesame'},
    {username: 'Bob', email: 'bob@projects.com', password: 'opensesame'}
  ], function(err, users) {
    if (err) return cb(err);

    //create the admin role
    Role.create({
      name: 'admin'
    }, function(err, role) {
      if (err) cb(err);

      //make bob an admin
      role.principals.create({
        principalType: RoleMapping.USER,
        principalId: users[2].id
      }, function(err, principal) {
        cb(err);
      });
    });
  });
```

Now you can use the role defined above in the access controls. For example, add the following to common/models/project.json to enable users in the "admin" role to call all REST APIs.

/common/models/model.json

```
{
  "accessType": "EXECUTE",
  "principalType": "ROLE",
  "principalId": "admin",
  "permission": "ALLOW",
  "property": "find"
}
```

Dynamic roles

Sometimes static roles aren't flexible enough. LoopBack also enables you to define *dynamic roles* that are defined at run-time.

LoopBack provides the following built-in dynamic roles.

Role object property	String value	Description
Role.OWNER	$owner	Owner of the object
Role.AUTHENTICATED	$authenticated	authenticated user
Role.UNAUTHENTICATED	$unauthenticated	Unauthenticated user
Role.EVERYONE	$everyone	Everyone

The first example used the "$owner" dynamic role to allow access to the owner of the requested project model.

> To qualify a $owner, the target model needs to have a belongsTo relation to the User model (or a model extends from User) and property matching the foreign key of the target model instance. The check for $owner is only performed for a remote method that has ':id' on the path, for example, GET /api/users/:id.

Use `Role.registerResolver()` to set up a custom role handler in a boot script. This function takes two parameters:

1. String name of the role in question.
2. Function that determines if a principal is in the specified role. The function signature must be `function(role, context, callback)`.

For example, here is the role resolver from loopback-example-access-control:

/server/boot/script.js

```
module.exports = function(app) {
  var Role = app.models.Role;
  Role.registerResolver('teamMember', function(role, context, cb) {
    function reject(err) {
      if(err) {
        return cb(err);
      }
      cb(null, false);
    }
    if (context.modelName !== 'project') {
      // the target model is not project
      return reject();
    }
    var userId = context.accessToken.userId;
    if (!userId) {
      return reject(); // do not allow anonymous users
    }
    // check if userId is in team table for the given project id
    context.model.findById(context.modelId, function(err, project) {
      if(err || !project) {
        reject(err);
      }
      var Team = app.models.Team;
      Team.count({
        ownerId: project.ownerId,
        memberId: userId
      }, function(err, count) {
        if (err) {
          return reject(err);
        }
        cb(null, count > 0); // true = is a team member
      });
    });
  });
};
```

Using the dynamic role defined above, we can restrict access of project information to users that are team members of the project.

/common/models/model.json

```
{
  "accessType": "READ",
  "principalType": "ROLE",
  "principalId": "teamMember",
  "permission": "ALLOW",
  "property": "findById"
}
```

Accessing related models

By default, all related model methods have a DENY ALL ACL set. You need to explicitly grant access. ACLs do not inherit from the target model's endpoint.

So, for example, even if the books model's default ACL is `ALLOW $authenticated` for `GET /books`, the route `GET /user/{id}/books` default will still be DENY ALL.

Related articles:

Restricting access to related models

When two models have a relationship between them (see Creating model relations), LoopBack automatically creates a set of *related model methods* corresponding to the API routes defined for the relationship.

In the following list, *modelName* is the name of the related model and *modelNamePlural* is the plural form of the related model name.

In the method names below, the separators are *double* underscores, __.

belongsTo:

- __get__*relatedModelName*

hasOne:

- __create__*relatedModelName*
- __get__*relatedModelName*
- __update__*relatedModelName*
- __destroy__*relatedModelName*

hasMany:

- __count__*relatedModelNamePlural*
- __create__*relatedModelNamePlural*
- __delete__*relatedModelNamePlural*
- __destroyById__*relatedModelNamePlural*
- __findById__*relatedModelNamePlural*
- __get__*relatedModelNamePlural*
- __updateById__*relatedModelNamePlural*

hasManyThrough:

- __count__*relatedModelNamePlural*
- __create__*relatedModelNamePlural*
- __delete__*relatedModelNamePlural*
- __destroyById__*relatedModelNamePlural*
- __exists__*relatedModelNamePlural* (through only)
- __findById__*relatedModelNamePlural*
- __get__*relatedModelNamePlural*
- __link__*relatedModelNamePlural* (through only)
- __updateById__*relatedModelNamePlural*
- __unlink__*relatedModelNamePlural* (through only)

You can use these related model methods to control access to the related routes.

By default, access to related model methods is denied. To call these methods, you must set the `permission` property to "ALLOW" as described below.

For example, if a **User hasMany projects**, LoopBack creates these routes (among others) and the corresponding related model methods:

- `/api/users/count` - standard method is `count`
- `/api/users/:id/projects` - related model method is `__get__projects`
- `/api/users/:id/projects/count` - related model method is `__count__projects`

To configure access control to such routes, set the permission on the related model methods in the model definition JSON file. For example, the

ACL for the User model definition JSON file (`user.json`) for these routes might look like this, for example:

/common/models/user.json

```
"acls": [{
 "principalType": "ROLE",
 "principalId": "$authenticated",
 "permission": "ALLOW",
 "property": "count"
},
{
 "principalType": "ROLE",
 "principalId": "$owner",
 "permission": "ALLOW",
 "property": "__get__projects"
},
{
 "principalType": "ROLE",
 "principalId": "$authenticated",
 "permission": "ALLOW",
 "property": "__count__projects"
}]
```

Querying related models

This feature requires LoopBack 2.16.0 or later.

When querying a model, you may also want to return data from its related models.

For example, suppose you have three models: `User`, `Report`, and `LineItem`, where:

- A user can have many reports; that is, there is a HasMany relation between `User` and `Report` (User hasMany Report).
- A report can have many line items; that is, there is a HasMany relation between `Report` and `Lineitem` (Report hasMany LineItem).

Additionally, the ReportModel is configured with the following ACLs so that authenticated users can create new records and users can update their own records:

```
[
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    },
    {
      "principalType": "ROLE",
      "principalId": "$owner",
      "permission": "ALLOW",
      "property": "findById"
    },
    ...
]
```

Assume the LineItem model has the same ACL defined.

Now, suppose you want to fetch a model owned by your user and also get at its related models. Here is how you do it with `findById()` using the Node API:

```
Report.findById({
  id:1,
  filter:{ include:'lineitems' }
});
```

Using the REST API:

```
GET /api/Reports/110?filter={"include":["lineItems"]}
```

Example results:

```
{
  "name": "january report - bob",
  "id": 110,
  "userId": 100,
  "lineItemModels": [
    {
      "name": "lunch",
      "id": 111,
      "reportModelId": 110
    },
    {
      "name": "dinner",
      "id": 112,
      "reportModelId": 110
    }
  ]
}
```

Creating a default admin user

LoopBack does not define a default administrator user, however you can define one when the application starts, as illustrated in the loopback-example-access-control example. Specifically, the example includes code in `server/boot/sample-models.js` that:

Related articles:

- Creates several users, along with instances of other models
- Defines relations among the models.
- Defines an admin role,
- Adds a role mapping to assign one of the users to the admin role.

Because this script is in `server/boot`, it is executed when the application starts up, so the admin user will always exist once the app initializes.

The following code creates three users named "John," "Jane," and "Bob, then (skipping the code that creates projects, project owners, and project team members) defines an "admin" role, and makes Bob an admin:

/server/boot/script.js

```
User.create([
    {username: 'John', email: 'john@doe.com', password: 'opensesame'},
    {username: 'Jane', email: 'jane@doe.com', password: 'opensesame'},
    {username: 'Bob', email: 'bob@projects.com', password: 'opensesame'}
], function(err, users) {
    if (err) return debug('%j', err);
    ...
    // Create projects, assign project owners and project team members
    ...
    // Create the admin role
    Role.create({
      name: 'admin'
    }, function(err, role) {
      if (err) return debug(err);
      debug(role);

      // Make Bob an admin
      role.principals.create({
        principalType: RoleMapping.USER,
        principalId: users[2].id
      }, function(err, principal) {
        if (err) return debug(err);
        debug(principal);
      });
    });
  });
};
```

The project model JSON (created by running `slc loopback:acl`, the ACL generator) file specifies that the admin role has unrestricted access to view projects (`GET /api/projects`):

/common/models/model.json

```
...
    {
      "accessType": "READ",
      "principalType": "ROLE",
      "principalId": "admin",
      "permission": "ALLOW",
      "property": "find"
    },
...
```

Security considerations

- Model REST APIs
  - Hiding properties
  - Disabling API Explorer
- CORS
- Mitigating XSS exploits

Related articles:

Model REST APIs

By default, LoopBack models you create expose a standard set of HTTP endpoints for create, read, update, and delete (CRUD) operations. The `public` property in model-config.json specifies whether to expose the model's REST APIs, for example:

/server/model-config.json

```
...
  "MyModel": {
    "public": true,
    "dataSource": "db"
  },
...
```

To "hide" the model's REST API, simply change `public` to `false`.

Hiding properties

To hide a property of a model exposed over REST, define a hidden property. See Model definition JSON file (Hidden properties).

Disabling API Explorer

LoopBack API Explorer is great when you're developing your application, but for security reasons you may not want to expose it in production.

For an application using loopback-component-explorer, to disable explorer in production:

- Set the NODE_ENV environment variable to "production".
- Then in `server/component-config.production.json`:

server/component-config.production.json

```
{
  "loopback-component-explorer": false
}
```

For an application using the old `loopback-explorer` (prior to version 2.0), disable API Explorer by deleting or renaming `server/boot/explorer.js`.

CORS

By default LoopBack enables Cross-origin resource sharing (CORS) using the cors package. Change the CORS settings in middleware.json.

If you are using a JavaScript client, you must also enable CORS on the client side. For example, one way to enable it with AngularJS is:

/client/app.js

```
var myApp = angular.module('myApp', [
    'myAppApiService']);

myApp.config(['$httpProvider', function($httpProvider) {
        $httpProvider.defaults.useXDomain = true;
        delete $httpProvider.defaults.headers.common['X-Requested-With'];
    }
]);
```

Mitigating XSS exploits

LoopBack stores the user's access token in a JavaScript object, which may make it susceptible to a cross-site scripting (XSS) security exploit. As a best practice to mitigate such threats, use appropriate Express middleware, for example:

- Lusca
- Helmet

See also:

- Express 3.x csrf() function.
- Cookies vs Tokens. Getting auth right with Angular.JS

Tutorial: access control

This article is reproduced from loopback-example-access-control

loopback-example-access-control

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

In this example, we create “Startkicker” (a basic Kickstarter-like application) to demonstrate authentication and authorization mechanisms in LoopBack. The application consists of four types of users:

- `guest`
- `owner`
- `team member`
- `administrator`

Each user type has permission to perform tasks based on their role and the application’s ACL (access control list) entries.

Prerequisites

Tutorials

- Getting started with LoopBack
- Tutorial series - step 1

- Tutorial series - step 2
- Tutorial series - step 3

Knowledge

- EJS
- body-parser
- JSON
- LoopBack models
- LoopBack adding application logic

Procedure

Create the application

Application information

- Name: `loopback-example-access-control`
- Directory to contain the project: `loopback-example-access-control`

```
$ slc loopback loopback-example-access-control
... # follow the prompts
$ cd loopback-example-access-control
```

Add the models

Model information

- Name: `user`
- Datasource: `db (memory)`
- Base class: `User`
- Expose via REST: `No`
- Custom plural form: *Leave blank*
- Properties
  - *None*
- Name: `team`
- Datasource: `db (memory)`
- Base class: `PersistedModel`
- Expose via REST: `No`
- Custom plural form: *Leave blank*
- Properties
  - `ownerId`
  - Number
  - Not required
  - `memberId`
  - Number
  - Required
- Name: `project`
- Datasource: `db (memory)`
- Base class: `PersistedModel`
- Expose via REST: `Yes`
- Custom plural form: *Leave blank*
- Properties
  - `name`
  - String
  - Not required
  - `balance`
  - Number
  - Not required

No properties are required for the `user` model because we inherit them from the built-in `User` model by specifying it as the base class.

```
$ slc loopback:model user
... # follow the prompts, repeat for `team` and `project`
```

Define the remote methods

Define three remote methods in `project.js`:

- `listProjects`
- `donate`
- `withdraw`

Create the model relations

Model relation information

- `user`
- has many
  - `project`
  - Property name for the relation: `projects`
  - Custom foreign key: `ownerId`
  - Require a through model: No
  - `team`
  - Property name for the relation: `teams`
  - Custom foreign key: `ownerId`
  - Require a through model: No
- `team`
- has many
  - `user`
  - Property name for the relation: `members`
  - Custom foreign key: `memberId`
  - Require a through model: No
- `project`
- belongs to
  - `user`
  - Property name for the relation: `user`
  - Custom foreign key: `ownerId`

Add model instances

Create a boot script named `sample-models.js`.

This script does the following:

- Creates 3 users (`John`, `Jane`, and `Bob`)
- Creates project 1, sets `John` as the owner, and adds `John` and `Jane` as team members
- Creates project 2, sets `Jane` as the owner and solo team member
- Creates a role named `admin` and adds a role mapping to make `Bob` an `admin`

Configure server-side views

> *LoopBack comes preconfigured with EJS out-of-box. This means we can use server-side templating by simply setting the proper view engine and a directory to store the views.*

Create a `views` directory to store server-side templates.

```
$ mkdir server/views
```

Add server-side templating configurations to `server.js`.

Create `index.ejs` in the views directory.

Configure `server.js` to use server-side
templating. Remember to import the `path` package.

Add routes

Create `routes.js`. This script does the following:

- Sets the `GET /` route to render `index.ejs`
- Sets the `GET /projects` route to render `projects.ejs`
- Sets the `POST /projects` route to to render `projects.ejs` when credentials are valid and renders `index.ejs` when credentials are invalid
- Sets the `GET /logout` route to log the user out

> *When you log in sucessfully, `projects.html` is rendered with the authenticated user's access token embedded into each link.*

Create the views

Create the `views` directory to store views.

In this directory, create `index.ejs` and `projects.ejs`.

Create a role resolver

Create `role-resolver.js`.

> *This file checks if the context relates to the project model and if the request maps to a user. If these two requirements are not met, the request is denied. Otherwise, we check to see if the user is a team member and process the request accordingly.*

Create ACL entries

> *ACLs are used to restrict access to application REST endpoints.*

ACL information

- Deny access to all project REST endpoints
- Select the model to apply the ACL entry to: `(all existing models)`
- Select the ACL scope: `All methods and properties`
- Select the access type: `All (match all types)`
- Select the role: `All users`
- Select the permission to apply: `Explicitly deny access`
- Allow unrestricted access to `GET /api/projects/listProjects`
- Select the model to apply the ACL entry to: `project`
- Select the ACL scope: `A single method`
- Enter the method name: `listProjects`
- Select the role: `All users`
- Select the permission to apply: `Explicitly grant access`
- Only allow admin unrestricted access to `GET /api/projects`
- Select the model to apply the ACL entry to: `project`
- Select the ACL scope: `A single method`
- Enter the method name: `find`
- Select the role: `other`
- Enter the role name: `admin`

- Select the permission to apply: `Explicitly grant access`
- Only allow team members access to `GET /api/projects/:id`
- Select the model to apply the ACL entry to: `project`
- Select the ACL scope: `A single method`
- Enter the method name: `findById`
- Select the role: `other`
- Enter the role name: `teamMember`
- Select the permission to apply: `Explicitly grant access`
- Allow authenticated users to access `POST /api/projects/donate`
- Select the model to apply the ACL entry to: `project`
- Select the ACL scope: `A single method`
- Enter the method name: `donate`
- Select the role: `Any authenticated user`
- Select the permission to apply: `Explicitly grant access`
- Allow owners access to `POST /api/projects/withdraw`
- Select the model to apply the ACL entry to: `project`
- Select the ACL scope: `A single method`
- Enter the method name: `withdraw`
- Select the role: `The user owning the object`
- Select the permission to apply: `Explicitly grant access`

```
$ slc loopback:acl
# follow the prompts, repeat for each ACL listed above
```

Try the application

Start the server (`node .`) and open `localhost:3000` in your browser to view the app. You will see logins and explanations related to each user type we created:

- Guest `Guest`
- Role = $everyone, $unauthenticated
- Has access to the “List projects” function, but none of the others
- John `Project owner`
- Role = $everyone, $authenticated, teamMember, $owner
- Can access all functions except “View all projects”
- Jane `Project team member`
- Role = $everyone, $authenticated, teamMember
- Can access all functions except “View all projects” and “Withdraw”
- Bob `Administator`
- Role = $everyone, $authenticated, admin
- Can access all functions except “Withdraw”

- Next tutorial
- All tutorials

Advanced topics: access control

Manually enabling access control

If you created your app with `slc loopback`, then you don't need to do anything to enable access control.

Otherwise, if you're adding access control manually, you must call the LoopBack `enableAuth()` method, for example:

/server/server.js

```
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.

/server/boot/script.js

```
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 (LDL).

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.

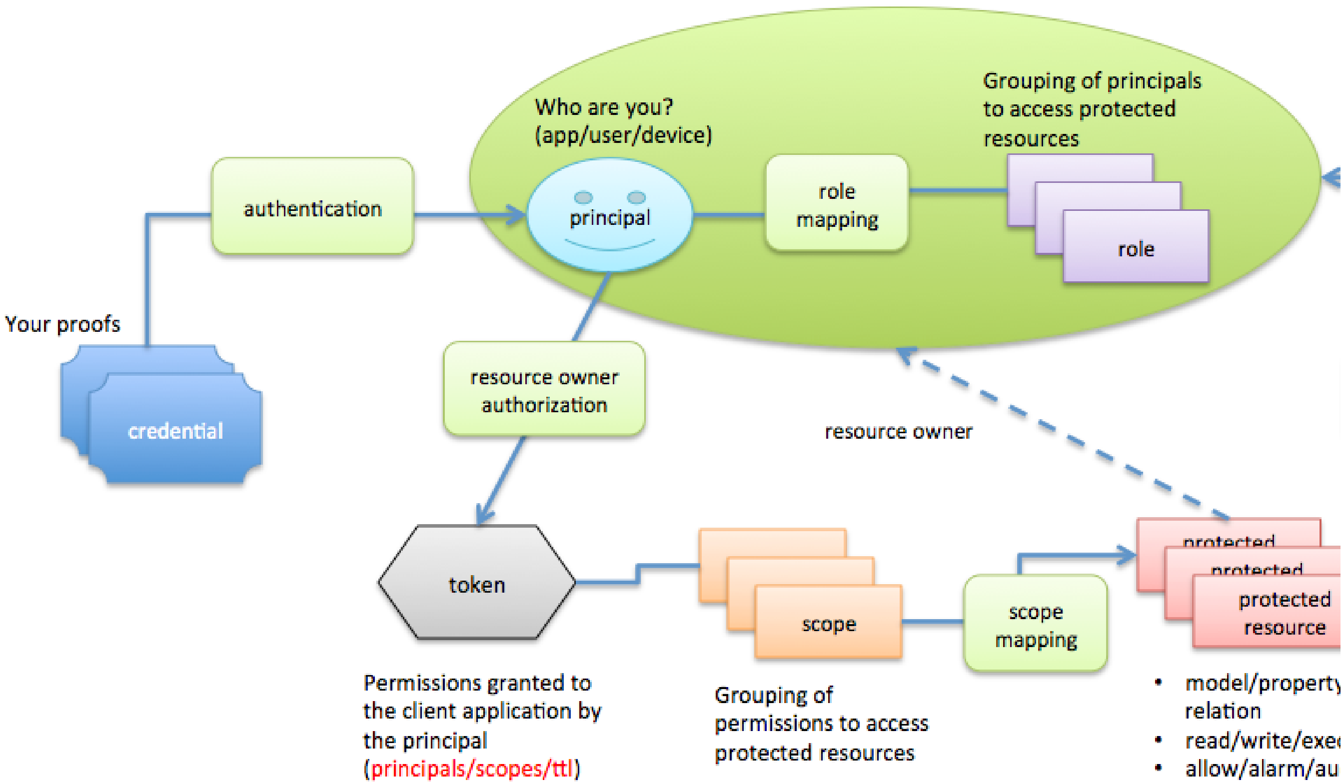
/server/boot/script.js

```
ACL.create({principalType: ACL.USER, principalId: 'u001', model: 'User', property:
ACL.ALL,
    accessType: ACL.ALL, permission: ACL.ALLOW}, function (err, acl) {...});

ACL.create({principalType: ACL.USER, principalId: 'u001', model: 'User', property:
ACL.ALL,
 accessType: ACL.READ, permission: ACL.DENY}, function (err, acl) {...});
```

See Using built-in models for more information.

Architecture

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



Defining models

Prerequisites

- Install StrongLoop software.
- Read LoopBack core concepts first.
- Follow Getting started with LoopBack for a basic introduction to LoopBack.

A *LoopBack model* represents data in backend systems such as databases, and by default has both Node and REST APIs. Additionally, you can add functionality such as validation rules and business logic to models.

Every LoopBack application has a set of predefined built-in models such as User, Role, and Application. You can extend built-in models to suit your application's needs.

Additionally, you can define your own custom models specific to your application:

- Use the `slc loopback:model` model generator to create cusom models from scratch. This creates a Model definition JSON file that defines your model in LoopBack.
- Use `Datasource.buildModelFromInstance()` to create *dynamic* schema-less models for data sources such as SOAP and REST services. See Creating models from unstructured data for more information.
- For data sources backed by a relational database, a model typically corresponds to a table. Use model discovery to create *static*, schema-driven models for database-backed data sources.   See Discovering models from relational databases for more information.

Creating models

Overview

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

- With the model generator, `slc loopback:model`.
- From an existing relational database using *model discovery*. Then you can keep your model synchronized with the database using LoopBack's schema / model synchronization API.
- By instance introspection for free-form data in NoSQL databases or REST APIs.

All three of these methods create a Model definition JSON file that defines your model in LoopBack, by convention in a LoopBack project's `common/models` directory; for example, `common/models/Account.json`.

You can also create and customize models programmatically using the LoopBack API, or by manually editing the Model definition JSON file. In most cases, you shouldn't need to use those techniques to create models, but generally will to customize models for your use.

Getting a reference to a model in JavaScript

The way that you get a reference (or "handle") to a model in JavaScript code depends on where the code is.

In model JavaScript file

In the model JavaScript file (for example) the models is passed into the top-level function, so the model object is available directly; for example:

/common/models/model.js

```
module.exports = function(Customer) {
  Customer.create( ... );  // Customer object is available
  ...
```

Promises
LoopBack also supports Promises in addition to callbacks for CRUD methods of a model and its related models.

In a boot script

In a boot script, use the `app.models` object to get a reference to any model; for example:

/server/boot/script.js

```
module.exports = function(app) {
  var User = app.models.user;
  var Role = app.models.Role;
  var RoleMapping = app.models.RoleMapping;
  var Team = app.models.Team;
  ...
```

Using the model generator

- Overview
- Basic use
    - Adding default values

If you already have a back-end schema (like a database), create models based on it using LoopBack's discovery feature. See Discovering models from relational databases.

Overview

The easiest way to create a new model is with `slc loopback:model`, the model generator. When creating a new model, the generator will prompt you for the properties in the model. Subsequently, you can add more properties to it using the Property generator.

When you create a model (for example, called "myModel"), the tool:

- Creates `/common/models/myModel.json`, the Model definition JSON file.
- Creates `/common/models/myModel.js`, where you can extend the model programmatically; for example to add remote methods. See Adding application logic for more information.
- Adds an entry to `/server/model-config.json` for the model, specifying the model's data source. See model-config.json for more information.

Once you've created your model, you may want to read:

- Customizing models
- Attaching models to data sources
- Exposing models over REST

Basic use

Use the LoopBack model generator to create a new model.   In your application root directory, enter the command (for example, to create a "books" model):

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

Then `slc` will prompt you to:

- Choose the data source to which the model will connect. By default, there will be only the in-memory data source (named "db"). When you create additional data sources with slc loopback:datasource, the data source generator, they will be listed as options.
- Choose the model's base class, from a list of built-in models classes and existing custom models in the application.

    In general, use `PersistedModel` as the base model when you want to store your data in a database using a connector such as MySQL or MongoDB. Use `Model` as the base for models that don't have CRUD semantics, for example, using connectors such as SOAP and REST.

- Choose whether to expose the model over REST; the default is yes.
- Enter a custom plural form; the default is to use standard plural (for example, "books").
- Add properties to the model; for each property it will prompt you for:

- Name of the property
- Type of the property; see LoopBack types. This sets the `type` property in the model definition JSON file.
- Whether the property is required. This sets the `required` property in the model definition JSON file.

Adding properties

After you create a model, you can add more properties with the property generator.

```
$ slc loopback:property
```

Then `slc` will prompt you to choose the model to which you want to add the property, along with the other property settings (as before). Then, `slc` will modify the model definition JSON file accordingly.

Adding default values

One way to set a default value for a property is to set the "default" property in the models JSON file. You can also set the defaultFn property to set the default value to a globally-unique identifer (GUID) or the timestamp.

For more information, see the model JSON file General property properties section.

Discovering models from relational databases

Overview

LoopBack makes it simple to create models from an existing relational database. This process is called *discovery* and is supported by the following connectors:

- MySQL connector
- PostgreSQL connector
- Oracle connector
- SQL Server connector

For NoSQL databases such as MongoDB, use instance introspection instead.

Data sources connected to relational databases automatically get the asychronous Database discovery API.

> The StrongLoop Arc graphical tool enables you to perform discovery without coding. See Discovering models from a database for more information.

Basic procedure

Follow these basic steps:

1. Use a script such as that below to discover the schema.
2. Use `fs.writeFile()` to save the output in `common/models/model-name.json`.
3. Add the new model entry to `server/model-config.json`.
4. Run the app:

    ```
    $  node .
    ```

5. Use LoopBack Explorer to verify the schema is defined properly.

Example discovery

For example, consider an Oracle database. First, the code sets up the Oracle data source. Then the call to `discoverAndBuildModels()` creates models from the database tables. Calling it with the `associations: true` option makes the discovery follow primary/foreign key relations.

/server/bin/script.js

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

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

Additional discovery functions

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

/server/bin/script.js

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

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

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

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

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

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

Database discovery API

See also:

- Discovering models from relational databases
- loopback-datasource-juggler API reference

Overview

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

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

Synchronous methods

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

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

Note there are performance implications in using synchronous methods.

Methods

In general, *schema/owner* is the name of the table schema. It's a namespace that contains a list of tables. Each database uses slightly different terminology:

- MySQL: databases and schemas are exactly the same thing.
- Oracle: the schema is the user/owner.
- PostgreSQL: a database contains one or more named schemas, which in turn contain tables. The schema defaults to "public".
- MS SQL Server: the schema defaults to "dbo".

discoverAndBuildModels

Discover and build models from the specified owner/modelName.

```
dataSource.discoverAndBuildModels(modelName [, options] [, cb])
```

Arguments

Name	Type	Description
modelName	`String`	The model name.
[options]	`Object`	Options; see below.
[cb]	`Function`	The callback function

Options

Name	Type	Description
owner / schema	`String`	Database owner or schema name.
relations	`Boolean`	True if relations (primary key/foreign key) are navigated; false otherwise.
all	`Boolean`	True if all owners are included; false otherwise.
views	`Boolean`	True if views are included; false otherwise.

discoverModelDefinitions

Call `discoverModelDefinitions()` to discover model definitions (table or collection names), based on tables or collections in a data source. This method returns list of table/view names.

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

Parameters

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

Options

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

views	Boolean	If true, include views.

Example of callback function return value:

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

Example

For example:

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

discoverModelProperties

Call `discoverModelProperties()` to discover metadata on columns (properties) of a database table. This method returns column information for a given table/view.

discoverModelProperties(*table*, *options*, *cb*)

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

Example return value of callback function:

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

discoverPrimaryKeys

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

discoverPrimaryKeys(*table*, *options*, *cb*)

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

Example return value of callback function:

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

discoverForeignKeys

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

discoverForeignKeys(*table*, *options*, *cb*)

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

Example return value of callback function:

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

discoverExportedForeignKeys

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

discoverExportedForeignKeys(*table*, *options*, *cb*)

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

Example return value of callback function:

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

discoverSchemas

Use `discoverSchema` to discover LDL models from a database. Starting with one table/view, if the `relations` option is set to true, it follows foreign keys to discover related models.

> discoverSchema(*modelName* [, *options*] [, *cb*])

Properties of options parameter:

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

Options

Name	Type	Description
owner / schema	`String`	Database owner or schema name.
relations	`Boolean`	If true, the function will follow foreign key relations to discover related tables.
all	`Boolean`	True to include all owners; false otherwise.
views	`Boolean`	True to include views; false otherwise.

Example

/server/script.js

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

The result is shown below.

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

/common/models/model.json

```
{
  "name":"Inventory",
  "options":{
    "idInjection":false,
    "mysql":{
      "schema":"STRONGLOOP",
      "table":"INVENTORY"
    }
  },
  "properties":{
    "productId":{
      "type":"String",
      "required":false,
      "length":60,
      "precision":null,
      "scale":null,
      "id":1,
      "mysql":{
        "columnName":"PRODUCT_ID",
        "dataType":"varchar",
        "dataLength":60,
        "dataPrecision":null,
        "dataScale":null,
        "nullable":"NO"
      }
    },
    "locationId":{
      "type":"String",
      "required":false,
      "length":60,
      "precision":null,
      "scale":null,
      "id":2,
      "mysql":{
        "columnName":"LOCATION_ID",
        "dataType":"varchar",
        "dataLength":60,
        "dataPrecision":null,
        "dataScale":null,
        "nullable":"NO"
      }
    },
    "available":{
      "type":"Number",
      "required":false,
      "length":null,
      "precision":10,
      "scale":0,
      "mysql":{
        "columnName":"AVAILABLE",
        "dataType":"int",
        "dataLength":null,
        "dataPrecision":10,
```

```
      "dataScale":0,
      "nullable":"YES"
    }
  },
  "total":{
    "type":"Number",
    "required":false,
    "length":null,
    "precision":10,
    "scale":0,
    "mysql":{
      "columnName":"TOTAL",
      "dataType":"int",
      "dataLength":null,
      "dataPrecision":10,
      "dataScale":0,
      "nullable":"YES"
    }
```

```
    }
  }
}
```

Example of building models via discovery

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

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

/server/script.js

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

Creating models from unstructured data

For unstructured data such as that in NoSQL databases and REST services, you can create models using *instance introspection*. Instance introspection creates a model from a single model instance using `buildModelFromInstance()`.

The following data sources support instance introspection:

- MongoDB data sources
- REST data sources
- SOAP data sources

For example:

/server/boot/script.js

```
module.exports = function(app) {
  var db = app.dataSources.db;

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

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

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

  console.log(obj.toObject());

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

Customizing models

Once you've created a model with the model generator (`slc loopback:model`), you can start customizing it.   You can customize it using slc, by editing the Model definition JSON file, and by adding JavaScript code.

Customizing a model with slc

Once you've created a model with the model generator (`slc loopback:model`), you can't modify the model with the model generator . However, you can customize the model to some degree with other `slc loopback` generators; see below.

You can use `slc` to customize a model after you initially create it; specifically, you can:

- Use `slc loopback:property` to add a property to the model. See Property generator for more information.
- Use `slc loopback:relation` to add add relations beteween models. See Relation generator for more information.
- Use slc loopback:acl to add access control to the model. See ACL generator for more information.

Customizing a model using JSON

You can customize a number of aspects of a model by simply editing the model definition JSON file in `common/models` (for example, `customer.json`), which by default looks like this:

common/models/model.json

```
{
  "name": "myModel",
  "base": "PersistedModel",
  "properties": {
     // Properties listed here depend on your responses to the CLI
  },
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

LoopBack *adds* the settings in the the model JSON file to those of the base model. In most cases, this is straightforward, but for ACL settings there can be complex interactions since some ACL settings take precedence over others. For more information, see ACL rule precedence for more information.

Extending another model

You can make a model extend or "inherit from" an existing model, either one of the built-in models such as User, or a custom model you've defined in your application. To do this with the model generator (`slc loopback:model` command), simply choose the desired model when you're prompted to "Select model's base class." Alternatively, you can edit the Model definition JSON file and set the "base" property to the name of the model you want to extend.

In general, use `PersistedModel` as the base model when you want to store your data in a database using a connector such as MySQL or MongoDB. Use `Model` as the base for models that don't have CRUD semantics, for example, using connectors such as SOAP and REST.

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

/common/models/model.json

```
{

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

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

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

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

/common/models/model.json

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

You can add new properties when you extend a model, for example:

/common/models/model.json

```
{
   "name": "Customer",
   "base": "User",
   "properties": {
      "favoriteMovie": {
        "type": "string"
      }
   }
}
```

See LoopBack types for information on data types supported.

Customizing other model settings

Here are some of the most important settings you can customize:

- **plural** - set to a custom string value to use, instead of the default standard plural form.
- **strict** - set to true to make the model save only instances that have the predefined set of properties. Any additional properties in a save or update operation are not persisted to the data source. False by default.
- **idInjection** - Whether to automatically add an id property to the model. True by default.
- **http.path** - customized HTTP path of REST endpoints.

See Model definition JSON file for more information.

Customizing a model with JavaScript code

The basic way to extend a model programmatically is to edit the model's JavaScript file in the `common/models/` directory. For example, a "customer" model will have a `common/models/customer.js` file (if you create the model using `slc loopback:model`, the model generator). The script is executed immediately after the model is defined. Treat the script as part of the model definition; use it for model configuration and registration. You could also add model relationships, complex validations, or default functions for certain properties: Basically, anything you cannot do in JSON. However, note that at this point the script doesn't have access to the app instance.

You can also extend a model by adding a remote method or a model hook.

If you don't want to expose the method over REST, then just omit the `remoteMethod()` call.

See Adding application logic for more information on customizing a model using JavaScript. See LoopBack types for information on data types supported.

Change the implementation of built-in methods

Via server boot script

When you attach a model to a persistent data source, it becomes a *persisted model* that extends PersistedModel, and LoopBack automatically adds a set of built-in methods for CRUD operations. In some cases, you might want to change the implementation; use a JavaScript file in the `/server/boot` directory to do this. For example, the following code shows how to reimplement `Note.find()` to override the built-in find() method.

/server/boot/script.js

```
module.exports = function(app) {
  var Note = app.models.Note;
  var find = Note.find;
  var cache = {};

  Note.find = function(filter, cb) {
    var key = '';
    if(filter) {
      key = JSON.stringify(filter);
    }
    var cachedResults = cache[key];
    if(cachedResults) {
      console.log('serving from cache');
      process.nextTick(function() {
        cb(null, cachedResults);
      });
    } else {
      console.log('serving from db');
      find.call(Note, function(err, results) {
        if(!err) {
          cache[key] = results;
        }
        cb(err, results);
      });;
    }
  }
}
```

Via your model's script

Use a JavaScript file in the `common/models` directory to do this

common/models/MyModel.js

```
module.exports = function(MyModel) {
  MyModel.on('dataSourceAttached', function(obj){
    var find = MyModel.find;
    MyModel.find = function(filter, cb) {
      return find.apply(this, arguments);
    };
  });
};
```

References:

- https://github.com/strongloop/loopback/issues/443
- https://github.com/strongloop/loopback-datasource-juggler/issues/427
- https://github.com/strongloop/loopback/issues/1077

Attaching models to data sources

Overview

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

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

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

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

Add a data source

To add a new data source, use the Data source generator:

shell

```
$ slc loopback:datasource
```

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

/server/datasources.json

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

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

Add data source credentials

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

/server/datasources.json

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

Make the model use the data source

When you create a new model with the model generator, you can specify the data source you want it to use from among those you've added to the application using the Data source generator and the default `db` data source (that uses the memory connector).

To change the data source a model uses after you've created the model, edit the application's `server/model-config.json` and set the dataSource property for the model. For example, to make myModel use the corp1 data source:

server/model-config.json

```
"myModel": {
    "dataSource": "corp1",
    "public": true
  }
```

By default, the model generator creates models to use the `db` data source.

Exposing models over REST

Overview

LoopBack models automaticaly have a standard set of HTTP endpoints that provide REST APIs for create, read, update, and delete (CRUD) operations on model data. The `public` property in model-config.json specifies whether to expose the model's REST APIs, for example:

/server/model-config.json

```
...
  "MyModel": {
    "public": true,
    "dataSource": "db"
  },
...
```

To "hide" the model's REST API, simply change `public` to `false`.

REST paths

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

- `Model.settings.http.path`
- `plural`, if defined in the Model definition JSON file.
- Automatically-pluralized model name (the default). For example, if you have a location model, by default it is mounted to `/locations`.

Using the REST Router

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

> If your application is scaffolded using `slc loopback`, LoopBack will automatically set up REST middleware and register public models. You don't need to do anything additional.

To manually expose a model over REST with the `loopback.rest` router, use the following code, for example:

/server/server.js

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

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

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

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

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

Request format

For POST and PUT requests, the request body can be JSON, XML or urlencoded format, with the **Content-Type** header set to `application/json, application/xml, or application/x-www-form-urlencoded`. The **Accept** header indicates its preference for the response format.

> Setting the request's **Accept** header to `application/vnd.api+json` will result in the response's **Content-Type** header being automatically set to `application/vnd.api+json` if `application/vnd.api+json` is in the array of supported types. Set the supported types with the `remoting.rest.supportedTypes` property in config.json.

Passing JSON object or array using HTTP query string

Some REST APIs take a JSON object or array from the query string. LoopBack supports two styles to encode the object/array value as query parameters.

- Syntax from node-querystring (qs)
- Stringified JSON

For example,

```
http://localhost:3000/api/users?filter[where][username]=john&filter[where][email]=call
back@strongloop.com
http://localhost:3000/api/users?filter={"where":{"username":"john","email":"callback@s
trongloop.com"}}
```

The table below illustrates how to encode the JSON object/array can be encoded in different styles:

JSON object/array for the filter object	qs style	Stringified JSON
`{ where:` `{ username: 'john', email: 'callback@strongloop.com' } }`	`?filter[where][username]=john &` `filter[where][email]=callback@strongloop.com`	`?filter={"where":` `{"username":"john", "email":"callback@stronglo`
`{ where: { username: {inq: ['john', 'mary']} } }`	`?filter[where][username][inq][0]=john &filter[where][username][inq][1]=mary`	`?filter=` `{"where": {"username":{"inq":["john",`
`{ include: ['a', 'b'] }`	`?filter[include]=a&filter[include]=b`	`?filter={"include":`

Response format

The response format for all requests is typically a JSON object/array or XML in the body and a set of headers. Some responses have an empty body. For example,

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Credentials: true
Content-Type: application/json; charset=utf-8
Content-Length: 59
Vary: Accept-Encoding
Date: Fri, 24 Oct 2014 18:02:34 GMT
Connection: keep-alive

{"title":"MyNote","content":"This is my first note","id":1}
```

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 JSON format:

- message: String error message.
- stack: String stack trace.
- statusCode: Integer HTTP status code.

For example,

```
{
"error": {
    "message": "could not find a model with id 1",
    "stack": "Error: could not find a model with id 1\n ...",
    "statusCode": 404
    }
}
```

Disabling API Explorer

LoopBack API Explorer is great when you're developing your application, but for security reasons you may not want to expose it in production.

For an application using loopback-component-explorer, to disable explorer in production:

- Set the NODE_ENV environment variable to "production".
- Then in `server/component-config.production.json`:

server/component-config.production.json

```
{
  "loopback-component-explorer": false
}
```

> For an application using the old `loopback-explorer` (prior to version 2.0), disable API Explorer by deleting or renaming `server/boot/explorer.js`.

Predefined remote methods

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

As an example, consider a simple model called `Location` (that provides business locations) to illustrate the REST API exposed by LoopBack. LoopBack automatically creates a number of Node methods with corresponding REST endpoints, including:

Model (Node) API	HTTP Method	Example Path
create()	POST	/locations
upsert()	PUT	/locations
exists()	GET	/locations/:id/exists
findById()	GET	/locations/:id
find()	GET	/locations
findOne()	GET	/locations/findOne
destroyById() or deleteById()	DELETE	/locations/:id
count()	GET	/locations/count
prototype.updateAttributes()	PUT	/locations/:id
createChangeStream()	POST	/locations/change-stream

updateAll()	POST	/locations/update

The above table provides a partial list of methods and REST endpoints. See the API documentation for a complete list of all the Node API methods. See PersistedModel REST API for details on the REST API.

Exposing and hiding models, methods, and endpoints

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

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

Hiding methods and REST endpoints

If you don't want to expose certain CRUD operations, you can easily hide them by calling `disableRemoteMethod()` on the model. For example, following the previous example, by convention custom model code would go in the file `common/models/location.js`. You would add the following lines to "hide" one of the predefined remote methods:

common/models/location.js

```
var isStatic = true;
MyModel.disableRemoteMethod('deleteById', isStatic);
```

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

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

common/models/location.js

```
var isStatic = false;
MyModel.disableRemoteMethod('updateAttributes', isStatic);
```

Be sure to call `disableRemoteMethod()` on your own custom model, not one of the built-in models; in the example below, for instance, the calls are `MyUser.disableRemoteMethod()` *not* `User.disableRemoteMethod()`.

Here's an example of hiding all methods of the `MyUser` model, except for `login` and `logout`:

```
MyUser.disableRemoteMethod("create", true);
MyUser.disableRemoteMethod("upsert", true);
MyUser.disableRemoteMethod("updateAll", true);
MyUser.disableRemoteMethod("updateAttributes", false);

MyUser.disableRemoteMethod("find", true);
MyUser.disableRemoteMethod("findById", true);
MyUser.disableRemoteMethod("findOne", true);

MyUser.disableRemoteMethod("deleteById", true);

MyUser.disableRemoteMethod("confirm", true);
MyUser.disableRemoteMethod("count", true);
MyUser.disableRemoteMethod("exists", true);
MyUser.disableRemoteMethod("resetPassword", true);

MyUser.disableRemoteMethod('__count__accessTokens', false);
MyUser.disableRemoteMethod('__create__accessTokens', false);
MyUser.disableRemoteMethod('__delete__accessTokens', false);
MyUser.disableRemoteMethod('__destroyById__accessTokens', false);
MyUser.disableRemoteMethod('__findById__accessTokens', false);
MyUser.disableRemoteMethod('__get__accessTokens', false);
MyUser.disableRemoteMethod('__updateById__accessTokens', false);
```

Read-Only endpoints example

You may want to only expose read-only operations on your model hiding all POST, PUT, DELETE verbs

common/models/model.js

```
Product.disableRemoteMethod('create', true);    // Removes (POST) /products
Product.disableRemoteMethod('upsert', true);    // Removes (PUT) /products
Product.disableRemoteMethod('deleteById', true);   // Removes (DELETE) /products/:id
Product.disableRemoteMethod("updateAll", true);    // Removes (POST) /products/update
Product.disableRemoteMethod("updateAttributes", false);  // Removes (PUT)
/products/:id
Product.disableRemoteMethod('createChangeStream', true); // removes (GET|POST)
/products/change-stream
```

Hiding endpoints for related models

To disable a REST endpoints for related model methods, use disableRemoteMethod().

For more information, see Accessing related models.

For example, if there are post and tag models, where a post hasMany tags, add the following code to `/common/models/post.js` to disable the remote methods for the related model and the corresponding REST endpoints:

common/models/model.js

```
module.exports = function(Post) {
  Post.disableRemoteMethod('__get__tags', false);
  Post.disableRemoteMethod('__create__tags', false);
  Post.disableRemoteMethod('__destroyById__accessTokens', false); // DELETE
  Post.disableRemoteMethod('__updateById__accessTokens', false); // PUT
};
```

Hiding properties

To hide a property of a model exposed over REST, define a hidden property. See Model definition JSON file (Hidden properties).

Validating model data

A *schema* imposes restrictions on the model, to ensure (for example) that the model will save data that matches the corresponding database table.

A model can validate data before passing it on to a data store such as a database to ensure that it conforms to the backend schema.

- Adding a schema to a model
- Using validation methods
- Localizing validation messages

Adding a schema to a model

One way to validate data is to create a model schema; LoopBack will then ensure that data conforms to that schema definition.

For example, suppose your app has a product model. The following code defines a schema and assigns it to the product model. The schema defines two properties: name, a required string property and price, an optional number property.

common/models/product.js

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

If a 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.

Using validation methods

Every model attached to a persistent data source has validations methods mixed in from Validatable.

Method	Description
validatesAbsenceOf	Validate absence of one or more specified properties. A model should not include a property to be considered valid; fails when validated field not blank.
validatesExclusionOf	Validate exclusion. Require a property value not be in the specified array.
validatesFormatOf	Validate format. Require a model to include a property that matches the given format.
validatesInclusionOf	Validate inclusion in set. Require a value for property to be in the specified array.

validatesLengthOf	Validate length. Require a property length to be within a specified range. Three kinds of validations: "min," "max," and "is." Default error messages are: • min: too short • max: too long • is: length is wrong
validatesNumericalityOf	Validate numericality. Requires a value for property to be either an integer or number.
validatesPresenceOf	Validate presence of one or more specified properties. Requires a model to include a property to be considered valid; fails when validated field is blank.
validatesUniquenessOf	Validate uniqueness. Ensure the value of the property is unique for the model. Not available for all connectors. Currently supported with these connectors: • In Memory • Oracle • MongoDB

Use these methods to perform specific data validation; for example:

common/models/user.js

```
module.exports = function(user) {
  user.validatesPresenceOf('name', 'email')
  user.validatesLengthOf('password', {min: 5, message: {min: 'Password is too
short'}});
  user.validatesInclusionOf('gender', {in: ['male', 'female']});
  user.validatesExclusionOf('domain', {in: ['www', 'billing', 'admin']});
  user.validatesNumericalityOf('age', {int: true});
  user.validatesUniquenessOf('email', {message: 'email is not unique'});
})
```

The validation methods are invoked when you call the `isValid()` method on a model instance, and automatically each time model instance is created or updated. You don't have to call `isValid()` to validate data.

To enforce validation constraints when calling `upsert()`, ensure that `validateUpsert` option is set to `true` in the model definition JSON file. By default, the model generator sets this property to true.

To invoke the validation constraints explicitly, call `isValid()`; for example:

```
user.isValid(function (valid) {
    if (!valid) {
        user.errors // hash of errors {attr: [errmessage, errmessage, ...], attr: ...}

    }
```

Another example of defining validation constraints, this time using a regular expresson:

common/models/user.js

```
var re =
/^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]
{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;

UserModel.validatesFormatOf('email', {with: re, message: 'Must provide a valid
email'});
if (!(UserModel.settings.realmRequired || UserModel.settings.realmDelimiter)) {
    UserModel.validatesUniquenessOf('email', {message: 'Email already exists'});
    UserModel.validatesUniquenessOf('username', {message: 'User already exists'});
}
```

To add validation to model for creating a new model instance, you *do not* need to call `isValid()`. You can add validation by simply adding the validator calls:

common/models/MyModel.js

```
module.exports = function(MyModel) {
  MyModel.validatesLengthOf('name', { min: 5, message: { min: 'Name should be 5+
characters' } });
  ...
};
```

Use `isValid()` as an additional *ad-hoc* way to check validity. You can also call `validate()` or `validateAsync()` with custom validation functions.

Localizing validation messages

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

Here is an example error response:

error.details.codes

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

Creating model relations

Related articles

- Creating models
- Customizing models
- Creating model relations
- Querying data
- Model definition JSON file
- PersistedModel REST API

Overview of model relations

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

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

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

You can define the following relations between models:

- BelongsTo relations
- HasOne relations
- HasMany relations
- HasManyThrough relations
- HasAndBelongsToMany relations
- Polymorphic relations
- Embedded relations (embedsOne and embedsMany)

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

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

Using the relation generator

The easiest way to create a new relation between existing models is to use the relation generator, `slc loopback:relation`. It will prompt you to enter the type of relation (belongsTo, hasMany, and so on) and the affected models.

> The name of the relation must be different than a property it references.

Relation options

There are three options for most relation types:

- Scope
- Properties
- Custom scope methods

Scope

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

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

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

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

For example:

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

Properties

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

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

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

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

To accomplish the same thing with a callback function:

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

invertProperties

Normally, `properties` are transferred from parent to child, but there are cases where it makes sense to do the opposite. To enable this, use the `invertProperties` option. See an example in Embedded models (embed with belongsTo).

Custom scope methods

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

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

For example:

```
var reorderFn = function(ids, cb) {
  console.log(this.name); // `this` refers to the RelationDefinition  - `images`
(relation name)
  // Do some reordering here & save cb(null, [3, 2, 1]); };
  // Manually declare remoting params
  reorderFn.shared = true;
  reorderFn.accepts = { arg: 'ids', type: 'array', http: { source: 'body' } };
  reorderFn.returns = { arg: 'ids', type: 'array', root: true };
  reorderFn.http = { verb: 'put', path: '/images/reorder' };
  Product.hasMany(Image, { scopeMethods: { reorder: reorderFn } });
}
```

Exposing REST APIs for related models

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

/server/script.js

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

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

Tutorial: model relations

Follow this tutorial to create a web application (loopback-example-relations) that demonstrates LoopBack model relations. The application's main page consists of links to query and filter data through an exposed REST API.

Related articles:

- Create application
- Create models
- Create the front-end
- Add sample data
- Create model relations
- Try the API

Create application

Begin by scaffolding the application with `slc loopback`:

```
$ slc loopback
```

You'll see:

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

Enter **loopback-example-relation** as the project name (referred to as the *project root* henceforth). Finish the creation process by following the prompts.

Create models

You'll use an in-memory database to hold data. Create a model named `Customer` as follows:

```
$ cd loopback-example-relation
$ slc loopback:model Customer
```

You'll see:

```
[?] Enter the model name: Customer
[?] Select the data-source to attach Customer to: db (memory)
[?] Expose Customer via the REST API? Yes
[?] Custom plural form (used to build REST URL):
Let's add some Customer properties now.

Enter an empty property name when done.
[?] Property name: name
   invoke   loopback:property
[?] Property type: string
[?] Required? No

Let's add another Customer property.
Enter an empty property name when done.
[?] Property name: age
   invoke   loopback:property
[?] Property type: number
[?] Required? No

Let's add another Customer property.
Enter an empty property name when done.
[?] Property name: #leave blank, press enter
```

Follow the prompts to finish creating the model. Repeat for `Review` and `Order` using the following properties:

- Review
  - product: String
  - star: Number
- Order
  - description: String
  - total: Number

> You'll see new files `customer.json`, `order.json`, and `review.json` in `/common/models` when you're done.

Create the front-end

Now create a front-end to make it easier to analyze the data. Install Embedded JavaScript (EJS), by running the following command from the project root:

```
$ npm install --save ejs
```

Then configure the application view engine by modifying `server/server.js` as follows:

```
...
// -- Mount static files here--
...
app.set('view engine', 'html');
app.engine('html', require('ejs').renderFile);
app.set('json spaces', 2); //pretty print results for easier viewing later
...
```

Next, modify `server/boot/root.js` as follows:

```
module.exports = function(server) {
  var router = server.loopback.Router();
  router.get('/', function(req, res) {
    res.render('index');
  });
  server.use(router);
};
```

Finally, create the `views` directory by running:

```
$ mkdir -p server/views
```

Inside the `views` directory, create `index.html` with the following contents:

Expand source

```
<DOCTYPE html>
<html>
  <head>
    <title>loopback-example-relation</title>
  </head>
  <body>
    <h1>loopback-example-relation</h1>
    <p>
      <a href="/explorer">API Explorer</a>
    </p>
    <h2>API</h2>
    <ul>
      <li><a href='/api/customers'>/api/customers</a>
      <li><a
href='/api/customers?filter[fields][name]=true'>/api/customers?filter[fields][name]=tr
ue</a>
      <li><a href='/api/customers/1'>/api/customers/1</a>
      <li><a href='/api/customers/youngFolks'>/api/customers/youngFolks</a>
      <li><a href='/api/customers/1/reviews'>/api/customers/1/reviews</a>
      <li><a href='/api/customers/1/orders'>/api/customers/1/orders</a>
      <li><a
href='/api/customers?filter[include]=reviews'>/api/customers?filter[include]=reviews</
a>
      <li><a
href='/api/customers?filter[include][reviews]=author'>/api/customers?filter[include][r
eviews]=author</a>
      <li><a
href='/api/customers?filter[include][reviews]=author&filter[where][age]=21'>/api/custo
mers?filter[include][reviews]=author&filter[where][age]=21</a>
      <li><a
href='/api/customers?filter[include][reviews]=author&filter[limit]=2'>/api/customers?f
ilter[include][reviews]=author&filter[limit]=2</a>
      <li><a
href='/api/customers?filter[include]=reviews&filter[include]=orders'>/api/customers?fi
lter[include]=reviews&filter[include]=orders</a>
    </ul>
  </body>
</html>
```

View what you have so far by entering this command:

```
$ cd loopback-example-relation
$ node .
```

Browse to localhost:3000 to see the home page. Then click on API Explorer and you'll see the models you created.

> You may notice some of the API endpoints return empty arrays or errors, because the database is empty. In addition, you need to define model relations for some of the API endpoints to work. Don't fret, you'll get to that very soon!

Add sample data

In `server/boot`, create a script named `sample-customers.js` with the following contents:

server/boot/sample-customers.js

Expand source

```
module.exports = function(app) {
  var Customer = app.models.Customer;
  var Order = app.models.Order;

  // define a custom scope
  Customer.scope('youngFolks', {where: {age: {lte: 22 }}});
  app.dataSources.db.automigrate('Customer', function(err) {
    if (err) throw err;

    var customers = [
      {name: 'Customer A', age: 21},
      {name: 'Customer B', age: 22},
      {name: 'Customer C', age: 23},
      {name: 'Customer D', age: 24},
      {age: 25}
      ];
    var orders = [
      {
        description: 'First order by Customer A',
        date: '01-01-2015'
      },
      {
        description: 'Second order by Customer A',
        date: '02-01-2015'
      },
      {
        description: 'Order by Customer B',
        date: '03-01-2015'
      },
      {
        description: 'Order by Customer C',
        date: '04-01-2015'
      },
      {
        description: 'Order by Anonymous',
        date: '05-01-2015'
      }
    ];

    // Create customers and orders
    Customer.create(customers[0], function(err, instance) {
      if (err) return console.error(err);
      console.log('Customter created: ', instance);
      orders[0].customerId = instance.id;
      orders[1].customerId = instance.id;
      Order.create(orders[0], function(err, instance) {
        if (err) return console.error(err);
        console.log('Order created: ', instance);
      });
      Order.create(orders[1], function(err, instance) {
        if (err) return console.error(err);
        console.log('Order created: ', instance);
      });
    });
    Customer.create(customers[1], function(err, instance) {
      if (err) return console.error(err);
```

```
    console.log('Customter created: ', instance);
    orders[2].customerId = instance.id;
    Order.create(orders[2], function(err, instance) {
      if (err) return console.error(err);
      console.log('Order created: ', instance);
    });
  });
  Customer.create(customers[2], function(err, instance) {
    if (err) return console.error(err);
    console.log('Customter created: ', instance);
    orders[3].customerId = instance.id;
    Order.create(orders[3], function(err, instance) {
      if (err) return console.error(err);
      console.log('Order created: ', instance);
    });
  });
  Customer.create(customers[3], function(err, instance) {
    if (err) return console.error(err);
    console.log('Customter created: ', instance);
    Order.create(orders[4], function(err, instance) {
      if (err) return console.error(err);
      console.log('Order created: ', instance);
    });
```

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

Create four more scripts in `server/boot` to add sample data:

- z-book-people.js
- z-customer-accounts.js
- z-customer-address.js
- z-customer-emails.js

When the application starts, it will automatically run the boot scripts and load the sample data. Note that the file names ensure that these scripts are run last when the application boots.

> `automigrate()` recreates the database table/index if it already exists. In other words, existing tables will be dropped and ALL EXISTING DATA WILL BE LOST. For more information, see Creating a database schema from models. Note also that `Model.scope()` is only in `create-customers.js`.

Create model relations

You're going to create four relations between the models you just created. The relations will describe that:

- A customer has many reviews (Customer *hasMany* Review).
- A customer has many orders (Customer hasMany Order).
- A review belongs to a customer (Review belongsTo Customer).
- An order belongs to a customer (Order belongsTo Customer).

From the project root, enter the command:

```
$ slc loopback:relation
```

Follow the prompts and create the following relationships:

Customer - *hasMany* Review

- property name for the relation: reviews
- custom foreign key: authorId

Customer - *hasMany* Order

- property name for the relation: orders
- custom foreign key: customerId

Review - *belongsTo* Customer

- property name for the relation: author
- custom foreign key: authorId

Order - *belongsTo* Customer

> For any item without *property name for the relation* or *custom foreign key*, just use the defaults. LoopBack will derive these values automatically when you don't specify one.

When you're done, your `common/models/customer.json` should look like:

Expand source

```
{
  "name": "Customer",
  "base": "PersistedModel",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "number"
    }
  },
  "validations": [],
  "relations": {
    "reviews": {
      "type": "hasMany",
      "model": "Review",
      "foreignKey": "authorId"
    },
    "orders": {
      "type": "hasMany",
      "model": "Order",
      "foreignKey": "customerId"
    }
  },
  "acls": [],
  "methods": []
}
```

```
common/models/reviews.json
```

should look like:

Expand source

```
{
  "name": "Review",
  "base": "PersistedModel",
  "properties": {
    "product": {
      "type": "string"
    },
    "star": {
      "type": "number"
    }
  },
  "validations": [],
  "relations": {
    "author": {
      "type": "belongsTo",
      "model": "Customer",
      "foreignKey": "authorId"
    }
  },
  "acls": [],
  "methods": []
}
```

and `common/models/order.json` should look like:

Expand source

```
{
  "name": "Order",
  "base": "PersistedModel",
  "properties": {
    "description": {
      "type": "string"
    },
    "total": {
      "type": "number"
    }
  },
  "validations": [],
  "relations": {
    "customer": {
      "type": "belongsTo",
      "model": "Customer",
      "foreignKey": ""
    }
  },
  "acls": [],
  "methods": []
}
```

Try the API

Restart application and browse to localhost:3000. Try out the API with the models and relations you defined. The following table describes many of the API endpoints.

API Endpoint	Description
/api/customers	List all customers
/api/customers?filter[fields][0]=name	List all customers, but only return the name property for each result
/api/customers/1	Look up a customer by ID
/api/customers/youngFolks	List a predefined scope named *youngFolks*
/api/customers/1/reviews	List all reviews posted by a given customer
/api/customers/1/orders	List all orders placed by a given customer
/api/customers?filter[include]=reviews	List all customers including their reviews
/api/customers?filter[include][reviews]=author	List all customers including their reviews which also include the author
/api/customers?filter[include][reviews]=author&filter[where][age]=21	List all customers whose age is 21, including their reviews which also include the author
/api/customers?filter[include][reviews]=author&filter[limit]=2	List first two customers including their reviews which also include the author
/api/customers?filter[include]=reviews&filter[include]=orders	List all customers including their reviews and orders

BelongsTo relations

- Overview
- Defining a belongsTo relation
- Methods added to the model

Related articles:

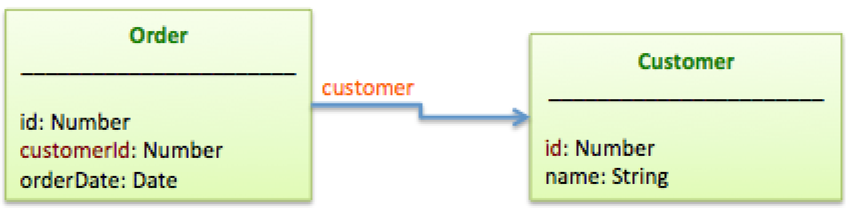
Overview

A `belongsTo` relation sets up either a many-to-one or a one-to-one connection with another model. In a many-to-one relationship each instance of the declaring model "belongs to" at most one instance of the other model, while the target model can have many of the declaring model.

For example, consider an application with customers and orders:

- Each order can be placed by exactly one customer: "Order belongsTo (one) Customer".
- A customer may place many orders: "Customer hasMany Orders" .

This kind of `belongsTo` relation is the logical reflection of a `hasMany` relation.



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

Another kind of `belongsTo` relation is a one-to-one relationship, which is similar to many-to-one, except that each instance of the owning model can have only one instance of the declaring model.

For example, consider an application that includes customers and addresses:

- Each address can be associated with exactly one customer: "Address belongsTo (one) Customer".
- A customer can have only one address: "Customer hasOne Address".

This kind of `belongsTo` relation is the logical reflection of a `hasOne` relation.

Defining a belongsTo relation

Use `slc loopback:relation` to create a relation between two models. The tool will prompt you to enter the name of the model, the name of related model, and other required information. The tool will then modify the Model definition JSON file (for example, `common/models/customer.json`) accordingly.

For more information, see Relation generator.

For example, here is the model JSON file for the order model in loopback-example-relations:

common/models/order.json

```
{
  "name": "Order",
  "base": "PersistedModel",
  ...
  "relations": {
    "customer": {
      "type": "belongsTo",
      "model": "Customer",
      "foreignKey": ""
    }
  },
...
```

Alternatively, you can define a “belongsTo” relation in code, though in general this is not recommended:

common/models/order.js

```
Order.belongsTo(Customer, {foreignKey: ‘customerId’});
```

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

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

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

Methods added to the model

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

Depending on the arguments, the method can be used to get or set the owning model instance. The results of method calls are cached internally and available via later synchronous calls to the method.

Example method	Description

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

`var customer = order.customer();`	Synchronously get the results of a previous get call to customer(...)
`order.customer(customer);`	Set the customer for the order

HasOne relations

- Overview
- Defining a hasOne relation
- Methods added to the model

Related articles:

Overview

A hasOne relation sets up a one-to-one connection with another model, such that each instance of the declaring model "has one" instance of the other model. A hasOne relation is a degenerate case of a hasMany relation.

Defining a hasOne relation

Use `slc loopback:relation` to create a relation between two models. The tool will prompt you to enter the name of the model, the name of related model, and other required information. The tool will then modify the Model definition JSON file (for example, `common/models/customer.json`) accordingly.

For more information, see Relation generator.

For example, consider two models: supplier and account.

common/models/supplier.json

```
{
  "name": "supplier",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "supplier_acct": {
      "type": "hasOne",
      "model": "account",
      "foreignKey": "supplierId"
    }
  },
  "acls": [],
  "methods": []
}
```

A supplier has one account, where the foreign key is on the declaring model: account.supplierId -> supplier.id.

common/models/account.json

```
{
  "name": "account",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "id": {
      "type": "number",
      "required": true
    },
    "acctmgr": {
      "type": "string"
    },
    "supplierId": {
      "type": "number",
      "required": true
    }
  },
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

Alternatively, you can define a “hasOne” relation in code, though in general this is not recommended:

common/models/supplier.js

```
Supplier.hasOne(Account, {foreignKey: 'supplierId', as: 'account'});
```

If the target model doesn’t have a foreign key property, LoopBack will add a property with the same name. The type of the property will be the same as the type of the target model’s **id** property. Please note the foreign key property is defined on the target model (for example, Account).

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

- Relation name: Camel case of the model name, for example, for the "supplier" model the relation is "supplier".
- Foreign key: The relation name appended with ‘Id’, for example, for relation name "supplier" the default foreign key is "supplierId".

Methods added to the model

Once you define the hasOne relation, LoopBack automatically adds a method with the relation name to the declaring model class’s prototype, for example: `supplier.prototype.account(...)`.

Example method	Description

`supplier.account(function(err, account) {`
`...`
`});`
	Find the supplier's account model.

`var supplier = supplier.account.build(data);`
 	
Or equivalently:
 	
`var account = new account({supplierId: supplier.id, ...});`
	Build a new account for the supplier with the supplierId to be set to the id of the supplier. No persistence is involved.

`supplier.account.create(data, function(err, account) {`
`...`
`});`
Or, equivalently:
`account.create({supplierId: supplier.id, ...}, function(err, account) {`
`...`
`});`
	Create a new account for the supplier. If there is already an account, an error will be reported.

`supplier.account.destroy(function(err) {`
`...`
`});`
	Remove the account for the supplier.

`supplier.account.update({balance: 100}, function(err, account) {`
`...`
`});`
	Update the associated account.

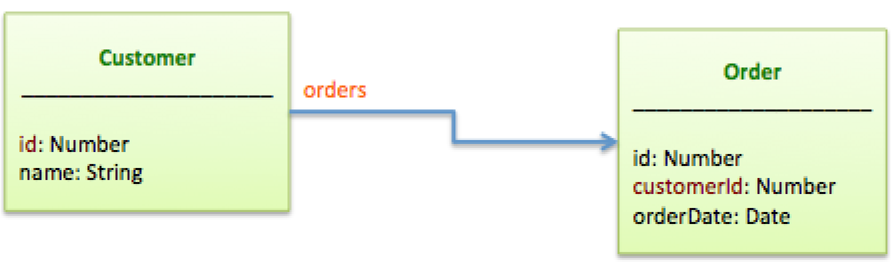
HasMany relations

- Overview
- Defining a hasMany relation
- Methods added to the model

Related articles:

Overview

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



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

Defining a hasMany relation

Use `slc loopback:relation` to create a relation between two models. The tool will prompt you to enter the name of the model, the name of related model, and other required information. The tool will then modify the Model definition JSON file (for example, `common/models/customer.json`) accordingly.

For more information, see Relation generator.

For example, here is the model JSON file for the customer model in loopback-example-relations:

common/models/customer.json

```
{
  "name": "Customer",
  "base": "PersistedModel",
  ...
  "relations": {
    "reviews": {
      "type": "hasMany",
      "model": "Review",
      "foreignKey": "authorId"
    },
...
```

Alternatively, you can define the relation in code, though in general this is not recommended:

common/models/customer.js

```
Customer.hasMany(Review, {as: 'reviews', foreignKey: 'authorId'});
```

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

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

Methods added to the model

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

Example method	Description

`customer.orders([where], function(err, orders) {`
`...`
`});`
	Find orders for the customer by the filter

`var order = customer.orders.build(data);`
 	
Or equivalently:
 	
`var order = new Order({customerId: customer.id, ...});`
	Build a new order for the customer with the customerId to be set to the id of the customer. No persistence is involved.

`customer.orders.create(data, function(err, order) {`
`...`
`});`
 	
Or, equivalently:
 	
`Order.create({customerId: customer.id, ...}, function(err, order) {`
`...`
`});`
	Create a new order for the customer.

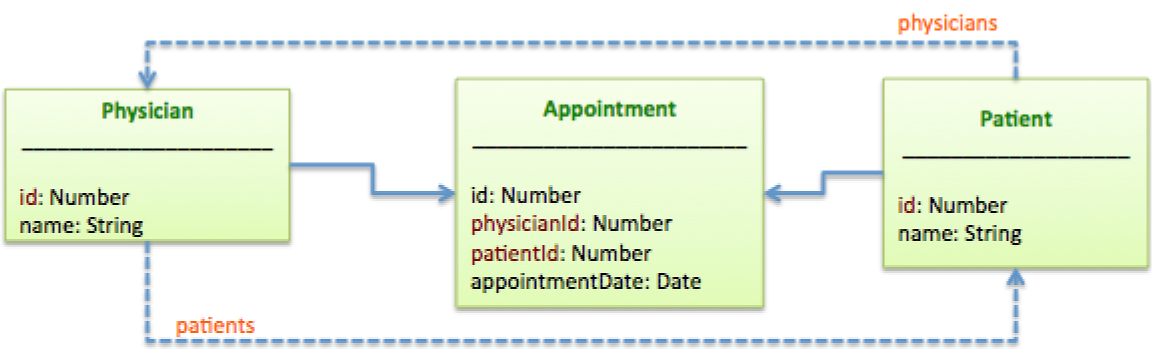
`customer.orders.destroyAll(function(err) { ... });`	Remove all orders for the customer.
`customer.orders.findById(orderId, function(err, order) { ... });`	Find an order by ID.
`customer.orders.destroy(orderId, function(err) { ... });`	Delete an order by ID.

HasManyThrough relations

Related articles:

Overview

A `hasManyThrough` relation sets up a many-to-many connection with another model. This relation indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model. For example, in an application for a medical practice where patients make appointments to see physicians, the relevant relation declarations might be:



The “through” model, **Appointment**, has two foreign key properties, **physicianId** and **patientId**, that reference the primary keys in the declaring model, **Physician**, and the target model, **Patient**.

Defining a hasManyThrough relation

Use `slc loopback:relation` to create a relation between two models. The tool will prompt you to enter the name of the model, the name of related model, and other required information. The tool will then modify the Model definition JSON file (for example, `common/models/customer.json`) accordingly.

For more information, see Relation generator.

To create a hasManyThrough relation, respond with **Yes** to the prompt for a "through" model, then specify the model:

```
[?] Require a through model? Yes
[?] Choose a through model: Appointment
```

For example:

common/models/physician.json

```
{
  "name": "Physician",
  "base": "PersistedModel",
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "patients": {
      "type": "hasMany",
      "model": "Patient",
      "foreignKey": "physicianId",
      "through": "Appointment"
    },
...
```

common/models/patient.json

```
{
  "name": "Patient",
  "base": "PersistedModel",
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "physicans": {
      "type": "hasMany",
      "model": "Physician",
      "foreignKey": "patientId",
      "through": "Appointment"
    },
...
```

common/models/appointment.json

```
{
  "name": "Appointment",
  "base": "PersistedModel",
  "properties": {
    "appointmentDate": {
      "type": "date"
    }
  },
  "validations": [],
  "relations": {
    "physician": {
      "type": "belongsTo",
      "model": "Physician",
      "foreignKey": "physicianId"
    },
    "patient": {
      "type": "belongsTo",
      "model": "Patient",
      "foreignKey": "patientId"
    },
...
```

You can also define a `hasManyThrough` relation in code, though this is not generally recommended:

common/models/physician.js

```
...
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
```

Defining the foreign key property

A `hasManyThrough` relation has a `keyThrough` property that indicates the foreign key property (field) name. If not specified, it defaults to the toModelName with `Id` appended; for example:

- `Physician.hasMany(Patient, {through: Appointment})` - keyThrough defaults to `patientId`.
- `Patient.hasMany(Physician, {through: Appointment})`- keyThrough defaults to `physicianId`.

The `keyThrough` properties above will be used to match these `foreignKeys` below:

```
Appointment.belongsTo(Physician, {as: 'foo', foreignKey: 'physicianId'});
Appointment.belongsTo(Patient, {as: 'bar', foreignKey: 'patientId'});
```

You can specify the keyThrough property explicitly:

```
Physician.hasMany(Patient, {through: Appointment, foreignKey: 'fooId', keyThrough:
'barId'});
Patient.hasMany(Physician, {through: Appointment, foreignKey: 'barId', keyThrough:
'fooId'});
// keyThroughs above will be used to match foreignKeys below
Appointment.belongsTo(Physician, {as: 'foo'}); // foreignKey defaults to 'fooId'
Appointment.belongsTo(Patient, {as: 'bar'});   // foreignKey defaults to 'barId'
```

keyThrough in JSON

Here is an example of defining a hasManyThrough relation with foreign keys. Consider the following tables:

- STUDENTS(ID,STUNAME): student information
- COURSES(ID,COURNAME): course information
- COURSTU(COURID,STUID): table with foreign keys that handle the many-to-many mapping

You can define the relations in JSON files in `common/models` as follows:

common/models/courses.json

```
...
  "relations": {
    "students": {
      "type": "hasMany",
      "model": "Students",
      "foreignKey": "courid",
      "through": "Courstu",
      "keyThrough": "stuid"
    }
...
```

common/models/students.json

```
"relations": {
    "courses": {
      "type": "hasMany",
      "model": "Courses",
      "foreignKey": "stuid",
      "through": "Courstu",
      "keyThrough": "courid"
    }
```

Self through

In some cases, you may want to define a relationship from a model to itself. For example, consider a social media application where users can follow other users. In this case, a user may follow many other users and may be followed by many other users. The code below shows how this might be defined, along with corresponding `keyThrough` properties:

common/models/user.js

```
User.hasMany(User, {as: 'followers', foreignKey: 'followeeId', keyThrough:
'followerId', through: Follow});
User.hasMany(User, {as: 'following', foreignKey: 'followerId', keyThrough:
'followeeId', through: Follow});
Follow.belongsTo(User, {as: 'follower'});
Follow.belongsTo(User, {as: 'followee'});
```

Methods added to the model

Once you define a "hasManyThrough" relation, LoopBack adds methods with the relation name to the declaring model class's prototype automatically, for example: `physician.patients.create(...)`.

Example method	Description

`physician.patients(filter, function(err, patients) {`
`...`
`});`
	Find patients for the physician.

`var patient = physician.patients.build(data);`	Create a new patient.

`physician.patients.create(data, function(err, patient) {`
`...`
`});`
	Create a new patient for the physician.

`physician.patients.destroyAll(function(err) {`
`...`
`});`
	Remove all patients for the physician

`physician.patients.add(patient, function(err, patient) {`
`...`
`});`
	Add a patient to the physician.

`physician.patients.remove(patient, function(err) {`
`...`
`});`
	Remove a patient from the physician.

`physician.patients.findById(patientId, function(err, patient) {`
`...`
`});`
	Find an patient by ID.

These relation methods provide an API for working with the related object (patient in the example above). However, they do not allow you to access both the related object (Patient) and the "through" record (Appointment) in a single call.

For example, if you want to add a new patient and create an appointment at a certain date, you have to make two calls (REST requests):

1. Create the patient via `Patient.create`

    ```
    POST /patients
    {
      "name": "Jane Smith"
    }
    ```

2. Create the appointment via `Appointment.create`, setting the `patientId` property to the `id` returned by `Patient.create`.

```
POST /appointments
{
  "patientId": 1,
  "physicianId": 1,
  "appointmentDate": "2014-06-01"
}
```

The following query can be used to list all patients of a given physician, including their appointment date:

```
GET /appointments?filter={"include":["patient"],"where":{"physicianId":2}}
```

Sample response:

```
[
  {
    "appointmentDate": "2014-06-01",
    "id": 1,
    "patientId": 1,
    "physicianId": 1,
    "patient": {
      "name": "Jane Smith",
      "id": 1
    }
  }
]
```

HasAndBelongsToMany relations

Related articles:

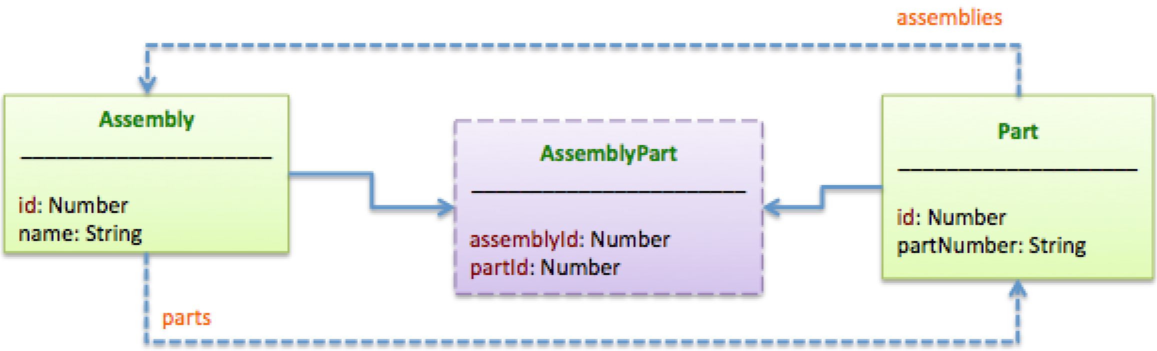
Overview

A hasAndBelongsToMany relation creates a direct many-to-many connection with another model, with no intervening model. For example, in an application with assemblies and parts, where each assembly has many parts and each part appears in many assemblies, you could declare the models this way:

Defining a hasAndBelongsToMany relation

Use `slc loopback:relation` to create a relation between two models. The tool will prompt you to enter the name of the model, the name of related model, and other required information. The tool will then modify the Model definition JSON file (for example, `common/models/customer.json`) accordingly.

For more information, see Relation generator.



For example, here is an excerpt from a model JSON file for a assembly model, expressing a hasAndBelongsToMany relation between assembly and part models:

/common/models/assembly.json

```
{
  "name": "Assembly",
  "plural": "Assemblies",
  "relations": {
    "parts": {
      "type": "hasAndBelongsToMany",
      "model": "Part"
    },
...
```

You can also define a hasAndBelongsToMany relation in code, though this is not recommended in general. For example:

/common/models/assembly.js

```
Part.hasAndBelongsToMany(Assembly);
Assembly.hasAndBelongsToMany(Part);
```

Adding a relation via REST API

When adding relation through the REST API, a join model must exist before adding relations.
For Example in the above example with "Assembly" and "Part" models, to add an instance of "Part" to "Assembly" through the REST API interface an "AssemblyPart" model must exist for it to work.

Most of the time you should add "hasAndBelongToMany" relations to models on server side using the method:

Example method

```
assembly.parts.add(part, function(err) {
  ...
});
```

Thus, if you need to add the relation using REST, first check if the "AssemblyPart" model exists first. Then add the relation using this code:

Rest Example Method

```
Assembly.Parts.link({id:assemblyId, fk: partId}, partInstance,  function(value,
header) {
  //success
});
```

Methods added to the model

Once you define a "hasAndBelongsToMany" relation, LoopBack adds methods with the relation name to the declaring model class's prototype automatically, for example: `assembly.parts.create(...)`.

Example method	Description
`assembly.parts(filter, function(err, parts) { ... });`	Find parts for the assembly.
`var part = assembly.parts.build(data);`	Build a new part.
`assembly.parts.create(data, function(err, part) { ... });`	Create a new part for the assembly.
`assembly.parts.add(part, function(err) { ... });`	Add a part to the assembly.
`assembly.parts.remove(part, function(err) { ... });`	Remove a part from the assembly.
`assembly.parts.findById(partId, function(err, part) { ... });`	Find a part by ID.
`assembly.parts.destroy(partId, function(err) { ... });`	Delete a part by ID.

Polymorphic relations

This documentation is still a work in progress.

LoopBack supports *polymorphic relations* in which a model can belong to more than one other model, on a single association. For example, you might have a Picture model that belongs to either an Author model or a Reader model.

Related articles:

- HasMany polymorphic relations
- BelongsTo polymorphic relations
- HasAndBelongsToMany polymorphic relations
- HasOne polymorphic relations
- Dealing with polymorphic.idType

The examples below use three example models: Picture, Author, and Reader, where a picture can belong to either an author or reader.

HasMany polymorphic relations

The usual options apply, for example: `as: 'photos'` to specify a different relation name/accessor.

common/models/author.json

```
{
  "name": "Author",
  "base": "PersistedModel",
  ...
  "relations": {
    "pictures": {
      "type": "hasMany",
      "model": "Picture",
      { "polymorphic": "imageable" }
    }
  }
...
```

And:

common/models/reader.json

```
{
  "name": "Reader",
  "base": "PersistedModel",
  ...
  "relations": {
    "pictures": {
      "type": "hasMany",
      "model": "Picture",
      "polymorphic": {
        "as": "imageable",
        "foreignKey": "imageableId",
        "discriminator": "imageableType"
       }
    }
  }
...
```

Alternatively, you can define the relation in code:

common/models/author.js

```
Author.hasMany(Picture, { polymorphic: 'imageable' });
```

And:

common/models/reader.js

```
Reader.hasMany(Picture, { polymorphic: { // alternative syntax
  as: 'imageable', // if not set, default to: reference
  foreignKey: 'imageableId', // defaults to 'as + Id'
  discriminator: 'imageableType' // defaults to 'as + Type'
  }
});
```

BelongsTo polymorphic relations

Because you define the related model dynamically, you cannot declare it up front. So instead of passing in the related model (name), you specify the name of the polymorphic relation.

common/models/picture.json

```
{ "name": "Picture",
  "base": "PersistedModel",
  ...
  "relations": {
    "author": {
      "type": "belongsTo",
      "model": "Author",
      "polymorphic": {
         "foreignKey": "imageableId",
         "discriminator": "imageableType"
      }
    }
  },
...
```

Or, in code:

common/models/picture.js

```
Picture.belongsTo('imageable', { polymorphic: true });

// Alternatively, use an object for setup:

Picture.belongsTo('imageable', { polymorphic: {
  foreignKey: 'imageableId',
  discriminator: 'imageableType'
} });
```

HasAndBelongsToMany polymorphic relations

This requires an explicit 'through' model, in this case: PictureLink

The relations `Picture.belongsTo(PictureLink)` and `Picture.belongsTo('imageable', { polymorphic: true });` will be setup automatically.

The same is true for the needed properties on PictureLink.

/common/models/model.js

```
Author.hasAndBelongsToMany(Picture, { through: PictureLink, polymorphic: 'imageable'
});
Reader.hasAndBelongsToMany(Picture, { through: PictureLink, polymorphic: 'imageable'
});

// Optionally, define inverse hasMany relations (invert: true):

Picture.hasMany(Author, { through: PictureLink, polymorphic: 'imageable', invert: true
});
Picture.hasMany(Reader, { through: PictureLink, polymorphic: 'imageable', invert: true
});
```

HasOne polymorphic relations

As shown here, you can specify `as: 'avatar'` to explicitly set the name of the relation. If not set, it defaults to the polymorphic name.

/common/models/model.js

```
Picture.belongsTo('imageable', { polymorphic: true });

Author.hasOne(Picture, { as: 'avatar', polymorphic: 'imageable' });
Reader.hasOne(Picture, { polymorphic: { as: 'imageable' } });
```

Dealing with polymorphic.idType

Because `modelTo` is unknown up-front (it's polymorphic), you cannot rely on `modelTo` for getting the `foreignKey` type. You can explicitly declare the idType as shown below.

The example below should provide you with the following results:

```
[ { url: 'john.jpg', imageableType: 'Author', imageableId: '1', id: 1 },
{ url: 'joe.jpg', imageableType: 'Reader', imageableId: '1', id: 2 } ]
Authors:
[ { username: 'John', id: 1 } ]
Readers:
[ { name: 'Joe', id: 1 } ]
```

```
var Author = app.models.Author;
var Reader = app.models.Reader;
var Picture = app.models.Picture;
Author.hasOne(Picture, {
  as: 'avatar',
  polymorphic: {
    foreignKey: 'imageableId',
    discriminator: 'imageableType'
  }
});
Reader.hasOne(Picture, {
  as: 'imageable',
  polymorphic: {
    foreignKey: 'imageableId',
    discriminator: 'imageableType'
  }
});
Picture.belongsTo('owner', {
```

```
  idName: 'username',
  polymorphic: {
    idType: Author.definition.properties.username.type,
    foreignKey: 'imageableId',
    discriminator: 'imageableType'
  }
});
//Creating demo author, reader pictures then listing them
function createAuthor(cb){
  Author.create({
    username: "John"
  }). then( function(author){
    author.avatar.create({url : "john.jpg"}, function(){
      cb();
    });
  });
}
function createReader(cb){
  Reader.create({
    name: "Joe"
  }). then( function(reader){
    reader.imageable.create({url : "joe.jpg"}, function(){
      cb();
    });
  });
}
function listPictures(){
  Picture.find(function(err,res){
    console.log("\nPictures:\n",res);
  })
}
function listReaders(){
  Reader.find(function(err,res){
    console.log("\nReaders:\n",res);
  })
}
function listAuthors(){
  Author.find(function(err,res){
    console.log("\nAuthors:\n",res);
  })
}
//executing the demo
createAuthor( function(){
  createReader(function(){
    listPictures();
    listAuthors();
    listReaders();
```

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

Querying related models

Related articles:

See also: Relation REST API.

Overview

A relation defines the connection between two models by connecting a foreign key property to a primary key property. For each relation type, LoopBack automatically mixes in helper methods to the model class to help navigate and associate the model instances to load or build a data graph.

Often, client applications want to select relevant data from the graph, for example to get user information and recently-placed orders. LoopBack provides a few ways to express such requirements in queries.

The LoopBack Relations example application provides some examples. For general information on queries, see Querying data.

Inclusion

To include related models in the response for a query, use the 'include' property of the query object or use the `include()` method on the model class. The 'include' can be a string, an array, or an object. For more information, see Include filter.

The following examples illustrate valid formats.

Load all user posts with only one additional request:

/server/script.js

```
User.find({include: 'posts'}, function() {
  ...
});
```

Or, equivalently:

/server/script.js

```
User.find({include: ['posts']}, function() {
  ...
});
```

Load all user posts and orders with two additional requests:

/server/script.js

```
User.find({include: ['posts', 'orders']}, function() {
  ...
});
```

Load all post owners (users), and all orders of each owner:

/server/script.js

```
Post.find({include: {owner: ‘orders’}}, function() {
  ...
});
```

Load all post owners (users), and all friends and orders of each owner:

/server/script.js

```
Post.find({include: {owner: [‘friends’, ‘orders’]}}, function() {
  ...
});
```

Load all post owners (users), all posts (including images), and orders of each owner:

/server/script.js

```
Post.find({include: {owner: [{posts: ‘images’} , ‘orders’]}}, function() {
  ...
});
```

The model class also has an `include()` method. For example, the code snippet below will populate the list of user instances with posts:

/server/script.js

```
User.include(users, 'posts', function() {
  ...
});
```

Scope

Scoping enables you to define a query as a method to the target model class or prototype. For example,

/server/boot/script.js

```
User.scope(‘top10Vips’, {where: {vip: true}, limit: 10});

User.top10Vips(function(err, vips) {
});
```

You can create the same function using a custom method too:

/server/boot/script.js

```
User.top10Vips = function(cb) {
  User.find({where: {vip: true}, limit: 10}, cb);
}
```

Using filters parameters with included relations

You can use parameters on filters such as where, order, fields, include filters when querying related models to return the data from the related models.

For example, consider Student, Class, and Teacher models, where a Student hasMany Classes, and a Teacher hasMany Classes.

Find ALL Student and also return ALL their Classes with the Teacher who teaches those Classes and also ALL of the Students enrolled in those Classes...

```
Student.find({"filter": {
                  "include": {"relation": "classes", "scope": {"include":
["teachers","students"]}}
             }
})
```

Another example: find a specific teacher and also return ALL their classes and also ALL of the students enrolled in those classes.

```
Teacher.find({"filter": {
                    "where": {"id": $state.params.id},
                    "include": {"relation": "classes", "scope": {"include":
["students"]}}
              }
})
```

Embedded models and relations

- Overview
- EmbedsOne
    - Define the relation in code
    - Parameters for the definition
    - Options
    - Define the relation in JSON
    - Helper methods
- EmbedsMany
    - Define the relation in code
    - Parameters for the definition
    - Options
    - Define the relation in JSON
    - Helper methods
- EmbedsMany with belongsTo
    - Define the embedsMany relation for Book
    - Define the polymorphic belongsTo relation for Link
- ReferencesMany
    - Parameters for the definition
    - Options
    - Define the relation in code
    - Helper methods
- Transient versus persistent for the embedded model
    - Define a transient data source
    - Use the transient data source for embedded models

Related articles:

Overview

LoopBack relations enable you to create connections between models and provide navigation/aggregation APIs to deal with a graph of model instances. In addition to the traditional ones, LoopBack also supports the following embedded relations:

- EmbedsOne - a model that embeds another model; for example, a Customer embeds one billingAddress.
- EmbedsMany - a model that embeds many instances of another model; for example, a Customer can have multiple email addresses and each email address is a complex object that contains label and address.
- EmbedsMany with belongsTo - a model that embeds many links to related people, such as an author or a reader.
- ReferencesMany

You can use embedded relations with relational databases, but be aware that the data will be stored in stringified-JSON format.

EmbedsOne

EmbedsOne is used to represent a model that embeds another model, for example, a Customer embeds one billingAddress.

Sample embedded model

```
{
  id: 1,
  name: 'John Smith',
  billingAddress: {
    street: '123 Main St',
    city: 'San Jose',
    state: 'CA',
    zipCode: '95124'
  }
}
```

Define the relation in code

common/models/customer.js

```
Customer.embedsOne(Address, {
  as: 'address', // default to the relation name - address
  property: 'billingAddress' // default to addressItem
});
```

Parameters for the definition

- methods - Scoped methods for the given relation
- properties - Properties taken from the parent object
- scope - Default scope
- options - Options
- default - Default value
- property - Name of the property for the embedded item
- as - Name of the relation

Options

- forceId - force generation of ida for embedded items, default to false
- validate - denote if the embedded items should be validated, default to true
- persistent - denote if the embedded items should be persisted, default to false

Define the relation in JSON

common/models/customer.json

```
{
  "name": "Customer",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "relations": {
    "address": {
      "type": "embedsOne",
      "model": "Address",
      "property": "billingAddress",
      "options": {
        "validate": true,
        "forceId": false
      }
    }
...
}
```

Helper methods

- customer.address()
- customer.address.build()
- customer.address.create()
- customer.address.update()
- customer.address.destroy()
- customer.address.value()

EmbedsMany

Use an embedsMany relation to indicate that a model can embed many instances of another model, for example, a Customer can have multiple email addresses and each email address is a complex object that contains label and address.

Sample model instance with many embedded models

```
{
  id: 1,
  name: 'John Smith',
  emails: [
  {
    label: 'work',
    address: 'john@xyz.com'
  },
  {
    label: 'home',
    address: 'john@gmail.com'
  }
  ]
}
```

Treat `embedsMany` as an actual relation, no different from `hasMany`, for example. This means that you cannot just POST the full object with embedded/nested data to create everything all at once. So using the example above to add a Customer and multiple email addresses would require two POST operations, one for the Customer record and one for the multiple email address data.

Define the relation in code

common/models/customer.js

```
Customer.embedsOne(EmailAddress, {
  as: 'emails', // default to the relation name - emailAddresses
  property: 'emailList' // default to emailAddressItems
});
```

Parameters for the definition

- methods
- properties
- scope
- options
- default
- property
- as

Options

- forceId
- validate
- persistent

Define the relation in JSON

common/models/customer.json

```
{
  "name": "Customer",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "relations": {
    "emails": {
      "type": "embedsMany",
      "model": "EmailAddress",
      "property": "emailList",
      "options": {
        "validate": true,
        "forceId": false
      }
    }
...
}
```

Helper methods

- customer.emails()
- customer.emails.create()
- customer.emails.build()
- customer.emails.findById()
- customer.emails.destroyById()
- customer.emails.updateById()
- customer.emails.exists()
- customer.emails.add()
- customer.emails.remove()
- customer.emails.get() - alias to findById
- customer.emails.set() - alias to updateById
- customer.emails.unset() - alias to destroyById
- customer.emails.at()
- customer.emails.value()

EmbedsMany with belongsTo

Use an embedsMany with belongsTo relation to indicate a model that can embed many links to other models; for example a book model that embeds many links to related people, such as an author or a reader. Each link belongs to a person and it's polymorphic, since a person can be an Author or a Reader.

Exampel embedsMany with belongsTo model instance

```
{
  id: 1
  name: 'Book 1',
  links: [
    { notes: 'Note 1',
      id: 1,
      linkedId: 1,
      linkedType: 'Author',
      name: 'Author 1' },
    { notes: 'Note 2',
      id: 2,
      linkedId: 1,
      linkedType: 'Reader',
      name: 'Reader 1' }
  ]
}
```

Define the embedsMany relation for Book

common/models/book.json

```
{
  "name": "Book",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "people": {
      "type": "embedsMany",
      "model": "Link",
      "scope": {
        "include": "linked"
      }
    }
  },
  "acls": [],
  "methods": []
}
```

Define the polymorphic belongsTo relation for Link

common/models/link.json

```
{
  "name": "Link",
  "base": "Model",
  "idInjection": true,
  "properties": {
    "id": {
      "type": "number",
      "id": true
    },
    "name": {
      "type": "string"
    },
    "notes": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "linked": {
      "type": "belongsTo",
      "polymorphic": {
        "idType": "number"
      },
      "properties": {
        "name": "name"
      },
      "options": {
        "invertProperties": true
      }
    }
  },
  "acls": [],
  "methods": []
}
```

ReferencesMany

Sample referencesMany model instance

```
{
  id: 1,
  name: 'John Smith',
  accounts: [
    "saving-01", "checking-01",
  ]
}
```

Parameters for the definition

- methods
- properties
- foreignKey

- scope
- options
- default
- property
- as

Options

- forceId
- validate
- persistent

Define the relation in code

common/models/customer.json

```
{
  "name": "Customer",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "relations": {
    "accounts": {
      "type": "referencesMany",
      "model": "Account",
      "property": "accountIds",
      "options": {
        "validate": true,
        "forceId": false
      }
    }
...
}
```

Helper methods

- customer.accounts()
- customer.accounts.create()
- customer.accounts.build()
- customer.accounts.findById()
- customer.accounts.destroy()
- customer.accounts.updateById()
- customer.accounts.exists()
- customer.accounts.add()
- customer.accounts.remove()
- customer.accounts.at()

Transient versus persistent for the embedded model

Define a transient data source

server/datasources.json

```
{
  ...
  "transient": {
    "name": "transient",
    "connector": "transient"
  }
}
```

Use the transient data source for embedded models

server/model-config.json

```
{
  ...
  "Customer": {
    "dataSource": "db",
    "public": true
  },
  "Address": {
    "dataSource": "transient",
    "public": false
  },
  "EmailAddress": {
    "dataSource": "transient",
    "public": false
  },
  "Account": {
    "dataSource": "db",
    "public": false
  }
}
```

Nested queries

In addition to querying data directly through models using LoopBack APIs, nested models can be API endpoints. To do this, you must use nestRemoting functions.

The easiest way to understand nested queries is through an example.

Suppose an app has book, chapter, page, and image models. And:

- Each book can have many pages and chapters.
- Each chapter and page can have many notes.
- Each book has an image.

The following blocks of JSON show the model for Book and relations for Page and Chapter models.

For more information on the model relations used, see BelongsTo relations and HasMany relations.

common/models/book.json

```
{
  "name": "Book",
  "base": "PersistedModel",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "properties": {
    "name": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "pages": {
      "type": "hasMany",
      "model": "Page",
      "foreignKey": "",
      "through": ""
    },
    "chapters": {
      "type": "hasMany",
      "model": "Chapter",
      "foreignKey": "",
      "through": ""
    }
  },
  "acls": [],
  "methods": {}
}
```

common/models/chapter.json

```
{
  "name": "Chapter",
  ...
  "relations": {
    "notes": {
      "type": "hasMany",
      "model": "Note",
      "foreignKey": "",
      "through": ""
    }
  },
  ...
}
```

common/models/page.json

```
{
  "name": "Page",
  ...
  "relations": {
    "notes": {
      "type": "hasMany",
      "model": "Note",
      "foreignKey": "",
      "through": ""
    }
  },
  ...
}
```

common/models/image.json

```
{
  "name": "Image",
  ...
  "relations": {
    "book": {
      "type": "belongsTo",
      "model": "Book",
      "foreignKey": "",
      "required": true
    }
  },
  ...
}
```

You can query pages of a specific book via regular relationships,as illustrated with the following API endpoints:

Endpoint	Output	Description
/api/books/123/pages	An array of pages data	Queries pages of a specific book
/api/books/123/pages/456	An object of a page data	Queries a page data of a specific page under a specific book

However, to query nested models more in depth and have them as API endpoints you need to use the model `nestRemoting()` function:

```
Book.nestRemoting('pages');
 Book.nestRemoting('chapters');
 Image.nestRemoting('book');
```

The above code enables the following nested queries:

Endpoint	Output	Description
/api/books/123/pages/456/notes	An array of notes objects	Queries all of the notes associated with a specific page under a specific book
/api/books/123/pages/456/notes/567	An object of a note data	Queries a specific note associated with a specific page under a specific book

Alternatively, since an image belongs to book instance; you can query their pages through their images:

Endpoint	Output	Description
/api/images/345/book/pages	An array of pages of a book	Queries all of the pages of the book, whose associated image id is 345
/api/images/345/book/pages/456	An object of a page data	Queries page with the id of 456 under the book, whose associated image id is 345

Using built-in models

Overview

Loopback provides useful built-in models for common use cases:

- **Application model** - contains metadata for a client application that has its own identity and associated configuration with the LoopBack server.
- **User model** - register and authenticate users of your app locally or against third-party services.
- **Access control models** - ACL, AccessToken, Scope, Role, and RoleMapping models for controlling access to applications, resources, and methods.
- **Email model** - send emails to your app users using SMTP or third-party services.

The built-in models (except for Email) extend PersistedModel, so they automatically have a full complement of create, update, and delete (CRUD) operations.

By default, only the User model is exposed over REST. To expose the other models, change the model's `public` property to true in `server/model-config.json`. See Exposing models for more information. **Use caution**: exposing some of these models over public API may be a security risk.

Application model

Use the Application model to manage client applications and organize their users.

The default model definition file is common/models/application.json in the LoopBack repository.

User model

The User model represents users of the application or API. The default model definition file is common/models/user.json in the LoopBack repository.

IMPORTANT
You must create your own custom model (named something other than "User," for example "Customer" or "Client") that extends the built-in User model rather than use the built-in User model directly. The built-in User model provides a great deal of commonly-used functionality that you can use via your custom model.

For more information, see Managing users.

Access control models

Use access control models to control access to applications, resources, and methods. These models include:

- ACL
- AccessToken
- Scope
- Role

- RoleMapping

ACL model

An ACL model connects principals to protected resources. The system grants permissions to principals (users or applications, that can be grouped into roles) .

- Protected resources: the model data and operations (model/property/method/relation)
- Whether a given client application or user is allowed to access (read, write, or execute) the protected resource.

Creating a new ACL instance.

server/boot/script.js

```
ACL.create( {principalType: ACL.USER,
             principalId: 'u001',
             model: 'User',
             property: ACL.ALL,
          accessType: ACL.ALL,
             permission: ACL.ALLOW},
             function (err, acl) { ACL.create( {principalType: ACL.USER,
                                                principalId: 'u001',
                                                model: 'User',
                                                property: ACL.ALL,
                                                accessType: ACL.READ,
                                                permission: ACL.DENY},
                                                function (err, acl) { }
                                                })
                                 }
             })
```

Email model

Set up an email data source by adding an entry to `/server/datasources.json`, such as the following (for example):

server/datasources.json

```
{
    ...
    "myEmailDataSource": {
       "connector": "mail",
       "transports": [{
         "type": "smtp",
         "host": "smtp.private.com",
         "secure": false,
         "port": 587,
         "tls": {
           "rejectUnauthorized": false
         },
         "auth": {
           "user": "me@private.com",
           "pass": "password"
         }
       }]
    }
    ...
}
```

See Email connector for more information on email data sources.

Then, reference the data source in `/server/model-config.json` as follows (for example):

server/model-config.json

```
{
  ...
  "Email": {
    "dataSource": "myEmailDataSource",
  },
  ...
}
```

Send email messages

The following example illustrates how to send emails from an app. Add the following code to a file in the `/models` directory:

server/models/model.js

```
module.exports = function(MyModel) {
  // send an email
  MyModel.sendEmail = function(cb) {
    MyModel.app.models.Email.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!');
      cb(err);
    });
  }
};
```

The default model definition file is common/models/email.json in the LoopBack repository.

The mail connector uses nodemailer. See the nodemailer docs for more information.

Confirming email address

See Verifying email addresses.

Extending built-in models

- Extending models using JSON
- Extending a model in JavaScript
    - Mixing in model definitions
    - Setting up a custom model

Extending models using JSON

When you create a model with the model generator, you choose a base model, that is, the model that your model will "extend" and from which it will inherit methods and properties. The tool will set the base property in the model definition JSON file accordingly. For example, for a model that extends PersistedModel:

/common/models/model.json

```
{
  "name": "Order",
  "base": "PersistedModel",
  ...
```

To change the base model, simply edit the JSON file and change the `base` property.

In general, use `PersistedModel` as the base model when you want to store your data in a database using a connector such as MySQL or MongoDB. Use `Model` as the base for models that don't have CRUD semantics, for example, using connectors such as SOAP and REST.

Use the built-in User model as the base for a model that represents app users; this model provides capabilities to register, login, and recover passwords. See Managing users for more information. When you extend the built-in User model, use a name other than "User", for example, user (with a lowercase "u"), "customer," or "client." Don't name it "User" because that will conflict with the built-in User model.

See Customizing models for general information on how to create a model that extends (or "inherits from") another model.

See LoopBack types for information on data types supported.

Extending a model in JavaScript

You can also extend models using JavaScript file in the model JavaScript file, `/common/models/`*`modelName`*`.js` (where *`modelName`* is the name of the model); for example:

/common/models/user.js

```
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);
```

See LoopBack types for information on data types supported.

Mixing in model definitions

You may want to create models that share a common set of properties and logic. LoopBack enables you to "mix-in" one or more other models into a single model. This is a special case of the general ability to mix in model properties and functions. See Defining mixins for more information.

For example:

common/models/myModel.js

```
var TimeStamp = modelBuilder.define('TimeStamp', {created: Date, modified: Date});
var Group = modelBuilder.define('Group', {groups: [String]});
User.mixin(Group, TimeStamp);
```

Setting up a custom model

You may want to perform additional setup for an custom model, such as adding remote methods of another model. To do so, implement a `setup()` method on the new model. The `loopback.Model.extend()` function calls `setup()` so code you put in `setup()` will automatically get executed when the model is created.

For example:

common/models/myModel.js

```
MyModel = Model.extend('MyModel');

MyModel.on('myEvent', function() {
  console.log('meep meep!');
});

MyExtendedModel = MyModel.extend('MyExtendedModel');

MyModel.emit('myEvent'); // nothing happens (no event listener)

// this is where `setup()` becomes handy

MyModel.setup = function() {
  var MyModel = this;
  // since setup is called for every extended model
  // the extended model will also have the event listener
  MyModel.on('myEvent', function() {
    MyModel.printModelName();
  });
}
```

Creating database tables for built-in models

LoopBack applications come with a small set of built-in models. To create database tables for these models, follow the general procedure for creating a database schema from models using *auto-migration*.

> If the database has existing tables, running `automigrate()` will drop and re-create the tables and thus may lead to loss of data. To avoid this problem use `autoupdate()`. See Creating a database schema from models for more information.

To create tables for LoopBack built-in models, follow this procedure:

1. Follow the basic procedure in Attaching models to data sources to change from the in-memory data source to the database you want to use.
2. Create `server/create-lb-tables.js` file with the following:

```
var server = require('./server');
var ds = server.dataSources.db;
var lbTables = ['User', 'AccessToken', 'ACL', 'RoleMapping', 'Role'];
ds.automigrate(lbTables, function(er) {
  if (er) throw er;
  console.log('Looback tables [' + lbTables + '] created in ', ds.adapter.name);
  ds.disconnect();
});
```

3. Run the script manually:

```
$ cd server
$ node create-lb-tables.js
```

Model property reference

This reference information is being moved to the API documentation. Until that is complete, it is provided here.

Application properties

The application model represents the metadata for a client application that has its own identity and associated configuration with the LoopBack server.

See http://apidocs.strongloop.com/loopback/#application for a list of the application model properties.

ACL properties

See http://apidocs.strongloop.com/loopback/#acl for a list of ACL model properties.

Role 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

LoopBack defines some special roles:

Identifier	Name	Description
Role.OWNER	$owner	Owner of the object
Role.RELATED	$related	Any user with a relationship to the object

Role.AUTHENTICATED	$authenticated	Authenticated user
Role.UNAUTHENTICATED	$unauthenticated	Unauthenticated user
Role.EVERYONE	$everyone	Everyone

Scope 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

RoleMapping properties

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

The following table describes the properties of the roleMapping model:

Property	Type	Description
id	String	ID
roleId	String	Role ID
principalType	String	Principal type, such as user, application, or role
principalId	String	Principal ID

Built-in models REST API

LoopBack provides a number of built-in models that have REST APIs. Many of them inherit endpoints from the generic PersistedModel REST API.

See also: Exposing models over REST

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.

The built-in models are:

- PersistedModel REST API
- Access token REST API
- ACL REST API
- Application REST API
- Email REST API
- Relation REST API
- Role REST API
- User REST API

PersistedModel REST API

You can use the **StrongLoop API Explorer** to quickly construct and make requests to a LoopBack app running on the server. If a LoopBack app is running on `localhost` at port `3000`, then by default API Explorer is available at http://localhost:3000/explorer/.

- Overview
- Create model instance
- Update / insert instance
- Check instance existence
- Find instance by ID
- Find matching instances
- Find first instance
- Delete model instance
- Delete all matching instances
- Get instance count
- Update model instance attributes
- Update matching model instances
- Create Change Stream

- Get Change Stream

Related articles

- Creating models
- Customizing models
- Creating model relations
- Querying data
- Model definition JSON file
- PersistedModel REST API

Overview

PersistedModel is the base class for models connected to persistent data sources such as databases and is also the base class for all built-in models (except Email). It provides all the standard create, read, update, and delete (CRUD) operations and exposes REST endpoints for them.

By default, LoopBack uses `/api` as the URI root for the REST API. You can change this by changing the `restApiRoot` property in the application `/server/config.json` file. See config.json for more information.

Model REST API endpoints are generally 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 JSON file.

> You can't customize the routes to PersistedModel REST API endpoints. However, you can control how REST API endpoints are constructed from custom models with the `rest.normalizeHttpPath` property in `server/config.json`; for more information, see config.json (Remoting properties).

Create model instance

Create a new instance of the model and persist it to the data source.

```
POST /modelName
```

Arguments

- Form data - Model instance data. Can be JSON representing a single model instance or an array of model instances.

Example

Request URL: `POST  http://localhost:3000/api/locations`
Request body:

```
{"name": "L1", "street": "107 S B St", "city": "San Mateo", "zipcode": "94401"}
```

Response status code: `200`
Response body:

```
{
  "id": "96",
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L1"
}
```

Update / insert instance

Update an existing model instance or insert a new one into the data source. The update will override any specified attributes in the request data object. It won't remove existing ones unless the value is set to null.

Performs upsert to detect if there is a matching instance; if not, then inserts (creates) a new instance. If there is a matching instance, updates it.

```
PUT /modelName
```

Arguments

- Form data - model instance data in JSON format.

Examples

Insert

Request URL: `PUT http://localhost:3000/api/locations`
Request body:

```
{"name": "L1", "street": "107 S B St", "city": "San Mateo", "zipcode": "94401"}
```

Response status code: `200`
Response body:

```
{
  "id": 98,
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L1"
}
```

Update

Request URL: `PUT http://localhost:3000/api/locations`
Request body:

```
{"id": "98", "name": "L4", "street": "107 S B St", "city": "San Mateo", "zipcode": "94401"}
```

Response status code: `200`
Response body:

```
{
  "id": 98,
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L4"
}
```

Check instance existence

Check whether a model instance exists by ID in the data source.

```
GET /modelName/modelID/exists
```

Arguments

- *modelID* - model instance ID

Example

Request URL: `GET http://localhost:3000/api/locations/88/exists`

Response status code: `200`
Response body:

```
{
    "exists": true
}
```

Find instance by ID

Find a model instance by ID from the data source.

GET */modelName/modelID?filter=[filterType1]=<val1>&filter[filterType2]=<val2>...*

See also Accessing related models for an example of fetching data from related models.

Arguments

- **modelID** - Model instance ID
- *filterType1*, *filterType2*, and so on, are the filter types. This operation supports only include and fields filters; see Include filter and Fields filter for more information.
- *val1*, *val2* are the corresponding values.

Example

Request URL: `GET  http://localhost:3000/api/locations/88`

Response status code: `200`
Response body:

```
{
    "id": 88,
    "street": "390 Lang Road",
    "city": "Burlingame",
    "zipcode": 94010,
    "name": "Bay Area Firearms"
}
```

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 `filter` HTTP query parameters, where:

- *filterType1*, *filterType2*, and so on, are the filter types.
- *val1*, *val2* are the corresponding values.

See Querying data for an explanation of filter syntax.

Example

Request without filter:

Request URL: `GET  http://localhost:3000/api/locations`

Request with a filter to limit response to two records:

Copyright 2016 StrongLoop, an IBM company

Request URL: `GET` `http://localhost:3000/api/locations?filter[limit]=2`

Response status code: `200`
Response body:


```
[
  {
    "id": "87",
    "street": "7153 East Thomas Road",
    "city": "Scottsdale",
    "zipcode": 85251,
    "name": "Phoenix Equipment Rentals"
  },
  {
    "id": "88",
    "street": "390 Lang Road",
    "city": "Burlingame",
    "zipcode": 94010,
    "name": "Bay Area Firearms"
  }
]
```


Find first instance

Find first instance of the model matched by filter from the data source.

```
GET /modelName/findOne?filter=[filterType1]=<val1>&filter[filterType2]=<val2>...
```

Arguments

Query parameters:

- **filter** - Filter that defines where, order, fields, skip, and limit. It's same as find's filter argument. See Querying data details.

Example

Request URL: `GET` `http://localhost:3000/api/locations/findOne?filter[where][city]=Scottsdale`

Response status code: `200`
Response body:


```
{
  "id": "87",
  "street": "7153 East Thomas Road",
  "city": "Scottsdale",
  "zipcode": 85251,
  "name": "Phoenix Equipment Rentals"
}
```


Delete model instance

Delete a model instance by ID from the data source

```
DELETE /modelName/modelID
```

Arguments

- **modelID** - model instance ID

Example

Request URL: `DELETE  http://localhost:3000/api/locations/88`

Response status code: `204`

Delete all matching instances

By default, this operation is not exposed over REST to prevent deleting data unintentionally.

Delete model instanced from the data source that match the specified where clause.

```
DELETE /modelName?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 Querying data details.

Example

Request URL: `DELETE  http://localhost:3000/api/locations?[where][city]=Dallas`

Response status code: `200`

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. See Where filter for more information.

Example

Count without "where" filter

Request URL: `GET  http://localhost:3000/api/locations/count`

Count with a "where" filter

Request URL: `GET  http://localhost:3000/api/locations/count?where[city]=Burlingame`

Response status code: `200`
Response body:

```
{
    count: 6
}
```

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 instance ID

Example

Request URL: `PUT  http://localhost:3000/api/locations/88`
Request body:

```
{"name": "L2"}
```

Response status code: `200`
Response body:

```
{
  "id": "88",
  "street": "390 Lang Road",
  "city": "Burlingame",
  "zipcode": 94010,
  "name": "L2"
}
```

Update matching model instances

Update attributes of matching model instances and persist into the data source.

POST /*modelName*/update?where[*property*]=*value*

Arguments

- data - An object containing property name/value pairs.
- *where* - The where object to select matching instances. See Where filter for more information.

Example

Request URL: `POST  http://localhost:3000/api/locations/update?where[city]=Burlingame`
Request body:

```
{"city": "San Jose"}
```

Response status code: `200`

Create Change Stream

Create a new change stream.

POST /*modelName*/change-stream?_format=event-stream

Arguments

- Form data - Model instance data. JSON representing a single model instance or an array of model instances.

Example

Request URL: `POST  http://localhost:3000/api/locations/`
Request body:

```
{"city": "San Jose"}
```

Get Change Stream

Fetch a change stream.

```
GET /modelName/change-stream?_format=event-stream
```

Access token REST API

All of the endpoints in the access token REST API are inherited from the generic PersistedModel REST 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.	JSON object (in request body)
`/accessTokens`	GET	Deny	Find instances of accessTokens that match specified filter.	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/accessTokens`	PUT	Deny	Update / insert access token instance and persist to data source.	JSON object (in request body)
`/accessTokens/id`	GET	Deny	Find access token by ID: Return data for the specified access token instance ID.	*id*, the access token instance ID (in URI path)
`/accessTokens/id`	PUT	Deny	Update attributes for specified access token ID and persist.	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/accessTokens/id`	DELETE	Deny	Delete access token with specified instance ID.	*id*, access token ID (in URI path)
`/accessTokens/id/exists`	GET	Deny	Check instance existence: Return true if specified access token ID exists.	URI path: • *id* - Model instance ID
`/accessTokens/count`	GET	Deny	Return the number of access token instances that matches specified where clause.	Where filter specified in query parameter
`/accessTokens/findOne`	GET	Deny	Find first access token instance that matches specified filter.	Same as Find matching instances.

ACL REST API

All of the endpoints in the ACL REST API are inherited from the PersistedModel REST API. The reference is provided here for convenience.

By default, the ACL REST API is not exposed. To expose it, add the following to models.json:

```
"acl": {
    "public": true,
    "options": {
      "base": "ACL"
    },
    "dataSource": "db"
  },
```

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/acls`	POST	Allow	Add ACL instance and persist to data source.	JSON object (in request body)
`/acls`	GET	Deny	Find instances of ACLs that match specified filter.	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/acls`	PUT	Deny	Update / insert ACL instance and persist to data source.	JSON object (in request body)
`/acls/`*`id`*	GET	Deny	Find ACL by ID: Return data for the specified ACL instance ID.	*id*, the ACL instance ID (in URI path)
`/acls/`*`id`*	PUT	Deny	Update attributes for specified ACL ID and persist.	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/acls/`*`id`*	DELETE	Deny	Delete ACL with specified instance ID.	*id*, acls ID (in URI path)
`/acls/`*`id`*`/exists`	GET	Deny	Check instance existence: Return true if specified ACL ID exists.	URI path: • *id* - Model instance ID
`/acls/count`	GET	Deny	Return the number of ACL instances that matches specified where clause.	Where filter specified in query parameter
`/acls/findOne`	GET	Deny	Find first ACL instance that matches specified filter.	Same as Find matching instances.

Application REST API

All of the endpoints in the Application REST API are inherited from the PersistedModel REST 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.	JSON object (in request body)

`/applications`	GET	Deny	Find instances of applications that match specified filter.	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/applications`	PUT	Deny	Update / insert application instance and persist to data source.	JSON object (in request body)
`/applications/id`	GET	Deny	Find application by ID: Return data for the specified application instance ID.	*id*, the application instance ID (in URI path)
`/applications/id`	PUT	Deny	Update attributes for specified application ID and persist.	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/applications/id`	DELETE	Deny	Delete application with specified instance ID.	*id*, application ID (in URI path)
`/applications/id/exists`	GET	Deny	Check instance existence: Return true if specified application ID exists.	URI path: • *id* - Model instance ID
`/applications/count`	GET	Deny	Return the number of application instances that matches specified where clause.	Where filter specified in query parameter
`/applications/findOne`	GET	Deny	Find first application instance that matches specified filter.	Same as Find matching instances.

Email REST API

- Operation name

Quick reference

URI Pattern	HTTP Verb	Default Permission	Action	Arguments
`/foo/bar/baz`	One of: GET, POST, PUT, DELETE	Allow / Deny	Description plus link to section with full reference. NOTE: Rand will add links to sections.	List arguments in POST body, query params, or path.

Operation name

Brief description goes here.

```
POST /modelName
```

Arguments

- List of all arguments in POST data or query string

Example

Request:

```
curl -X POST -H "Content-Type:application/json"
-d '{... JSON ... }'
http://localhost:3000/foo
```

Response:

```
// Response JSON
```

Errors

List error codes and return JSON format if applicable.

Relation REST API

These endpoints are part of the PersistedModel REST API, but are presented in a separate page for ease of reference.

Get related model instances

Follow the relations from one model to another one to get instances of the associated model.

GET /*model1-name*/*instanceID*/*model2-name*

Arguments

- *instanceID* - ID of instance in model1.
- *model1-name* - name of first model.
- *model2-name* - name of second related model.

Example

Request:

```
GET http://localhost:3000/locations/88/inventory
```

`Response:`

```
[
  {
    "productId": "2",
    "locationId": "88",
    "available": 10,
    "total": 10
  },
  {
    "productId": "3",
    "locationId": "88",
    "available": 1,
    "total": 1
  }
]
```

Get hasMany related model instances

List related model instances for specified *model-name* identified by the *instance-ID*, for hasMany relationship.

GET */model-name/instanceID/hasManyRelationName*

Create hasMany related model instance

Create a related model instance for specified *model-name* identified by *instance-ID*, for hasMany relationship.

POST */model1-name/instanceID/hasManyRelationName*

Delete hasMany related model instances

Delete related model instances for specified *model-name* identified by *instance-ID*, for hasMany relationship.

DELETE */model1-name/instance-ID/hasMany-relation-name*

List belongsTo related model instances

List the related model instances for the given model identified by *instance-ID*, for hasMany relationship.

GET /*model-name*/*instance-ID*/*belongsToRelationName*

Aggregate models following relations

It's often desirable to include related model instances in the response to a query so that the client doesn't have to make multiple calls.

GET /*model1-name?filter[include]=...*

Arguments

- *include* - The object that describes a hierarchy of relations to be included

Example

Retrieve all members including the posts with the following request:

`GET /api/members?filter[include]=posts`

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2
      }
    ]
  },
... ]
```


The following request retrieves all members, including the posts, which further includes the author:

`GET /api/members?filter[include][posts]=author`

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2,
        "author": {
          "name": "Member B",
          "age": 22,
          "id": 2
        }
      }
    ]
  }, ... ]
```

The following request retrieves all members who are 21 years old, including the posts, which further includes the author:

`GET /api/members?filter[include][posts]=author&filter[where][age]=21`

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  }
]
```

The following request retrieves two members, including the posts, which further includes the author:

`GET /api/members?filter[include][posts]=author&filter[limit]=2`

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2,
        "author": {
          "name": "Member B",
          "age": 22,
          "id": 2
        }
      }
    ]
  }
]
```

The following request retrieves all members, including the posts and passports.

`GET /api/members?filter[include]=posts&filter[include]=passports`

The API returns the following JSON:


```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1
      }
    ],
    "passports": [
      {
        "number": "1",
        "id": 1,
        "ownerId": 1
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2
      }
    ],
    "passports": [
      {
        "number": "2",
        "id": 2,
        "ownerId": 2
      }
    ]
  }, ... ]
```


Errors

None

Role REST API

All of the endpoints in the Role REST API are inherited from the generic PersistedModel REST 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.	JSON object (in request body)
`/roles`	GET	Deny	Find instances of roles that match specified filter.	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/roles`	PUT	Deny	Update / insert role instance and persist to data source.	JSON object (in request body)
`/roles/`*`id`*	GET	Deny	Find role by ID: Return data for the specified role instance ID.	*id*, the role instance ID (in URI path)
`/roles/`*`id`*	PUT	Deny	Update attributes for specified role ID and persist.	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/roles/`*`id`*	DELETE	Deny	Delete role with specified instance ID.	*id*, role ID (in URI path)
`/roles/`*`id`*`/exists`	GET	Deny	Check instance existence: Return true if specified role ID exists.	URI path: • *id* - Model instance ID
`/roles/count`	GET	Deny	Return the number of role instances that matches specified where clause.	Where filter specified in query parameter
`/roles/findOne`	GET	Deny	Find first role instance that matches specified filter.	Same as Find matching instances.

User REST API

You can use the **StrongLoop API Explorer** to quickly construct and make requests to a LoopBack app running on the server. If a LoopBack app is running on `localhost` at port `3000`, then by default API Explorer is available at http://localhost:3000/explorer/.

All of the endpoints in the table below are inherited from PersistedModel REST API, except for the following:

- Log in user
- Log out user
- Confirm email address
- Reset password

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments

`/users`	POST	Allow	Add user instance and persist to data source. Inherited from PersistedModel API.	JSON object (in request body) providing User object properties: `username`, `password`, `email`. LoopBack sets values for `emailVerified` and `verificationToken`. NOTE: A value for `username` is not required, but a value for `email` is. LoopBack validates a unique value for `password` is provided. LoopBack does not automatically maintain values of the `created` and `lastUpdated` properties; you can set them manually if you wish.
`/users`	GET	Deny	Find matching instances of users that match specified filter. Inherited from PersistedModel API.	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/users`	PUT	Deny	Update / insert user instance and persist to data source. Inherited from PersistedModel API.	JSON object (in request body) Same as for `POST /users`
`/users/`*`id`*	GET	Deny	Find user by ID: Return data for the specified user ID. Inherited from PersistedModel API.	*id*, the user ID (in URI path)
`/users/`*`id`*	PUT	Deny	Update user attributes for specified user ID and persist. Inherited from PersistedModel API.	Query parameters: • data An object containing property name/value pairs • *id* The model id
`/users/`*`id`*	DELETE	Deny	Delete user with specified instance ID. Inherited from PersistedModel API.	*id*, user ID (in URI path)
`/users/`*`id`*`/accessTokens`	GET	Deny	Returns access token for specified user ID.	• *id*, user ID, in URI path • where in query parameters
`/users/`*`id`*`/accessTokens`	POST	Deny	Create access token for specified user ID.	*id*, user ID, in URI path
`/users/`*`id`*`/accessTokens`	DELETE	Deny	Delete access token for specified user ID.	*id*, user ID, in URI path
`/users/confirm`	GET	Deny	Confirm email address for specified user.	Query parameters: • uid • token • redirect

`/users/count`	GET	Deny	Return number of user instances that match specified where clause. Inherited from PersistedModel API.	"Where" filter specified in query parameter
`/users/id/exists`	GET	Deny	Check instance existence: Return true if specified user ID exists. Inherited from PersistedModel API.	URI path: • *users* - Model name • *id* - Model instance ID
`/users/findOne`	GET	Deny	Find first user instance that matches specified filter. Inherited from PersistedModel API.	Same as Find matching instances.
`/users/login[?include=user]`	POST	Allow	Log in the specified user.	Username and password in POST body. If query parameter is `include=user`, then returns the user object.
`/users/logout`	POST	Allow	Log out the specified user.	Access token in POST body.
`/users/reset`	POST		Reset password for the specified user.	In POST body

Log in user

```
POST /users/login
```

You must provide a username or an email, and the password in the request body. 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).

You may also specify how long you would like the access token to be valid by providing a `ttl` (time to live) property with value in **milliseconds**.

Example

Request URL: `POST  http://localhost:3000/users/login`
Request body:

```
{
    "email": "foo@bar.com",
    "password": "bar",
    "ttl": 1209600000
  }
```

Response status code: `200`
Response body:

```
{
  "id": "PqosmmPCdQgwerDYwQcVCxMakGQV0BSUwG4iGVLvD3XUYZRQky1cmG8ocmzsVpEE",
  "ttl": 1209600,
  "created": "2014-12-23T08:31:33.464Z",
  "userId": 1
}
```

The access token for the user's session is returned in the `id` key of the response. It must be specified in the query parameter `access_token` for all the APIs that requires the user to be logged in. For example:

`http://localhost:3000/api/Users/logout?access_token=PqosmmPCdQgwerDYwQcVCxMakGQV0BSUwG4iGVLvD3XUYZRQky1cmG8ocmzsVpEE`.

Log out user

```
POST /users/logout
```

Example

Request URL: `POST http://localhost:3000/api/Users/logout?access_token=PqosmmPCdQgwerDYwQcVCxMakGQV0BSUwG4iGVLvD3XUYZRQky1cmG8ocmzsVpEE.`

Response status code: `204`

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.

```
GET /users/confirm
```

Parameters

Query parameters:

- uid
- token
- redirect

Return value

If token invalid: HTTP 400

If user not found: HTTP 404

If successful: HTTP 204

Reset password

```
POST  /users/reset
```

Parameters

POST payload:

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

Return value

```
200 OK
```

You must the handle the '`resetPasswordRequest`' event to send a reset email containing an access token to the correct user.

The example below shows how to get an access token that a user can use to reset their password.

common/models/user.js

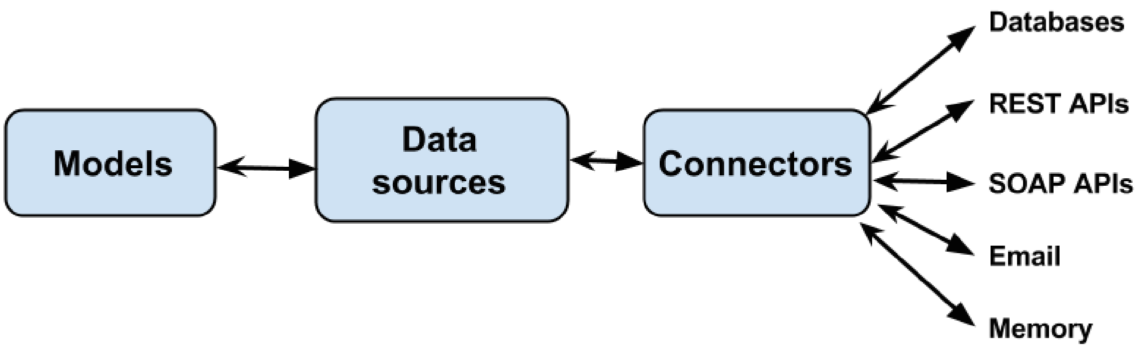
```
User.on('resetPasswordRequest', function (info) {
  console.log(info.email); // the email of the requested user
  console.log(info.accessToken.id); // the temp access token to allow password reset

  // requires AccessToken.belongsTo(User)
  info.accessToken.user(function (err, user) {
    console.log(user); // the actual user
  });
});
```

See also Verifying email addresses (Registering users).

Connecting models to data sources

Overview

LoopBack models connect to backend systems such as databases via *data sources* that provide create, retrieve, update, and delete (CRUD) functions. LoopBack also generalizes other backend services, such as REST APIs,



SOAP web services, and storage services, and so on, as data sources.

Data sources are backed by *connectors* that implement the data exchange logic using database drivers or other client APIs. In general, applications don't use connectors directly, rather they go through data sources using the DataSource and PersistedModel APIs.

Basic procedure

To connect a model to a data source, follow these steps:

1. Use the data source generator, `slc loopback:datasource`, to create a new data source. For example:

```
$ slc loopback:datasource
? Enter the data-source name: mysql-corp
? Select the connector for mysql: MySQL (supported by StrongLoop)
```

   Follow the prompts to name the datasource and select the connector to use. See Connecting models to data sources for more information. This adds the new data source to `datasources.json`.
2. Edit `server/datasources.json` to add the necessary authentication credentials: typically hostname, username, password, and database name. For example:

   server/datasources.json

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

   For information on the specific properties that each connector supports, see:
   - Cloudant connector
   - DB2 connector
   - Memory connector
   - MongoDB connector
   - MySQL connector
   - Oracle connector
   - PostgreSQL connector
   - Redis connector
   - SQL Server connector

3. Install the corresponding connector as a dependency of your app with `npm`, for example:

```
$ cd <your-app>
$ npm install --save loopback-connector-mysql
```

   See Connectors for the list of connectors.
4. Use the model generator, `slc loopback:model`, to create a model. When prompted for the data source to attach to, select the one you just created.

   > The model generator lists the memory connector, "no data source," and data sources listed in `datasources.json`. That's why you created the data source first in step 1.

```
$ slc loopback:model
? Enter the model name: myModel
? Select the data-source to attach myModel to: mysql (mysql)
? Select model's base class: PersistedModel
? Expose myModel via the REST API? Yes
? Custom plural form (used to build REST URL):
Let's add some test2 properties now.
...
```

You can also create models from an existing database; see Creating models for more information.

Connectors

The following LoopBack connectors are available:

Database connectors		
Connector	**Module**	**Installation**
Memory connector	Built in to LoopBack	Not required; suitable for development and debugging only.
MongoDB	loopback-connector-mongodb	`npm install --save loopback-connector-mongodb`
MySQL	loopback-connector-mysql	`npm install --save loopback-connector-mysql`
Oracle	loopback-connector-oracle	`npm install --save loopback-connector-oracle`
PostgreSQL	loopback-connector-postgresql	`npm install --save loopback-connector-postgresql`
SQL Server	loopback-connector-mssql	`npm install --save loopback-connector-mssql`
Other connectors		
Email connector	Built in to LoopBack	Not required
Push connector	loopback-component-push	`npm install --save loopback-component-push`
Remote connector	loopback-connector-remote	`npm install --save loopback-connector-remote`
REST	loopback-connector-rest	`npm install --save loopback-connector-rest`
SOAP	loopback-connector-soap	`npm install --save loopback-connector-soap`
Storage connector	loopback-component-storage	`npm install --save loopback-component-storage`

In addition to the connectors listed above that StrongLoop provides, Community connectors developed and maintained by the LoopBack community enable you to connect to CouchDB, Neo4j, Elasticsearch, and many others. See Community connectors for more information.

Installing a connector

Run `npm install --save` *`<connector-module>`* in your application root directory to add the dependency to `package.json`; for example, to install the Oracle database connector:

```
$ cd <your-app>
$ npm install --save loopback-connector-oracle
```

This command adds the following entry to `package.json`:

/package.json

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

Creating a data source

Use the Data source generator to create a new data source:

```
$ slc loopback:datasource
```

Follow the prompts to add the desired data source.

You can also create a data source programmatically; see Advanced topics: data sources for more information.

Data source properties

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

Property	Type	Description
connector	String	Connector name; one of: • "memory" • "loopback-connector-mongodb" or "mongodb" • "loopback-connector-mysql" or "mysql" • "loopback-connector-oracle" or "oracle" • "loopback-connector-postgresql" or "postgresql" • "loopback-connector-rest" or "rest" • "loopback-connector-mssql" or "mssql"
database	String	Database name
debug	Boolean	If true, turn on verbose mode to debug database queries and lifecycle.
host	String	Database host name
password	String	Password to connect to database
port	Number	Database TCP port
url	String	Combines and overrides `host`, `port`, `user`, `password`, and `database` properties. Only valid with MongoDB connector, PostgreSQL connector, and SQL Server connector.
username	String	Username to connect to database

Creating a database schema from models

LoopBack *auto-migration* creates a database schema based on your application's models. In relational databases, auto-migration creates a table for each model, and a column in the table for each property in the model. Auto-migration creates tables for all models attached to a data source, including built-in models

Once you have defined a model, LoopBack can create or update (synchronize) the database schemas accordingly, if you need to adjust the database to match the models. LoopBack provides two ways to synchronize model definitions with table schemas:

- **Auto-migrate**: Automatically create or re-create the table schemas based on the model definitions.
- **Auto-update**: Automatically alter the table schemas based on the model definitions.

Auto-migration will drop an existing table if its name matches a model name. When tables with data exist, use auto-update to avoid data loss.

Auto-migrate

StrongLoop Arc enables you to perform auto-migration without coding. For more information, see Creating and editing models (Migrating a model).

See also: automigrate() in LoopBack API reference.

The following data sources support auto-migration:

- Oracle
- PostgreSQL
- MySQL
- SQL Server
- MongoDB

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

/common/models/model.json

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

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

/common/models/model.js

```
var ds = require('../data-sources/db')('oracle');
var Customer = require('../models/customer');
ds.createModel(schema_v1.name, schema_v1.properties, schema_v1.options);

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

This creates the following objects in the Oracle database:

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

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

/common/models/model.json

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

MongoDB indexes

Running `autoMigrate()` creates missing indexes but it doesn't modify them if their definitions change. If a model's index definitions change, you must either modify them via the MongoDB shell, or delete them and re-create them. For more information, see the MongoDB documentation.

Auto-update

See also: See also autoupdate() in LoopBack API reference.

If there are existing tables in a database, running `autoMigrate()` will drop and re-create the tables: Therefore, data will be lost. To avoid this problem use `auto-update()`. Instead of dropping tables and recreating them, `autoupdate()` calculates the difference between the LoopBack model and the database table definition and alters the table accordingly. This way, the column data will be kept as long as the property is not deleted from the model.

For example:

/server/script.js

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

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

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

/server/script.js

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

Database connectors

- Cloudant connector
- DB2 connector
- Memory connector
- MongoDB connector
- MySQL connector
- Oracle connector
- PostgreSQL connector
- Redis connector
- SQL Server connector

LoopBack provides connectors for popular relational and NoSQL databases. These connectors implement CRUD operations as a common set of methods of PersistedModel. When you attach a model to a data source backed by one of the database connectors, the model automatically acquires the CRUD methods from PersistedModel. The data access methods on a persisted model are exposed to REST by default; see PersistedModel REST API for the endpoints.

You can connect models using relations to reflect relationships among data. For more information about relations, see Creating model relations.

Cloudant connector

- Overview
    - Key features
- Installation
- Configuring the Cloudant datasource
    - Model-specific configuration
    - Example

See also: Getting Started with LoopBack and IBM Cloudant

Overview

IBM Cloudant® is a NoSQL database platform built for the cloud. You can use Cloudant as a fully-managed DBaaS running on public cloud platforms like Bluemix, SoftLayer or via an on-premise version called Cloudant Local. For more information, see Getting Started with Cloudant NoSQL DB.

Key features

- Uses Cloudant Query (Lucene) to support ad-hoc searching.
- Loopback query support for fields, limit, order, skip and where filters.
- Performs query and filtering on the database for optimal efficiency.
- Uses different DB instances per model definition.
- Supports basic model discovery.

Installation

To install the connector, enter the following command when you are in the top level of your Loopback application directory:

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

The --save option automatically updates the dependency in the application's `package.json` file.

Configuring the Cloudant datasource

Use the data source generator to add the Cloudant data source to your application. The entry in the applications `server/datasources.json` w ill look like this:

```
"mydb": {
  "name": "mydb",
  "connector": "cloudant",
  "username": "XXXX-bluemix",
  "password": "YYYYYYYYYYYY",
  "database": "test"
}
```

Edit the `datasources.json` (or use StrongLoop Arc) to add other supported properties as required:

Property	Type	Description
database	String	Database name
username	String	Cloudant username, use either 'url' or username/password
password	String	Cloudant password
url	String	Cloudant URL containing both username and password
modelIndex	String	Specify the model name to document mapping, defaults to 'loopback__model__name'

Model-specific configuration

Per-model configuration is also supported for database selection and to specify different LoopBack model to document mappings:

common/models/MyUser.json

```
{
  "name": "MyUser",
  "base": "PersistedModel",
  "idInjection": true,
  ...
  "settings": {
    "cloudant": {
        "modelIndex": "myPropertyName",
        "database": "test2"
    }
  },
  ...
```

Model specific configuration settings:

Property	Type	Description
database	String	Database name
modelIndex	String	Specify the model name to document mapping, defaults to 'loopback__model__name'

Example

```
var DataSource = require ('loopback-datasource-juggler').DataSource,
    Cloudant   = require ('loopback-connector-cloudant');

var config = {
    username: 'XXXXX-bluemix',
    password: 'YYYYYYYYYYYYY',
    database: 'test'
};

var db = new DataSource (Cloudant, config);

User = db.define ('User', {
  name: { type: String },
  email: { type: String }
});

User.create ({
  name: "Tony",
  email: "tony@t.com"
}, function (err, user) {
  console.log (user);
});

User.find ({ where: { name: "Tony" }}, function (err, users) {
  console.log (users);
});

User.destroyAll (function () {
  console.log ('test complete');
})
```

DB2 connector

Overview

IBM® DB2® is the database of choice for robust, enterprise-wide solutions handling high-volume workloads. It is optimized to deliver industry-leading performance while lowering costs.

The LoopBack DB2 connector supports:

- All CRUD operations.
- Queries with fields, limit, order, skip and where filters.

The following features are not yet implemented:

- Model discovery
- Auto migration and update

For more information on using LoopBack on BlueMix, see Creating apps with LoopBack Starter.

Installation

Enter the following in the top-level directory of your LoopBack application:

```
$ npm install loopback-connector-db2 --save
```

The `--save` option adds the dependency to the application's package.json file.

Configuration

Use the data source generator to add the DB2 data source to your application. The entry in the applications `server/datasources.json` will look something like this:

server/datasources.json

```
"mydb": {
  "name": "mydb",
  "connector": "db2",
  "username": <username>,
  "password": <password>,
  "database": <database name>,
  "hostname": <db2 server hostname>,
  "port":     <port number>
}
```

Edit the `datasources.json` to add other supported properties as required:

Property	Type	Description
database	String	Database name
hostname	String	DB2 server hostname or IP address
password	String	DB2 password associated with the username above
port	String	DB2 server TCP port number
useLimitOffset	Boolean	LIMIT and OFFSET must be configured on the DB2 server before use (compatibility mode)

username	String	DB2 Username

Example

Here's an example of configuring the data source in JavaScript code:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var DB2 = require('loopback-connector-db2');

var config = {
  username: process.env.DB2_USERNAME,
  password: process.env.DB2_PASSWORD,
  hostname: process.env.DB2_HOSTNAME,
  port: 50000,
  database: 'SQLDB',
};

var db = new DataSource(DB2, config);

var User = db.define('User', {
  name: { type: String },
  email: { type: String },
});

db.autoupdate('User', function(err) {
  if (err) {
    console.log(err);
    return;
  }

  User.create({
    name: 'Tony',
    email: 'tony@t.com',
  }, function(err, user) {
    console.log(err, user);
  });

  User.find({ where: { name: 'Tony' }}, function(err, users) {
    console.log(err, users);
  });

  User.destroyAll(function() {
    console.log('example complete');
  });
});
```

Memory connector

Overview

LoopBack's built-in memory connector enables you to test your application without connecting to an actual persistent data source such as a database. Although the memory connector is very well tested it is not suitable for production.

The memory connector supports:

- Standard query and create, read, update, and delete (CRUD) operations, so you can test models against an in-memory data source.
- Geo-filtering when using the `find()` operation with an attached model. See GeoPoint class for more information on geo-filtering.

Limitations
The memory connector is designed for development and testing of a single-process application without setting up a database. It cannot be used in a cluster as the worker processes will have their own isolated data not shared in the cluster.

You can persist data between application restarts using the file property. See Data persistence for more information.

Creating a data source

By default, an application created with the Application generator has a memory data source defined; for example:

/server/datasources.json

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

Use the Data source generator to add a new memory data source to your application.

Memory connector properties

Property	Type	Description
name	String	Name by which you refer to the data source.
connector	String	Must be "memory" to use the memory connector.
file	String	Path to file where the connector will store data, relative to application root directory. NOTE: The connector will create the file if necessary, but the directory containing the file must exist.

If you specify the file property, the connector will save data there that will persist when you restart the application. Otherwise, the memory connector does not persist data after an application stops.

Data persistence

By default, data in the memory connector are transient. When an application using the memory connector exits, all model instances are lost. To maintain data across application restarts, specify a JSON file in which to store the data with the `file` property when creating the data source.

The simplest way to do this is by editing `server/datasources.json`; for example:

server/datasources.json

```
{

  "db": {
    "name": "db",
    "connector": "memory",
    "file": "mydata.json"
  }
}
```

You can also set the persistence file in a boot script; for example:

server/boot/script.js

```
var memory = loopback.createDataSource({
  connector: loopback.Memory,
  file: "mydata.json"
});
```

When the application exits, the memory connector will then store data in the `mydata.json` file, and when it restarts will load the saved data from that file.

MongoDB connector

Related articles:

See also: loopback-example-mongodb

The MongoDB connector requires MongoDB 2.6 - 3.x.

Installation

The MongoDB connector indirectly uses bson, that requires you to have a standard set of compiler tools on your system. See Installing compiler tools for details.

In your application root directory, enter:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a MongoDB data source

Use the Data source generator to add a MongoDB data source to your application. The entry in the application's `/server/datasources.json` will look like this:

/server/datasources.json

```
"mydb": {
    "name": "mydb",
    "connector": "mongodb",
}
```

Edit `datasources.json` to add other properties that enable you to connect the data source to a MongoDB database.

Properties

Property	Type	Description
connector	String	Connector name, either "loopback-connector-mongodb" or "mongodb"
database	String	Database name
debug	Boolean	If true, turn on verbose mode to debug database queries and lifecycle.
host	String	Database host name or IP address.
password	String	Password to connect to database, if required.
port	Number	Database TCP port
url	String	Connection string URI; see http://docs.mongodb.org/manual/reference/connection-string/. See Replica set configuration below.
username	String	Username to connect to database, if required.

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

For example:

/server/datasources.json

```
{
  "mongodb_dev": {
    "name": "mongodb_dev",
    "connector": "mongodb",
    "host": "127.0.0.1",
    "database": "devDB",
    "username": "devUser",
    "password": "devPassword",
    "port": 27017
  },
  "mongodb_staging": {
    "name": "mongodb_staging",
    "connector": "mongodb",
    "host": "127.0.0.1",
    "database": "stagingDB",
    "username": "stagingUser",
    "password": "stagingPassword",
    "port": 27017
  }
}
```

Using the MongoDB connector

LoopBack does not currently support property mapping for MongoDB; you can customize only collection names.

Using MongoDB operators in update operations

Enable the `allowExtendedOperators` option to include MongoDB operators in update operations. There are two ways to enable the allowExtendedOperators flag: in the model definition JSON file and as an option passed to the update method.

To set the option in the model definition file, set the property `settings.mongodb.allowExtendedOperators` to true. For example:

common/models/my-model.json

```
{
  "name": "MyModel",
  "settings": {
    "mongodb": {
      "allowExtendedOperators": true
    }
  ...
```

To set the option when calling an update method from code, set it in the options object; for example the following call to `updateAll()` uses the `$rename` operator:

```
User.updateAll(
  { name: 'Al' },
  { '$rename': { name: 'firstname' }},
  { allowExtendedOperators: true }
);
```

The following operators are compatible and a default value of $set will be applied if no operator is used:

Update Operator	Description
$inc	Increments the value of the field by the specified amount.
$mul	Multiplies the value of the field by the specified amount.
$rename	Renames a field.
$setOnInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
$set	Sets the value of a field in a document. (Default)
$unset	Removes the specified field from a document.
$min	Only updates the field if the specified value is less than the existing field value.
$max	Only updates the field if the specified value is greater than the existing field value.

Customizing the collection name

You might want to customize the collection name for a LoopBack model. It can be done in the model definition JSON file. In the example below, the Post model will be mapped to the PostCollection collection in MongoDB.

/common/models/model.json

```
{
  "name": "Post",
  "mongodb": {
    "collection": "PostCollection"
  },
  "properties": {
    ...
  }
}
```

Replica set configuration

The LoopBack MongoDB connector supports the replica set configuration using the MongoDB connection string URI format. For example, here is a snippet for the data source configuration:

/server/datasources.json

```
{
 “connector": “mongodb”,
 “url”: "mongodb://example1.com,example2.com,example3.com/?readPreference=secondary"
}
```

Handling MongoDB server restart

MongoDB has some options to control the `reconnect`.

- auto_reconnect: true, // default to true
- reconnectTries: 30, // default to 30
- reconnectInterval: 1000 // default to 1000ms

By default, after a connection failure, mongodb driver tries to reconnect up to 30 times, once per second. If your server doesn't come back within 30 seconds, the driver gives up. You can bump up the `reconnectTries` or `reconnectInterval`.

For example:

server/datasources.json

```
{
  "accountDB": {
    "name": "accountDB",
    "connector": "mongodb",
    "host": "localhost",
    "port": 27017,
    "server": {
      "auto_reconnect": true,
      "reconnectTries": 100,
      "reconnectInterval": 1000
    },
    "database": "demo"
  }
}
```

About MongoDB _id field

MongoDB uses a specific ID field with BSON `ObjectID` type, named `_id`

The MongoDB connector does not expose the MongoDB `_id` field, to be consistent with other connectors. Instead, it is transparently mapped to the `id` field, which is declared by default in the model if you do not define any `id`.

To access the `_id` property, you must define it explicitly as your model ID, along with its type; For example:

/server/script.js

```
var ds = app.dataSources.db;
MyModel = ds.createModel('mymodel', {
    _id: { type: ds.ObjectID, id: true }
});
```

Example with a Number _id :

/server/script.js

```
MyModel = ds.createModel('mymodel', {
    _id: { type: Number, id: true }
});
```

MongoDB query examples

Query with logical operators (since v1.2.3)

MongoDB supports queries with logical operators such as $and, $or, and $nor. See Logical Query Operators (MongoDB documentation) for more information.

To use the logical operators with LoopBack's query filter, use a `where` clause as follows (for example):

/server/script.js

```
// Find posts that have title = 'My Post' and content = 'Hello'
Post.find({where: {and: [{title: 'My Post'},
                         {content: 'Hello'}]}},
          function (err, posts) {
            ...
});

// Find posts that either have title = 'My Post' or content = 'Hello'
Post.find({where: {or: [{title: 'My Post'},
                        {content: 'Hello1'}]}},
          function (err, posts) {
            ...
});

// Find posts that neither have title = 'My Post1' nor content = 'Hello1'
Post.find({where: {nor: [{title: 'My Post1'},
                         {content: 'Hello1'}]}},
          function (err, posts) {
            ...
});
```

Case-insensitive query

Since version 1.8.0, the MongoDB connector supports using a regular expression as the value of the `like` operator; this enables case-insensitive queries, for example:

```
var pattern = new RegExp('.*'+query+'.*', "i"); /* case-insensitive RegExp search */
Post.find({ where: {title: { like: pattern} } };
```

Using the REST API:

```
?filter={"where":{"title":{"like":"someth.*","options":"i"}}}
```

Aggregate (group by) query

To perform an aggregate (group by) query on a MongoDB data source, follow this example:

```
var bookCollection = Book.getDataSource().connector.collection(Book.modelName);
bookCollection.aggregate({
  $match: {}
},
{
  $group: {
    _id: { category: "$category", author: "$author" },
    total: { $sum: 1 }
  }
},
...,
function(err, groupByRecords) {
  if(err) {
    next(err);
  } else {
    next();
  }
});
```

Connecting to MongoDB

This article is reproduced from loopback-example-mongodb

Error rendering macro 'markdown-url' : URL was not found.

Using MongoLab

If you are using MongoLab to host your MongoDB database, use the LoopBack `url` property to configure your data source, since the connection string is dynamically generated. For example, the entry in `datasources.json` might look like this:

/server/datasources.json

```
"mongodb": {
   "defaultForType": "mongodb",
   "connector": "loopback-connector-mongodb",
   "url": "mongodb://localhost:27017/mydb"
}
```

For information on how to get your connection URI, see the MongoLab documentation.

MySQL connector

Related articles:

See also:

- loopback-example-mysql
- Discovering models from relational databases
- Database discovery API

The MySQL connector requires MySQL 5.0+.

Installation

In your application root directory, enter this command to install the connector:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a MySQL data source

Use the Data source generator to add a MySQL data source to your application. The entry in the application's `/server/datasources.json` will look like this:

/server/datasources.json

```
"mydb": {
    "name": "mydb",
    "connector": "mysql",
}
```

Edit `datasources.json` to add other properties that enable you to connect the data source to a MySQL database.

Properties

Property	Type	Description
connector	String	Connector name, either "loopback-connector-mysql" or "mysql"
database	String	Database name
debug	Boolean	If true, turn on verbose mode to debug database queries and lifecycle.
host	String	Database host name
password	String	Password to connect to database
port	Number	Database TCP port
username	String	Username to connect to database

In addition to these properties, you can use additional parameters supported by `node-mysql`, for example `password` and `collation`. `Collation` currently defaults to `utf8_general_ci`. The `collation` value will also be used to derive the connection charset.

Type mappings

See LoopBack types for details on LoopBack's data types.

LoopBack to MySQL types

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

MySQL to LoopBack types

MySQL Type	LoopBack Type
CHAR	String
CHAR(1)	Boolean
VARCHAR TINYTEXT MEDIUMTEXT LONGTEXT TEXT ENUM SET	String
TINYBLOB MEDIUMBLOB LONGBLOB BLOB BINARY VARBINARY BIT	Node.js Buffer object
TINYINT SMALLINT INT MEDIUMINT YEAR FLOAT DOUBLE NUMERIC DECIMAL	Number
DATE TIMESTAMP DATETIME	Date

Using the datatype field/column option with MySQL

`loopback-connector-mysql` allows mapping of LoopBack model properties to MySQL columns using the 'mysql' property of the property definition. For example:

/common/models/model.json

```
"locationId":{
    "type":"String",
    "required":true,
    "length":20,
    "mysql":
    {
        "columnName":"LOCATION_ID",
        "dataType":"VARCHAR2",
        "dataLength":20,
        "nullable":"N"
    }
}
```

You can also use the dataType column/property attribute to specify what MySQL column type to use for many loopback-datasource-juggler types. The following type-dataType combinations are supported:

- Number
- integer
- tinyint
- smallint
- mediumint
- int
- bigint

Use the `limit` option to alter the display width. Example:

```
`{ count : { type: Number, dataType: 'smallInt' }}`
```

Floating-point types

For Float and Double data types, use the `precision` and `scale` options to specify custom precision. Default is (16,8). For example:

```
{ average : { type: Number, dataType: 'float', precision: 20, scale: 4 }}
```

Fixed-point exact value types

For Decimal and Numeric types, use the `precision` and `scale` options to specify custom precision. Default is (9,2). These aren't likely to function as true fixed-point.

Example:

```
{ stdDev : { type: Number, dataType: 'decimal', precision: 12, scale: 8 }}
```

Other types

Convert String / DataSource.Text / DataSource.JSON to the following MySQL types:

- varchar
- char
- text
- mediumtext
- tinytext
- longtext

Example:

```
{ userName : { type: String, dataType: 'char', limit: 24 }}
```

Example:

```
{ biography : { type: String, dataType: 'longtext' }}
```

Convert JSON Date types to datetime or timestamp

Example:

```
{ startTime : { type: Date, dataType: 'timestamp' }}
```

Enum

Enums are special. Create an Enum using Enum factory:

```
var MOOD = dataSource.EnumFactory('glad', 'sad', 'mad');
MOOD.SAD; // 'sad'
MOOD(2); // 'sad'
MOOD('SAD'); // 'sad'
MOOD('sad'); // 'sad'
{ mood: { type: MOOD }}
{ choice: { type: dataSource.EnumFactory('yes', 'no', 'maybe'), null: false }}
```

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Discovering models from relational databases and Database discovery API.

Connecting to MySQL

This tutorial is reproduced from loopback-example-mysql.

Error rendering macro 'markdown-url' : URL was not found.

Oracle connector

The Oracle connector does not work on Mac OSX 10.11 (El Capitan) due to SIP runtime protections and restrictions. Specifically, DYLD environment variables are now ignored; hence, rendering the Oracle connector inoperable due to the DYLD_LIBRARY_PATH dependency. For a workaround, see How to Disable SIP in OS X El Capitan.

Related articles:

See also:

- loopback-example-oracle
- Database discovery API

The Oracle connector requires Oracle 8.x - 12.x.

Installation

In your application root directory, enter this command to install the connector:

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

See Installing the Oracle connector for further installation instructions.

On 64-bit Windows systems, the Oracle connector runs only on 64-bit version of Node.js.

Connector properties

The connector properties depend on naming methods you use for the Oracle database. LoopBack supports three naming methods:

- Easy connect: host/port/database.
- Local naming (TNS): alias to a full connection string that can specify all the attributes that Oracle supports.
- Directory naming (LDAP): directory for looking up the full connection string that can specify all the attributes that Oracle supports.

Easy Connect

Easy Connect is the simplest form that provides out-of-the-box TCP/IP connectivity to databases. The data source then has the following settings.

Property	Type	Default	Description
host or hostname	String	localhost	Host name or IP address of the Oracle database server
port	Number	1521	Port number of the Oracle database server
username or user	String		User name to connect to the Oracle database server
password	String		Password to connect to the Oracle database server
database	String	XE	Oracle database listener name

For example:

/server/datasources.json

```
{
  "demoDB": {
    "connector": "oracle",
    "host": "oracle-demo.strongloop.com",
    "port": 1521,
    "database": "XE",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

Local and directory naming

Both local and directory naming require that you place configuration files in a TNS admin directory, such as `/oracle/admin`.

sqlnet.ora (specifying the supported naming methods)

```
NAMES.DIRECTORY_PATH=(LDAP,TNSNAMES,EZCONNECT)
```

tnsnames.ora (mapping aliases to connection strings)

```
demo1=(DESCRIPTION=(CONNECT_DATA=(SERVICE_NAME=))(ADDRESS=(PROTOCOL=TCP)(HOST=demo.str
ongloop.com)(PORT=1521)))
```

ldap.ora (configuring the LDAP server)

```
DIRECTORY_SERVERS=(localhost:1389)
DEFAULT_ADMIN_CONTEXT="dc=strongloop,dc=com"
DIRECTORY_SERVER_TYPE=OID
```

Set up TNS_ADMIN environment variable

For the Oracle connector to pick up the configurations, you must set the environment variable 'TNS_ADMIN' to the directory containing the `.ora` files.

```
export TNS_ADMIN=<directory containing .ora files>
```

Now you can use either the TNS alias or LDAP service name to configure a data source:

```
var ds = loopback.createDataSource({
"tns": "demo", // The tns property can be a tns name or LDAP service name
"username": "demo",
"password": "L00pBack"
});
```

Here is an example for `datasources.json`:

/server/datasources.json

```
{
  "demoDB": {
    "connector": "oracle",
    "tns": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

Connection pooling options

Property name	Description	Default value
minConn	Maximum number of connections in the connection pool	1
maxConn	Minimum number of connections in the connection pool	10
incrConn	Incremental number of connections for the connection pool.	1
timeout	Time-out period in seconds for a connection in the connection pool. The Oracle connector will terminate connections in this connection pool that are idle longer than the time-out period.	10

For example,

/server/datasources.json

```
{
  "demoDB": {
    "connector": "oracle",
    "minConn":1,
    "maxConn":5,
    "incrConn":1,
    "timeout": 10,
    ...
  }
}
```

Model definition for Oracle

The model definition consists of the following properties:

- name: Name of the model, by default, it's the camel case of the table.
- options: Model level operations and mapping to Oracle schema/table.
- properties: Property definitions, including mapping to Oracle column.

/common/models/model.json

```
{
    "name":"Inventory",
    "options":{
      "idInjection":false,
      "oracle":{
        "schema":"STRONGLOOP",
        "table":"INVENTORY"
      }
    },
    "properties":{
      "productId":{
        "type":"String",
        "required":true,
        "length":20,
        "id":1,
        "oracle":{
          "columnName":"PRODUCT_ID",
          "dataType":"VARCHAR2",
          "dataLength":20,
          "nullable":"N"
        }
      },
      "locationId":{
        "type":"String",
        "required":true,
        "length":20,
        "id":2,
        "oracle":{
          "columnName":"LOCATION_ID",
          "dataType":"VARCHAR2",
          "dataLength":20,
          "nullable":"N"
        }
      },
      "available":{
        "type":"Number",
        "required":false,
        "length":22,
        "oracle":{
          "columnName":"AVAILABLE",
          "dataType":"NUMBER",
          "dataLength":22,
          "nullable":"Y"
        }
      },
      "total":{
        "type":"Number",
        "required":false,
        "length":22,
        "oracle":{
          "columnName":"TOTAL",
          "dataType":"NUMBER",
          "dataLength":22,
          "nullable":"Y"
        }
      }
    }
  }
```

Type mapping

See LoopBack types for details on LoopBack's data types.

JSON to Oracle Types

LoopBack Type	Oracle Type
String JSON Text default	VARCHAR2 Default length is 1024
Number	NUMBER
Date	DATE
Timestamp	TIMESTAMP(3)
Boolean	CHAR(1)

Oracle Types to JSON

Oracle Type	LoopBack Type
CHAR(1)	Boolean
CHAR(n) VARCHAR VARCHAR2, LONG VARCHAR NCHAR NVARCHAR2	String
LONG, BLOB, CLOB, NCLOB	Node.js Buffer object
NUMBER INTEGER DECIMAL DOUBLE FLOAT BIGINT SMALLINT REAL NUMERIC BINARY_FLOAT BINARY_DOUBLE UROWID ROWID	Number
DATE TIMESTAMP	Date

Destroying models

Destroying models may result in errors due to foreign key integrity. Make sure to delete any related models first before calling delete on model's with relationships.

Auto-migrate / Auto-update

LoopBack *auto-migration* creates a database schema based on your application's models. Auto-migration creates a table for each model, and a column in the table for each property in the model. Once you have defined a model, LoopBack can create or update (synchronize) the database schemas accordingly, if you need to adjust the database to match the models. See Creating a database schema from models for more information.

After making changes to your model properties call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on new models since it will drop existing tables.

LoopBack Oracle connector creates the following schema objects for a given model:

- A table, for example, PRODUCT
- A sequence for the primary key, for example, PRODUCT_ID_SEQUENCE
- A trigger to generate the primary key from the sequnce, for example, PRODUCT_ID_TRIGGER

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Creating a database schema from models.

Installing the Oracle connector

Overview

The Oracle connector depends on strong-oracle module as the Node.js driver for Oracle databases. Since strong-oracle is a C++ addon, the installation usually requires the presence of C++ development tools to compile and build the module from source code. At runtime, strong-oracle also requires dynamic libraries from Oracle Database Instant Client. To simplify the whole process, we use a helper module LoopBack Oracle Installer to take care of the binary dependencies. 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.

To install the Oracle connector, use the following command:

shell

```
npm install loopback-connector-oracle --save
```

If you need to use the Oracle driver directly, see https://github.com/strongloop/strong-oracle

Post installation setup

Before you run the application, you **MUST** configure the environment variable depending on the target platform to make sure the dynamic libraries from Oracle Instant Client will be available to your Node process.

MacOS X or Linux

During npm install, the change is made in `$HOME/strong-oracle.rc`.

```
export DYLD_LIBRARY_PATH="$DYLD_LIBRARY_PATH:/Users/<user>/<myapp>/node_modules/loopback-connector-oracle/node_modules/instantcl
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/Users/<user>/<myapp>/node_modules/loopback-connector-oracle/node_modules/instantclient" (I
```

libaio requirement for Linux

```
libaio library is required on Linux systems and you might have to install
it.

On Ubuntu/Debian:

sudo apt-get install libaio1
```

```
On Fedora/CentOS/RHEL:

sudo yum install libaio
```

To activate the strong-oracle settings for your terminal window, add the following statements to $HOME/.bash_profile (or .profile depending on what shell you use):

~/.bashrc | ~/.bash_profile

```
if [ -f ~/strong-oracle.rc ]; then
  source ~/strong-oracle.rc
fi
```

You need to open a terminal window or run *source ~/.bash_profile* to make the change take effect.

Windows

The change is made to the PATH environment variable for the logged in user. Please note the PATH setting will NOT be effective immediately. You have to activate it using one of the methods below:

1. Log off the current user session and log in.
2. Open Control Panel --> System --> Advanced System Settings --> Environment Variables. Examine the Path under User variables, and click OK to activate it. You need to open a new Command Prompt. Please run 'path' command to verify.

Installation from behind a proxy server

This feature is supported by loopback-oracle-installer vesion 1.1.3 or later.

If your system is behind a corporate HTTP/HTTPS proxy to access the internet, you'll need to set the proxy for npm before running 'npm install'. For example,

shell

```
$ npm config set proxy http://proxy.mycompany.com:8080
$ npm config set https-proxy http://https-proxy.mycompany.com:8080
```

If the proxy url requires username/password, you can use the following syntax:

shell

```
$ npm config set proxy http://youruser:yourpass@proxy.mycompany.com:8080
$ npm config set https-proxy http://youruser:yourpass@https-proxy.mycompany.com:8080
```

The proxy can also be set as part of the npm command as follows:

```
$ npm --proxy=http://proxy.mycompany.com:8080 install
$ npm --https-proxy=http://https-proxy.mycompany.com:8080 install
```

Please note that npm's default value for proxy is from the HTTP_PROXY or http_proxy environment variable. And the default value for https-proxy is from the HTTPS_PROXY, https_proxy, HTTP_PROXY, or http_proxy environment variable. So you can configure the proxy using environment variables too.

Linux or Mac:

shell

```
HTTP_PROXY=http://proxy.mycompany.com:8080 npm install
```

Windows:

shell

```
set HTTP_PROXY=http://proxy.mycompany.com:8080
npm install
```

Connecting to Oracle

This article is reproduced from loopback-example-oracle

Error rendering macro 'markdown-url' : URL was not found.

PostgreSQL connector

Related articles:

See also:

- loopback-example-postgresql
- Database discovery API

The PostgreSQL connector requires PostgreSQL 8.x or 9.x.

Installation

In your application root directory, enter this command to install the connector:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the Data source generator to add a PostgreSQL data source to your application.

The entry in the application's `server/datasources.json` will look like this:

/server/datasources.json

```
"mydb": {
    "name": "mydb",
    "connector": "postgresql"
}
```

Edit `datasources.json` to add other properties that enable you to connect the data source to a PostgreSQL database.

Properties

Property	Type	Description
connector	String	Connector name, either "loopback-connector-postgresql" or "postgresql"
database	String	Database name
debug	Boolean	If true, turn on verbose mode to debug database queries and lifecycle.
host	String	Database host name
password	String	Password to connect to database
port	Number	Database TCP port
url	String	Use instead of the `host`, `port`, `user`, `password`, and `database` properties. For example: 'postgres://test:mypassword@localhost:5432/dev'.
username	String	Username to connect to database

By default, the 'public' schema is used for all tables.

Connecting to UNIX domain socket

A common PostgreSQL configuration is to connect to the UNIX domain socket `/var/run/postgresql/.s.PGSQL.5432` instead of using the TCP/IP port. For example:

```
{
  "postgres": {
    "host": "/var/run/postgresql/",
    "port": "5432",
    "database": "dbname",
    "username": "dbuser",
    "password": "dbpassword",
    "name": "postgres",
    "debug": true,
    "connector": "postgresql"
  }
}
```

Defining models

The model definition consists of the following properties.

Property	Default	Description
name	Camel-case of the database table name	Name of the model.

options	N/A	Model level operations and mapping to PostgreSQL schema/table
properties	N/A	Property definitions, including mapping to PostgreSQL column

For example:

/common/models/model.json

```
{"name": "Inventory",
   "options": {
      "idInjection": false,
      "postgresql": {
        "schema": "strongloop",
        "table": "inventory"
      }
   },
   "properties": {
      "id": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "postgresql": {
          "columnName": "id",
          "dataType": "character varying",
          "dataLength": 64,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "NO"
        }
      },
      "productId": {
        "type": "String",
        "required": false,
        "length": 20,
        "precision": null,
        "scale": null,
        "id": 1,
        "postgresql": {
          "columnName": "product_id",
          "dataType": "character varying",
          "dataLength": 20,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "YES"
        }
      },
      "locationId": {
        "type": "String",
        "required": false,
        "length": 20,
        "precision": null,
        "scale": null,
        "id": 1,
        "postgresql": {
          "columnName": "location_id",
          "dataType": "character varying",
```

```
        "dataLength": 20,
        "dataPrecision": null,
        "dataScale": null,
        "nullable": "YES"
      }
    },
    "available": {
      "type": "Number",
      "required": false,
      "length": null,
      "precision": 32,
      "scale": 0,
      "postgresql": {
        "columnName": "available",
        "dataType": "integer",
        "dataLength": null,
        "dataPrecision": 32,
        "dataScale": 0,
        "nullable": "YES"
      }
    },
    "total": {
      "type": "Number",
      "required": false,
      "length": null,
      "precision": 32,
      "scale": 0,
      "postgresql": {
        "columnName": "total",
        "dataType": "integer",
        "dataLength": null,
        "dataPrecision": 32,
        "dataScale": 0,
        "nullable": "YES"
```

```
        }
      }
    }}
```

Destroying models

If you destroy models, you may get errors due to foreign key integrity. Make sure to delete any related models first before calling `delete()` on models that have relationships.

Auto-migrate and auto-update

After making changes to your model properties, you must call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on new models since it will drop existing tables. These methods will

- Define a primary key for the properties whose `id` property is true (or a positive number).
- Create a column with ‘SERIAL’ type if the `generated` property of the `id` property is true.

See Creating a database schema from models for more information.

Type mapping

See LoopBack types for details on LoopBack's data types.

LoopBack to PostgreSQL types

LoopBack Type	PostgreSQL Type
String JSON Text Default	VARCHAR2 Default length is 1024
Number	INTEGER
Date	TIMESTAMP WITH TIME ZONE
Boolean	BOOLEAN

PostgreSQL types to LoopBack

PostgreSQL Type	LoopBack Type
BOOLEAN	Boolean
VARCHAR CHARACTER VARYING CHARACTER CHAR TEXT	String
BYTEA	Node.js Buffer object
SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL DOUBLE SERIAL BIGSERIAL	Number
DATE TIMESTAMP TIME	Date

POINT	GeoPoint

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Database discovery API.

Connecting to PostgreSQL

This article is reproduced from loopback-example-postgresql

Error rendering macro 'markdown-url' : URL was not found.

Redis connector

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see StrongLoop Labs.

StrongLoop Labs

The Redis connector requires Redis 3.0.3+.

Installation

In your application root directory, enter this command to install the connector:

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

This will install the module and add it as a dependency to the application's `package.json` file.

Creating a Redis data source

Use the data source generator to add a Redis data source to your application. When prompted for the connector, choose **other,** then enter **redis** for the connector name. The entry in the application's `server/datasources.json` will look like this:

server/datasources.json

```
"redisDS": {
    "name": "redisDS",
    "connector": "redis",
}
```

Edit `datasources.json` to add other properties that enable you to connect the data source to a Redis database.

Properties

Property	Type	Description

connector	String	Connector name, either "loopback-connector-redis" or "redis"
database	String	Database name
host	String	Database host name
password	String	Password to connect to database
port	Number	Database TCP port
url	String	Use instead host and port properties.
username	String	Username to connect to database

SQL Server connector

Related articles:

See also:

- loopback-example-mssql
- Database discovery API

The SQL Server connector requires SQL Server 2005+.

Installation

In your application root directory, enter:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a SQL Server data source

Use the Data source generator to add a SQL Server data source to your application. The generator will add the following entry to the `/server/datasources.json` file:

/server/datasources.json

```
"sqlserverdb": {
    "name": "sqlserverdb",
    "connector": "mssql"
}
```

Edit `datasources.json` to add other properties that enable you to connect the data source to a SQL Server database.

To connect to a SQL Server instance running in Azure, you must specify a qualified user name with hostname, and add to the following to the data source declaration:

```
"options": {
"encrypt": true
}
```

Connector settings

To configure the data source to use your MS SQL Server database, edit `datasources.json` and add the following settings as appropriate. The MSSQL connector uses node-mssql as the driver. For more information about configuration parameters, see node-mssql documentation.

Property	Type	Default	Description
connector	String		Either "loopback-connector-mssql" or "mssql"
database	String		Database name
debug	Boolean		If true, turn on verbose mode to debug database queries and lifecycle.
host	String	localhost	Database host name
password	String		Password to connect to database
port	Number	1433	Database TCP port
schema	dbo	Database schema	
url	String		Use instead of the `host`, `port`, `user`, `password`, and `database` properties. For example: 'mssql://test:mypassword@localhost:1433/dev'.
user	String		Qualified username with host name, for example “user@your.sqlserver.dns.host".

For example:

/server/datasources.json

```
...
"accountDB": {
    "connector": "mssql",
    "host": "demo.strongloop.com",
    "port": 1433,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
...
```

Alternatively you can use a single 'url' property that combines all the database configuration settings, for example:

```
"accountDB": {
    "url": "mssql://test:mypassword@localhost:1433/demo?schema=dbo"
}
```

The application will automatically load the data source when it starts. You can then refer to it in code, for example:

/server/boot/script.js

```
var app = require('./app');
var dataSource = app.dataSources.accountDB;
```

Alternatively, you can create the data source in application code; for example:

/server/script.js

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var dataSource = new DataSource('mssql', config);
config = { ... };  // JSON object as specified above in "Connector settings"
```

Defining models

The model definition consists of the following properties:

- name: Name of the model, by default, it's the camel case of the table
- options: Model level operations and mapping to Microsoft SQL Server schema/table
- properties: Property definitions, including mapping to Microsoft SQL Server columns

For example:

/common/models/model.json

Expand source

```
{"name": "Inventory",
     "options": {
       "idInjection": false,
       "mssql": {
         "schema": "strongloop",
         "table": "inventory"
       }
     }, "properties": {
      "id": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "mssql": {
          "columnName": "id",
          "dataType": "varchar",
          "dataLength": 64,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "NO"
        }
      },
      "productId": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "id": 1,
        "mssql": {
          "columnName": "product_id",
          "dataType": "varchar",
          "dataLength": 64,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "YES"
        }
      },
```

```
    "locationId": {
      "type": "String",
      "required": false,
      "length": 64,
      "precision": null,
      "scale": null,
      "id": 1,
      "mssql": {
        "columnName": "location_id",
        "dataType": "varchar",
        "dataLength": 64,
        "dataPrecision": null,
        "dataScale": null,
        "nullable": "YES"
      }
    },
    "available": {
      "type": "Number",
      "required": false,
      "length": null,
      "precision": 10,
      "scale": 0,
      "mssql": {
        "columnName": "available",
        "dataType": "int",
        "dataLength": null,
        "dataPrecision": 10,
        "dataScale": 0,
        "nullable": "YES"
      }
    },
    "total": {
      "type": "Number",
      "required": false,
      "length": null,
      "precision": 10,
      "scale": 0,
      "mssql": {
        "columnName": "total",
        "dataType": "int",
        "dataLength": null,
        "dataPrecision": 10,
        "dataScale": 0,
        "nullable": "YES"
```

```
        }
      }
    }}
```

Auto migrating and auto-updating

After making changes to model properties you must call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on a new model, since it will drop existing tables. See Creating a database schema from models for more information.

For each model, the LoopBack SQL Server connector creates a table in the 'dbo' schema in the database.

Destroying models

Destroying models may result in errors due to foreign key integrity. First delete any related models first calling delete on models with relationships.

Type mapping

See LoopBack types for details on LoopBack's data types.

LoopBack to SQL Server types

LoopBack Type	SQL Server Type
Boolean	BIT
Date	DATETIME
GeoPoint	FLOAT
Number	INT
String JSON	NVARCHAR

SQL Server to LoopBack types

SQL Server Type	LoopBack Type
BIT	Boolean
BINARY VARBINARY IMAGE	Node.js Buffer object
DATE DATETIMEOFFSET DATETIME2 SMALLDATETIME DATETIME TIME	Date
POINT	GeoPoint
BIGINT NUMERIC SMALLINT DECIMAL SMALLMONEY INT TINYINT MONEY FLOAT REAL	Number

CHAR VARCHAR TEXT NCHAR NVARCHAR NTEXT CHARACTER VARYING CHARACTER	String

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Database discovery API.

Connecting to Microsoft SQL Server

This article is reproduced from loopback-example-mssql

Error rendering macro 'markdown-url' : URL was not found.

Executing native SQL

This feature has not been fully tested and is not officially supported: the API may change in future releases.

In general, it is always better to perform database actions through connected models. Directly executing SQL may lead to unexpected results, corrupted data, and other issues.

To execute SQL directly against your data-connected model, use the following:

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

or

dataSource.connector.query(sql, params, cb); // For 1.x connectors
```

Where:

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

The actual method signature depends on the specific connector being used. See connector source code. For example, loopback-connector-mysql.

Use caution and be advised that the API may change in the future.

Non-database connectors

LoopBack supports a number of connectors to backend systems beyond databases:

- Email connector
- Push connector
- Remote connector
- REST connector
- SOAP connector
- Storage connector

These types of connectors often implement specific methods depending on the underlying system. For example, the REST connector delegates calls to REST APIs while the Push connector integrates with iOS and Android push notification services.

Models attached to non-database data sources can serve as controllers (a model class that only has methods). Such models usually don't have property definitions as the backing connector doesn't support CRUD operations. For example, to define a model for an external REST API, we can have a model as follows:

common/models/my-rest-service.json

```
{
  "name": "MyRestService",
  "base": "Model",
  "properties": {},
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

The model is configured to attach to the REST data source.

server/model-config.json

```
...
  "MyRestService": {
    "dataSource": "myRestDataSource",
    "public": true
  }
...
```

Email connector

The email connector is built in to LoopBack, so you don't need to install it.

Creating an email data source

Create a new email data source with the data source generator:

```
$ slc loopback:datasource
```

When prompted, select **Email** as the connector. This creates an entry in `datasources.json` like this (for example):

server/datasources.json

```
...
 "myEmailDataSource": {
    "name": "myEmailDataSource",
    "connector": "mail"
  }
 ...
```

Configuring an email data source

Configure the email data source by editing `/server/datasources.json` (for example):

server/datasources.json

```
{
    ...
    "myEmailDataSource": {
       "connector": "mail",
       "transports": [{
         "type": "smtp",
         "host": "smtp.private.com",
         "secure": false,
         "port": 587,
         "tls": {
           "rejectUnauthorized": false
         },
         "auth": {
           "user": "me@private.com",
           "pass": "password"
         }
       }]
    }
    ...
}
```

Using GMail

With GMail, you may need to eanble the "access for less secure apps" option. See Nodemailer - Using GMail and Nodemailer - Authentication for more information.

For GMail, configure your email data source as follows:

server/datasources.json

```
...
"Email": {
    "name": "mail",
    "defaultForType": "mail",
    "connector": "mail",
    "transports": [
      {
        "type": "SMTP",
        "host": "smtp.gmail.com",
        "secure": true,
        "port": 465,
        "auth": {
          "user": "name@gmail.com",
          "pass": "pass"
        }
      }
    ]
  }
...
```

Connecting a model to the email data source

Then, connect models to the data source in `/server/model-config.json` as follows (for example):

server/model-config.json

```
{
  ...
  "Email": {
    "dataSource": "myEmailDataSource"
  },
  ...
}
```

Push connector

- Installation
- Creating a push data source
- Configuring a push data source
- Defining a push model
- Connect model to push data source

Installation

If you haven't yet installed the Push component, in your application root directory, enter:

```
$ npm install loopback-component-push --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a push data source

Create a new push data source with the data source generator:

```
$ slc loopback:datasource
```

When prompted, select **other** as the connector.

At the prompt "**Enter the connector name without the loopback-connector- prefix,**" enter **push**.

This creates an entry in `datasources.json` like this (for example):

/server/datasources.json

```
...
 "myPushDataSource": {
    "name": "myPushDataSource",
    "connector": "push"
  }
 ...
```

Configuring a push data source

To configure a push data source, edit the `datasources.json` file; for example as shown in the push example:

/server/datasources.json

```
"myPushDataSource": {
    "name": "myPushDataSource",
    "connector": "push",
    "installation": "installation",
    "notification": "notification",
    "application": "application"
  }
}
```

Defining a push model

Then define a push model in the Model definition JSON file, for example:

/server/models/push.json

```
{
  "name": "push",
  "base": "Model",
  "plural": "Push",
  "properties": {},
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

Connect model to push data source

Connect the model to the data source:

/server/model-config.json

```
"push": {
    "public": true,
    "dataSource": "myPushDataSource"
  }
```

Remote connector

See also: Example application: loopback-example-remote

The remote connector enables you to use a LoopBack application as a data source via REST. The client can be a LoopBack application, a Node application, or a browser-based application running LoopBack in the client. The connector uses Strong Remoting.

In general, using the remote connector is more convenient than calling into REST API, and enables you to switch the transport later if you need to.

Installation

In your application root directory, enter:

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

This will install the module and add it as a dependency to the application's `package.json` file.

Creating an remote data source

Create a new remote data source with the datasource generator:

```
$ slc loopback:datasource
```

When prompted:

- For connector, scroll down and select **other**.
- For connector name without the loopback-connector- prefix, enter **remote**.

This creates an entry in `datasources.json`; Then you need to edit this to add the data source properties, for example:

/server/datasources.json

```
...
 "myRemoteDataSource": {
    "name": "myRemoteDataSource",
    "connector": "remote",
    "url": "http://localhost:3000/api"
  }
 ...
```

The `url` property specifies the root URL of the LoopBack API.

Remote data source properties

Property	Type	Description
host	String	Hostname of LoopBack application providing remote data source.
port	Number	Port number of LoopBack application providing remote data source.
root	String	Path to API root of LoopBack application providing remote data source.
url	String	Full URL of LoopBack application providing remote connector. Use instead of host, port, and root properties.

Configuring authentication

The remote connector does not support JSON-based configuration of the authentication credentials; see issue #3. You can use the following code as a workaround. It assumes that your data source is called "remote" and the AccessToken id is provided in the variable "token".

```
app.dataSources.remote.connector.remotes.auth = {
  bearer: new Buffer(token).toString('base64'),
  sendImmediately: true
};
```

Using with MongoDB connector

When using the **MongoDB** connector on the server and a **Remote** connector on the client, the following **id** property should be used.

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

Remote connector example

This article is reproduced from loopback-example-remote

Error rendering macro 'markdown-url' : URL was not found.

Strong Remoting

See also Strong remoting API

Overview

Objects (and, therefore, data) in Node applications commonly need to be accessible by other Node processes, browsers, and even mobile clients. Strong remoting:

- Makes local functions remotable, exported over adapters.
- Supports multiple transports, including custom transports.
- Manages serialization to JSON and deserialization from JSON.
- Supports multiple client SDKs, including mobile clients.

Client SDK support

For higher-level transports, such as REST and Socket.IO, existing clients will work well. If you want to be able to swap out your transport, use one of our supported clients. The same adapter model available on the server applies to clients, so you can switch transports on both the server and all clients without changing your application-specific code.

Installation

Quick start

The following example illustrates how to set up a basic strong-remoting server with a single remote method, user.greet.

Then, invoke `User.greet()` easily with `curl` (or any HTTP client)!

```
$ curl http://localhost:3000/user/greet?str=hello
```

Result:

```
{
  "msg": "hello world"
}
```

Concepts

Remote objects

Most Node applications expose a remotely-available API. Strong-remoting enables you to build your app in vanilla JavaScript and export remote objects over the network the same way you export functions from a module. Since they're just plain JavaScript objects, you can always invoke methods on your remote objects locally in JavaScript, whether from tests or other, local objects.

Remote object collections

Collections that are the result of require('strong-remoting').create() are responsible for binding their remote objects to transports, allowing you to swap out the underlying transport without changing any of your application-specific code.

Adapters

Adapters provide the transport-specific mechanisms to make remote objects (and collections thereof) available over their transport. The REST adapter, for example, handles an HTTP server and facilitates mapping your objects to RESTful resources. Other adapters, on the other hand, might provide a less opionated, RPC-style network interface. Your application code doesn't need to know what adapter it's using.

Hooks

Hooks enable you to run code before remote objects are constructed or methods on those objects are invoked. For example, you can prevent actions based on context (HTTP request, user credentials, and so on).

See the before-after example for more info.

Streams

Strong-remoting supports methods that expect or return Readable and Writeable streams. This enables you to stream raw binary data such as files over the network without writing transport-specific behavior.

For example, the following code exposes a method of the `fs` Remote Object, `fs.createReadStream`, over the REST adapter:

Then you can invoke fs.createReadStream() using curl as follows:

```
$ curl http://localhost:3000/fs/createReadStream?path=some-file.txt
```

...

REST connector

See also:

- REST connector API doc
- loopback-example-rest-connector

Overview

The LoopBack REST connector enables applications to interact with other (third party) REST APIs using a template-driven approach. It supports two different styles of API invocations:

- Resource CRUD
- Defining a custom method using a template

Installation

In your application root directory, enter:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a REST data source

Use the DataSource generator to add a REST data source to your application.

```
$ slc loopback:datasource
```

When prompted, scroll down in the list of connectors and choose **REST services (supported by StrongLoop)**. This adds an entry to datasources.json (for example):

```
...
  "myRESTdatasource": {
    "name": "myRESTdatasource",
    "connector": "rest"
  }
...
```

Configuring a REST data source

Configure the REST connector by editing `datasources.json` manually (for example using the Google Maps API):

/server/datasources.json

```
...
    "geoRest": {
        "connector": "rest",
        "debug": "false",
        "operations": [{
            "template": {
                "method": "GET",
                "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
                "headers": {
                    "accepts": "application/json",
                    "content-type": "application/json"
                },
                "query": {
                    "address": "{street},{city},{zipcode}",
                    "sensor": "{sensor=false}"
                },
                "responsePath": "$.results[0].geometry.location"
            },
            "functions": {
                "geocode": ["street", "city", "zipcode"]
            }
        }]
    }
...
```

For a REST data source, you can define an array of *operation* objects to specify the REST API mapping. Each operation object can have the following two properties:

- template: See how to define a custom method template below.
- functions: An object that maps a JavaScript function to a list of parameter names. For example, a function geocode(street, city, zipcode) will be created so that the first argument will be the value of street variable in the template, second for city, and third for zipcode. The function can be executed anywhere by the server (in a boot script, through middleware, or within a model's .js file if it is attached to the REST datasource).

Configure options for request

The REST connector uses the request module as the HTTP client. You can configure the same options as for the `request()` function; see `request(options, callback)` .

You can configure options `options` property at two levels:

- Data source level (common to all operations)
- Operation level (specific to the declaring operation)

For example, the following example sets `Accept` and `Content-Type` to `"application/json"` for all requests. It also sets `strictSSL` to false so that the connector allows self-signed SSL certificates.

/server/datasources.json

```
{
  "connector": "rest",
  "debug": false,
  "options": {
    "headers": {
      "accept": "application/json",
      "content-type": "application/json"
    },
    "strictSSL": false
  },
  "operations": [
    {
      "template": {
        "method": "GET",
        "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
        "query": {
          "address": "{street},{city},{zipcode}",
          "sensor": "{sensor=false}"
        },
        "options": {
          "strictSSL": true,
          "useQuerystring": true
        },
        "responsePath": "$.results[0].geometry.location"
      },
      "functions": {
        "geocode": ["street", "city", "zipcode"]
      }
    }
  ]
}
```

Resource CRUD

If the REST API supports create, read, update, and delete (CRUD) operations for resources, such as users or orders, you can simply bind the model to a REST endpoint that follows REST conventions.

For example, the following methods would be mixed into your model class:

- create: POST /users
- findById: GET /users/:id
- delete: DELETE /users/:id
- update: PUT /users/:id
- find: GET /users?limit=5&username=ray&order=email

For example:

/server/boot/script.js

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

Setting the resource URL

You can set the remote URL when using CRUD functionality by setting the `resourceName` property on a model definition. This allows for a local model name that is different from the remote resource name.

For example:

```
var config = {
  "name": "ServiceTransaction",
  "base": "PersistedModel",
  "resourceName": "transactions"
}

var ServiceTransaction = ds.createModel('ServiceTransaction', {}, config);
```

Now there will be a resource model named ServiceTransaction, but whose CRUD URLs call out to `baseUrl + '/transactions'`

Without setting `resourceName` the calls would have been made to `baseUrl + '/ServiceTransaction'`.

Defining a custom method using a template

Imagine that you use a web browser or REST client to test drive a REST API; you will specify the following HTTP request properties:

- method: HTTP method
- url: The URL of the request
- headers: HTTP headers
- query: Query strings
- responsePath: an optional JSONPath applied to the HTTP body. See https://github.com/s3u/JSONPath for syntax of JSON paths.

Then you 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:

Variable definition	Description
`'{x=100:number}'`	Define a variable x of number type and default value 100
`'{x:number}'`	Define a variable x of number type
`'{x}'`	Define a variable x
`'{x=100}ABC{y}123'`	Define two variables x and y. The default value of x is 100. The resolved value will be a concatenation of x, 'ABC', y, and '123'. For example, x=50, y=YYY will produce '50ABCYYY123'
`'{!x}'`	Define a required variable x

`'{x=100}ABC{^y}123'`	Define two variables x and y. The default value of x is 100. y is required.

To use custom methods, you can configure the REST connector with the `operations` property, which is an array of objects that contain `template` and `functions`. The `template` property defines the API structure while the `functions` property defines JavaScript methods that takes the list of parameter names.

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

var ds = loopback.createDataSource({
    connector: require("loopback-connector-rest"),
    debug: false,
    operations: [
    {
        template: {
            "method": "GET",
            "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
            "headers": {
                "accepts": "application/json",
                "content-type": "application/json"
            },
            "query": {
                "address": "{street},{city},{zipcode}",
                "sensor": "{sensor=false}"
            },
            "responsePath": "$.results[0].geometry.location"
        },
        functions: {
           "geocode": ["street", "city", "zipcode"]
        }
    }
]});
```

Now you can invoke the geocode API as follows:

```
Model.geocode('107 S B St', 'San Mateo', '94401', processResponse);
```

By default, LoopBack REST connector also provides an 'invoke' method to call the REST API with an object of parameters, for example:

```
 Model.invoke({street: '107 S B St', city: 'San Mateo', zipcode: '94401'},
processResponse);
```

REST connector API

Module: loopback-connector-rest

- exports.initialize
- 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
- restConnector.getTypes

REST resource API

Module: loopback-connector-rest

- RestResource
- wrap
- restResource.create
- restResource.update
- restResource.delete
- restResource.deleteAll
- restResource.find
- restResource.all

Request builder API

Module: loopback-connector-rest

- debug
- RequestBuilder
- isObject
- 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._request
- RequestBuilder.resource

REST example with SharePoint

Imagine that you need to get document details from a Microsoft Sharepoint server repository published as a lightweight JSON API that can be consumed by various client apps. This tutorial walks you through how to do this with LoopBack.

- REST example - creating the back-end
- REST example - adding a client app

REST example - creating the back-end

- Setup
- Create REST data source with custom methods
- Add model for CRUD operations
- Add model with custom logic
- Run the application
  - Get list of all documents from Sharepoint
  - Get individual documents from Sharepoint filtered by ID

Setup

Start with the following app from GitHub and use `npm install` to install all the app's dependencies:

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

If you're impatient or just lazy, you can download the completed application from GitHub:

```
$ git clone https://github.com/strongloop/loopback-example-customAPI.git
```

Create REST data source with custom methods

First, create a custom data source using the LoopBack REST connector.

```
$ slc loopback:datasource
[?] Enter the data-source name: Sharepoint
[?] Select the connector for Sharepoint: REST services (supported by StrongLoop)
```

Now open the `datasources.json` file and add custom logic and methods. Copy the code below starting with the line after

`"connector": "rest",`

and paste it into `datasources.json`, so it looks as shown below when you're done.

> The API endpoint http://sharepoint.global.strongloop.com/... is just an example; it doesn't actually exist. Below you'll use the `documentsList.json` file to mock up data that would come from the SharePoint REST API.

```
{
  "Sharepoint": {
    "name": "Sharepoint",
    "connector": "rest",
    "debug": "true",
    "operations": [{
        "template": {
           "method": "GET",
           "url":
"http://sharepoint.global.strongloop.com/Corporate/_api/web/lists/getByTitle('StrongLo
opCorporate')/items",
           "headers": {
               "accept": "application/json; odata=verbose",
               "content-type": "application/json"
           },
           "query": {
               "$orderby": "{orderby}",
                "$top": "{top}"
              },
           "responsePath": "$.results[0]"
         },
         "functions": {
             "getFileList": ["orderby", "top"]
         }
     } ,
         {
          "template": {
             "method": "GET",
             "url":
"http://sharepoint.global.strongloop.com/Corporate/_api/web/lists/GetByTitle('StrongLo
opCorporate')/items({fileID})/File",
             "headers": {
                 "accept": "application/json; odata=verbose",
                 "content-type": "application/json"
              },
             "responsePath": "$.results[0]"
            },
            "functions": {
               "getFileAttributes": ["fileID"]
            }
     }]
  },
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "accountDB": {
    "host": "demo.strongloop.com",
    "port": 3306,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack",
    "name": "accountDB",
    "connector": "mysql"
  }
}
```

The LoopBack REST connector enables Node.js applications to interact with HTTP REST APIs using a template-driven approach. It supports two different styles of API invocations:

- Resource create, read, update, and delete (CRUD)
- Defining a custom method using REST template

In the code above, you can see that you added template-based modeling logic to the REST connector. You are making two independent API calls to Sharepoint: one to get a list of files/documents and another to return the attributes of each file within the list.

Add model for CRUD operations

If you were only interested in CRUD operations, as usual with the generators-based methodology, you can create a Document model using Yeoman and attach it to the Sharepoint REST connector.

```
$ slc loopback:model
[?] Enter the model name: Document
[?] Select the data-source to attach Document to:
 Sharepoint (rest)
  accountDB (mysql)
  db (memory)
[?] Expose Document via the REST API? (Y/n) :Y
[?] Custom plural form (used to build REST URL):
```

When the generator prompts you to add properties, follow the prompts to add these properties:

Property	Type	Required?
name	string	Yes
type	string	Yes
size	string	Yes
date_created	date	Yes
last_modified	date	Yes

You can see the corresponding changes made to `/common/models/document.json`.

Add model with custom logic

If you're not interested only in CRUD operations, but want to implement custom logic, skip the Yeoman steps and add custom logic for post-processing the data returned by the Sharepoint API call.

- Copy and paste the code below.
- Save it to `/server/boot/document.js`. That's where you put custom models and other code you want executed on app startup.

document.js Expand source

```
var loopback = require("loopback");
var app = require('../server');
var fs = require("fs");

var ds = app.dataSources.Sharepoint;
var fileListModel = ds.createModel ('Documents',{},{base:loopback.Model});
console.log(fileListModel.super_.modelName);

module.exports=fileListModel;
var queryParam = {
    arg1: 'Modified desc',
    arg2: '5'
};
```

```
var fileIDParam = {
        fileID: '8'
};

fileListModel.getFileList = function(orderBy, top, cb) {
        var file = __dirname + '/documentList.json';
        fs.readFile(file, 'utf8', function (err, data) {
        if (err) {
          console.log('Error: ' + err);
          cb(err);
        } else {
          data = JSON.parse(data);
          cb(null, data);
        }
   });
};

fileListModel.getFileList.shared = true;
fileListModel.getFileList.accepts = [{arg: 'orderBy', type: 'string', http: {source:
'query'}},
{arg: 'top', type: 'number', http: {source: 'query'}}];
fileListModel.getFileList.returns = [{arg: 'data', type: 'array', root: true} ];
fileListModel.getFileList.http = {verb: 'get', path: '/'};

fileListModel.getFileAttributes = function(id, cb) {
        console.log("findbyid", id);
        var file = __dirname + '/documentList.json';
        fs.readFile(file, 'utf8', function (err, data) {
        if (err) {
          console.log('Error: ' + err);
          cb(err);
        } else {
          data = JSON.parse(data);
          console.log(data[id-1]);
          cb(null, data[id-1]);
        }
});
};

fileListModel.getFileAttributes.shared = true;
fileListModel.getFileAttributes.accepts = [{arg: 'id', type: 'number', http: {source:
'path'}}];
fileListModel.getFileAttributes.returns = [{arg: 'data', type: 'object', root: true}
];
```

```
fileListModel.getFileAttributes.http = {verb: 'get', path: '/:id'};

app.model(fileListModel);
```

This file does the following:

- Overrides the default persistence model created by the generators and adds custom model to extend the base model.
- Extends the defined custom methods in the data source setup and converts them into API endpoints.
- Provides dummy data (`documentsList.json`) to represent data that in practice would come from the Sharepoint Server REST interface.
- Exposes local server-side methods that other Node functions can invoke directly without having to interact with the JSON API endpoint.

The dummy data in `documentList.json` is shown below. This test data is the same format that the backend Sharepoint call would return.

documentList.json

Expand source

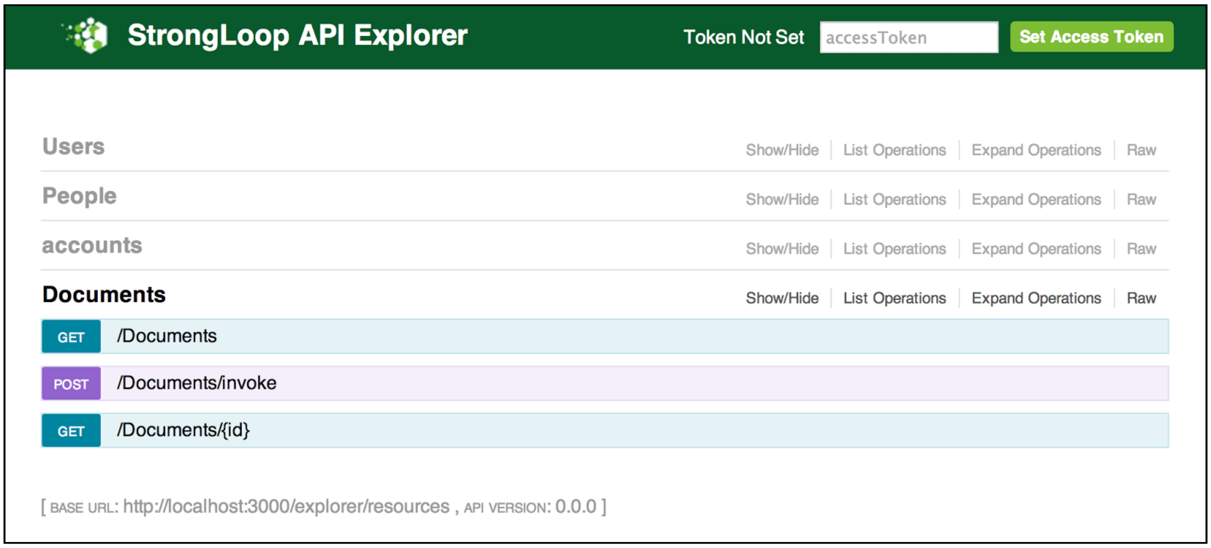
```
[
  {
    "name": "PriceList",
    "created_by": "Michelle Williams",
    "type": "xls",
    "size": "20Kb",
    "date_created": "2011-06-23T18:25:43.511Z",
    "last_modified": "2014-03-23T18:25:43.511Z",
    "id": 1
  },
  {
    "name": "CustomerList",
    "created_by": "Gorge Clooney",
    "type": "xls",
    "size": "10Kb",
    "date_created": "2010-05-23T18:25:43.511Z",
    "last_modified": "2012-09-23T18:25:43.511Z",
    "id": 2
  },
  {
    "name": "SalesPipeline",
    "created_by": "Brad Pitt",
    "type": "pdf",
    "size": "2Kb",
    "date_created": "2011-02-25T18:25:43.511Z",
    "last_modified": "2012-04-23T18:25:43.511Z",
    "id": 3
  },
 {
    "name": "Forecast",
    "created_by": "Olivia Wilde",
    "type": "xls",
    "size": "2MB",
    "date_created": "2012-01-23T18:25:43.511Z",
    "last_modified": "2012-04-23T18:25:43.511Z",
    "id": 4
  },
  {
    "name": "ProductRoadMap",
    "created_by": "Ryan Gosling",
    "type": "pdf",
    "size": "200Kb",
    "date_created": "2012-07-23T18:25:43.511Z",
    "last_modified": "2014-04-23T18:25:43.511Z",
    "id": 5
  }
]
```

Run the application

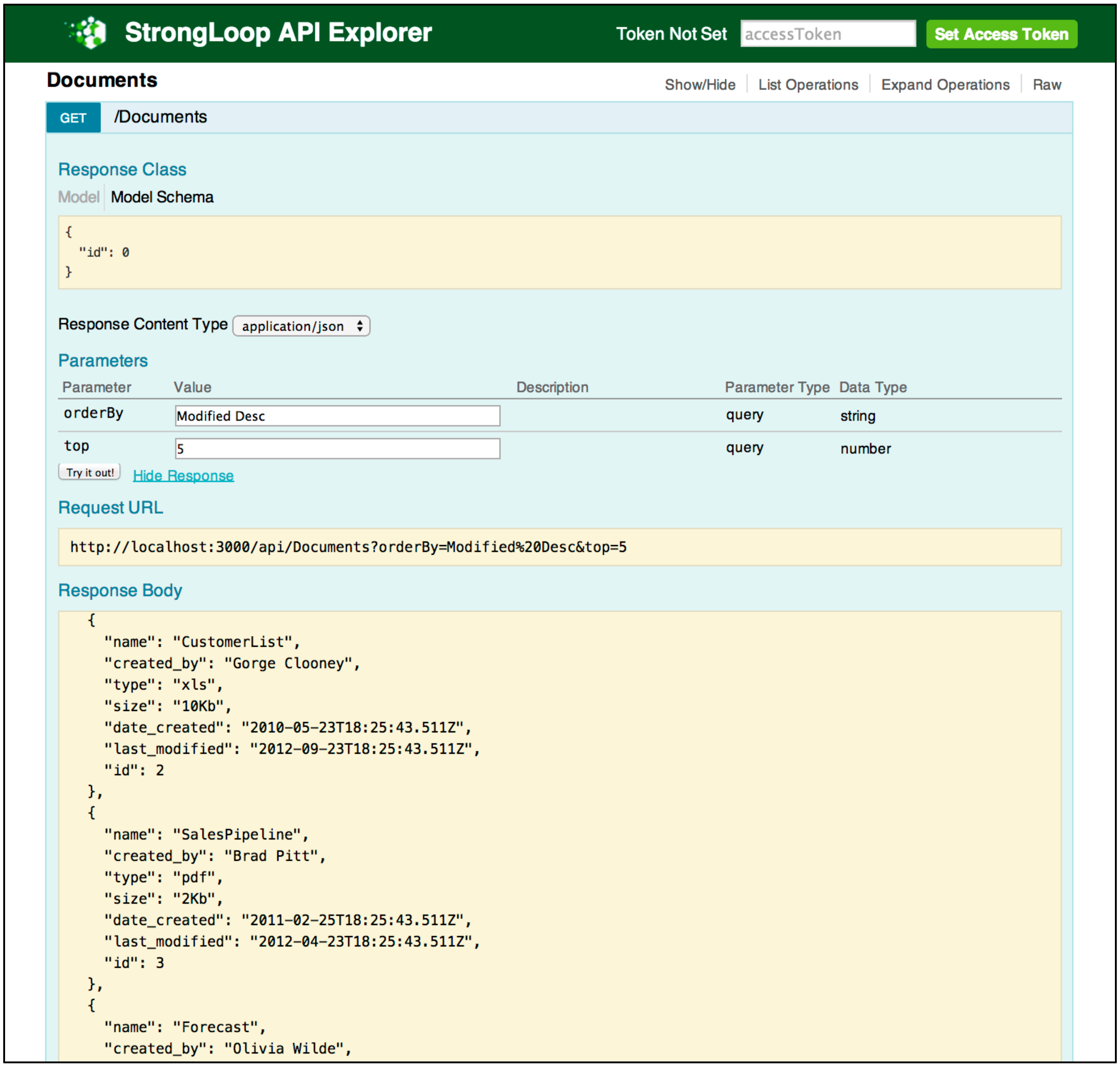
Now run the application:

```
$ node .
```

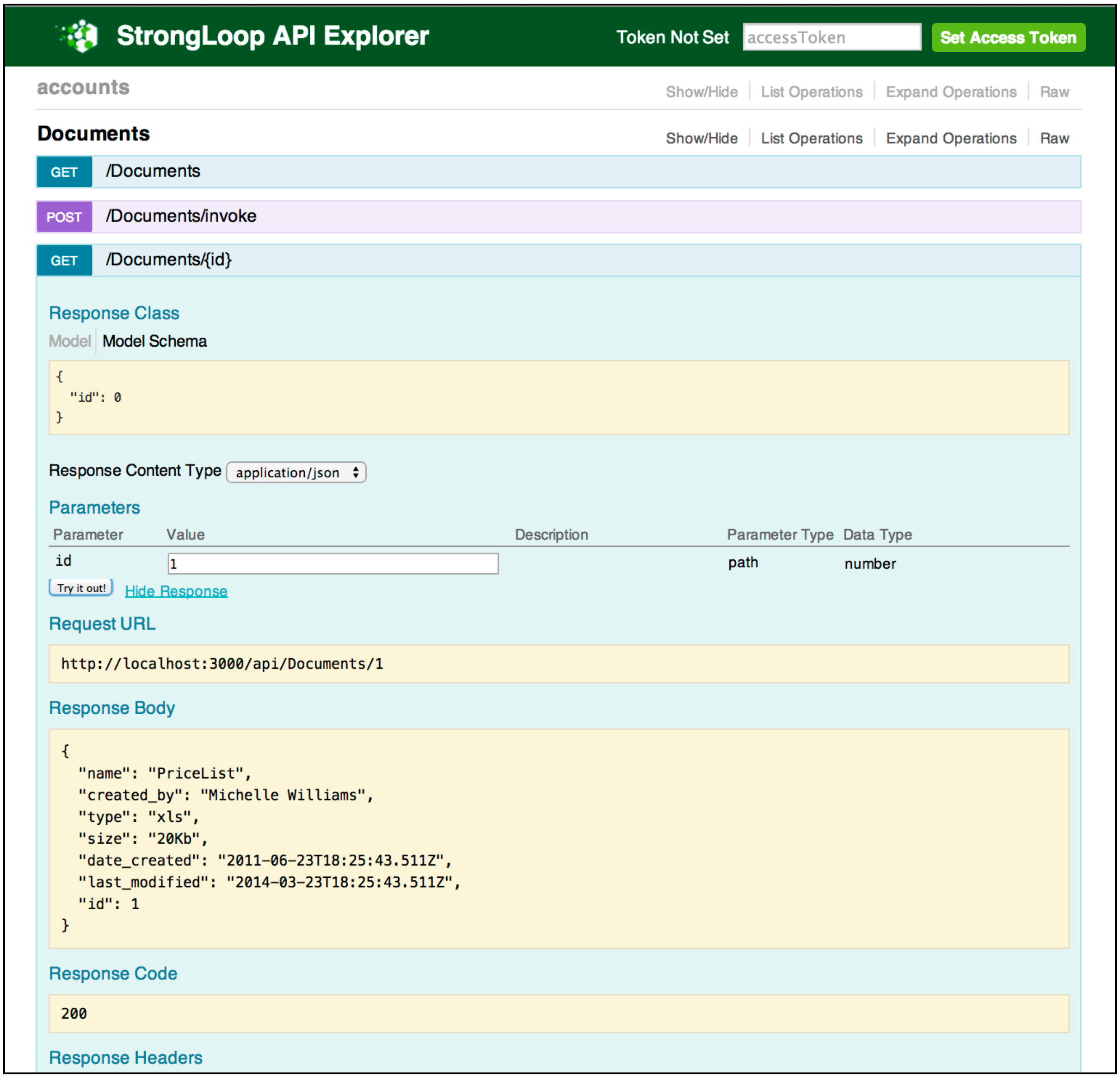
Browsing the API explorer, you can see both the API endpoints as well as individual query results.



Get list of all documents from Sharepoint



Get individual documents from Sharepoint filtered by ID



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

REST example - adding a client app

This article explains how to add an iOS client app to connect to the custom API that connects to SharePoint you created previously.

Create the iOS app starting point

Follow the steps listed in this example of a sample Books collection application

- Creating a LoopBack iOS app: part one
- Creating a LoopBack iOS app: part two

Our app is going to be a replica copy of this app, just that we will make some visual updates to reflect a “Sharepoint Library” instead of a “Books collection”

Alternatively, you can clone the following GitHub repository:

```
$ git clone https://github.com/strongloop/loopback-example-APIClientApp.git
```

Modify the iOS app

Key aspects of using the LoopBack API in an IOS App are:

- Import the Loopback framework (iOS SDK) as a library into the application
- Import Looback models as Prototypes
- Import the `LoopBack.h` header into your application just as you would `Foundation/Foundation.h`. Type this line:

```
#import <LoopBack/LoopBack.h>
```

- You need an Adapter to tell the SDK where to find the server. Enter this code:

```
LBRESTAdapter *adapter =
  [LBRESTAdapter adapterWithURL:[NSURL URLWithString:@"http://example.com"]];
```

- `LBRESTAdapter` provides the starting point for all interactions with the server.
- Once we have access to adapter (for the sake of example, we'll assume the Adapter is available through our AppDelegate), we can create basic `LBModel` and `LBModelRepository` objects. Assuming we've previously created a model named "product":

```
LBRESTAdapter *adapter = [[UIApplication sharedApplication] delegate].adapter;
LBModelRepository *productReposiory = [adapter
repositoryWithModelName:@"products"];
LBModel *pen = [Product modelWithDictionary:@{ "name": "Awesome Pen" }];
```

Once you have an adapter, you can create a repository instance.

```
WidgetRepository *repository = (WidgetRepository *)[adapter
repositoryWithModelClass:[WidgetRepository class]];
```

Now that you have a `WidgetRepository` instance, you can create, save, find, and delete widgets, as illustrated below.

- Create a Widget:

```
Widget *pencil = (Widget *)[repository modelWithDictionary:@{ @"name": @"Pencil",
@"price": @1.50 }];
```

Save a Widget:

```
[pencil saveWithSuccess:^{
                    // Pencil now exists on the server!
                }
                failure:^(NSError *error) {
                    NSLog("An error occurred: %@", error);
                }];
```

Find another Widget:

```
[repository findWithId:@2
              success:^(LBModel *model) {
                  Widget *pen = (Widget *)model;
              }
              failure:^(NSError *error) {
                  NSLog("An error occurred: %@", error);
              }];
```

Remove a Widget:

```
[pencil destroyWithSuccess:^{
                       // No more pencil. Long live Pen!
                   }
```

SOAP connector

See also: Turn SOAP into REST APIs with LoopBack (Blog post)

The SOAP connector enables LoopBack applications to interact with SOAP-based web services described using WSDL.

Installation

In your application root directory, enter:

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

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the Data source generator to add a SOAP data source to your application.

```
$ slc loopback:datasource
```

Choose "SOAP webservices" as the data source type when prompted.

SOAP data source properties

The following table describes the SOAP data source properties you can set in `datasources.json`.

Property	Type	Description
url	String	URL to the SOAP web service endpoint. If not present, defaults to the `location` attribute of the SOAP address for the service/port from the WSDL document; for example: `<wsdl:service name="Weather"> <wsdl:port name="WeatherSoap" binding="tns:WeatherSoap"> <soap:address **location="http://wsf.cdyne.com/WeatherWS/Weather.asmx"** /> </wsdl:port> ... </wsdl:service>`

wsdl	String	HTTP URL or local file system path to the WSDL file, if not present, defaults to `?wsdl`.
remotingEnabled	Boolean	Indicates whether the operations are exposed as REST APIs. To expose or hide a specific method, you can override this with: `<Model>.<method>.shared = true / false;`
operations	Object	Maps WSDL binding operations to Node.js methods. Each key in the JSON object becomes the name of a method on the model. See Operations property below.

Operations property

The operations property value is a JSON object that has a property (key) for each method being defined for the model. The corresponding value is an object with the following properties:

Property	Type	Description
service	String	WSDL service name
port	String	WSDL port name
operation	String	WSDL operation name

Here is an example operations property for the stock quote service:

```
operations: {
  // The key is the method name
  stockQuote: {
    service: 'StockQuote', // The WSDL service name
    port: 'StockQuoteSoap', // The WSDL port name
    operation: 'GetQuote' // The WSDL operation name
  },
  stockQuote12: {
    service: 'StockQuote',
    port: 'StockQuoteSoap12',
    operation: 'GetQuote'
  }
}
```

A complete example datasource.json:

```
{
  "WeatherServiceDS": {
    "url": "http://wsf.cdyne.com/WeatherWS/Weather.asmx",
    "name": "WeatherServiceDS",
    "connector": "soap",
    "wsdl": "http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL",
    "remotingEnabled": true,
    "operations": {
      "stockQuote": {
        "service": "StockQuote",
        "port": "StockQuoteSoap",
        "operation": "GetQuote"
      },
      "stockQuote12": {
        "service": "StockQuote",
        "port": "StockQuoteSoap12",
        "operation": "GetQuote"
      }
    }
  }
}
```

Creating a model from a SOAP data source

The SOAP connector loads WSDL documents asynchronously. As a result, the data source won't be ready to create models until it's connected. The recommended way is to use an event handler for the 'connected' event; for example:

```
ds.once('connected', function () {
   // Create the model
   var WeatherService = ds.createModel('WeatherService', {});
   ...
}
```

Extending a model to wrap and mediate SOAP operations

Once you define the model, you can extend it to wrap or mediate SOAP operations and define new methods. The following example simplifies the `GetCityForecastByZIP` operation to a method that takes `zip` and returns an array of forecasts.

```
// Refine the methods
WeatherService.forecast = function (zip, cb) {
    WeatherService.GetCityForecastByZIP({ZIP: zip || '94555'}, function (err,
response) {
        console.log('Forecast: %j', response);
        var result = (!err && response.GetCityForecastByZIPResult.Success) ?
        response.GetCityForecastByZIPResult.ForecastResult.Forecast : [];
        cb(err, result);
    });
};
```

The custom method on the model can be exposed as REST APIs. It uses the `loopback.remoteMethod` to define the mappings.

```
// Map to REST/HTTP
loopback.remoteMethod(
    WeatherService.forecast, {
       accepts: [
         {arg: 'zip', type: 'string', required: true, http: {source: 'query'}}
       ],
       returns: {arg: 'result', type: 'object', root: true},
       http: {verb: 'get', path: '/forecast'}
    }
);
```

Use boot script to create model and expose apis to explorer

The SOAP connector is a bit special as it builds the operations from WSDL asynchronously. To expose such methods over REST, you need to do the following with a boot script, such as server/a-soap.js:

```
module.exports = function(app, cb) {
  var ds = app.dataSources.WeatherServiceDS;
  if (ds.connected) {
    var weather = ds.createModel('weather', {}, {base: 'Model'});
    app.model(weather);
    process.nextTick(cb);
  } else {
    ds.once('connected', function() {
      var weather = ds.createModel('weather', {}, {base: 'Model'});
      app.model(weather);
      cb();
    });
  }
};
```

Examples

The loopback-connector-soap repository provides several examples:

Get stock quotes by symbols: stock-ws.js. Run with the command:

```
$ node example/stock-ws
```

Get weather and forecast information for a given zip code: weather-ws.js. Run with the command:

```
$ node example/weather-ws
```

Expose REST APIs to proxy the SOAP web services: weather-rest.js.   Run with the command:

```
$ node example/weather-rest
```

View the results at http://localhost:3000/explorer.

Storage connector

- Installation

- Creating a storage data source
- Configuring a storage data source
- Creating a storage model
- Connect the model to the storage data source

Installation

If you haven't yet installed the storage component, in your application root directory, enter:

```
$ npm install loopback-component-storage --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a storage data source

Create a new push data source with the data source generator:

```
$ slc loopback:datasource
```

When prompted, select **other** as the connector.

At the prompt "**Enter the connector name without the loopback-connector- prefix,**" enter **storage**.

This creates an entry in `datasources.json` like this (for example):

/server/datasources.json

```
...
 "myStorageDataSource": {
    "name": "myStorageDataSource",
    "connector": "storage"
  }
 ...
```

Configuring a storage data source

Configure a storage data source by editing the datasources.json file, for example as shown in the storage service example:

/server/datasources.json

```
...
  "myStorageDataSource": {
    "name": "myStorageDataSource",
    "connector": "storage",
    "provider": "filesystem",
    "root": "./server/storage"
  }
...
```

Creating a storage model

Use the model generator to create a new model, then edit the model.json file, as shown in the storage service example:

/server/models/container.json

```
{
  "name": "container",
  "base": "Model",
  "properties": {},
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

Connect the model to the storage data source

/server/model-config.json

```
...
  "container": {
    "dataSource": "myStorageDataSource",
    "public": true
  }
...
```

Community connectors

In addition to the connectors that StrongLoop provides and maintains, there are a number of connectors created by the open-source community.

StrongLoop does not support the connectors listed here; they are maintained by the LoopBack community and are listed here for convenience.

Please contact StrongLoop to request support for one of these connectors or to request an additional connector.

The following table lists some of the community connectors. See npmjs.org for a complete list.

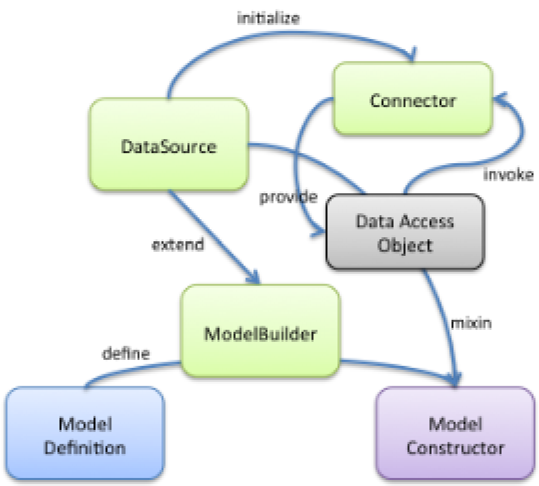
See also https://github.com/pasindud/awesome-loopback for an extensive list of LoopBack community resources.

Data source	Connector	Notes
Apache CouchDB	loopback-connector-couch	
Apache Kafka	loopback-connector-kafka	Provided as option by data source generator.
ArangoDB	loopback-connector-arango	
Couchbase	loopback-connector-couchbase	Example at loopback-example-couchbase
Elasticsearch	loopback-connector-elastic-search	
Mandrill	lb-connector-mandrill	Enables applications to send emails via Mandrill
Neo4j	loopback-connector-neo4j	Provided as option by data source generator. NOTE: This connector has known issues.

RavenDB	loopback-connector-ravendb	
Riak	loopback-connector-riak	
SAP HANA	loopback-connector-saphana	Provided as option by data source generator.
SQLite	loopback-connector-sqlite	
Twilio	loopback-connector-twilio	Example in GitHub.

Advanced topics: data sources

Overview



The diagram illustrates the relationship between LoopBack `Model`, `DataSource`, and `Connector`.

1. Define the model.
2. Create an instance of `ModelBuilder` or `DataSource`. `DataSource` extends from `ModelBuilder`. ModelBuilder is responsible for compiling model definitions to JavaScript constructors representing model classes. DataSource inherits that function from ModelBuilder.
3. Use `ModelBuilder` or `DataSource` to build a JavaScript constructor (i.e. the model class) from the model definition. Model classes built from `ModelBuilder` can be later attached to a DataSource to receive the mixin of data access functions.
4. As part of step 2, `DataSource` initializes the underlying `Connector` with a settings object which provides configurations to the connector instance. `Connector` collaborates with `DataSource` to define the functions as `DataAccessObject` to be mixed into the model class. The `DataAccessObject` consists of a list of static and prototype methods. It can be CRUD operations or other specific functions depending on the connector's capabilities.

The `DataSource` object is the unified interface for LoopBack applications to integrate with backend systems. It's a factory for data access logic around model classes. With the ability to plug in various connectors, `DataSource` provides the necessary abstraction to interact with databases or services to decouple the business logic from plumbing technologies.

Creating a DataSource programmatically

The `DataSource` constructor accepts the following arguments:

- name: Optional name of the data source instance being created.
- settings: An object of properties to configure the connector. Must include a `connector` property, specifying the connector to use. See Connecting models to data sources (Connectors).

For example:

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

var dataSource = new DataSource({
    connector: require('loopback-connector-mongodb'),
    host: 'localhost',
    port: 27017,
    database: 'mydb'
});
```

The `connector` argument passed the `DataSource` constructor can be one of the following:

- The connector module from `require(connectorName)`
- The full name of the connector module, such as 'loopback-connector-oracle'
- The short name of the connector module, such as 'oracle', which will be converted to 'loopback-connector-'
- A local module under ./connectors/ folder

```
var ds1 = new DataSource('memory');
var ds2 = new DataSource('loopback-connector-mongodb'));
var ds3 = new DataSource(require('loopback-connector-oracle'));
```

LoopBack provides the built-in memory connector that uses in-memory store for CRUD operations.

The `settings` argument configures the connector. Settings object format and defaults depends on specific connector, but common fields are:

- `host`: Database host
- `port`: Database port
- `username`: Username to connect to database
- `password`: Password to connect to database
- `database`: Database name
- `debug`: Turn on verbose mode to debug db queries and lifecycle

For more information, see Connecting models to data sources (Connectors). For connector-specific settings, see the connector's documentation.

Creating a model from a data source

`DataSource` extends from `ModelBuilder`, which is a factory for plain model classes that only have properties. `DataSource` connects to databases and other backend systems using `Connector`.

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

var User = ds.define('User', {
    name: String,
    bio: String,
    approved: Boolean,
    joinedAt: Date,
    age: Number
});
```

All model classes within single data source share the same connector type and one database connection or connection pool. But it's possible to use more than one data source to connect to different databases.
Alternatively, you can attach a plain model constructor created from `ModelBuilder` to a `DataSource`.

```
var ModelBuilder = require('loopback-datasource-juggler').ModelBuilder;
var builder = new ModelBuilder();

var User = builder.define('User', {
        name: String,
        bio: String,
        approved: Boolean,
        joinedAt: Date,
        age: Number
});

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

User.attachTo(ds); // The CRUD methods will be mixed into the User constructor
```

Creating a data source for a connector

Application code does not directly use a connector. Rather, you create a DataSource to interact with the connector.

The simplest example is for the in-memory connector:

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

Here is another example, this time for the Oracle connector:

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

var ds = new DataSource(oracleConnector, {
    host : 'localhost',
    database : 'XE',
    username : 'username',
    password : 'password',
    debug : true
});
```

The connector argument passed the DataSource constructor can be one of the following:

- The connector module from `require('connectorName')`
- The full name of the connector module, such as `'loopback-connector-oracle'`.
- The short name of the connector module, such as `'oracle'`, that LoopBack converts to `'loopback-connector-oracle'` (for example).
- A local module in the `/connectors` folder

Initializing a connector

The connector module can export an `initialize` function to be called by the owning DataSource instance.

```
exports.initialize = function (dataSource, postInit) {

    var settings = dataSource.settings || {}; // The settings is passed in from the
dataSource

    var connector = new MyConnector(settings); // Construct the connector instance
    dataSource.connector = connector; // Attach connector to dataSource
    connector.dataSource = dataSource; // Hold a reference to dataSource
    ...
};
```

The DataSource calls the `initialize` method with itself and an optional `postInit` callback function. The connector receives the settings from the `dataSource` argument and use it to configure connections to backend systems.

Please note connector and dataSource set up a reference to each other.

Upon initialization, the connector might connect to database automatically. Once connection established dataSource object emit 'connected' event, and set `connected` flag to true, but it is not necessary to wait for 'connected' event because all queries cached and executed when dataSource emit 'connected' event.

To disconnect from database server call `dataSource.disconnect` method. This call is forwarded to the connector if the connector have ability to connect/disconnect.

Building a connector

- Overview
- Understand a connector's responsibilities
- Understand a database connector with CRUD operations
- Define a module and export the initialize function
- Create a subclass of SqlConnector
- Implement lifecyle methods
    - Connect to the database
    - Disconnect from the database
    - Ping the database

Related articles:

Overview

LoopBack provides connectors for:

- Popular relational and NoSQL databases; see Database connectors.
- Backend systems beyond databases; see Non-database connectors.

Also see Community connectors for a list of connectors developed by the StrongLoop developer community.

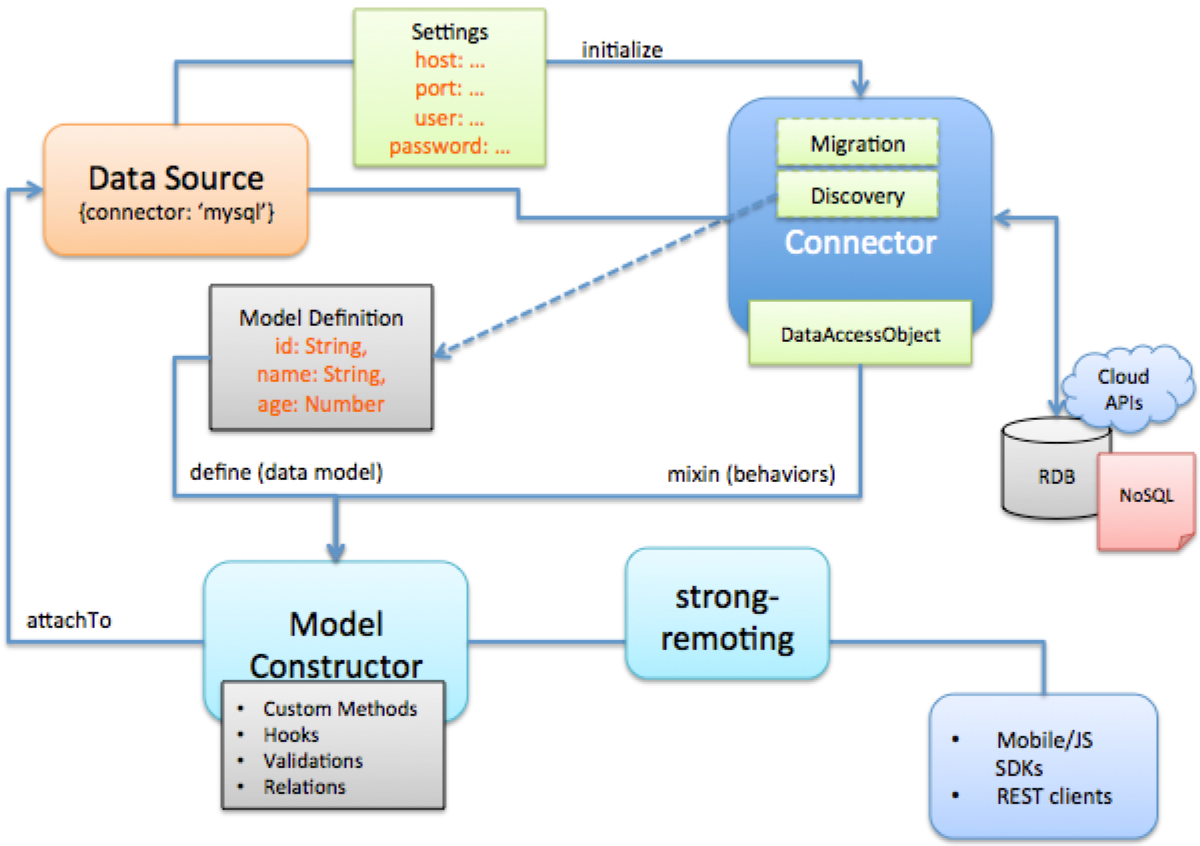
This article is for developers who want to create a new connector type to connect to a data source not currently supported. It walks you through the MySQL connector implementation to teach you how to develop a connector for relational databases. However, many of the concepts also apply to creating a connector to other types of data sources.

Understand a connector's responsibilities

LoopBack abstracts the backend persistence layer as *data sources* that can be databases, or other backend services such as REST APIs, SOAP web services, storage services, and so on. Each data source is backed a *connector* that implements the interactions between Node.js and the underlying backend system. Connectors are responsible for mapping model methods to backend functions, such as database operations or calls to REST or SOAP APIs.

LoopBack models encapsulate business data and logic as JavaScript properties and methods. One of the powerful features of LoopBack is that connectors provide most common model behaviors "out of the box", so application developers don't have to implement them. For example, a model automatically receives the create, retrieve, update, and delete (CRUD) functions when attached to a data source for a database.

The following diagram illustrates how connectors fit into the LoopBack API framework.



You don't always have to develop a connector to enable your application to interact with other systems. You can use custom methods on a model to provide *ad-hoc* integration. The custom methods can be implemented using other Node modules, such as drivers or clients to your backend.

You may want to develop a connector to:

- Integrate with a backend such as databases.
- Provide reusable logic to interact with another system.

There are a few typical types of connectors based on what backends they connect to and interact with.

- Databases that support full CRUD operations
  - Oracle, SQL Server, MySQL, Postgresql, MongoDB, In-memory DB
- Other forms of existing APIs
  - REST APIs exposed by your backend
  - SOAP/HTTP web services
- Services
  - E-mail
  - Push notification
  - Storage

The connectors are mostly transparent to models. Their functions are mixed into model classes through data source attachments.

Most connectors need to implement the following logic:

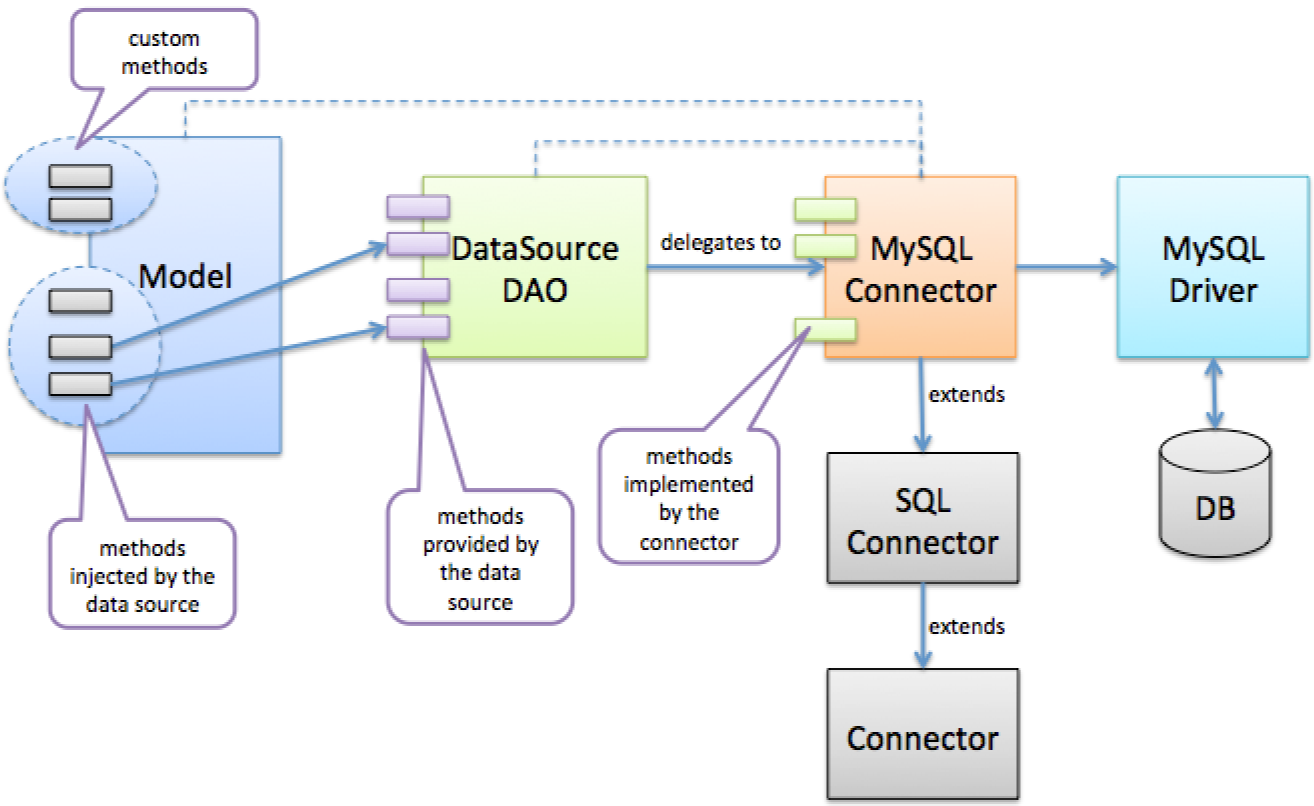
- Lifecycle handlers
  - initialize: receive configuration from the data source settings and initialize the connector instance
  - connect: create connections to the backend system
  - disconnect: close connections to the backend system
  - ping (optional): check connectivity
- Model method delegations
  - Delegating model method invocations to backend calls, for example CRUD
- Connector metadata (optional)
  - Model definition for the configuration, such as host/URL/username/password
  - What data access interfaces are implemented by the connector (the capability of the connector)
  - Connector-specific model/property mappings

To mix-in methods onto model classes, a connector must choose what functions to offer. Different types of connectors implement different interfaces that group a set of common methods, for example:

- Database connectors
    - CRUD methods, such as create, find, findById, deleteAll, updateAll, count
- E-mail connector
    - send()
- Storage connector
    - Container/File operations, such as createContainer, getContainers, getFiles, upload, download, deleteFile, deleteContainer
- Push Notification connector
    - notify()
- REST connector
    - Map operations from existing REST APIs
- SOAP connector
    - Map WSDL operations

This article focuses on building a connector for databases that provide full CRUD capabilities.

Understand a database connector with CRUD operations



LoopBack unifies all CRUD based database connectors so that a model can choose to attach to any of the supported database. There are a few classes involved here:

1. PersistedModelClass defines all the methods mixed into a model for persistence.
2. The DAO facade maps the PersistedModel methods to connector implementations.
3. CRUD methods need to be implemented by connectors.

The following sections use the MySQL connector as an example to walk through how to implement a connector for a relational database.

Define a module and export the initialize function

A LoopBack connector is packaged as a Node.js module that can be installed using `npm install`. The LoopBack runtime loads the module via `require` on behalf of data source configuration, for example, `require('loopback-connector-mysql');`. The connector module should export an `initialize` function as follows:

```
// Require the DB driver
var mysql = require('mysql');
// Require the base SqlConnector class
var SqlConnector = require('loopback-connector').SqlConnector;
// Require the debug module with a pattern of loopback:connector:connectorName
var debug = require('debug')('loopback:connector:mysql');

/**
 * Initialize the  connector against the given data source
 *
 * @param {DataSource} dataSource The loopback-datasource-juggler dataSource
 * @param {Function} [callback] The callback function
 */
exports.initialize = function initializeDataSource(dataSource, callback) {
  ...
};
```

After initialization, the `dataSource` object will have the following properties added:

- `connector`: The connector instance
- `driver`: The module for the underlying database driver (`mysql` for MySQL)

The `initialize` function calls the `callback` function once the connector has been initialized.

Create a subclass of SqlConnector

Connectors for relational databases have a lot of things in common. They are responsible for mapping CRUD operations to SQL statements. LoopBack provides a base class called `SqlConnector` that encapsulates the common logic for inheritance. The following code snippet is used to create a subclass of SqlConnector. The settings parameter is an object containing the data source settings.

```
function MySQL(settings) {
  // Call the super constructor with name and settings
  SqlConnector.call(this, 'mysql', settings);
  ...
}
// Set up the prototype inheritence
require('util').inherits(MySQL, SqlConnector);
```

Implement lifecyle methods

A connector must implement `connect()`, `disconnect()`, and optionally `ping()` methods to communicate with the underlying database.

Connect to the database

The `connect` method establishes a connection to the database. In most cases, it creates a connection pool based on the data source settings, including `host`, `port`, `database`, and other configuration properties.

```
MySQL.prototype.connect = function (cb) {
  // ...
};
```

Disconnect from the database

The `disconnect` method closes a connection to the database. Most database drivers provide an API to disconnect.

```
MySQL.prototype.disconnect = function (cb) {
  // ...
};
```

Ping the database

Optionally, implement a `ping` method to test if the connection to the database is healthy. Most connectors implement it by executing a simple SQL statement.

```
MySQL.prototype.ping = function(cb) {
  // ...
};
```

Implementing auto-migration

Related articles:

Overview

It's often desirable to apply model definitions to the underlying relational database to provision or update schema objects so that they stay synchronized with the model definitions. In LoopBack, this is called *auto-migration*. Implementing auto-migration is optional for connector.

There are two variations:

- **Auto-migration**: Drop existing schema objects if they exist, and re-create them based on model definitions. Existing data will be lost.
- **Auto-update**: Detect the difference between schema objects and model definitions, and alter the database schema objects. Keep existing data.

See Creating a database schema from models for a general introduction to auto-migration auto-update.

Define autoupdate and automigrate functions

These are the two top-level functions that actually create the database schema and call the other functions.

For both functions, the parameters are:

- `models` (optional): a string model name or an array of string model names. If not present, apply to all models
- `cb`: callback function

```
MySQL.prototype.autoupdate = function (models, cb) {
  // ...
};

MySQL.prototype.automigrate = function (models, cb) {
  // ...
};
```

The `automigrate` and `autoupdate` operations are usually mapped to a sequence of data definition language (DDL) statements.

Defining helper functions

First, define a few helper functions.

Build a CREATE TABLE statement

Define a function to create a database table for a model.

Parameters:

- `model`: Model name
- `cb`: Callback function

```
MySQL.prototype.createTable = function (model, cb) {
  // ...
};
```

Check if models have corresponding tables

Define a function to check if the specified models exist.

Parameters:

- `models` (optional): a string model name or an array of string model names. If not present, apply to all models
- `cb`: callback function

```
MySQL.prototype.isActual = function(models, cb) {
  // ...
};
```

Alter a table

```
MySQL.prototype.alterTable = function (model, actualFields, actualIndexes, done,
checkOnly) {
  // ...
};
```

Define metadata definition functions

Define functions to create column and index definitions for models and model properties.

Build column definition clause for a given model

```
MySQL.prototype.buildColumnDefinitions =
MySQL.prototype.propertiesSQL = function (model) {
  // ...
};
```

Build index definition clause for a given model property

```
MySQL.prototype.buildIndex = function(model, property) {
  // ...
};
```

Build indexes for a given model

```
MySQL.prototype.buildIndexes = function(model) {
  // ...
};
```

Build column definition for a given model property

```
MySQL.prototype.buildColumnDefinition = function(model, prop) {
  // ...
};
```

Build column type for a given model property

```
MySQL.prototype.columnDataType = function (model, property) {
  // ...
};
```

Implementing CRUD methods

Related articles:

Overview

A relational database connector is responsible for implementing a number of methods for create, read, update, and delete (CRUD) operations. The base `SqlConnector` has most of the methods implemented with the extension point to override certain behaviors that are specific to the underlying database.

To extend from `SqlConnector`, you must implement the minimum set of methods listed below.

Implementing basic CRUD methods

Execute a SQL statement with parameters

The `executeSQL` method is the core function that a connector must implement. Most of other CRUD methods are delegated to the `query` function. It executes a SQL statement with an array of parameters. `SELECT` statements will produce an array of records representing matching rows from the database while other statements such as `INSERT`, `DELETE`, or `UPDATE` will report the number of rows changed during the operation.

The function's parameters are:

- `sql`: A string containing the SQL statement to execute, possibly with placeholders for parameters.
- `params` (optional): An array of parameter values.
- `options`: Options passed to the CRUD method.
- `callback`: Callback function called after the SQL statement is executed.

```
MySQL.prototype.executeSQL = function (sql, params, options, callback) {
  // ...
};
```

Map values between a model property and a database column

Define a `toColumnValue()` function that converts a model property value into the form required by the database column. The result should be one of following forms:

- `{sql: "point(?,?)", params:[10,20]}`
- `{sql: "'John'", params: []}`
- `"John"`

The function returns database column value as an ParameterizedSQL object

Parameters are:

- `propertyDef`: Object containing the model property definition.
- `value`: Model property value (any type).

```
SqlConnector.prototype.toColumnValue = function(propertyDef, value) {
  /*jshint unused:false */
  throw new Error('toColumnValue() must be implemented by the connector');
};
```

Define a `fromColumnValue()` function that converts the data from database column to model property. It returns a model property value.

Parameters are:

- `propertyDef`: Model property definition in an object.
- `value`: Column value (any type)

```
SqlConnector.prototype.fromColumnValue = function(propertyDef, value) {
  /*jshint unused:false */
  throw new Error('fromColumnValue() must be implemented by the connector');
};
```

Helpers to generate SQL statements and parse responses from DB drivers

Define an `applyPagniation()` method to build a new SQL statement with pagination support by wrapping the specified SQL.

The parameters are:

- `model`: String model name
- `stmt`: The SQL statement as a ParameterizedSQL object.
- `filter` The filter object from the query

```
SqlConnector.prototype.applyPagination = function(model, stmt, filter) {
  throw new Error('applyPagination() must be implemented by the connector');
};
```

Implement a `getCountForAffectedRows()` method to parse the result for SQL UPDATE/DELETE/INSERT for the number of rows affected.

Parameters are:

- `model`: model name (string)
- `info`: Status object

The method returns the number of rows affected.

```
SqlConnector.prototype.getCountForAffectedRows = function(model, info) {
  /*jshint unused:false */
  throw new Error('getCountForAffectedRows() must be implemented by the connector');
};
```

Implement `getInsertedId()` to parse the result for SQL INSERT for newly inserted ID.

Parameters:

- `model`: Model name
- `info`: The status object from driver

It returns the inserted ID value.

```
SqlConnector.prototype.getInsertedId = function(model, info) {
  /*jshint unused:false */
  throw new Error('getInsertedId() must be implemented by the connector');
};
```

Implement `escapeName()` and `escapeValue()` methods to escape the name and value for the underlying database. They both return a string that is an escaped name for SQL.

Parameter:

- `name` The name (string).

```
SqlConnector.prototype.escapeName = function(name) {
  /*jshint unused:false */
  throw new Error('escapeName() must be implemented by the connector');
};

SqlConnector.prototype.escapeValue = function(value) {
  /*jshint unused:false */
  throw new Error('escapeValue() must be implemented by the connector');
};
```

Implement `getPlaceholderForIdentifier()` to get the placeholder in SQL for identifiers, such as ??. Implement `getPlaceholderForValue()` to get the placeholder in SQL for identifiers, such as :1 or ?.

Both methods return the placeholder as a string.

The `key` parameter is an optional key, such as 1 or id.

```
SqlConnector.prototype.getPlaceholderForIdentifier = function(key) {
  throw new Error('getPlaceholderForIdentifier() must be implemented by the
connector');
};

SqlConnector.prototype.getPlaceholderForValue = function(key) {
  throw new Error('getPlaceholderForValue() must be implemented by the connector');
};
```

Override other methods

There are a list of methods that serve as default implementations in the SqlConnector. The connector can choose to override such methods to customize the behaviors. Please see a complete list athttp://apidocs.strongloop.com/loopback-connector/.

Implementing transaction methods

To support database local transactions, the connector must implement the following methods.

Begin transaction

```
/**
 * Begin a new transaction
 * @param {String} isolationLevel
 * @param {Function} cb Callback function
 */
 MySQL.prototype.beginTransaction = function(isolationLevel, cb) {
   // get a connection from the pool
   // set up the isolation level
   // call back with the connection object
  };
```

Commit

```
/**
 * Commit a transaction
 * @param {Object} connection The connection object associated with the transaction
 * @param {Function} cb Callback function
 */
 MySQL.prototype.commit = function(connection, cb) {
   // commit the transaction
   // release the connection back to the pool
   // callback
  };
```

Rollback

```
/**
 * Rollback a transaction
 * @param {Object} connection The connection object associated with the transaction
 * @param {Function} cb Callback function
 */
 MySQL.prototype.rollback = function(connection, cb) {
   // rollback the transaction
   // release the connection back to the pool
   // callback
  };
```

ExecutecSQL

The transaction object is passed in via the *options.transaction.* The execution logic should check the presence of the *transaction* property and use the underlying *connection* so that the SQL statement will be executed as part of the transaction. For example,

```
if (transaction && transaction.connection &&
  transaction.connector === this) {
  if (debugEnabled) {
    debug('Execute SQL within a transaction');
  }
  executeWithConnection(null, transaction.connection);
} else {
  // Get a connection from the pool
  client.getConnection(executeWithConnection);
}
```

Implementing model discovery

Related articles:

For relational databases that have schema definitions, the connector can implement *discovery* to reverse engineer database schemas into model definitions. Implementing discovery is optional for a connector.

See Discovering models from relational databases for a general introduction to LoopBack model discovery.

Implementing functions to build SQL statements

You first need to implement methods to list schemas, tables, and views.

Build a SQL statement to list schemas

Implement a `querySchemas()` function that constructs and returns an SQL statement that lists all schemas (databases in MySQL).

It has a single parameter that is an options object.

It must return the SQL statement in a string.

```
function querySchemas(options) {
   // ...
 }
```

Build a SQL statement to list tables

Implement a `queryTables()` function that constructs and returns an SQL statement that lists tables in a given schema (database).

It has a single parameter that is an options object that specifies the owner or schema, or "all" for all owners/schemas.

It must return the SQL statement in a string.

```
function queryTables(options) { // ... }
```

Build a SQL statement to list views

Implement a `queryViews()` function that constructs and returns an SQL statement that lists views in a given database.

It has a single parameter that is an options object that specifies the owner, or "all" for all owners.

It must return the SQL statement in a string.

```
function queryViews(options) { // ... }
```

Build SQL statements to discover database objects

```
/**
 * Build the sql statement to query columns for a given table
 * @param schema
 * @param table
 * @returns {String} The sql statement
 */
 function queryColumns(schema, table) {
   // ...
 }

/**
 * Build the sql statement for querying primary keys of a given table
 * @param schema
 * @param table
 * @returns {string}
 */
 function queryPrimaryKeys(schema, table) {
   // ...
 }

/**
 * Build the sql statement for querying foreign keys of a given table
 * @param schema
 * @param table
 * @returns {string}
 */
 function queryForeignKeys(schema, table) {
   // ...
 }

/**
 * 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.
 * @param schema
 * @param table
 * @returns {string}
 */
 function queryExportedForeignKeys(schema, table) {
   // ...
 }
```

Implementing discovery functions

Discover schemas

```
MySQL.prototype.discoverDatabaseSchemas = function(options, cb) {
   // ...
 };
```

Discover a list of models

```
/**
 * Discover model definitions
 *
 * @param {Object} options Options for discovery
 * @param {Function} [cb] The callback function
 */
 MySQL.prototype.discoverModelDefinitions = function(options, cb) {
   // ...
 };
```

Discover a list of model properties for a given table

```
/**
 * Discover model properties from a table
 * @param {String} table The table name
 * @param {Object} options The options for discovery
 * @param {Function} [cb] The callback function
 *
 */
 MySQL.prototype.discoverModelProperties = function(table, options, cb) {
   // ...
 };
```

Discover primary keys for a given table

```
/**
 * Discover primary keys for a given table
 * @param {String} table The table name
 * @param {Object} options The options for discovery
 * @param {Function} [cb] The callback function
 */
 MySQL.prototype.discoverPrimaryKeys = function(table, options, cb) {
   // ...
 };
```

Discover foreign keys for a given table

```
/**
 * Discover foreign keys for a given table
 * @param {String} table The table name
 * @param {Object} options The options for discovery
 * @param {Function} [cb] The callback function
 */
 MySQL.prototype.discoverForeignKeys = function(table, options, cb) {
   // ...
 };
```

Discover exported foreign keys for a given table

```
/**
 * Discover foreign keys that reference to the primary key of this table
 * @param {String} table The table name
 * @param {Object} options The options for discovery
 * @param {Function} [cb] The callback function
 */
 MySQL.prototype.discoverExportedForeignKeys = function(table, options, cb) {
   // ...
 };
```

Discover indexes for a given table

```
MySQL.prototype.discoverIndexes = function(table, options, cb) {
    // ...
  };
```

Map column definition to model property definition

```
MySQL.prototype.buildPropertyType = function(columnDefinition) {
    // ...
  }
```

Working with data

Once you have defined a model, then you can use create, read, update, and delete (CRUD) operations to add data to the model, manipulate the data, and query it. All LoopBack models that are connected to persistent data stores (such as a database) automatically have the CRUD operations of the PersistedModel class.

Operation	REST	LoopBack model method (Node API)*	Corresponding SQL Operation
Create	PUT */modelName* POST */modelName*	`create()`*	INSERT
Read (Retrieve)	GET /modelName?filter=...	`find()`*	SELECT
Update (Modify)	POST */modelName* PUT /modelName	`updateAll()`*	UPDATE
Delete (Destroy)	DELETE */modelName/modelID*	`destroyById()`*	DELETE

*Methods listed are just prominent examples; other methods may provide similar functionality; for example: `findById()`, `findOne()`, and `findOrCreate()`. See PersistedModel API documentation for more information.

See the following articles for more information:

- Creating, updating, and deleting data
- Querying data
  - Fields filter
  - Include filter
  - Limit filter
  - Order filter
  - Skip filter
  - Where filter
- Using database transactions
- Realtime server-sent events

> Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

Creating, updating, and deleting data

> Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

PersistedModel has a large set of methods for creating, updating, and deleting data.

Model data is also called a *model instance*; in database terminology, conceptually a model corresponds to a table, and a model instance corresponds to a *row* or *record* in the table.

> For information on model *read* operations, see Querying data.

- Creating data (model instances)
- Updating data (model instances)
  - Performing bulk updates
- Deleting data

Creating data (model instances)

Use the following PersistedModel methods to add data, that is to insert or create instances:

- create - creates a new model instance (record).
- upsert - checks if the instance (record) exists, based on the designated ID property, which must have a unique value; if the instance already exists, the method updates that instance. Otherwise, it inserts a new instance.
- findOrCreate - Find one instance matching the filter object provided as the first parameter. If found, returns the object. If not found, creates a new instance (record).

  > Be sure to include a `where` clause in the filter object. Without the `where`, the `findOrCreate` finds and returns the first record in the collection, without error, which can lead to unintended behavior.
- save - Save model instance. If the instance doesn't have an ID, then calls create instead. Triggers: validate, save, update, or create.

Updating data (model instances)

Static method (called on the Model object):

- updateAll - updates multiple instances (records) that match the specified where clause.

> The where clause used with `updateAll()` is slightly different than that for queries. Omit `{ where : ... }` from the where clause. Simply provide the condition as the first argument.
>
> For more information, see Where filter (Where clause for updates and deletes).

Instance methods (called on a single model instance):

- updateAttribute - Update a single attribute (property).
- updateAttributes - Update set of attributes (proerties). Performs validation before updating.

Performing bulk updates

- createUpdates
- bulkUpdate

Deleting data

Static methods (called on the Model object):

- destroyAll - Delete all model instances that match the optional Where filter.
- destroyById - Delete the model instance with the specified ID.

> The where clause with `destroyAll()` is slightly different than that for queries. Omit `{ where : ... }` from the where clause. Simply provide the condition as the first argument.
>
> For more information, see Where filter (Where clause for updates and deletes).

Querying data

> Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

- Overview
  - Examples
- Filters
  - REST syntax
  - Node syntax
  - Using "stringified" JSON in REST queries
  - Filtering arrays of objects

Related articles:

See also: Querying related models.

Overview

A *query* is a read operation on models that returns a set of data or results. You can query LoopBack models using a Node API and a REST API, using *filters*, as outlined in the following table. Filters specify critera for the returned data set. The capabilities and options of the two APIs are the same–the only difference is the syntax used in HTTP requests versus Node function calls. In both cases, LoopBack models return JSON.

Query	Model API (Node)	REST API
Find all model instances using specified filters.	`find(filter, callback)` Where filter is a JSON object containing the query filters. See Filters below.	`GET /modelName?filter...` See Model REST API - Find matching instances. See Filters below.
Find first model instance using specified filters.	`findOne(filter, callback)` Where filter is a JSON object containing the query filters. See Filters below.	`GET /modelName/findOne?filter ...` See Model REST API - Find first instance. See Filters below.
Find instance by ID.	`findById(id, [filter,] callback)` Where optional filter is a JSON object containing the query filters. See Filters below.	`GET /modelName/modelID` See Model REST API - Find instance by ID.

A REST query must include the literal string "filter" in the URL query string. The Node API call does not include the literal string "filter" in the JSON.

LoopBack API Explorer adds "filter" to the query string, but you must enter Stringified JSON in the **filter** field. Also make sure that the quotes you use are proper straight quotes ("), not curved or typographic quotation marks (“ or ”). These can often be hard to distinguish visually.

If you are trying query filters with curl, use the `-g` or `--globoff` option to use brackets `[` and `]` in request URLs.

LoopBack supports the following kinds of filters:

- Fields filter
- Include filter
- Limit filter
- Order filter
- Skip filter
- Where filter

See Filters below for more information.

Examples

See additional examples of each kind of filter in the individual articles on filters (for example Where filter).

An example of using the `find()` method with both a *where* and a *limit* filter:

```
Account.find({where: {name: 'John'}, limit: 3}, function(err, accounts) { ... });
```

Equivalent using REST:

```
/accounts?filter[where][name]=John&filter[limit]=3
```

Filters

In both REST and Node API, you can use any number of filters to define a query.

LoopBack supports a specific filter syntax: it’s a lot like SQL, but designed specifically to serialize safely without injection and to be native to JavaScript. Previously, only the `PersistedModel.find()` method (and related methods) supported this syntax.

The following table describes LoopBack's filter types:

Filter type	Type	Description
fields	Object, Array, or String	Specify fields to include in or exclude from the response. See Fields filter.
include	String, Object, or Array	Include results from related models, for relations such as *belongsTo* and *hasMany*. See Include filter.
limit	Number	Limit the number of instances to return. See Limit filter.
order	String	Specify sort order: ascending or descending. See Order filter.
skip (offset)	Number	Skip the specified number of instances. See Skip filter.

where	Object	Specify search criteria; similar to a WHERE clause in SQL. See Where filter.

REST syntax

Specify filters in the HTTP query string:

```
?filterfilterType=spec&filterType=spec....
```

The number of filters that you can apply to a single request is limited only by the maximum URL length, which generally depends on the client used.

There is no equal sign after `?filter` in the query string; for example `http://localhost:3000/api/books?filter[where][id]=1`

See https://github.com/hapijs/qs for more details.

Node syntax

Specify filters as the first argument to `find()` and `findOne()`:

```
{ filterType: spec, filterType: spec, ... }
```

There is no theoretical limit on the number of filters you can apply.

Where:

- *filterType* is the filter: where, include, order, limit, skip, or fields.
- *spec* is the specification of the filter: for example for a *where* filter, this is a logical condition that the results must match; for an *include* filter it specifies the related fields to include.

Using "stringified" JSON in REST queries

Instead of the standard REST syntax described above, you can also use "stringified JSON" in REST queries. To do this, simply use the JSON specified for the Node syntax, as follows:

```
?filter={ Stringified-JSON }
```

where *Stringified-JSON* is the stringified JSON from Node syntax. However, in the JSON all text keys/strings must be enclosed in quotes (").

When using stringified JSON, you must use an equal sign after `?filter` in the query string; for example `http://localhost:3000/api/books?filter={%22where%22:{%22id%22:2}}`

For example:

```
GET /api/activities/findOne?filter={"where":{"id":1234}}
```

Filtering arrays of objects

The loopback-filters module implements LoopBack's filter syntax. Using this module, you can filter arrays of objects using the same filter syntax supported by `MyModel.find(filter)`.

We plan to convert all modules to use `loopback-filter`, so it will become LoopBack's common “built-in” filtering mechanism.

Here is a basic example using the new module.

```
var data = [{n: 1}, {n: 2}, {n: 3, id: 123}];
var filter = {where: {n: {gt: 1}}, skip: 1, fields: ['n']};
var filtered = require('loopback-filters')(data, filter);
console.log(filtered); // => [{n: 3}]
```

For a bit more detail, say you are parsing a comma-separated value (CSV) file, and you need to output all values where the price column is between 10 and 100. To use the LoopBack filter syntax you would need to either create your own CSV connector or use the memory connector, both of which require some extra work not related to your actual goal.

Once you've parsed the CSV (with a module like `node-csv`) you will have an array of objects like this, for example (but with, say, 10,000 unique items):

```
[
  {price: 85, id: 79},
  {price: 10, id: 380},
  ...
]
```

To filter the rows you could use generic JavaScript like this:

```
data.filter(function(item) {
return item.price < 100 && item.price >= 10
});
```

This is pretty simple for filtering, but sorting, field selection, and more advanced operations become a bit tricky. On top of that, you are usually accepting the parameters as input; for example:

```
var userInput = {min: 10, max: 100}

data.filter(function(item) {
   return item.price < userInput.min && item.price >= userInput.max
});
```

You can rewrite this easily as a LoopBack filter:

```
filter(data, {where: {input: {gt: userInput.min, lt: userInput.max}}})
```

Or if you just adopt the filter object syntax as user input:

```
filter(data, userInput)
```

But `loopback-filters` supports more than just excluding and including. It supports field selection (including / excluding fields), sorting, geo/distance sorting, limiting and skipping. All in a declarative syntax that is easily created from user input.

As a LoopBack user this is a pretty powerful thing. Typically, you will have learned how to write some complex queries using the `find()` filter syntax; before you would need to figure out how to do the same thing in JavaScript (perhaps using a library such as underscore). Now with the `loopback-filters` module, in your client application you can re-use the same exact filter object you were sending to the server to filter the database without having to interact with a LoopBack server at all.

Fields filter

A *fields* filter specifies properties (fields) to include or exclude from the results.

- REST API
- Node API
- Examples

REST API

```
filter[fields][propertyName]=<true|false>&filter[fields][propertyName]=<true|false>...
```

Note that to include more than one field in REST, use multiple filters.

You can also use stringified JSON format in a REST query.

Node API

Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

```
{ fields: {propertyName: <true|false>, propertyName: <true|false>, ... } }
```

Where:

- *propertyName* is the name of the property (field) to include or exclude.
- <true|false> signifies either `true` or `false` Boolean literal.   Use `true` to include the property or `false` to exclude it from results.
  You can also use 1 for true and 0 for false.

By default, queries return all model properties in results. However, if you specify at least one fields filter with a value of `true`, then by default the query will include **only** those you specifically include with filters.

Examples

Return only `id`, `make`, and `model` properties:

REST

```
?filter[fields][id]=true&filter[fields][make]=true&filter[fields][model]=true
```

Node API

```
{ fields: {id: true, make: true, model: true} }
```

Returns:

```
[
  {
    "id": "1",
    "make": "Nissan",
    "model": "Titan"
  },
  {
    "id": "2",
    "make": "Nissan",
    "model": "Avalon"
  },
  ...
]
```

Exclude the `vin` property:

REST

```
?filter[fields][vin]=false
```

Node API

```
{ fields: {vin: false} }
```

Include filter

See also: Querying related models.

An *include* filter enables you to include results from related models in a query, for example models that have belongsTo or hasMany relations, to optimize the number of requests. See Creating model relations for more information.

The value of the include filter can be a string, an array, or an object.

You can use an *include* filter with `find(), findOne()` and `findById()`.

REST API

filter[include][*relatedModel*]=*propertyName*

You can also use stringified JSON format in a REST query.

Node API

Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

{include: '*relatedModel*'}
{include: ['*relatedModel1*', '*relatedModel2*', ...]}
{include: {*relatedModel1*: [{*relatedModel2*: '*propertyName*'} , '*relatedModel*']}}

Where:

- *relatedModel*, *relatedModel1*, and *relatedModel2* are the names (pluralized) of related models.
- *propertyName* is the name of a property in the related model.

Examples

Include relations without filtering

```
User.find({include: 'posts'}, function() { ... });
```

Return all user posts and orders with two additional requests:

```
User.find({include: ['posts', 'orders']}, function() { ... });
```

Return all post owners (users), and all orders of each owner:

```
Post.find({include: {owner: 'orders'}}, function() { ... });
```

Return all post owners (users), and all friends and orders of each owner:

```
Post.find({include: {owner: ['friends', 'orders']}}, function() { ... });
```

Return all post owners (users), and all posts and orders of each owner. The posts also include images.

```
Post.find({include: {owner: [{posts: 'images'} , 'orders']}}, function() { ... });
```

Include with filters

In some cases, you may want to apply filters to related models to be included. LoopBack supports that with the following syntax (for example):

```
Post.find({
  include: {
    relation: 'owner', // include the owner object
    scope: { // further filter the owner object
      fields: ['username', 'email'], // only show two fields
      include: { // include orders for the owner
        relation: 'orders',
        scope: {
          where: {orderId: 5} // only select order with id 5
        }
      }
    }
  }
}, function() { ... });
```

For real-world scenarios where only users in `$authenticated` or `$owner` roles should have access, use `findById()`. For example, the following example uses filters to perform pagination:

```
Post.findById('123', {
  include: {
    relation: 'orders',
    scope: { // fetch 1st "page" with 5 entries in it
      skip:0,
      limit:5
    }
  }
}, function() { ... });
```

Access included objects

With Node.js API, you need to call toJSON() to convert the returned model instance with related items into a plain JSON object. For example:

```
Post.find({include: {owner: [{posts: 'images'} , 'orders']}}, function(err, posts) {
 posts.forEach(function(post) {
   // post.owner points to the relation method instead of the owner instance
   var p = post.toJSON();
   console.log(p.owner.posts, p.owner.orders);
 });
 ...
});
```

Please note the relation properties such as *post.owner* points to a JavaScript **function** for the relation method.

REST examples

These examples assume a customer model with a hasMany relationship to a reviews model.

Return all customers including their reviews:

```
/customers?filter[include]=reviews
```

Return all customers including their reviews which also includes the author:

```
/customers?filter[include][reviews]=author
```

Return all customers whose age is 21, including their reviews which also includes the author:

```
/customers?filter[include][reviews]=author&filter[where][age]=21
```

Return first two customers including their reviews which also includes the author

```
/customers?filter[include][reviews]=author&filter[limit]=2
```

Return all customers including their reviews and orders

```
/customers?filter[include]=reviews&filter[include]=orders
```

Limit filter

A *limit* filter limits the number of records returned to the specified number (or less).

Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

REST API

filter[limit]=*n*

Node API

{limit: *n*}

You can also use stringified JSON format in a REST query.

Where *n* is the maximum number of results (records) to return.

Examples

Return only the first five query results:

REST

```
/cars?filter[limit]=5
```

Node API

```
Cars.find( {limit: 5},  function() { ... } )
```

Order filter

An *order* filter specifies how to sort the results: ascending (ASC) or descending (DESC) based on the specified property.

- REST API
- Node API
- Examples

REST API

Order by one property:

```
filter[order]=propertyName <ASC|DESC>
```

Order by two or more properties:

```
filter[order][0]=propertyName <ASC|DESC>&filter[order][1]propertyName]=<ASC|DESC>...
```

You can also use stringified JSON format in a REST query.

> Default ordering can be configured in default scope.

Node API

> Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

Order by one property:

```
{ order: 'propertyName <ASC|DESC>' }
```

Order by two or more properties:

```
{ order: ['propertyName <ASC|DESC>', 'propertyName <ASC|DESC>',...] }
```

Where:

- *propertyName* is the name of the property (field) to sort by.
- `<ASC|DESC>` signifies either ASC for ascending order or DESC for descending order.

Examples

Return the three loudest three weapons, sorted by the `audibleRange` property:

REST

```
/weapons?filter[order]=audibleRange%20DESC&filter[limit]=3
```

Node API

```
weapons.find({
 order: 'price DESC',
 limit: 3 });
```

Skip filter

A skip filter omits the specified number of returned records. This is useful, for example, to paginate responses.

Use `offset` as an alias for `skip`.

Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

REST API

?filter=[skip]=*n*

Node

{ skip: *n* }

You can also use stringified JSON format in a REST query.

Where *n* is the number of records to skip.

Examples

This REST request skips the first 50 records returned:

```
/cars?filter[skip]=50
```

The equivalent query using the Node API:

```
Cars.find( {skip: 50},  function() { ... } )
```

Pagination example

The following REST requests illustrate how to paginate a query result. Each request request returns ten records: the first returns the first ten, the second returns the 11th through the 20th, and so on...

```
/cars?filter[limit]=10&filter[skip]=0
/cars?filter[limit]=10&filter[skip]=10
/cars?filter[limit]=10&filter[skip]=20
...
```

Using the Node API:

```
Cars.find( {limit: 10, skip: 0},  function() { ... } );
Cars.find( {limit: 10, skip: 10},  function() { ... } );
Cars.find( {limit: 10, skip: 20}, function() { ... } );
```

Where filter

A *where* filter specifies a set of logical conditions to match, similar to a WHERE clause in a SQL query.

- near
- like and nlike
- inq

REST API

In the first form below, the condition is equivalence, that is, it tests whether *property* equals *value*. The second form below is for all other conditions.

filter[where][*property*]=*value*

filter[where][*property*][*op*]=*value*

For example, if there is a cars model with a `odo` property, the following query finds instances where the `odo` is exactly equal to 5000:

```
/cars?filter[where][odo][gt]=5000
```

For example, here is a query to find cars with `odo` is less than 30,000:

```
/cars?filter[where][odo][lt]=30000
```

You can also use stringified JSON format in a REST query.

Node API

Methods of models in the AngularJS client have a different signature than those of the Node API. For more information, see AngularJS SDK API.

Where clause for queries

For query methods such as `find()`, `findOrCreate()`, or `findOne()`, use the first form below to test equivalence, that is, whether *property* e quals *value*. Use the second form below for all other conditions.

{where: {*property*: *value*}}

{where: {*property*: {*op*: *value*} } }

Where:

- *property* is the name of a property (field) in the model being queried.
- *value* is a literal value.
- *op* is one of the operators listed below.

```
Cars.find({ where: {carClass:'fullsize'} });
```

The above where clause syntax is for queries. For all other methods, omit the `{ where : ... }` wrapper; see Where clause for other methods below.

Where clause for other methods

When you call the Node APIs *for methods other than queries*, that is for methods that update and delete, don't wrap the where clause in a `{ where : ... }` object, simply use the condition as the argument as shown below. See examples below.

In the first form below, the condition is equivalence, that is, it tests whether *property* equals *value*. The second form is for all other conditions.

{*property*: *value*}

{*property*: {*op*: *value*}}

Where:

- *property* is the name of a property (field) in the model being queried.
- *value* is a literal value.
- *op* is one of the operators listed below.

For example, below shows a where clause in a call to a model's updateAll() method. Note the lack of `{ where : ... }` in the argument.

```
var myModel = req.app.models.Thing;
var theId = 12;
myModel.updateAll( {id: theId}, {regionId: null}, function(err, results) {
 return callback(err, results);
});
```

More examples, this time in a call to destroyAll():

```
var RoleMapping = app.models.RoleMapping;
RoleMapping.destroyAll( { principalId: userId }, function(err, obj) { ... } );
```

To delete all records where the cost property is greater than 100:

```
productModel.destroyAll( { cost: {gt: 100} }, function(err, obj) { ... } )
```

Operators

This table describes the operators available in "where" filters. See Examples below.

Operator	Description
and	Logical AND operator
or	Logical OR operator
gt, gte	Numerical greater than (>); greater than or equal (>=). Valid only for numerical and date values. For Geopoint values, the units are in miles by default. See Geopoint for more information.
lt, lte	Numerical less than (<); less than or equal (<=). Valid only for numerical and date values. For geolocation values, the units are in miles by default. See Geopoint for more information.
between	True if the value is between the two specified values: greater than or equal to first value and less than or equal to second value. For geolocation values, the units are in miles by default. See Geopoint for more information.
inq, nin	In / not in an array of values.
near	For geolocations, return the closest points, sorted in order of distance. Use with `limit` to return the n closest points.
neq	Not equal (!=)
like, nlike	LIKE / NOT LIKE operators for use with regular expressions. The regular expression format depends on the backend data source.
regexp	Regular expression.

AND and OR operators

Use the AND and OR operators to create compound logical filters based on simple where filter conditions, using the following syntax.

Node API

{where: {<and|or>: [*condition1*, *condition2*, ...] } }

REST

[where][<and|or>][0]condition1&[where][<and|or>]*condition2*...

Where *condition1* and *condition2* are a filter conditions.

See examples below.

Regular expressions

You can use regular expressions in a where filter, with the following syntax. You can use a regular expression in a where clause for updates and deletes, as well as queries.

Essentially, `regexp` is just like an operator in which you provide a regular expression value as the comparison value.

> A regular expression value can also include one or more flags. For example, append `/i` to the regular expression to perform a case-insensitive match.

Node API

{ where: {*property*: {regexp: *expression*} } }

Where *expression* can be a:

- String defining a regular expression (for example, `'^foo'`).
- Regular expression literal (for example, `/^foo/`).
- Regular expression object (for example, `new RegExp(/John/)`).

Or, in a simpler format:

{ where: {*property*: *expression* } } }

Where *expression* can be a:

- Regular expression literal (for example, `/^foo/`).
- Regular expression object (for example, `new RegExp(/John/)`).

For more information on JavaScript regular expressions, see Regular Expressions (Mozilla Developer Network).

REST

filter[where][*property*][regexp]=*value*

Where:

- *property* is the name of a property (field) in the model being queried.
- *expression* is the JavaScript regular expression string. See Regular Expressions (Mozilla Developer Network).

A regular expression value can also include one or more flags. For example, append `/i` to the regular expression to perform a case-insensitive match.

> The above where clause syntax is for queries. For updates and deletes, omit the `{ where : ... }` wrapper; see Where clause for updates and deletes below.

For example, this query will return all cars for which the model starts with a capital "T":

```
Cars.find( {"where": {"model": {"regexp": "^T"}}} );
```

Or, using the simplified form:

```
Cars.find( {"where": {"model": /^T/} } );
```

Equivalently, with REST:

```
/api/cars?filter[where][make][regexp]=^T
```

Or to match models that start with either an uppercase "T" or lowercase "t":

```
/api/cars?filter[where][make][regexp]=^t/i
```

Examples

Equivalence

Weapons with name M1911:

REST

```
/weapons?filter[where][name]=M1911
```

Cars where carClass is "fullsize":

REST

```
/api/cars?filter[where][carClass]=fullsize
```

Equivalently, in Node:

```
Cars.find({ where: {carClass:'fullsize'} });
```

gt and lt

```
ONE_MONTH = 30 * 24 * 60 * 60 * 1000;  // Month in milliseconds
transaction.find({
      where: {
        userId: user.id,
        time: {gt: Date.now() - ONE_MONTH}
      }
    }
```

For example, the following query returns all instances of the employee model using a *where* filter that specifies a date property after (greater than) the specified date:

```
/employees?filter[where][date][gt]=2014-04-01T18:30:00.000Z
```

The same query using the Node API:

```
Employees.find({
  where: {
    date: {gt: new Date('2014-04-01T18:30:00.000Z')}
  }
});
```

The top three weapons with a range over 900 meters:

```
/weapons?filter[where][effectiveRange][gt]=900&filter[limit]=3
```

Weapons with audibleRange less than 10:

```
/weapons?filter[where][audibleRange][lt]=10
```

and / or

The following code is an example of using the "and" operator to find posts where the title is "My Post" and content is "Hello".

```
Post.find({where: {and: [{title: 'My Post'}, {content: 'Hello'}]}},
          function (err, posts) {
            ...
});
```

Equivalent in REST:

```
?filter[where][and][0][title]=My%20Post&filter[where][and][1][content]=Hello
```

Example using the "or" operator to finds posts that either have title of "My Post" or content of "Hello".

```
Post.find({where: {or: [{title: 'My Post'}, {content: 'Hello'}]}},
          function (err, posts) {
            ...
});
```

More complex example. The following expresses `(field1= foo and field2=bar) OR field1=morefoo`:

```
{
  "or": [
 "and": [ {"field1": "foo"}, {"field2": "bar"}  ],
  "field1": "morefoo"
   ]
}
```

between

Example of between operator:

```
filter[where][price][between][0]=0&filter[where][price][between][1]=7
```

In Node API:

```
Shirts.find({where: {size: {between: [0,7]}}}, function (err, posts) { ... } )
```

near

Example using the **near** operator that returns the three closest locations to a given geo point:

```
/locations?filter[where][geo][near]=153.536,-28.1&filter[limit]=3
```

like and nlike

The like and nlike (not like) operators enable you to match SQL regular expressions. The regular expression format depends on the backend data source.

Example of like operator:

```
Post.find({where: {title: {like: 'M.+st'}}}, function (err, posts) { ... });
```

Example of nlike operator:

```
Post.find({where: {title: {nlike: 'M.+XY'}}}, function (err, posts) {
```

When using the memory connector:

```
User.find({where: {name: {like: '%St%'}}}, function (err, posts) { ... });
User.find({where: {name: {nlike: 'M%XY'}}}, function (err, posts) { ... });
```

inq

The inq operator checks whether the value of the specified property matches any of the values provided in an array. The general syntax is:

{ where: { *property*: { inq: [*val1*, *val2*, ...] } } }

where:

- *property* is the name of a property (field) in the model being queried.
- *val1, val2*, and so on, are literal values in an array.

Example of inq operator:

```
Posts.find({where: {id: {inq: [123, 234]}}},
  function (err, p){... });
```

REST:

```
/medias?filter[where][keywords][inq]=foo&filter[where][keywords][inq]=bar
```

Or

```
?filter={"where": {"keywords": {"inq": ["foo", "bar"]}}}
```

Using database transactions

Transaction support was added in loopback-datasource-juggler version 2.28.0.

Overview

A *transaction* is a sequence of data operations performed as a single logical unit of work. Many relational databases support transactions to help enforce data consistency and business logic requirements.

A LoopBack model can perform operations in a transaction when the model is attached to one of the following connectors:

- MySQL connector (IMPORTANT: Only with InnoDB as the storage engine).
- PostgreSQL connector
- SQL Server connector
- Oracle connector

Transaction APIs

See the API reference for full transaction API documentation.

Performing operations in a transaction typically involves the following steps:

- Start a new transaction.
- Perform create, read, update, and delete (CRUD) operations in the transaction.
- Commit or rollback the transaction.

Start transaction

Use the beginTransaction method to start a new transaction. For example, for a `Post` model:

```
Post.beginTransaction({isolationLevel: Post.Transaction.READ_COMMITTED}, function(err,
tx) {
  // Now we have a transaction (tx)
});
```

Isolation levels

When you call beginTransaction(), you can optionally specify a transaction isolation level. LoopBack transactions support the following isolation levels:

- Transaction.READ_UNCOMMITTED
- Transaction.READ_COMMITTED (default)

- Transaction.REPEATABLE_READ
- Transaction.SERIALIZABLE

If you don't specify an isolation level, the transaction uses READ_COMMITTED .

Oracle only supports READ_COMMITTED and SERIALIZABLE.

For more information about database-specific isolation levels, see:

- MySQL SET TRANSACTION Syntax
- Oracle Isolation Levels
- PostgreSQL Transaction Isolation
- SQL Server SET TRANSACTION ISOLATION LEVEL

Perform operations in a transaction

To perform CRUD operations in the transaction, add a second argument consisting of the transaction object to the standard `create()`, `upsert()`, `destroyAll()` (and so on) methods.

For example, again assuming a `Post` model:

```
Post.create({title: 't1', content: 'c1'}, {transaction: tx}, function(err, post) {
  post.updateAttributes({content: 'c2', {transaction: tx}, function(err, newPost) {
    //
    newPost.reviews.create({content: 'r1'}, {transaction: tx}, function(err, newPost)
{
    });
  }
});
```

Commit or rollback

Commit the transaction:

```
transaction.commit(function(err) {
});
```

Or to rollback the transaction:

```
transaction.rollback(function(err) {
});
```

Please note all three APIs support the Promise flavor. See an example at https://github.com/strongloop/loopback-connector-mysql/blob/master/test/transaction.promise.test.js.

Set up timeout

You can specify a timeout (in milliseconds) to begin a transaction. If a transaction is not finished (committed or rolled back) before the timeout, it will be automatically rolled back upon timeout by default. The timeout event can be trapped using the timeout hook. For example, again assuming a `Post` model:

```
Post.beginTransaction({
    isolationLevel: Transaction.READ_COMMITTED,
    timeout: 30000 // 30000ms = 30s
  }, function(err, tx) {
    tx.observe('timeout', function(context, next) {
      // handle timeout
      next();
    });
});
```

Propagate a transaction

Propagating a transaction is explicit by passing the transaction object via the options argument for all CRUD and relation methods. For example, again assuming a `Post` model:

```
var options = {transaction: tx};
Post.create({title: 't1', content: 'c1'}, options, function(err, post) {
  post.updateAttributes({content: 'c2', options, function(err, newPost) {
    //
    newPost.reviews.create({content: 'r1'}, options, function(err, newPost) {
    });
  }
});
```

Set up transaction hooks

There are four types of observable events for a transaction:

- before commit
- after commit
- before rollback
- after rollback
- timeout

```
tx.observe('before commit', function(context, next) {
  // ...
  next();
});
tx.observe('after commit', function(context, next) {
  // ...
  next();
});
tx.observe('before rollback', function(context, next) {
  // ...
  next();
});
tx.observe('after rollback', function(context, next) {
  // ...
  next();
});
```

Avoid long waits or deadlocks

Please be aware that a transaction with certain isolation level will lock database objects. Performing multiple methods within a transaction asynchronously has the great potential to block other transactions (explicit or implicit). To avoid long waits or even deadlocks, you should:

1. Keep the transaction as short-lived as possible
2. Don't serialize execution of methods across multiple transactions

Realtime server-sent events

See also:

- Angular Live-set example
- Blog post

Overview

The PersistedModel API supports streaming changes from servers to clients using a combination of the CRUD methods and the `createChangeStream()` method.

A ChangeStream enables a server to send model changes to a client.   A client makes an initial request to be notified of changes and then the server "pushes" these changes to the client.

Creating ChangeStreams on the server

Setup

First, add event-stream to your Node app as follows:

```
$ npm install -save event-stream
```

This will add a line something like this to the app's `package.json` file:

```
...
  "event-stream": "^3.3.1",
...
```

Disable compression

Event streams don't work with Node compression. To disable npm compression, delete the entry from `server/middleware.json` so it looks like this:

```
...
"compression": {
  "enabled":false
},
...
```

Script

Below is a basic example using the `createChangeStream()` method in a LoopBack application.

server/boot/realtime.js

```
var es = require('event-stream');
module.exports = function(app) {
  var MyModel = app.models.MyModel;
  MyModel.createChangeStream(function(err, changes) {
    changes.pipe(es.stringify()).pipe(process.stdout);
  });
  MyModel.create({foo: 'bar'});
}
```

This example will print the following to the console:

```
{"target":1,"data":{"foo":"bar","id":1},"type":"create"}
```

Pushing data to clients

This example shows how to consume the `ChangeStream` from a browser using the EventSource API, which is built in to JavaScript implemented in most browsers. The example code below assumes a model called `MyModel` and simply logs the response to the browser JavaScript console.

Browser script

```
var urlToChangeStream = '/api/MyModels/change-stream?_format=event-stream';
var src = new EventSource(urlToChangeStream);
src.addEventListener('data', function(msg) {
  var data = JSON.parse(msg.data);
  console.log(data); // the change object
});
```

To push data, the model on the server must change; for example, if you add a new record (model instance).

When this occurs, then in the browser JavaScript console, you will see (for example):

```
Object {target: 2, data: Object, type: "create"}
```

Using ChangeStreams in AngularJS

The angular-live-set module makes it easy to use `ChangeStream` in your AngularJS applications.

Adding application logic

Overview

When building an application, you'll generally need to implement custom logic to process data and perform other operations before responding to client requests. In LoopBack, there are three ways to do this:

- **Adding logic to models** - adding remote methods, remote hooks and operation hooks.
- **Defining boot scripts** - writing scripts (in the `/server/boot` directory) that run when the application starts.
- **Defining middleware** - adding custom middleware to the application .

Adding logic to models

There are three ways to add custom application logic to models:

- Remote methods - REST endpoints mapped to Node functions.
- Remote hooks - Logic that triggers when a remote method is executed (before or after).
- Operation hooks - Logic triggered when a model performs CRUD (create, read, update, and delete) operations.

See also:

- Defining boot scripts
- Defining middleware

You can further refine the timing of custom logic by configuring how you call each method. In any case, you will be required to code your own logic as LoopBack simply provides the mechanisms to trigger your logic.

Remote methods

Related articles:

Overview

A *remote method* is a static method of a model, exposed over a custom REST endpoint. Use a remote method to perform operations not provided by LoopBack's standard model REST API.

For an introductory example of defining a remote method, see Extend your API in Getting Started.

How to define a remote method

To define a remote method:

1. Edit the Model definition JSON file in `/common/models` directory; for example, to attach a remote method to the Person model, edit `/common/models/person.js`. If you created the model with the Model generator, then this file will already exist.

   The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

2. Define a static method that will handle the request.
3. Call `remoteMethod()`, to register the method, calling it with two parameters:
   a. First parameter is a string that is the name of the method you defined in step 2
   b. Second (optional) parameter provides addtional configuration for the REST endpoint.

Example

See additional introductory examples in Extend your API.

Suppose you have a Person model and you want to add a REST endpoint at `/greet` that returns a greeting with a name provided in the request. You add this code to `/common/models/person.js`:

/common/models/person.js

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

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

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

Now, for example, a request to

`POST /api/people/greet`

with data `{"msg": "John"}`

will return:

shell

```
Greetings... John!
```

> Notice the REST API request above uses the plural form "people" instead of "person". LoopBack exposes the plural form of model names for REST API routes.

Registering a remote method

All LoopBack models have a `remoteMethod()` static method that you use to register a remote method:

model.remoteMethod(*requestHandlerFunctionName*, [*options*])

where:

- *`model`* is the model object to which you're adding the remote method. In our example, `Person`.
- *`requestHandlerFunctionName`* is a string that specifies name of the remote method, for example `'greet'`.
- *`options`* is an object that specifies parameters to configure the REST endpoint; see below.

Options

The options argument is a Javascript object containing key/value pairs to configure the remote method REST endpoint.

> All of the options properties are optional. However, if the remote method requires arguments, you must specify `accepts`; if the remote method returns a value, you must specify `returns`.

Option	Description	Example

<table>
<tr><td>accepts</td><td>Defines arguments that the remote method accepts. These arguments map to the static method you define. For the example above, you can see the function signature:<pre><code>Person.greet(name, age, callback)...</code></pre>`name` is the first argument, `age` is the second argument and callback is automatically provided by LoopBack (do not specify it in your `accepts` array). For more info, see Argument descriptions.

Default if not provided is the empty array, <code>[]</code>.</td><td><pre><code>{ ...
  accepts: [
   {arg: 'name',
type: 'string'},
   {arg: 'age',
type:
'number'},...],
  ... }</code></pre></td></tr>
<tr><td>description</td><td>Text description of the method, used by API documentation generators such as Swagger.

You can put long strings in an array if needed (see note below).</td><td></td></tr>
<tr><td>http.path</td><td>HTTP path (relative to the model) at which the method is exposed.</td><td><pre><code>http: {path:
'/sayhi'}</code></pre></td></tr>
<tr><td>http.verb</td><td>HTTP method (verb) at which the method is available. One of:
get
post (default)
patch
put
del
all</td><td><pre><code>http: {path:
'/sayhi', verb:
'get'}</code></pre></td></tr>
<tr><td>http.status</td><td>Default HTTP status set when the callback is called without an error.</td><td><pre><code>http: {status:
201}</code></pre></td></tr>
<tr><td>http.errorStatus</td><td>Default HTTP status set when the callback is called with an error.</td><td><pre><code>http:
{errorStatus:
400}</code></pre></td></tr>
<tr><td>isStatic</td><td>Boolean. Whether the method is static (eg. <code>MyModel.myMethod</code>). Use <code>false</code> to define the method on the prototype (for example, <code>MyModel.prototype.myMethod</code>). Default is true.</td><td></td></tr>
<tr><td>notes</td><td>Additional notes, used by API documentation generators like Swagger.

You can put long strings in an array if needed (see note below).</td><td></td></tr>
<tr><td>returns</td><td>Describes the remote method's callback arguments; See Argument descriptions. The <code>err</code> argument is assumed; do not specify.

Default if not provided is the empty array, <code>[]</code>.</td><td><pre><code>returns: {arg:
'greeting',
type: 'string'}</code></pre></td></tr>
</table>

You can split long strings in the `description` and `notes` options into arrays of strings (lines) to keep line lengths manageable. For example:

```
[
"Lorem ipsum dolor sit amet, consectetur adipiscing elit,"
"sed do eiusmod tempor incididunt ut labore et dolore",
"magna aliqua."
]
```

Argument descriptions

The `accepts` and `returns` options properties define either a single argument as an object or an ordered set of arguments as an array. The following table describes the properties of each individual argument.

Property (key)	Type	Description
arg	String	Argument name

description	String or Array	A text description of the argument. This is used by API documentation generators like Swagger.
		
		You can split long descriptions into arrays of strings (lines) to keep line lengths manageable.
		
		`[`
		`"Lorem ipsum dolor sit amet, consectetur adipiscing elit,"`
		`"sed do eiusmod tempor incididunt ut labore et dolore",`
		`"magna aliqua."`
		`]`

http	Object or Function	For input arguments: a function or an object describing mapping from HTTP request to the argument value. See HTTP mapping of input arguments below.
http.target	String	Map the callback argument value to the HTTP response object. The following values are supported. • `status` sets the `res.statusCode` to the provided value • `header` sets the `http.header` or `arg` named header to the value
required	Boolean	True if argument is required; false otherwise.
root	Boolean	For callback arguments: set this property to `true` if your function has a single callback argument to use as the root object returned to remote caller. Otherwise the root object returned is a map (argument-name to argument-value).
type	String	Argument datatype; must be a Loopback type.
default	String	Default value that will be used to populate loopback-explorer input fields and swagger documentation. **Note**: This value will not be passed into remote methods function if argument is not present.

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

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

Multiple arguments, specified as an array:

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

HTTP mapping of input arguments

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

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

Using an object with a source property

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

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

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

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

Another example showing the Express HTTP request and response objects:

```
[
 {arg: 'req', type: 'object', 'http': {source: 'req'}},
 {arg: 'res', type: 'object', 'http': {source: 'res'}}
]
```

Using a custom mapping function

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

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

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

    // 2. Get 'a' and 'b' from query string or form data and return their sum.
    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 an HTTP request parameter `args` with JSON content, then it uses the value of `args['name']`.
2. Otherwise, it uses `req.param('name')`.

Returning data outside of a JSON field

Specifying a return argument with the *arg* property will automatically return a JSON object with your data stored within a field of the same name.

If you want to return data as the main response, for example an array, you can do so by setting the *root* property within the returns object and omitting *arg*.

```
returns: { type: 'array', root: true }
```

Setting a remote method route

By default, a remote method is exposed at:

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

Where

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

Following the above example, then by default the remote method is exposed at:

```
POST /api/people/greet
```

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

/common/models/model.js

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

This call changes the default route to

```
GET /api/people/sayhi
```

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

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

Adding ACLs to remote methods

To constrain access to custom remote methods, use the ACL generator in the same way you control access to any model API. The access type for custom remote methods is Execute.

Basic use

For example, to deny invocation of the `greet` method used in the examples above:

shell

```
 slc loopback:acl
[?] Select the model to apply the ACL entry to: Person
[?] Select the ACL scope: A single method
[?] Enter the method name: greet
[?] Select the access type: Execute
[?] Select the role: All users
[?] Select the permission to apply: Explicitly deny access
```

The tool then creates the following access control specification:

/common/models/person.json

```
...
  "acls": [
    {
      "principalType": "ROLE",
      "principalId": "$everyone",  // apply the ACL to everyone
      "permission": "DENY",        // DENY attempts to invoke this method
      "property": "greet"          // applies the access control to the greet() method
    }
  ],
...
```

Advanced use

Another example, to allow invocation of the a remote method only for the `$owner` of that model object:

/common/models/YourModel.js

```
module.exports = function(YourModel) {
  ...
  YourModel.remoteMethod(
    'someRemoteMethod',
    {
      accepts: [
        {arg: 'id', type: 'number', required: true}
      ],
      // mixing ':id' into the rest url allows $owner to be determined and used for
access control
      http: {path: '/:id/some-remote-method', verb: 'get'}
    }
  );
};
```

Formatting remote method responses

You can reformat the response returned by all remote methods by adding a boot script that modifies the object returned by `app.remotes()` as follows:

/server/boot/hook.js

```
module.exports = function(app) {
  var remotes = app.remotes();
  // modify all returned values
  remotes.after('**', function (ctx, next) {
    ctx.result = {
      data: ctx.result
    };

    next();
  });
};
```

Remote hooks

Related articles:

Overview

LoopBack provides two kinds of hooks:

- **Remote hooks**, that execute before or after calling a remote method, either a custom remote method or a standard CRUD method inherited from PersistedModel.   See PersistedModel REST API for information on how the Node methods correspond to REST operations.
- **Operation hooks** that execute when models perform CRUD operations. NOTE: Operation hooks replace model hooks, which are now deprecated.

A *remote hook* enables you to execute a function before or after a remote method is called by a client:

- `beforeRemote()` runs before the remote method.
- `afterRemote()` runs after the remote method has finished successfully.
- `afterRemoteError()` runs after the remote method has finished with an error.

Use beforeRemote hooks to validate and sanitize inputs to a remote method. Because a beforeRemote hook runs *before* the remote method is executed, it can access the inputs to the remote method, but not the result.

Use afterRemote hooks to modify, log, or otherwise use the results of a remote method before sending it to a remote client. Because an afterRemote hook runs *after* the remote method is executed, it can access the result of the remote method, but cannot modify the input arguments.

Signature

Both `beforeRemote()` and `afterRemote()` have the same signature; below syntax uses beforeRemote but afterRemote is the same.

For static remote methods, including custom remote methods:

```
modelName.beforeRemote( methodName, function( ctx, next) {
 ...
 next();
});
```

For instance methods:

```
modelName.beforeRemote( methodName, function( ctx, modelInstance, next) {
 ...
 next();
});
```

The hook `afterRemoteError()` has a slightly different signature: The handler function has only two arguments:

```
modelName.afterRemoteError( methodName, function( ctx, next) {
 ...
 next();
});
```

Where:

- *`modelName`* is the name of the model to which the remote hook is attached.
- *`methodName`* is the name of the method that triggers the remote hook. This may be a custom remote method or a standard CRUD method inherited from PersistedModel. It may include wildcards to match more than one method (see below).
- `ctx` is the context object.
- *`modelInstance`* is the affected model instance.

The syntax above includes a call to `next()` as a reminder that you must call `next()` at some point in the remote hook callback function. It doesn't necessarily have to come at the end of the function, but must be called at some point before the function completes.

Wildcards

You can use the following wildcard characters in *`methodName`*:

- Asterisk `'*'` to match any character, up to the first occurance of the delimiter character `'.'` (period).
- Double-asterisk to match any character, including the delimiter character `'.'` (period).

For example, use `'*.*'` to match any static method; use `'prototype.*'` to match any instance method.

Examples

The following example defines beforeRemote and afterRemote hooks for the `revEngine()` remote method:

common/models/car.js

```
module.exports = function(Car) {
  // remote method
  Car.revEngine = function(sound, cb) {
    cb(null, sound + ' ' + sound + ' ' + sound);
  };
  Car.remoteMethod(
    'revEngine',
    {
      accepts: [{arg: 'sound', type: 'string'}],
      returns: {arg: 'engineSound', type: 'string'},
      http: {path:'/rev-engine', verb: 'post'}
    }
  );
  // remote method before hook
  Car.beforeRemote('revEngine', function(context, unused, next) {
    console.log('Putting in the car key, starting the engine.');
    next();
  });
  // remote method after hook
  Car.afterRemote('revEngine', function(context, remoteMethodOutput, next) {
    console.log('Turning off the engine, removing the key.');
    next();
  });
...
}
```

The following example uses wildcards in the remote method name. This remote hook is called whenever any remote method whose name ends with "save" is executed:

common/models/customer.js

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

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

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

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

common/models/customer.js

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

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

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

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

  next();
});
```

A safer means of effectively white-listing the fields to be returned by copying the values into new objects:

common/models/account.js

```
var WHITE_LIST_FIELDS = ['account_id', 'account_name'];

Account.afterRemote('**', function(ctx, modelInstance, next) {
  if (ctx.result) {
    if (Array.isArray(modelInstance)) {
      var answer = [];
      ctx.result.forEach(function (result) {
        var replacement ={};
        WHITE_LIST_FIELDS.forEach(function(field) {
          replacement[field] = result[field];
        });
        answer.push(replacement);
      });
    } else {
      var answer ={};
      WHITE_LIST_FIELDS.forEach(function(field) {
        answer[field] = ctx.result[field];
      });
    }
    ctx.result = answer;
  }
  next();
});
```

Examples of afterRemoteError

Perform an additional action when the instance method `speak()` fails:

common/models/dog.js

```
 Dog.afterRemoteError('prototype.speak', function(ctx, next) {
  console.log('Cannot speak!', ctx.error);
  next();
});
```

Attach extra metadata to error objects:

common/models/dog.js

```
Dog.afterRemoteError('**', function(ctx, next) {
  ie (!ctx.error.details) ctx.result.details = {};
  ctx.error.details.info = 'intercepted by a hook';
  next();
})
```

Report a different error back to the caller:

common/models/dog.js

```
Dog.afterRemoteError('prototype.speak', function(ctx, next) {
  console.error(ctx.error);
  next(new Error('See server console log for details.'));
});
```

Context object

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

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

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

The context object passed to `afterRemoteError()` hooks has an additional property `ctx.error` set to the Error reported by the remote method.

Other properties:

- `ctx.args` - Object containing the HTTP request argument definitions. Uses the arg definition to find the value from the request. These are the input values to the remote method.
- `ctx.result` - An object keyed by the argument names.
  Exception: If the root property is true, then it's the value of the argument that has root set to true.

ctx.req.accessToken

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

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

ctx.result

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

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

If a remote method explicitly specifies the returned value, only then would `ctx.result` be set. So your remote method must do something like:

```
MyModel.remoteMethod(
    'doSomething',
    {
      ...
      returns: {arg: 'redirectUrl', type: 'string'}
    }
  );
```

Operation hooks

- Overview
  - Operation hook context object
  - Checking for support of ctx.isNewInstance
  - Accessing the affected instance
- Hooks
  - access
  - before save
  - after save

Related articles:

Overview

Operation hooks are not tied to a particular method, but rather are triggered from all methods that execute a particular high-level create, read, update, or delete (CRUD) "operation".   These are all methods of PersistedModel that application models inherit. Using operation hooks enables you to intercept CRUD-related actions independent of the specific method that invokes them (for example, `create`, `save`, or `updateOrCreate`).

> In general, use operation hooks instead of deprecated model hooks to do something when a model performs a specific operation.

The API is simple: the method `Model.observe(`*`name`*`, `*`observer`*`)`, where *`name`* is the string name of the operation hook, for example "before save", and *`observer`* is `function observer(context, callback)`. Child models inherit observers, and you can register multiple observers for a hook.

The following table summarizes the operation hooks invoked by PersistedModel CRUD methods.

Method **Operation hook**	**find findOne findById**	**exists**	**count**	**create**	**upsert**	**findOrCreate**	**deleteAll deleteById**	**updateAll**	**prototype .save**	**prototype .delete**	**protot .updat**
access	X	X	X		X	X	X	X			
before save				X	X	X		X	X		
after save				X	X	X		X	X		
before delete							X			X	
after delete							X			X	
loaded	X	X	X	X	X	X			X		
persist				X	X	X		X	X		

Operation hook context object

The `context` object is specific to operation hooks and does not have any relation to the context object passed to remote hooks registered via `Model.beforeRemote` and `Model.afterRemote`. See Remote hooks for more information. Note that the context object is not related to the "current context" provided by `loopback.getCurrentContext()` either.

Properties common for all hooks and operations

Target model

The property `context.Model` is set to the constructor of the model that is the target of the operation. For example `Product.find()` sets `context.Model = Product`.

Operation options

The context object has an `options` property that enables hooks to access any options provided by the caller of the specific model method (operation). For example:

```
var FILTERED_PROPERTIES = ['immutable', 'birthday'];
MyModel.observe('before save', function filterProperties(ctx, next) {
  if (ctx.options && ctx.options.skipPropertyFilter) return next();
  if (ctx.instance) {
    FILTERED_PROPERTIES.forEach(function(p) { ctx.instance.unsetAttribute(p); });
  } else {
    FILTERED_PROPERTIES.forEach(function(p) { delete ctx.data[p]; });
  }
  next();
});

// immutable is not updated
MyModel.updateOrCreate({ id: 1, immutable: 'new value' }, cb);

// immutable is changed
MyModel.updateOrCreate({ id: 2, immutable: 'new value' }, { skipPropertyFilter: true
}, cb);
```

Shared hookState property

The `ctx.hookState` property is preserved across all hooks invoked for a single operation.

For example, both "access", "before save" and "after save" invoked for `Model.create()` have the same object passed in `ctx.hookState`.

This way the hooks can pass state date between "before" and "after" hook.

Hook and operation specific properties

Besides the common properties listed above, each hook provides additional properties identifying the model instance(s) affected by the operation and the changes applied. The general rule is that the context provides either an `instance` property or a pair of `data` and `where` properties.

instance

This property is provided when the operation affects a single instance *and* performs a full update/create/delete of all model properties, for example `PersistedModel.create()`.

where + data

When the operation affects multiple instance (e.g. `PersistedModel.updateAll()`) *or* performs a partial update of a subset of model properties (e.g. `PersistedModel.prototype.updateAttributes()`), the context provides a where filter used to find the affected records and plain `data` object containing the changes to be made.

isNewInstance

Some operations provide a flag to distinguish between a CREATE operation and an UPDATE operation. See the documentation of individual hooks for more information.

> Only certain connectors support `ctx.isNewInstance`. With other connectors it is undefined. See Checking for support of ctx.isNewInstance.

currentInstance

This property is provided by hooks that perform a partial change of a single instance. It contains the affected model instance, you should treat the value as read only (immutable).

Checking for support of ctx.isNewInstance

The initial implementation of `ctx.isNewInstance` included only support for memory, MongoDB, and MySQL connectors. You can check whether your connector supports this feature by testing the value returned in "after save" hook.

For example:

```
MyModel.observe('after save', function(ctx, next) {
  console.log('supports isNewInstance?', ctx.isNewInstance !== undefined);
  next();
});
// It's important to provide a value for the id property
// Include also values for any required properties
MyModel.updateOrCreate({ id: 123 }, console.log);
```

Please report a GitHub issue in the connector project if the feature is not supported.

Accessing the affected instance

Operations affecting a single instance only (i.e. all CRUD operations except `PersistedModel.deleteAll` and `PersistedModel.updateAll`) usually provide the affected instance in the context object. However, depending on the operation, this instance is provided either as modifiable `ctx.instance` or as read-only `ctx.currentInstance`:

	before save	persist	after save	before delete	after delete
`create`	`ctx.instance`	`ctx.currentInstance`	`ctx.instance`	---	---
`findOrCreate`	`ctx.instance`	`ctx.currentInstance`	`ctx.instance`	---	---
`updateOrCreate`	*n/a**	`ctx.currentInstance`	`ctx.instance`	---	---
`updateAll`	*n/a*	*n/a*	*n/a*	---	---
`prototype.save`	`ctx.instance`	`ctx.currentInstance`	`ctx.instance`	---	---
`prototype.updateAttributes`	`ctx.currentInstance`	`ctx.currentInstance`	`ctx.instance`	---	---
`prototype.delete`	---	---	---	`ctx.instance`	`ctx.instanc`
`deleteAll`	---	---	---	*n/a*	*n/a*

* The operation `updateOrCreate` does not provide any instance in the "before save" hook. Because we cannot tell in advance whether the operation will result in UPDATE or CREATE, we cannot tell whether there is any existing "currentInstance" affected by the operation.

See the following sections for more details.

Hooks

LoopBack provides the following operation hooks:

- access
- before save
- after save
- before delete
- after delete
- loaded
- persist

The following table lists hooks that PersistedModel methods invoke.

Method name	Hooks invoked
all find findOne findById exists count	access, loaded
create	before save, after save, loaded, persist
upsert (aka updateOrCreate)	access, before save, after save, loaded, persist
findOrCreate	access, before save*, after save*, loaded, persist

deleteAll (aka destroyAll) deleteById (aka destroyById)	access, before delete, after delete
updateAll	access, before save, after save, persist
prototype.save	before save, after save, persist, loaded
prototype.delete	before delete, after delete
prototype.updateAttributes	before save, after save, loaded, persist

* When `findOrCreate` finds an existing model, the save hooks are not triggered. However, connectors providing atomic implementation may trigger `before save` hook even when the model is not created, since they cannot determine in advance whether the model will be created or not.

access

The `access` hook is triggered whenever a database is queried for models, that is when *any* CRUD method of PersistedModel is called. Observers may modify the query, for example by adding extra restrictions.

> Prototype methods don't trigger the `access` hook because the hook was already triggered by the method that loaded the model instance from the database.
>
> For example, when you call a prototype method via the REST API, two model calls are made: static `findById()` (that triggers the "access" hook) and then the prototype method as requested.

Context properties

- `Model` - the constructor of the model that will be queried
- `query` - the query containing fields `where`, `include`, `order`, etc.

Examples:

```
MyModel.observe('access', function logQuery(ctx, next) {
  console.log('Accessing %s matching %s', ctx.Model.modelName, ctx.query.where);
  next();
});

MyModel.observe('access', function limitToTenant(ctx, next) {
  ctx.query.where.tenantId = loopback.getCurrentContext().tenantId;
  next();
});
```

before save

The `before save` hook is triggered before a model instance is modified (created, updated), specifically when the following methods of PersistedModel are called:

- create()
- upsert()
- findOrCreate()*
- updateAll()
- prototype.save()
- prototype.updateAttributes()

* When `findOrCreate` finds an existing model, the save hooks are not triggered. However, connectors providing atomic implementation may trigger `before save` hook even when the model is not created, since they cannot determine in advance whether the model will be created or not.

The hook is triggered *before* model validation functions are called.

> Since the `before save` hook is triggered before validators are called, you can use it to ensure that empty or missing values are filled with default values.

Depending on which method triggered this hook, the context will have one of the following sets of properties:

- Full save of a single model
  - `Model` - the constructor of the model that will be saved
  - `instance` - the model instance to be saved. The value is an instance of `Model` class.
- Partial update of possibly multiple models
  - `Model` - the constructor of the model that will be saved
  - `where` - the where filter describing which instances will be affected
  - `data` - the (partial) data to apply during the update
  - `currentInstance` - the instance being affected, see Triggering with prototype.updateAttributes below.

ctx.isNewInstance

The before save hook provides the `ctx.isNewInstance` property when `ctx.instance` is set, with the following values:

- True for all CREATE operations
- False for all UPDATE operations
- Undefined for `updateOrCreate`, `prototype.save`, `prototype.updateAttributes`, and `updateAll` operations.

Manipulating model data in "before save" hook

As explained above, the context provides either an `instance` property or a pair of `data` and `where` properties. Exposing a full model instance in `ctx.instance` allows hooks to call custom model instance methods (for example , the hook can call `order.recalculateShippingAndTaxes()`whenever order data like address was changed). That's why LoopBack CRUD operation provide the instance in as many cases as possible.

There are two notable exception when it is not feasible to provide the instance object:

1. `PersistedModel.updateAll` updates multiple instances matching the provided query. LoopBack does not even load their data from the database, it's up to the database to find these instances and apply necessary changes.
2. `PersistedModel.updateAttributes` performs a partial update, only a subset of model properties is modified. While LoopBack has a model instance available, it also needs to know which of model properties should be changed in the database. Passing the operation payload in `ctx.data` - a plain object containing only those properties which should be modified - makes it easy for hook implementations to add/remove the properties to modify. You can still access the model instance to be modified via `ctx.currentInstance` as long as you treat it as immutable (read-only).

Examples

```
MyModel.observe('before save', function updateTimestamp(ctx, next) {
  if (ctx.instance) {
    ctx.instance.updated = new Date();
  } else {
    ctx.data.updated = new Date();
  }
  next();
});

MyModel.observe('before save', function computePercentage(ctx, next) {
  if (ctx.instance) {
    ctx.instance.percentage = 100 * ctx.instance.part / ctx.instance.total;
  } else if (ctx.data.part && ctx.data.total) {
    ctx.data.percentage = 100 * ctx.data.part / ctx.data.total;
  } else if (ctx.data.part || ctx.data.total) {
    // either report an error or fetch the missing properties from DB
  }
  next();
});
```

Removing unneeded properties

To remove (unset) a property in a model instance, it is not enough the set its value to undefined and/or delete the property. One has to call `unsetAttribute(name)` instead. However, don't forget to handle the case where the context has a data property instead! Since the data object is a plain object, you can remove properties the usual way via delete operator.

Example:

```
MyModel.observe('before save', function removeUnwantedField(ctx, next) {
  if (ctx.instance) {
    ctx.instance.unsetAttribute('unwantedField');
  } else {
    delete ctx.data.unwantedField;
  }
  next();
});
```

This completely removes the field and prevents inserting spurious data into the database.

after save

The `after save` hook is called after a model change was successfully persisted to the datasource, specifically when the following methods of PersistedModel are called:

- create()
- upsert()
- findOrCreate()*
- updateAll()
- prototype.save()
- prototype.updateAttributes()

* When `findOrCreate` finds an existing model, the save hooks are not triggered. However, connectors providing atomic implementation may trigger `before save` hook even when the model is not created, since they cannot determine in advance whether the model will be created or not.

Depending on which method triggered this hook, the context will have one of the following sets of properties:

- A single model was updated:
  - `Model` - the constructor of the model that will be saved.
  - `instance` - the model instance that was saved. The value is an instance of `Model` class and contains updated values computed by datastore (for example, auto-generated ID).

  > The after save hook returns the changes made to `ctx.instance` to the caller (REST client), but does not persist them to the database!

- Partial update of more model instances via `Model.updateAll`:
  - `Model` - the constructor of the model that will be saved.
  - `where` - the where filter describing which instances were queried. See caveat below.
  - `data`- the (partial) data applied during the update.

  > You cannot reliably use the "where" query in an after save hook to find which models were affected. Consider the following call:
  >
  > ```
  > MyModel.updateAll({ color: 'yellow' }, { color: 'red' }, cb);
  > ```
  >
  > At the time the "after save" hook is run, no records will match the query `{ color: 'yellow' }`.

The `after save` hook provides the `ctx.isNewInstance` property whenever `ctx.instance` is set, with the following values:

- True after all CREATE operations.
- False after all UPDATE operations.
- The operations `updateOrCreate`, `prototype.save`, and `prototype.updateAttributes` require connectors to report whether a new instance was created or an existing instance was updated. When the connector provides this information, `ctx.isNewInstance` is True or False. When the connector does not support this feature yet (see below), the value is undefined.

> Only certain connectors support `ctx.isNewInstace`. With other connectors it is undefined. See Checking for support of ctx.isNewInstance.

Examples:

```
MyModel.observe('after save', function(ctx, next) {
  if (ctx.instance) {
    console.log('Saved %s#%s', ctx.Model.modelName, ctx.instance.id);
  } else {
    console.log('Updated %s matching %j',
      ctx.Model.pluralModelName,
      ctx.where);
  }
  next();
});
```

before delete

The `before delete` hook is triggered before a model is removed from a datasource, specifically when the following methods of PersistedModel are called:

- destroyAll() (same as `deleteAll()`)
- destroyById() (same as `deleteById()`)
- prototype.destroy() (same as `prototype.delete()`)

> The `before delete` operation hook does not receive a list of deleted model instance IDs, because backend data stores such as relational or NoSQL databases don't provide this information. However, *when deleting a single model instance* the hook receives `ctx.instance` that contains the instance being deleted.

Context properties

- `Model` - the constructor of the model that will be queried
- `where` - the where filter describing which instances will be deleted.

Example:

```
MyModel.observe('before delete', function(ctx, next) {
  console.log('Going to delete %s matching %j',
    ctx.Model.pluralModelName,
    ctx.where);
  next();
});
```

To reject the deletion of a model based on some condition, call `next()` with an error to abort the delete operation. For example:

```
if (subscriptions.length > 0) {
  //Stop the deletion of this Client
  var err = new Error("Client has an active subscription, cannot delete");
  err.statusCode = 400;
  console.log(err.toString());
  next(err);
} else {
  next();
}
```

after delete

> The `after delete` operation hooks do not receive a list of deleted model instance IDs, because backend data stores such as relational or NoSQL databases don't provide this information. However, *when deleting a single model instance* the hook receives `ctx.instance` that contains the instance being deleted.

The `after delete` hook is triggered after some models are removed from the datasource, specifically when the following methods of PersistedModel are called:

- destroyAll() (same as `deleteAll()`)
- destroyById() (same as `deleteById()`)
- prototype.destroy() (same as `prototype.delete()`)

Context properties

- `Model` - the constructor of the model that will be queried
- `where` - the where filter describing which instances were deleted.

Example:

```
MyModel.observe('after delete', function(ctx, next) {
  console.log('Deleted %s matching %j',
    ctx.Model.pluralModelName,
    ctx.where);
  next();
});
```

loaded

This hook is triggered by the following methods of PersistedModel:

- find()
- findOne()
- findById()
- exists()
- count()
- create()
- upsert() (same as `updateOrCreate()`)
- findOrCreate()*
- prototype.save()
- prototype.updateAttributes()

> By default, `create` and `updateAttributes` do not apply database updates to the model instance returned to the callback, therefore any changes made by "loaded" hooks are discarded. To change this behavior, set a per-model option `updateOnLoad: true`.

LoopBack invokes this hook after the connector fetches data, but before creating a model instance from that data. This enables hooks to decrypt data (for example). NOTE: This hook is called with the raw database data, not a full model instance.

Context properties

- `data` - the data returned by the connector (loaded from the database)

persist

This hook is triggered by operations that persist data to the datasource, specifically, the following methods of PersistedModel:

- create()
- upsert() (same as `updateOrCreate()`)
- findOrCreate()*
- prototype.save()
- prototype.updateAttributes()
- updateAll()

Don't confuse this hook with the "before save" hook:

- **before save** – Use this hook to observe (and operate on) model instances that are about to be saved (for example, when the country code is set and the country name not, fill in the country name).
- **persist** – Use this hook to observe (and operate on) data just before it is going to be persisted into a data source (for example, encrypt the values in the database).

During `create` the updates applied through `persist` hook are reflected into the database, but the same updates are NOT reflected in the `instance` object obtained in callback of `create`.

Secondly, for connectors implementing atomic `findOrCreate`, a new instance of the object is created every time, even if an existing record is later found in the database. So:

- Both `ctx.data.id` and `ctx.currentInstance.id` are set to new ID.
- `ctx.isNewInstance` is `true`

Context properties

- `data` - the data that will be sent to the connector (saved to the database)
- `currentInstance` - the affected model instance
- `isNewInstance` - see below.

For this hook, `ctx.isNewInstance` is:

- True for all CREATE operations
- False for all UPDATE operations
- Undefined for updateOrCreate, prototype.save, prototype.updateAttributes, and updateAll operations.

afterInitialize hook

`afterInitialize` is not strictly an operation hook. It is actually the only model hook that is not deprecated.

It is a synchronous method and does not take a callback function: You do not need to call `next()` after performing your logic in the hook.

This hook is called after a model is initialized.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterInitialize = function() {
  //your logic goes here
};
...
```

Most operations require initializing a model before actually performing an action, but there are a few cases where the initialize event is not triggered, such as HTTP requests to the `exists`, `count`, or bulk update REST endpoints.

Migration guide

The following table shows which new hook to use for each of the old model hooks:

Model hook	Operation hook to use instead
beforeValidate	before save
afterValidate	persist
beforeCreate	before save
afterCreate	after save
beforeSave	before save
afterSave	after save
beforeUpdate	before save
afterUpdate	after save
beforeDestroy	before delete
afterDestroy	after delete

Model hooks

Model hooks are deprecated, except for afterInitialize.

Please use operation hooks instead.

Related articles:

Overview

Use model hooks to add custom logic to models that extend PersistedModel. Each hook is called before or after a specific event in the model's lifecycle.

You can define the following model hooks, listed in the order that the events occur in a model lifecycle:

- `afterInitialize` - triggers after a model has been initialized.
- `beforeValidate` - triggers before validation is performed on a model.
- `afterValidate` - triggers after validation is performed on a model.
- `beforeSave` - triggers before a model is saved to a data source.
- `afterSave` - triggers after a model is saved to a data source.
- `beforeCreate` - triggers before a model is created.
- `afterCreate` - triggers after a model is created.
- `beforeUpdate` - triggers before a model is updated.
- `afterUpdate` - triggers after a model is updated.
- `beforeDestroy` - triggers before a model is destroyed.
- `afterDestroy` - triggers after a model is destroyed.

Best practice is to register model hooks in `/common/models/your-model.js`. This ensures hooks are registered during application initialization. If you need to register a hook at runtime, get a reference to the `app` object and register it right then and there.

afterInitialize

This hook is called after a model is initialized.

This model hook is *not* deprecated and is still useful. It is a synchronous method: there is no callback function.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterInitialize = function() {
  //your logic goes here
};
...
```

Most operations require initializing a model before actually performing an action, but there are a few cases where the initialize event is not triggered, such as HTTP requests to the `exists`, `count`, or bulk update REST endpoints.

This is the only hook that does not require you to explicitly call `next()` after performing your logic.

beforeValidate

This hook is called before validatation is performed on a model.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.beforeValidate = function(next, modelInstance) {
  //your logic goes here - don't use modelInstance
  next();
};
...
```

In the beforeValidate hook, use `this` instead of `modelInstance` to get a reference to the model being validated. In this hook, `modelInstance` is not valid.

You must call `next()` to let LoopBack now you're ready to go on after the hook's logic has completed.

If you don't call `next()`, the application will appear to "hang".

afterValidate

This hook is called after validation is performed on a model.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterValidate(next) {
  //your logic goes here
  next();
};
...
```

You must call `next()` to let LoopBack now you're ready to go on after the hook's logic has completed.

beforeCreate

This hook is called just before a model is created.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.beforeCreate = function(next, modelInstance) {
  //your logic goes here
  next();
};
...
```

LoopBack provides `modelInstance` as a reference to the model being created.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

afterCreate

This hook is called after a model is created.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterCreate = function(next) {
  //your logic goes here
  this.name = 'New coffee shop name; //you can access the created model via `this`
  next();
};
...
```

Access the model being created with `this`.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

beforeSave

This hook is called just before a model instance is saved.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.beforeSave = function(next, modelInstance) {
  //your logic goes here
  next();
};
...
```

LoopBack provides `modelInstance` as a reference to the model being saved.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

afterSave

This hook is called after a model is saved.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterSave = function(next) {
  //your logic goes here
  this.name = 'New coffee shop name; //you can access the created model via `this`
  next();
};
...
```

Access the model being saved with `this`.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

beforeUpdate

This hook is called just before a model is updated.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.beforeUpdate = function(next, modelInstance) {
  //your logic goes here
  next();
};
...
```

LoopBack provides `modelInstance` as a reference to the model being updated.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

afterUpdate

This hook is called after a model is updated.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterUpdate = function(next) {
  //your logic goes here
  this.name = 'New coffee shop name; //you can access the created model via `this`
  next();
};
...
```

LoopBack provides `modelInstance` as a reference to the model being saved.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

beforeDestroy

This hook is called just before a model is destroyed.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.beforeDestroy = function(next, modelInstance) {
  //your logic goes here
  next();
};
...
```

LoopBack provides `modelInstance` as a reference to the model being saved.

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

afterDestroy

This hook is called after a model is destroyed.

Example

/common/models/coffee-shop.js

```
...
CoffeeShop.afterDestroy = function(next) {
  //your logic goes here
  next();
};
...
```

You must call `next()` to continue execution after the hook completes its logic. If you don't the application will appear to hang.

Connector hooks

Related articles:

Overview

Connectors are responsible for interacting with the backend systems on behalf of model methods. The connector hooks enable applications to intercept the connector execution.

Hooks

before execute

The 'before execute' hook is invoked before a connector sends a request to the backend.

```
var connector = MyModel.getDataSource().connector;
connector.observe('before execute', function(ctx, next) {
  // ...
  next();
});
```

To terminate the invocation, call `ctx.end(err, result)`, for example:

```
var connector = MyModel.getDataSource().connector;
connector.observe('before execute', function(ctx, next) {
  // ...
  ctx.end(null, cachedResponse);
});
```

after execute

The 'after execute' hook is invoked after the connector receives a response from the backend.

```
connector.observe('after execute', function(ctx, next) {
  // ...
  next();
});
```

Context

The context object contains information for the hooks to act on. It varies based on the type of connectors.

SQL based connectors (MySQL, PostgreSQL, SQL Server, Oracle)

`before: {req: {sql: 'SELECT ...', params: [1, 2]}, end: ...}`

`after: {req: {sql: 'SELECT ...', params: [1, 2]}, res: ..., end: ...}`

MongoDB connector

`before: {req: {command: ..., params: ...}, end: ...}`

`after: {req: {...}, res: {...}, end: ...}`

req.command is the command for the mongodb collection.

req.params is the parameters passing to the mongodb driver.

res is the object received from the mongodb driver.

REST connector

`before: {req: {...}, end: ...}`

`after: {req: {...}, res: {...}, end: ...}`

req is the object passing to request module.

res is the object received from request module.

SOAP connector

`before: {req: {...}, end: ...}`

`after: {req: {...}, res: {...}, end: ...}`

req is the object passing to request module.

res is the object received from request module.

Tutorial: Adding application logic

This article is reproduced from loopback-example-app-logic

loopback-example-app-logic

```
$ git clone https://github.com/strongloop/loopback-example-app-logic.git
$ cd loopback-example-app-logic
$ npm install
$ node .
# then in a different tab, run ./bin/remote-method-request or
./bin/datetime-request
```

In this example, we demonstrate remote methods, remote hooks, model operation hooks, boot scripts, middleware, and email-connector as solutions for integrating user-defined logic into a LoopBack application.

Prerequisites

Tutorials

- Getting started with LoopBack
- Tutorial series - step 1
- Tutorial series - step 2

Knowledge

- LoopBack models
- LoopBack adding application logic

Procedure

Create the application

Application information

- Name: `loopback-example-app-logic`
- Directory to contain the project: `loopback-example-app-logic`

```
$ slc loopback loopback-example-app-logic
... # follow the prompts
$ cd loopback-example-app-logic
```

Add a model

Model information

- Name: `car`
- Datasource: `db (memory)`
- Base class: `PersistedModel`

- Expose via REST: `Yes`
- Custom plural form: *Leave blank*
- Properties
    - `make`
    - String
    - Not required
    - `model`
    - String
    - Not required

```
$ slc loopback:model car
... # follow the prompts
```

Define a remote method

Define a remote method in `car.js`.

> *The remote method takes a "sound" and repeats it three times.*

Test it by starting the server (using `node .`) and running `curl -XPOST localhost:3000/api/cars/rev-engine -H 'content-type:application/json' -d '{"sound":"vroom"}'`.

> *If you are using Windows, single quotes are treated as backticks in `cmd`. This means you will have to modify the `curl` command to use and escape double quotes instead: `curl -XPOST localhost:3000/api/cars/rev-engine -H "content-type:application/json" -d "{\"sound\":\"vroom\"}"`.*

You should see:

```
...
{"engineSound":"vroom vroom vroom"}
```

Define a remote method before hook

Define a remote method before hook in `car.js`.

> *The second parameter `unused` must be provided for legacy reasons. You may simply ignore it, but you must declare it to ensure `next` is the third parameter. This is a side effect of inheriting from the `jugglingdb` library.*
>
> *`context` contains the Express request and response objects (ie. `context.req` and `context.res`).*

This method is triggered right before `revEngine` is called and prints a message to the console.

Restart the server.

```
$ ./bin/remote-method-request
```

You should see:

```
...
Putting in the car key, starting the engine.
```

Define a remote method after hook

Define a remote method after hook in `car.js`.

This method is triggered after `revEngine` finishes execution and prints a message to the console.

Restart the server.

```
$ ./bin/remote-method-request
```

You should see:

```
...
Turning off the engine, removing the key.
```

Create a boot script

Create `print-models.js` in the `boot` directory.

> *The app argument is provided by LoopBack. You can use it to access the application context, which is required when you want to retrieve models, configs, and so on.*
>
> ***Asynchronous boot scripts***
>
> *To use asynchronous boot scripts, you have to modify `boot` to take callback. You will also need to provide an additional `callback` argument in your boot scripts.*

Restart the server.

In the server output, you should see:

```
...
Models:  [ 'User', 'AccessToken', 'ACL', 'RoleMapping', 'Role', 'car' ]
...
```

Define a model operation hook

Define a model operation hook in `car.js`.

Copy the `create-car.js` script to the `server/boot` directory.

```
$ cp examples/async-boot-script/create-car.js server/boot/
```

Restart the server.

You should see:

```
...
About to save a car instance: { make: 'honda', model: 'civic' }
A `car` instance has been created from a boot script: { make: 'honda',
model: 'civic', id: 1 }
...
```

This model operation hook is triggered **before** saving any `car` model instance.

> *Many other operation hooks are available, such as `access`, `before save`, `after save`, `before delete`, and `after delete`. See the model operation hooks documentation for more information.*

Add pre-processing middleware

Create the `middleware` directory to store middleware files.

```
$ mkdir server/middleware
```

Create the `tracker` middleware to respond with the request processing time.

Register the `tracker` middleware in `middleware.json`.

> *We register `tracker` in the `initial` phase because we want it configured before other middleware. See the official middleware phases documentation.*

Restart the server.

```
$ ./bin/remote-method-request
```

You should see:

```
...
The request processing time is 28.472051 ms.
```

> *Your time will be different.*

Add post-processing middleware

Create the `datetime` middleware, which responds with the current date and time when a request is made to `localhost:3000/datetime`.

Register the `tracker` middleware in `middleware.json`.

Create a shell script to test the middleware.

Restart the server.

```
$ ./bin/datetime-request
```

You should see:

```
...
{"started":"2015-01-14T22:54:35.708Z","uptime":3.494}
```

Your date and time will be different.

Add an email connector

How do you send email?

1. Configure an email datasource
2. Map the built-in `Email` model to the the email datasource
3. Send an email using the configured model

Notes

- This example contains a boot script that sends an email every time you start the application.
- Be sure to use **YOUR** email configurations in `datasources.json`
- You will need to configure `boot()` in `server.js` to take a callback for the application to start up properly because we use an asynchronous boot script for this example

---

- Next tutorial
- All tutorials

Defining boot scripts

- Overview
  - Using boot scripts
- Predefined boot scripts
- Using the boot script generator
- Synchronous and asynchronous boot scripts
  - Bootstrap function arguments
  - Asynchronous boot scripts
  - Synchronous boot scripts
- Boot script loading order

See also:

- Adding logic to models
- Defining middleware

Overview

The LoopBack bootstrapper, loopback-boot, performs application initialization (also called *bootstrapping*). When an application starts, the bootstrapper:

- Configures data sources.
- Defines custom models
- Configures models and attaches models to data-sources.
- Configures application settings
- Runs boot scripts in the `/server/boot` directory.

The loopback-boot module exports a `boot()` function that initializes an application. For example, from the standard scaffolded server.js script:

/server/server.js

```
var loopback = require('loopback');
var boot = require('loopback-boot');
var app = module.exports = loopback();
...
boot(app, __dirname);
...
```

See loopback-boot API docs for details.

If you create your application with `slc loopback`, the Application generator, then you don't need to do anything to bootstrap your application--the above code is automatically scaffolded for you!

Using boot scripts

Use *boot scripts* to perform custom initialization in addition to that performed by the LoopBack bootstrapper. When an application starts, LoopBack loads all the scripts in the `server/boot` directory. By default, LoopBack loads boot scripts in alphabetical order. You can customize the boot script load order using the options argument of `boot()`; see Boot script loading order for details.

Predefined boot scripts

The standard scaffolded LoopBack application created by the application generator contains the following standard boot scripts (in `/server/boot`) that perform basic initialization:

- `authentication.js` - Enables authentication for the application by calling `app.enableAuth()`.
- `root.js` - Binds `loopback.status()` middleware at the root endpoint ("/") to provide basic status information.

Prior to generator-loopback v. 1.12, the application generator created two additional boot scripts, but this functionality is now handled in middleware:

- `explorer.js` - Enables API Explorer.
- `rest-api.js` - Exposes the application's models over REST using `loopback.rest()` middleware.

Using the boot script generator

In addition to the predefined boot scripts, you can define custom boot scripts to perform your own logic when an application starts.

Use the Boot script generator, `slc loopback boot-script`, to quickly generate boot script templates. Depending on how you respond to the generator's prompts, it will generate a template for either a synchronous or asynchronous boot script:

Synchronous boot script template

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

};
```

Comments are omitted here for brevity.

Asynchronous boot script template

```
module.exports = function(app, cb) {

  process.nextTick(cb); // Remove if you pass `cb` to an async function yourself
};
```

Simply add to the function the code you want to execute at boot time.

Synchronous and asynchronous boot scripts

LoopBack supports both synchronous and asynchronous boot scripts. The type to use depends on the nature of the task. Use asynchronous boot scripts for tasks for which you don't want to block program execution, such as database requests or network operations.

Both types of boot script must export a function that contains the actions of the script. The signature of this function is similar for both types of boot scripts, but asynchronous boot script functions take an additional callback argument.

Bootstrap function arguments

module.exports = function(app, [*callback*]) { ... }

Name	Type	Required	Description
app	Object	Yes	The application context object. Provides a handle the the application, so (for example) you can get model objects: `var User = app.models.User;`
callback	Function	Only for asynchronous boot scripts	Call the callback function when your application logic is done.

Asynchronous boot scripts

An asynchronous boot script must export a function that takes two arguments:

1. The application object, `app`. This object enables you to access system-defined variables and configurations.
2. A callback function that enables you to time your response according to your application logic.

> You must call the callback function when the script is finished to pass control back to the application.

For example, this boot script prints "hello world" and triggers the callback function after three seconds (3000 millseconds).

/server/boot/script.js

```
module.exports = function(app, callback) {
  setTimeout(function() {
    console.log('Hello world');
    callback();
  }, 3000);
};
```

If you add this boot script to an application, it will display "Hello world" to the console when it starts.

Synchronous boot scripts

A synchronous boot script must export a function that takes one argument, the application object, `app`. This object enables you to access system-defined variables and configurations.

For example, this boot script retrieves the names of all models registered with the application and displays them to the console.

/server/boot/script.js

```
module.exports = function(app) {
  var modelNames = Object.keys(app.models);
  var models = [];
  modelNames.forEach(function(m) {
    var modelName = app.models[m].modelName;
    if (models.indexOf(modelName) === -1) {
      models.push(modelName);
    }
  });
  console.log('Models:', models);
};
```

If you add this boot script to an "empty" application, you will see this:

shell

```
...
Models: [ 'User', 'AccessToken', 'ACL', 'RoleMapping', 'Role' ]
...
```

Boot script loading order

The easiest way to specify the order of loading boot scripts is by scripts' file names, since LoopBack always executes boot scripts in alphabetical order. For example, you could name boot scripts `01-your-first-script.js`, `02-your-second-script.js`, and so forth. This ensures LoopBack loads scripts in the order you want, for example before default boot scripts in `/server/boot`.

You can also specify the loading order with options to the `boot()` function call in `/server/server.js`. Replace the default scaffolded function call:

/server/server.js

```
...
boot(app, __dirname);
...
```

With something like this:

```
...
bootOptions = { "appRootDir": __dirname,
                "bootScripts" : [ "/full/path/to/boot/script/first.js",
"//full/path/to/boot/script/second.js", ... ] };
boot(app, bootOptions);
...
```

Then the application will then execute scripts in the order specified in the `bootScripts` array. Specify the full directory path to each script.

> Using the technique shown above, the application will still run all the boot scripts in `/server/boot` in alphabetical order (unless you move or delete them) after your custom-ordered boot scripts specified in `bootScripts`.

If desired, you can also specify one or more directories in the `bootDirs` property, and the application will run scripts in that directory in alphabetical order after those specified in `bootScripts` but before those in the `/server/boot` directory.

Defining mixins

See also:

- loopback-example-mixins
- loopback-ds-timestamp-mixin

Overview

Use mixins to apply common logic to a set of models. For example, a timestamp mixin could inject "created" and "modified" properties to model definitions.

You can apply mixins to any model, including built-in models.

Built-in model mixins

Basic model

By default, the basic LoopBack Model object has properties and methods "mixed in" from:

- Inclusion object - Enables you to load relations of several objects and optimize numbers of requests.
- Validateable object - provides validation methods; see Validating model data.

When you define relations between models, the RelationMixin object object also gets mixed in to the model object.

Connected model

In addition to the methods of the Basic model object, the following are mixed in when a model is connected to a data source:

- RelationMixin class
- PersistedModel class

Create a mixin script

Mixin scripts are JavaScript files in one of the following folders, depending on the scope of the mixin:

- `common/mixins/`*`mixinName`*`.js`, for example `common/mixins/timeStamp.js`.
- `server/mixins/`*`mixinName`*`.js`, for example `server/mixins/timeStamp.js`.

If the mixin applies to both client and server models, put it in the `common/mixins` directory; if it applies only to server models, put it in the `server/mixins` directory.

Good to Know
The above locations are just recommendations. You are free to put mixin scripts in any project directory as long as you set the location with the `mixins` property in `model-config.js`.

You can use mixins to perform different common actions on models such as observing changes using operation hooks and adding model attributes.

For example:

common/mixins/timestamp.js

```
module.exports = function(Model, options) {
  // Model is the model class
  // options is an object containing the config properties from model definition
  Model.defineProperty('created', {type: Date, default: '$now'});
  Model.defineProperty('modified', {type: Date, default: '$now'});
}
```

Above `Timestamp` mixin adds two properties: `'created'` and `'modified'` of type: `date`, with default values set to the current date, to `Model`

`.json`.

Where as, in the example below, the mixin observes the change using `before save` operation hook and manipulates input (see complete example: loopback-example-mixins):

/server/mixins/squirrel.js

```
module.exports = function(Model, options) {
  'use strict';
  Model.observe('before save', function event(ctx, next) { //Observe any insert/update
event on Model
    if (ctx.instance) {
      ctx.instance.squirrel = true;
    } else {
      ctx.data.squirrel = true;
    }
    next();
  });
};
```

Reference mixins in model-config.js

The configuration file `server/model-config.json` specifies the list of directories to be searched for mixin scripts. By default, the application generator `slc loopback` sets them up as follows:

server/model-config.json

```
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  ...
}
```

Enable a model with mixins

To apply a mixin to a model, add "mixins" to the model definition JSON file. The value of mixins is an object keyed by normalized mixin names. The value for each mixin is passed into the script as the options argument and these options are implemented by mixins.

common/models/note.json

```
{
  "name": "note",
  "base": "PersistedModel",
  ...
  "mixins": {
     "Timestamp": {
       "myOption": 1,
       "anotherOpt": 2
     }
   },
  "properties": {
    ...
  },
  ...
}
```

In the example above, `common/mixins/timestamp.js` will be invoked with `(note, {"myOption": 1, "anotherOpt":2})`

Defining Mixins - Draft

See also:

- loopback-example-mixins
- loopback-ds-timestamp-mixin

Overview

Mixins are used to apply common logic to a set of models. For example, a timestamp mixin could inject "created" and "modified" properties to model definitions.

You can apply mixins to any model, including built-in models.

Built-in model mixins

Basic model

By default, the basic LoopBack Model object has properties and methods "mixed in" from:

- Inclusion object - Enables you to load relations of several objects and optimize numbers of requests.
- Validateable object - provides validation methods; see Validating model data.

When you define relations between models, the RelationMixin object object also gets mixed in to the model object.

Connected model

In addition to the methods of the Basic model object, the following are mixed in when a model is connected to a data source:

- RelationMixin class
- PersistedModel class

Create a mixin script

Mixin scripts are JavaScript files in one of the following folders, depending on the scope of the mixin:

- `common/mixins/`*`mixinName`*`.js`, for example `common/mixins/timeStamp.js`.
- `server/mixins/`*`mixinName`*`.js`, for example `server/mixins/timeStamp.js`.

If the mixin applies to both client and server models, put it in the `common/mixins` directory; if it applies only to server models, put it in the `server/mixins` directory.

Good to Know:
Please note that above is just a recommendation on where to put `mixin` scripts, ideally, in a project directory structure. You are free to

put these scripts in any project directory and they should be referenced in `model-config.js`

Mixins can be used to perform different common actions on models like observing changes using Operation hooks and adding model attributes

For example:

common/mixins/timestamp.js

```
module.exports = function(Model, options) {
  // Model is the model class
  // options is an object containing the config properties from model definition
  Model.defineProperty('created', {type: Date, default: '$now'});
  Model.defineProperty('modified', {type: Date, default: '$now'});
}
```

Above `Timestamp` mixin adds two properties: `'created'` and `'modified'` of type: `date`, with default values set to the current date, to `Model.json`.

Where as, in the example below, the mixin observes the change using `before save` operation hook and manipulates input (see complete example: loopback-example-mixins):

/server/mixins/squirrel.js

```
module.exports = function(Model, options) {
  'use strict';
  Model.observe('before save', function event(ctx, next) { //Observe any insert/update event on Model
    if (ctx.instance) {
      ctx.instance.squirrel = true;
    } else {
      ctx.data.squirrel = true;
    }
    next();
  });
};
```

Reference mixins in model-config.js

The configuration file `server/model-config.json` specifies the list of directories to be searched for mixin scripts. By default, the Application generator `slc loopback` sets them up as follows:

server/model-config.json

```
{
  "_meta": {
    "sources": [
      "loopback/common/models",
      "loopback/server/models",
      "../common/models",
      "./models"
    ],
    "mixins": [
      "loopback/common/mixins",
      "loopback/server/mixins",
      "../common/mixins",
      "./mixins"
    ]
  },
  ...
}
```

Enable a model with mixins

To apply a mixin to a model, add "mixins" to the model definition JSON file. The value of mixins is an object keyed by normalized mixin names. The value for each mixin is passed into the script as the options argument and these options are implemented by mixins.

common/models/note.json

```
{
  "name": "note",
  "base": "PersistedModel",
  ...
  "mixins": {
     "Timestamp": {
       "myOption": 1,
       "anotherOpt": 2
     }
   },
  "properties": {
    ...
  },
  ...
}
```

In the example above, `common/mixins/timestamp.js` will be invoked with `(note, {"myOption": 1, "anotherOpt":2})`

Defining middleware

See also:

- middleware.json
- Upgrading applications to use phases
- Example app loopback-faq-user-management

Overview

Middleware refers to functions executed when HTTP requests are made to REST endpoints. Since LoopBack is based on Express, LoopBack middleware is the same as Express middleware. However, LoopBack adds the concept of *middleware phases*, to clearly define the order in which middleware is called. Using phases helps to avoid ordering issues that can occur with standard Express middleware.

LoopBack supports the following types of middleware:

- **Pre-processing middleware** for custom application logic. See example of static middleware.
- **Dynamic request handling middleware** to serve dynamically-generated responses, for example HTML pages rendered from templates and JSON responses to REST API requests. See example of pre-processing middleware.
- **Static middleware** to serve static client-side assets. See example of static middleware.
- **Error-handling middleware** to deal with request errors. See example of error-handling middleware.

How to add middleware

To add middleware to your application:

1. **Specify the middleware function**:
    a. If using an existing function or package, add the code to your application or install the package.
    b. If you are creating a new middleware function, write it. See Defining a new middleware handler function.
2. **Register the middleware**:
    - Edit `server/middleware.json`. This is the recommended way to register middleware. See Registering middleware in middleware.json.
    - Alternatively, register the middleware in application code. See Registering middleware in JavaScript.

Middleware phases

LoopBack defines a number of *phases*, corresponding to different aspects of application execution. When you register middleware, you can specify the phase in which the application will call it. See Registering middleware in middleware.json and Using the LoopBack API. If you register middleware (or routes) with the Express API, then it is executed at the beginning of the `routes` phase.

The predefined phases are:

1. **`initial`** - The first point at which middleware can run.
2. **`session`** - Prepare the session object.
3. **`auth`** - Handle authentication and authorization.
4. **`parse`** - Parse the request body.
5. **`routes`** - HTTP routes implementing your application logic. Middleware registered via the Express API `app.use`, `app.route`, `app.get` (and other HTTP verbs) runs at the beginning of this phase. Use this phase also for sub-apps like `loopback/server/middleware/rest` or `loopback-explorer`.
6. **`files`** - Serve static assets (requests are hitting the file system here).
7. **`final`** - Deal with errors and requests for unknown URLs.

Each phase has "before" and "after" subphases in addition to the main phase, encoded following the phase name, separated by a colon. For example, for the "initial" phase, middleware executes in this order:

1. `initial:before`
2. `initial`
3. `initial:after`

Middleware within a single subphase executes in the order in which it is registered. However, you should not rely on such order. Always explicitly order the middleware using appropriate phases when order matters.

Specifying a middleware function

Using built-in middleware

LoopBack provides convenience middleware for commonly-used Express/Connect middleware, as described in the following table.

When you use this middleware, you don't have to write any code or install any packages; you just specify in which phase you want it to be called; see Registering middleware in middleware.json.

Middleware ID	Code accessor	External package
`loopback#favicon`	`loopback.favicon()`	serve-favicon
`loopback#rest`	`loopback.rest()`	N/A
`loopback#static`	`loopback.static()`	serve-static
`loopback#status`	`loopback.status()`	N/A
`loopback#token`	`loopback.token()`	N/A
`loopback#urlNotFound`	`loopback.urlNotFound()`	N/A

To simplify migration from LoopBack 1.x and Express 3.x, LoopBack provides middleware that was built-in to in Express 3.x, as shown in the following table. Best practice is to load this middleware directly via `require()` and not rely on LoopBack's compatibility layer.

You can use any middleware compatible with Express; see Express documentation for a partial list. Simply install it:

```
$ npm install --save <module-name>
```

Then simply register it so that it is called as needed; see Registering middleware in middleware.json and Registering middleware in JavaScript.

Defining a new middleware handler function

If no existing middleware does what you need, you can easily write your own middleware handler function. To register the middleware function in `middleware.json`, you need to create a constructor (factory) function that returns the middleware function.

By convention, place middleware functions in the `server/middleware` directory.

A middleware handler function accepts three arguments, or four arguments if it is error-handling middleware. The general form is:

```
function myMiddlewareFunc([err,] req, res, next) { ... };
```

Name	Type	Optional?	Description
err	Object	Required for error-handling middleware.	Use *only* for error-handling middleware. Error object, usually an instance or `Error`; for more information, see Error object.
req	Object	No	The Express request object.
res	Object	No	The Express response object.
next	Function	No	Call `next()` after your application logic runs to pass control to the next middleware handler.

An example of a middleware function with three arguments, called to process the request when previous handlers did not report an error:

Regular middleware

```
return function myMiddleware(req, res, next) {
    // ...
}
```

Here is a constructor (factory) for this function; use this form when registering it in `middleware.json`:

Regular middleware

```
module.exports = function() {
  return function myMiddleware(req, res, next) {
    // ...
  }
}
```

An example a middleware function with four arguments, called only when an error was encountered.

Error handler middleware

```
function myErrorHandler(err, req, res, next) {
  // ...
}
```

Packaging a middleware function

To share middleware across multiple projects, create a package that exports a middleware constructor (factory) function that accepts configuration options and returns a middleware handler function; for example, as shown below.

If you have an existing project created via `slc loopback`, to implement a new middleware handler that you can share with other projects, place the file with the middleware constructor in the `server/middleware` directory, for example, `server/middleware/myhandler.js`.

```
module.exports = function(options) {
  return function customHandler(req, res, next) {
    // use options to control handler's behavior
  }
};
```

For details about the `options` object, refer to Middleware configuration properties.

Registering middleware in middleware.json

The easiest way to register middleware is in `server/middleware.json`. This file specifies all an application's middleware functions and the phase in which they are called.

When you create an application using the `slc loopback` application generator, it creates a default `middleware.json` file that looks as follows:

server/middleware.json

```
{
  "initial:before": {
    "loopback#favicon": {}
  },
  "initial": {
    "compression": {}
  },
  "session": {
  },
  "auth": {
  },
  "parse": {
  },
  "routes": {
  },
  "files": {
  },
  "final": {
    "loopback#urlNotFound": {}
  },
  "final:after": {
    "errorhandler": {}
  }
}
```

Each top-level key in `middleware.json` defines a middleware phase or sub-phase, for example "initial", "session:before", or "final". Phases run in the order they occur in the file.

Each phase is a JSON object containing a key for each middleware function to be called in that phase. For example, "loopback/server/middleware/favicon" or "compression".

In general, each phase has the following syntax:

```
phase[:sub-phase] : {
 middlewarePath : {
 [ enabled: [true | false] ]
 [, name: nameString ]
 [, params : paramSpec ]
 [, methods: methodSpec ]
 [ paths : routeSpec ]
 }
}
```

Where:

- *phase* is one of the predefined phases listed above (initial, session, auth, and so on) or a custom phase; see Adding a custom phase.
- *sub-phase* (optional) can be `before` or `after`.
- name: optional middleware name. See Middleware configuration properties below.
- *middlewarePath*: path to the middleware function. See Path to middleware function below.
- *paramSpec*: value of the middleware parameters, typically a JSON object. See Middleware configuration properties below.
- *methodSpec*: HTTP methods, such as 'GET', 'POST', and 'PUT'. If not present, applies to all methods.
- *routeSpec*: REST endpoint(s) that trigger the middleware.

Path to middleware function

Specify the path to the middleware function (*middlewarePath)* in the following ways:

- For an external middleware module installed in the project, just use the name of the module; for example `compression`. See Using other middleware.

- For a script in a module installed in the project, use the path to the module; for example `loopback/server/middleware/rest`.
- For a script with a custom middleware function, use the path relative to `middleware.json`, for example `./middleware/custom`.
- Absolute path to the script file (not recommended).

Additionally, you can use the shorthand format `{`*`module`*`}#{`*`fragment`*`}`, where *fragment* is:

- A property exported by *module*, for example `loopback#favicon` is resolved to `require('loopback').favicon`.
- A file in *module's* `server/middleware` directory, for example `require('loopback/server/middleware/favicon')`
- A file in *modules*' `middleware` directory, for example `require('loopback/middleware/favicon')`

Middleware configuration properties

You can specify the following properties in each middleware section. They are all optional:

Property	Type	Description	Default
name	String	An optional name for the entry. It can be used to identify an entry within the same phase/path for the purpose of merging	N/A
enabled	Boolean	Whether to register or enable the middleware. You can override this property in environment-specific files, for example to disable certain middleware when running in production. For more information, see Environment-specific configuration	true
params	Object or Array	Parameters to pass to the middleware handler (constructor) function. Most middleware constructors take a single "options" object parameter; in that case the `params` value is that object. To specify a project-relative path (for example, to a directory containing static assets), start the string with the prefix `$!`. Such values are interpreted as paths relative to the file `middleware.json`. See examples below.	N/A
methods	String[]	Specifies the HTTP methods, such as 'GET', 'POST', and 'PUT'. If not present, it will apply to all methods.	N/A
paths	String[]	Specifies the REST endpoint(s) that trigger the middleware. In addition to a literal string, route can be a path matching pattern, a regular expression, or an array including all these types. For more information, see the app.use (Express documentation).	Triggers on all routes
optional	Boolean	Specify whether the middleware is optional. Optional middleware do not throw, even if they are not installed or cannot be resolved in the file system.	N/A

Example of a typical middleware function that takes a single "options" object parameter:

```
"compression": {
    "params": { "threshold": 512 }
  }
```

Example of a middleware function that takes more than one parameter, where you use an array of arguments:

```
"morgan": {
    "params": ["dev", { "buffer": true } ]
  }
```

Example or an entry for static middleware to serve content from the `client` directory in the project's root:

```
...
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  }
...
```

Example or an entry for static middleware to serve content from the `multiple` directories in the project's root:

```
...
  "files": {
    "loopback#static": [{
      "name": "x",
      "paths": ["/x"],
      "params": "$!../client/x"
    },
    {
      "name": "y",
      "paths": ["/y"],
      "params": "$!../client/y"
    }]
  }
...
```

Using variables in values

For any middleware configuration property, you can specify a variable in the value using the following syntax:

```
${var}
```

Where *var* is a property of the app object. These properties include:

- Application-wide properties such as those defined in `config.json`.
- Express app object properties.

For example, the following `middleware.json` configuration will load LoopBack's built-in rest middleware (loopback.rest) during the routes phase at the path resolved by `app.get('restApiRoot')`, which defaults to `/api`.

```
{
  "routes": {
    "loopback#rest": {
      "paths": ["${restApiRoot}"]
    }
  }
}
```

The following example loads hypothetical middleware named `environmental` during the routes phase at the return value of `app.get(env)`, typically either `/development` or `/production`.

```
{
  "routes": {
    "environmental": {
      "paths": "${env}"
    }
  }
}
```

Adding a custom phase

In addition to the predefined phases in `middleware.json`, you can add your own custom phase simply by adding a new top-level key.

For example, below is a `middleware.json` file defining a new phase "log" that comes after "parse" and before "routes":

server/middleware.json

```
{
  ...
  "parse": {},
  "log": { ... },
  "routes": {}
  ...
}
```

Environment-specific configuration

You can further customize configuration through `middleware.local.js`, `middleware.local.json`, and `middleware.*env*.js` or `middleware.*env*.json`, where *env* is the value of `NODE_ENV` environment variable (typically `development` or `production`).

See Environment-specific configuration for more information.

Registering middleware in JavaScript

You can register middleware in JavaScript code with:

- LoopBack API; you can specify the phase in which you want the middleware to execute.
- Express API; the middleware is executed at the beginning of the `routes` phase.

Using the LoopBack API

To register middleware with the LoopBack phases API, use the following `app` methods:

- `middleware()`
- `middlewareFromConfig()`
- `defineMiddlewarePhases()`

For example:

server/server.js

```
var loopback = require('loopback');
var morgan = require('morgan');
var errorhandler = require('error-handler');

var app = loopback();

app.middleware('routes:before', morgan('dev'));
app.middleware('final', errorhandler());
app.middleware('routes', loopback.rest());
```

Using the Express API

When you register middleware with the Express API, it is always executed at the beginning of the `routes` phase.

You can define middleware the "regular way" you do with Express in the main application script file, `/server/server.js` by calling `app.use()` to specify middleware for all HTTP requests to the specified route; You can also use `app.get()` to specify middleware for only GET requests, a `pp.post()` to specify middleware for only POST requests, and so on. For more information, see app.METHOD in Express documentation.

Here is the general signature for `app.use()`:

```
app.use([route], function([err,] req, res, next) {
  ...
  next();
});
```

As usual, app is the LoopBack application object: `app = loopback()`.

The parameters are:

1. *`route`*, an optional parameter that specifies the URI route or "mount path" to which the middleware is bound. When the application receives an HTTP request at this route, it calls (or *triggers*) the handler function. See Specifying routes.
2. The middleware handler function (or just "middleware function"). See Defining a new middleware handler function.

For example:

server/server.js

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

var app = module.exports = loopback();

// Bootstrap the application, configure models, datasources and middleware.
// Sub-apps like REST API are mounted via boot scripts.
boot(app, __dirname);
// this middleware is invoked in the "routes" phase
app.use('/status', function(req, res, next) {
  res.json({ running: true });
});
```

Specifying routes

The *`route`* parameter is a string that specifies the REST endpoint that will trigger the middleware. If you don't provide the parameter, then the middleware will trigger on all routes. In addition to a literal string, *`route`* can be a path matching pattern, a regular expression, or an array including all these types. For more information, see the Express documentation for app.use().

For example, to register middleware for all endpoints that start with "/greet":

/server/server.js

```
app.use('/greet', function(req, res, next ) {
  ...
})
```

The above middleware is triggered by all routes that begin with "/greet", so "/greet/you", "greet/me/and/you" will all trigger it..

To register middleware for *all* endpoints:

server/server.js or server/boot/scripts.js

```
app.use(function(req, res, next ) {
  ...
})
```

Caveats

There are some things to look out for when using middleware, mostly to do with middleware declaration order. Be aware of the order of your middleware registration when using "catch-all" routes. For example:

server/server.js

```
...
app.get('/', function(req, res, next) {
  res.send('hello from `get` route');
});
app.use(function(req, res, next) {
  console.log('hello world from "catch-all" route');
  next();
});
app.post('/', function(req, res, next) {
  res.send('hello from `post` route')
});
...
```

In this case, since the `GET /` middleware ends the response chain, the "catch-all" middleware is never triggered when a get request is made. However, when you make a POST request to /, the "catch-all" route is triggered because it is declared **before** the post route. Doing a POST will show the console message from both the "catch-all" route and the `POST /` route.

Examples

Static middleware

Example or an entry for static middleware to serve content from the `client` directory in the project's root:

```
...
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  }
...
```

Pre-processing middleware

Use pre-processing middleware to apply custom logic for various endpoints in your application. Do this by registering handler functions to perform certain operations when HTTP requests are made to specific endpoint or endpoints.

> Always register pre-processing middleware in a phase before `routes`, for example `initial` or `parse`.

Pre-processing middleware must call `next()` at the end of the handler function to pass control to the next middleware. If you don't do this, your application will essentially "freeze." Technically, `next()` doesn't have to occur at the end of the function (for example, it could occur in an `if`/`else` block), but the handler function must call it eventually. For example:

```
module.exports = function() {
  return function tracker(req, res, next) {
    console.log('Request tracking middleware triggered on %s', req.url);
    var start = process.hrtime();
    res.once('finish', function() {
      var diff = process.hrtime(start);
      var ms = diff[0] * 1e3 + diff[1] * 1e-6;
      console.log('The request processing time is %d ms.', ms);
    });
    next();
  };
};
```

This middleware tells the server to display the time it takes to process the incoming HTTP request on all application routes.

You can see this middleware in action in using the basic LoopBack application from Getting started with LoopBack (or any standard LoopBack application):

1. Add the code above to `server/middleware/tracker.js`.
2. Edit (or create) the file `server/middleware.json` and register the new middleware in the "initial" phase:

   server/middleware.json

   ```
   {
     "initial": {
       "./middleware/tracker": {}
     }
   }
   ```

3. Start the application:

   ```
   $ node .
   ```

4. Load http://localhost:3000/ in your browser.

In the console, you will see (for example):

```
...
Request tracking middleware triggered on /.
The request processing time is 4.281957 ms. //your results will vary
```

Routing middleware

For routes serving JSON, best practice is to create a new model and implement the routes as remote methods. For routes serving non-JSON responses, best practice is to define them the standard "Express way" in `server.js` or a boot script. For more information, see Routing and Routing (Express documentation).

> If you add middleware on the `route` or `route:after` phase, it will not execute *after* the route is matched. Instead, it will be ignored because the route was already matched.

Error-handling middleware

Use error-handling middleware to deal with request errors. While you are free to register any number of error-handling middleware, be sure to register them in the "final" phase. LoopBack registers two error-handling middleware by default:

- `urlNotFound` middleware converts all requests that reach the middleware into an Error object with status 404, so that 404 error responses are consistent with "usual" error responses.
- `errorhandler` middleware is from the errorhandler module, previously available in Express v.3 as `express.errorHandler`. For information on customizing this error handler, see Customizing REST error handling.

Example of a custom error processing middleware:

```
module.exports = function() {
   return function logError(err, req, res, next) {
             console.log('ERR', req.url, err);
          };
};
```

To register this middleware:

1. Add the code above to `/server/middleware/log-error.js`.
2. Edit `/server/middleware.json` and register the new middleware in the "final" phase:

   ```
   { "final": { "./middleware/log-error": {} } }
   ```

3. Start the application.
4. Load http://localhost:3000/url-does-not-exist in your browser.

Upgrading applications to use phases

Introduction

LoopBack version 2.8 introduced middleware phases. Before that, middleware was registered in `server/server.js`, for example:

```
// Set up the /favicon.ico
app.use(loopback.favicon());

// request pre-processing middleware
app.use(loopback.compress());

// -- Add your pre-processing middleware here --

// boot scripts mount components like REST API
boot(app, __dirname);

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

// Requests that get this far won't be handled
// by any middleware. Convert them into a 404 error
// that will be handled later down the chain.
app.use(loopback.urlNotFound());

// The ultimate error handler.
app.use(loopback.errorHandler());
```

To upgrade your project to use middleware phases, you need to move the middleware configuration from `server/server.js` to `server/middleware.json`.

Middleware added after boot

Error handlers

Replace the line `app.use(loopback.errorHandler())` with the following entry in the JSON file:

```
{
  "final:after": {
    "errorhandler": {}
  }
}
```

404 handler

Replace the line `app.use(loopback.urlNotFound())` with the following entry in the JSON file:

```
{
  "final": {
    "loopback#urlNotFound": {}
  }
}
```

Static middleware

Register middleware serving static assets (files) in the `files` phase. Prefix the relative path to assets with `$!` so the path is resolved relative to `m` `iddleware.json`.

For example, replace the following line in `server/server.js`:

```
app.use(loopback.static(path.resolve(__dirname, '../client')));
```

with this middleware entry:

```
...
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  },
...
```

Other post-boot handlers

If your `server/server.js` file registers any other middleware after calling `boot(app, __dirname)`, you need to move that registration to `server/middleware.json`. Insert a new phase just before `final` and register the middleware there.

Middleware added before boot

The middleware registered **before** `boot(app, __dirname)` is usually pre-processing the requests.

LoopBack provides multiple phases for such middleware, you need to pick up the right phase for each of them.

- `favicon` should be called as the very first middleware, even before request loggers. Register it in `initial:before` phase.
- `compression` should be called very early in the middleware chain to enable response compression as soon as possible. Register it in `initial` phase.
- `express-session` (a.k.a. `loopback.session()`) should be registered in `session` phase.
- `loopback#token` (a.k.a. `loopback.token()`) belongs to `auth` phase.
- `morgan` (a.k.a. `loopback.logger()`) should usually go to `initial` phase.
- Request body parsers like `bodyParser#json` belong to `parse` phase.

Middleware registered from boot scripts

Any middleware installed via `app.use(fn)` is added to the `routes` phase. Call`app.middleware(phase, fn)` to add the middleware to a different phase.

Example:

```
module.exports = function(app) {
  app.middleware('initial', mylogger());
};
```

Example of migrated files

After you have applied the steps outlined above to the example `server/server.js` listed at the beginning of this guide, you should end up with the following two files.

The main server script does not register any middleware, but just calls `boot()`:

server/server.js

```
// boot scripts mount components like REST API
boot(app, __dirname);
```

All the middleware registration now occurs in `middleware.json`:

server/middleware.json

```
{
  "initial:before": {
    "loopback#favicon": {}
  },
  "initial": {
    "compression": {}
  },
  "session": {
  },
  "auth": {
  },
  "parse": {
  },
  "routes": {
  },
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  },
  "final": {
    "loopback#urlNotFound": {}
  },
  "final:after": {
    "errorhandler": {}
  }
}
```

Working with LoopBack objects

Related articles:

Overview

The primary LoopBack JavaScript objects are:

- App object
- Models
- Data sources

How to get a reference to these objects depends on where the code is (in a boot script, in a model JavaScript file `/common/models/model.js`, and so on) as well as which object you want to reference.

Getting the app object

Getting a reference to the `app` object is crucial, since from it you can obtain references to other objects such as models and data sources. You'll typically want to get a handle on the `app` object in:

- Model scripts: `/common/models/`*`modelName`*`.js` (where *`modelName`* is the name of the model).
- Boot scripts in `/server/boot`
- Middleware (the ones you register in boot scripts and the ones in `/server/server.js`)
- Your own custom scripts

The `app` object provides context into various parts of a typical LB app.

From a boot script

To get a reference to the `app` object in a boot script, pass it as the first parameter in the exported function.

Asynchronous boot script with a callback function:

Asynchronous boot script - /server/boot/your-script.js

```
module.exports = function(app, cb) { //app is injected by LoopBack
  ...
};
```

Synchronous boot script without a callback function:

Synchronous boot script - /server/boot/your-script.js

```
module.exports = function(app) { //app is injected by loopback
  ...
};
```

As you can see from both examples, LoopBack provides the `app` object automatically as the first parameter in your boot scripts.

See Defining boot scripts for more information about boot scripts.

From middleware

LoopBack sets `app` object automatically in the `request` object in middleware (actually, under the hood, Express does it). You can access in `server/server.js` as follows:

Middleware - /server/server.js

```
...
app.use(function(req, res, next) {
  var app = req.app;
  ...
});
...
```

See Defining middleware for more information on middleware.

From a custom script

If you need a reference to `app` in your own custom scripts, simply require it (as in the models example):

A custom script - /server/your-script.js

```
var app = require('/server/server');
...
```

You simply require `/server/server.js` as you would any Node module.

From a model script

To get a handle on the `app` object in a model scaffolded by the Model generator, "require" it as you would any Node module:

Model - /common/models/book.js

```
var app = require('../../server/server'); //require `server.js` as in any node.js app

module.exports = function(Book) {
  ...
};
```

With models, there is a special case. From anywhere except `/common/models/model.js`, you can actually get a reference to `app` through a model using `model.app`. For instance:

```
...
Book.app
...
```

However, the one caveat to this is that you cannot reference `model.app` in `/common/model/model.js` because this file does not add the `app` property until bootstrapping has finished. This means you **cannot** do the following in `/common/models/model.js`:

CANNOT do this in a model script

```
module.exports = function(Book) {
  Book.app... //won't work because `.app` has not been added to the Book object yet
});
```

However, you can get a reference to the app INSIDE remote methods, remote hooks, and model hooks because those are trigger after the application finishes loading (that is, after loopback.boot runs | after /server/server.js calls boot(..)). This means you CAN do:

```
module.exports = function(Book) {
  Book.read(cb) {
    var app = Book.app;
    console.log(app.models...)
    cb();
  };
  Book.remoteMethod(
    'read',
    ...
  });
};
```

Of course, you can do the same in remote hooks and remote methods, but be aware of the load timing. Simply put, `model.app` will not be available until the application has completed bootstrapping, that is run `boot()` in `/server/server.js`. The idea here is to define our models *b efore* they are added to the application. Once the application finishes bootstrapping, you can then access a model's `app` property.

The easiest way of accessing the app object is via `Model.on('attached')` event.

```
module.exports = function(MyModel) {
  var app;
  MyModel.on('attached', function(a) {
    app = a;
    // perform any setup that requires the app object
  });
};
```

Working with the app object

The LoopBack app object is defined in the main script as follows:

/server/server.js

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

The app object extends the Express app object; it inherits all of its properties and methods, as well as all the additional properties and methods of the LoopBack app object.

In some places such as boot scripts, `server` is used as the name of this object instead of `app`.

Working with model objects

Getting references to models

Once you get a handle on the `app` object, you can get a handle on to specific models via the `models` property on the `app` object.

Boot script - /server/boot/your-script.js

```
module.exports = function(app) {
  var app = app.models.Book;
  ...
};
```

In your own custom script:

A custom script - /server/your-script.js

```
var app = require('/server/server');
```

Using model objects

For information on the basic model object, see Basic model object. For information on model object when the model is connected to a persistent data source, see Connected model object.

Working with data source objects

Getting references to data sources

Similar to getting a handle on a model you first get a handle onto the `app` object, then you access the `app.datasources` property:

Boot script - /server/boot/your-script.js

```
module.exports = function(app) {
  var dataSource = app.datasources.db; //db can be any registered datasource in
`/server/datasources.json`
  ...
};
```

And in your own script:

A custom script - /server/your-script.js

```
var app = require('./server/server');
...
var datasource = app.datasources.db;
...
```

In middleware:

Middleware - /server/server.js

```
...
app.use(function(req, res, next) {
  var dataSource = app.datasources.db;
  ...
});
...
```

In models:

Model - /common/models/model.js

```
module.exports = function(Book) {
  Book.read = function() {
    var dataSource = Book.app.datasources.db;
  };
  Book.remoteMethod(
    'read',
     ...
  );
};
```

Be careful in models, because the following *will not work*:

Model - /common/models/model.js

```
module.exports = function(Book) {
  Book.app... //`Book` is not registered yet! This WON'T WORK.
};
```

Using data source objects

This section is still in progress. Thanks for your patience.

Prototype versus instance methods

Question

When I use the swagger spec to generate a javascript client I end up with something like the attached UsedModel client file. From a Node.js best practices perspective, I want to find out: what's the point of dividing methods between prototype and object in a client?

Why have:

```
this.login = function(parameters) {...};
UserModel.prototype.updateAttributes = function(parameters) {...};
```

When both can be:

```
this.login = function(parameters) {...};
this.updateAttributes = function(parameters) {...};
```

Answer

One visible difference is in the URLs. Static methods have URL `/api/modelName/methodName`, while prototype methods have URL `/api/modelName/id/methodName`.

The second difference is in the way how to use the method on the server. Static methods always have to fetch the model instance from the database, which is inefficient when a single HTTP request involves several calls. On the other hand, prototype methods operate on the same instance.
Example (using sync code to keep it simple):

```
/** static-only methods **/
OrderItem.addCount(id, count); // database calls - read, write
OrderItem.updatePrice(id); // 2 database calls - read, write

/* prototype methods */
var order = OrderItem.findById(123); // database call - read
order.addCount(1);
order.updatePrice();
order.save(); // database call - write
```

When it comes to client SDKs, the situation is little bit more complex. The isomorphic client can share the implementation of `addCount` and `updatePrice` with the server, thus the code above would involve two REST requests only - `GET /api/OrderItems/123` and `PUT /api/OrderItems/123`.

Other clients (iOS, Android, Angular, swagger-generated js client) are not able to use that. However, it is always possible to wrap the code in a new shared method and call this new method from the client, e.g. `POST /api/OrderItems/123/add/1` or `POST /api/OrderItems/add { id: 123, amount: 1 }`.

As a rule of thumb, you should use static methods for actions that are not necessarily bound to a model instance known by a client (e.g. `User.login`, `PersistedModel.create`,`PersistedModel.find`). Use prototype (instance) methods for actions that operate on a single given model instance (`PersistedModel.prototype.updateAttributes`, `OrderItem.prototype.updatePrice()`).

Looking at the problem from the client side, the benefit of saving DB calls is not applicable. It all boils down to what kind of API do you prefer. Since all prototype methods require model id, you can convert prototype methods to static methods with an extra argument - this is exactly what we do in Angular SDK.

```
// server
var pm = new PersistedModel({ id: modelId });
pm.updateAttributes(data, cb)

// Angular client
PersistedModel.prototype$updateAttributes({ id: modelId }, data);
If your client is smart enough to map model properties to request parameters (e.g.
path params), then a prototype method may keep the code more concise.

OrderItem.find({ where: { productName: 'pen' } }, function(err, list) {
  async.each(list, function(it, next) {
    // 1. prototype method
    it.updateAttributes({ count: 2 }, next); // automagically build URL using `it.id`

    // 2. static method
    OrderItem.updateAttributes(it.id, { count: 2 }, next);

  }, cb);
});
```

To make it short, there is no hard rule for the client SDK (API) prescribing how to map static and prototype methods. Use whatever works best for you.

Using current context

LoopBack applications sometimes need to access context information to implement the business logic, for example to:

See also: Example in LoopBack repository.

- Access the currently logged-in user.
- Access the HTTP request (such as URL and headers).

A typical request to invoke a LoopBack model method travels through multiple layers with chains of asynchronous callbacks. It's not always possible to pass all the information through method parameters.

Configure context propagation

LoopBack context is now enabled by default for REST APIs via `loopback.rest()` middleware. Configure it in `server/config.json` as follows:

```
"remoting": {
  "context": {
    "enableHttpContext": false
  },
  ...
}
```

By default, the HTTP req/res objects are not set onto the current context. You need to set `enableHttpContext` to true to enable automatic population of req/res objects.

Use the current context

Once you've enabled context propagation, you can access the current context object using `loopback.getCurrentContext()`. The context should be available in middleware (those come after the context middleware), remoting hooks, model hooks, and custom methods.

```
MyModel.myMethod = function(cb) {
  var ctx = loopback.getCurrentContext();
  // Get the current access token
  var accessToken = ctx.get('accessToken');
  ...
  // Set more information on current context
  ctx.set('foo', { bar: 'val' } );

  ...
}
```

Use current authenticated user in remote methods

The `loopback.context()` has been added to `loopback.rest()` to ensure that all REST applications have the context available, even if they don't add the middleware explicitly. In advanced use cases, for example when you want to add custom middleware, you have to add the context middleware at the right position in the middleware chain (before the middleware that depends on `loopback.getCurrentContext`).

> `loopback.context()` detects the situation when it is invoked multiple times on the same request and returns immediately in subsequent runs.

Here's sample code which uses a middleware function to place the currently authenticated user into the context so that remote methods may use it:

/server/server.js Expand source

```
...
// -- Add your pre-processing middleware here --
app.use(loopback.context());
app.use(loopback.token());
app.use(function setCurrentUser(req, res, next) {
  if (!req.accessToken) {
    return next();
  }
  app.models.UserModel.findById(req.accessToken.userId, function(err, user) {
    if (err) {
      return next(err);
    }
    if (!user) {
      return next(new Error('No user with this access token was found.'));
    }
    var loopbackContext = loopback.getCurrentContext();
    if (loopbackContext) {
      loopbackContext.set('currentUser', user);
    }
    next();
  });
});


// boot scripts mount components like REST API
...
```

/common/models/YourModel.js

Expand source

```
var loopback = require('loopback');
module.exports = function(YourModel) {
  ...
  //remote method
  YourModel.someRemoteMethod = function(arg1, arg2, cb) {
    var ctx = loopback.getCurrentContext();
    var currentUser = ctx && ctx.get('currentUser');
    console.log('currentUser.username: ', currentUser.username); // voila!
    ...
    cb(null);
  };
  ...
};
```

Events

Related articles:

In addition to the standard Node events, LoopBack applications and models emit other events.

Application events

By default, an application scaffolded with slc loopback emits a 'started' event when it starts up, after running boot scripts.

Model events

All models emit the following events:

- changed
- deleted
- deletedAll
- attached
- dataSourceAttached
- set

By default, the basic LoopBack Model object has properties and methods "mixed in" from:

- Inclusion object - Enables you to load relations of several objects and optimize numbers of requests.
- Validateable object - provides validation methods; see Validating model data.

When you define relations between models, the RelationMixin object object also gets mixed in to the model object.

changed

Emitted after a model has been successfully created, saved, or updated. Argument: `inst`, model instance, object. For example:

```
MyModel.on('changed', function(inst) {
  console.log('model with id %s has been changed', inst.id);
  // => model with id 1 has been changed
});
```

deleted

Emitted after an individual model has been deleted. Argument: `id`, model ID (number). For example:

```
MyModel.on('deleted', function(id) {
  console.log('model with id %s has been deleted', id);
  // => model with id 1 has been deleted
});
```

deletedAll

Emitted after an individual model has been deleted. Argument: `where` (optional), where filter, JSON object. For example:

```
MyModel.on('deletedAll', function(where) {
  if (where) {
    console.log('all models where ', where, ' have been deleted');
    // => all models where
    // => {price: {gt: 100}}
    // => have been deleted
  }
});
```

attached

Emitted after a `Model` has been attached to an `app`.

dataSourceAttached

Emitted after a `Model` has been attached to a `DataSource`.

set

Emitted when model property is set. Argument: `inst`, model instance, object. For example:

```
MyModel.on('set', function(inst) {
  console.log('model with id %s has been changed', inst.id);
  // => model with id 1 has been changed
});
```

Arguments: data, an object.

PersistedModel events

PersistedModels also have a changed event that listens for model changes. For example:

```
MyPersistedModel.on('changed', function(obj) {
   console.log(obj) // => the changed model
});
```

User model events

The User model `User.resetPassword()` method emits the 'resetPasswordRequest' event.

Running and debugging apps

In general, when you are developing an application, use the `node` command to run it. This enables you to see stack traces and console output immediately. For example:

```
$ cd myapp
$ node .
```

When you're ready to start tuning your app, use `slc start` to run it locally under the control of StrongLoop Process Manager; this enables you to profile it and monitor app metrics to help find memory leaks and optimize performance. See Profiling and Monitoring app metrics for more information.

> When running an application, you can specify debug strings that the application will display to the console (or save to a file), and you can also use Node Inspector to debug the running app. For more information, see Setting debug strings.

Running local apps with slc

Run a Node application (including a LoopBack application) under control of StrongLoop PM to:

- View CPU profiles and heap snapshots to optimize performance and diagnose memory leaks.
- Keep processes and clusters alive forever.
- View performance metrics.
- Run the app as a cluster of Node processes.

For more information, see Using Process Manager.

> The `slc start` command does not run on Windows systems. However, you can run StrongLoop PM on a Windows system and deploy to it. See Building and deploying for more information.

To run an app locally under control of StrongLoop Process Manager:

```
$ cd <app-root-dir>
$ slc start
```

Where `<app-root-dir>` is the application's root directory.

This starts a local instance of StrongLoop Process Manager (StrongLoop PM) and runs the specified application under its control. If PM is unable to start the application, it will periodically try to start the app until the PM is shut down.

StrongLoop PM will display some suggested commands, followed by information from the log file, for example:

```
...
--- tail of /Users/rand/.strong-pm/start.log ---
slc start(32276): StrongLoop PM v5.0.49 (API v6.1.0) on port `8701`
slc start(32276): Base folder `/Users/rand/.strong-pm`
slc start(32276): Applications on port `3000 + service ID`
Run request for commit "default-app/local-directory" on current (none)
Start Runner: commit default-app/local-directory
2015-08-14T17:07:23.361Z pid:32289 worker:0 INFO strong-agent v1.6.54 profiling app
'default-app' pid '32289'
2015-08-14T17:07:23.368Z pid:32289 worker:0 INFO strong-agent[32289] started profiling
agent
2015-08-14T17:07:23.369Z pid:32289 worker:0 INFO supervisor starting (pid 32289)
2015-08-14T17:07:23.374Z pid:32289 worker:0 INFO strong-agent strong-agent using
strong-cluster-control v2.1.2
2015-08-14T17:07:23.377Z pid:32289 worker:0 INFO supervisor reporting metrics to
`internal:`
2015-08-14T17:07:23.388Z pid:32289 worker:0 INFO supervisor size set to 1
2015-08-14T17:07:23.500Z pid:32289 worker:0 INFO supervisor started worker 1 (pid
32290)
2015-08-14T17:07:23.501Z pid:32289 worker:0 INFO supervisor resized to 1
2015-08-14T17:07:23.828Z pid:32290 worker:1 INFO strong-agent v1.6.54 profiling app
'default-app' pid '32290'
2015-08-14T17:07:23.832Z pid:32290 worker:1 INFO strong-agent[32290] started profiling
agent
```

You can also run an application from another directory. For example, if your application is in the `myApp` directory under the current working directory:

```
$ slc start myapp
```

To run an application remotely, use Process Manager; see Operating Node applications for more information.

To view the status of the application, use the `slc ctl` command, which by default displays the status of the locally-running StrongLoop PM:

```
$ slc ctl
Service ID: 1
Service Name: myapp
Environment variables:
  No environment variables defined
Instances:
    Version  Agent version  Cluster size
     4.1.1       1.5.1            4
Processes:
        ID      PID   WID  Listening Ports  Tracking objects?  CPU profiling?
    1.1.48554  48554   0
    1.1.48557  48557   1     0.0.0.0:3001
    1.1.48563  48563   2     0.0.0.0:3001
    1.1.48565  48565   3     0.0.0.0:3001
    1.1.48566  48566   4     0.0.0.0:3001
```

Where `myapp` is your application's service name (by default, the `name` property in `package.json`).

To see log output (including error messages and a stack trace), use this command:

```
$ slc ctl log-dump myapp
```

If PM cannot successfully run the application, you can make changes to the application code, and PM will automatically try to run it: you don't have to use `slc start` again.

Stop the application with:

```
$ slc ctl stop myapp
```

Setting debug strings

See also: Debugging applications for information on using the Node Inspector debugger.

You can specify debug strings when you run an application, as explained below, to display specific log output to the console. You can also redirect the output to a file, if desired. These techniques are often helpful in debugging applications.

Using debug strings

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

$ DEBUG=<*pattern*>[,<*pattern*>...] node .

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

For example:

```
$ DEBUG=loopback:datasource node .
```

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

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

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

```
$ DEBUG=*oracle node .
```

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

Debug string format

These strings have the format

module[:*area*]:*fileName*

Where

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

For example

`loopback:security:access-context`

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

Debug strings reference

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

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

Preparing for deployment

See also:

- Using Process Manager
- Building and deploying
- Deployment best practices
- Scaling

Configuration for deployment

When you move from deployment to production or staging, you typically want to change your datasource from your internal testing database (or even the in-memory data store) to a production database where your live application data will reside. Additionally, you may want to change application properties such as host name and port number.

By default, a LoopBack application created with the application generator has two kinds of configuration files in the server directory that you use to configure production settings:

- `config.json` containing general application configuration. You can override the settings in this file with `config.`*`env`*`.json`, where *env* is the value of NODE_ENV environment variable.
- `datasources.json` containing data source configuration. You can override the settings in this file with `datasources.`*`env`*`.json`, where *env* is the value of NODE_ENV environment variable.

Set NODE_ENV to a string reflecting the host environment, for example "development" or "production".

To get ready for production, create at least two copies of these files: `config.production.json` and `config.development.json`; and `datasources.production.json` and `datasources.development.json`. You can create additional files (for example, `config.staging.jso`

n) if desired. Then, make sure on your development system you set the NODE_ENV to "development" and on your production system you set NODE_ENV to "production".

> Setting NODE_ENV to "production" will automatically turn off stack traces in JSON responses.

For more information, see Environment-specific configuration.

Disabling API Explorer

LoopBack API Explorer is great when you're developing your application, but for security reasons you may not want to expose it in production.

For an application using loopback-component-explorer, to disable explorer in production:

- Set the NODE_ENV environment variable to "production".
- Then in `server/component-config.production.json`:

server/component-config.production.json

```
{
  "loopback-component-explorer": false
}
```

> For an application using the old `loopback-explorer` (prior to version 2.0), disable API Explorer by deleting or renaming `server/boot/explorer.js`.

Other changes

When you move your app from development to staging, you may want to make additional changes; for example, you might want to customize REST error responses. See Customizing REST error handling for more information.

Using SSL

For a working example app demonstrating how to use SSL with LoopBack, see loopback-example-ssl. The example code below is drawn from that repository.

Generate your own SSL certificate

Here's an example of generating an SSL certificate:

```
$ openssl genrsa -out privatekey.pem 1024
$ openssl req -new -key privatekey.pem -out certrequest.csr
$ openssl x509 -req -in certrequest.csr -signkey privatekey.pem -out certificate.pem
```

Load the SSL certificate

Once you've generated a certificate, load it in your app, for example:

ssl-config.js

```
var path = require('path'),
fs = require("fs");
exports.privateKey = fs.readFileSync(path.join(__dirname,
'./private/privatekey.pem')).toString();
exports.certificate = fs.readFileSync(path.join(__dirname,
'./private/certificate.pem')).toString();
```

Create the HTTPS server

server/server.js

```
var https = require('https');
var sslConfig = require('./ssl-config');
...
var options = {
  key: sslConfig.privateKey,
  cert: sslConfig.certificate
};
...

server.listen(app.get('port'), function() {
    var baseUrl = (httpOnly? 'http://' : 'https://') + app.get('host') + ':' +
app.get('port');
    app.emit('started', baseUrl);
    console.log('LoopBack server listening @ %s%s', baseUrl, '/');
});
return server;
```

Using StrongLoop Process Manager

Until now, you've run your application using the `node` command. This is fine for development, since it enables you to see stack traces directly in your console window, and to easily stop and restart your app when you need to. Once you're ready to move to production, however, you should start running your application with StrongLoop Process Manager (StrongLoop PM) because it enables you to:

- View CPU profiles and heap memory snapshots, to help you optimize performance and resource consumption. Often, you'll want to profile your application before you deploy to production, and then periodically thereafter as needed.
- Set up your build and deployment process so you can easily modify your app as needed. StrongLoop PM enables you to do this with zero downtime.
- Automatically restart the application if it crashes, providing high availability and high reliability.
- Cluster your app to use multiple CPU cores. This enables rapid vertical scalability to meet growing traffic.
- Scale horizontally using StrongLoop PM's built-in NGINX integration when your app outgrows a single host.
- Monitor application metrics such as event loop times, CPU and memory consumption to ensure that app performance is acceptable.

There are two ways to use StrongLoop PM:

- Run your app locally under management of PM with `slc start`. This is a quick way to start using StrongLoop PM, and is typical when your preparing for deployment.
- When you're ready to go to production, set up a production host with StrongLoop PM then build and deploy your app to PM using `slc` or StrongLoop Arc.

Tutorials and examples

This document is the central organizational point for LoopBack examples, which are divided into the following broad categories:

- Tutorials
- Topic-specific examples
- Client SDK examples

Tutorials

Each tutorial provides step-by-step instructions to create an example app that illustrates one or more core LoopBack features. In general, these tutorials are intended to be followed in order.

Topic-specific examples

These examples illustrate specific LoopBack features.

Client SDK examples

These examples illustrate use of LoopBack's client SDKs.

Blog posts

The StrongLoop blog contains lots of helpful posts relevant to LoopBack. Much of the information gets incorporated into the documentation, but you can still find lots of helpful tidbits in the blog!

LoopBack

Error rendering macro 'rss' : javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure

Latest blog posts (any topic)

Error rendering macro 'rss' : javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure

API tips

Error rendering macro 'rss' : javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure

Setting up an MBaas

End-to-end instructions on setting up an MBaaS with:

- Push
- Geopoint
- Offline sync and replication
- Social login

Using promises

> Promise support in LoopBack is not complete. See LoopBack issue #418 for current status.
>
> PLEASE NOTE: This documentation is a work in progress. The API documentation does not yet reflect promise support.

- What is a promise?
- LoopBack support for promises
- Setup
- Using promises in LoopBack

Promises are an alternative for asynchronous operations that can be simpler to write and understand than traditional callback-based approaches. Promises also enable you to handle asynchronous errors with something similar to a synchronous `try/catch` block.

What is a promise?

A promise represents the result of an asynchronous operation. A promise is in one of the following states:

- **pending** - The initial state of a promise (neither fulfilled nor rejected).
- **fulfilled** - The action relating to the promise succeeded. When a successful promise is fulfilled, all of the pending callbacks are called with the value. If more callbacks are registered in the future, they will be called with the same value. Fulfillment is the asynchronous analog for returning a value.
- **rejected** - The action relating to the promise failed. When a promise is rejected it invokes the errbacks that are waiting and remembers the error that was rejected for future errbacks that are attached. Rejection is the asynchronous analog for throwing an exception.
- **settled** - The promise has been fulfilled or rejected. Once a promise is settled, it is immutable (it can never change again).

Additional terminology:

- *Callback*: A function executed if a a promise is fulfilled with a value.
- *Errback*: A function executed if a promise is rejected, with an exception.
- *Progressback*: A function executed to show that progress has been made toward resolution of a promise.

For more general information on promises, see:

- Understanding JavaScript Promises
- JavaScript Promises: There and Back Again
- Promises (by Forbes Lindsay)
- Promise (Mozilla Developer Network)

LoopBack support for promises

Promise support in LoopBack is still in progress. The following are complete:

- Strong Remoting
- Operation hooks
- DAO and relation methods
- loopback-datasource-jugglermethods like `automigrate`, `autoupdate`, `discover*`, etc.
- The following built-in models support the promises API:
  - User
  - Application
  - PersistedModel

See LoopBack issue #418 for details.

Setup

When using Node v0.12+ or io.js 1.0+, you can use the native global Promise object.

With earlier versions of Node, use Bluebird. When running in an environment that supports native promises, Bluebird will automatically "fall back" to use them, so typically, it's easier to set up Bluebird so you don't need to be concerned with platform support. Simply, enter this command to update your application's dependencies in `package.json`:

```
$ npm install -save bluebird
```

Then, in your code:

```
var Promise = require('bluebird');
...
```

Using promises in LoopBack

> If using CoffeeScript, make sure your models don't accidentally return a Promise.

For example, here is how you would call a CRUD operation on a model that extends PersistedModel with standard callbacks:

```
MyModel.find(function(err, result){
  ...
  if (err) cb(err)
})
```

With promises, you would instead do the following:

```
MyModel.find()
.then(function(result){
  ... // Called if the operation succeeds.
})
.catch(function(err){
  ... // Called if the operation encounters an error.
})
```

Another example:

```
var Promise = require('bluebird');
CoffeeShop.status = function() {
  var currentDate = new Date();
  var currentHour = currentDate.getHours();
  var OPEN_HOUR = 6;
  var CLOSE_HOUR = 20;
  console.log('Current hour is ' + currentHour);
  var response;
  if (currentHour > OPEN_HOUR && currentHour < CLOSE_HOUR) {
    response = 'We are open for business.';
  } else {
    response = 'Sorry, we are closed. Open daily from 6am to 8pm.';
  }

  return Promise.resolve(response);
};
```

Promises can simplify, for example, defining asynchronous remote methods. Instead of:

common/models/my-model.js

```
module.exports = function(MyModel) {
 MyModel.myFunc = function(input, cb) {
   Todo.find(function(err, data) {
     if(err) return cb(err);
     cb(null, generateStats(input, data));
   });
 };
```

With promises, this is reduced to:

common/models/my-model.js

```
MyModel.myFunc = function(input, cb) {
   return Todo.find()
     .map(generateStats(input));
 };
 MyModel.remoteMethod('myFunc', ...);
}
```

Reference

- Command-line reference (slc loopback)
- Project layout reference
- Basic model object
- Connected model object
- LoopBack middleware
- LoopBack API reference
- LoopBack types
- Valid names in LoopBack
- Release notes
- LoopBack Definition Language (LDL)

- Error object

Command-line reference (slc loopback)

Prerequisite: Install StrongLoop software following the instructions in Getting started with LoopBack.

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

The `slc loopback` command provides an Application generator to create a new LoopBack application and a number of sub-generators to scaffold an application, as described in the following table. The commands are listed roughly in the order that you would use them.

Command	**See**	**Description**
`slc loopback`	Application generator	Create a new LoopBack application.
`slc loopback:datasource`	Data source generator	Add a new data source to a LoopBack application
`slc loopback:model`	Model generator	Add a new model to a LoopBack application.
`slc loopback:property`	Property generator	Add a new property to an existing model.
`slc loopback:acl`	ACL generator	Add a new access control list (ACL) entry to the LoopBack application.
`slc loopback:relation`	Relation generator	Add a new model relationship.
`slc loopback:middleware`	Middleware generator	Add a new middleware configuration.
`slc loopback:boot-script`	Boot script generator	Add a new boot scripts.
`slc loopback:export-api-def`	API definition generator	Export Swagger API definition.
`slc loopback:swagger`	Swagger generator	Generates a fully-functional application that provides the APIs conforming to the Swagger 2.0 specification.

The `slc` command has many additional sub-commands not specific to LoopBack for building, deploying, and managing Node applications. See Operating Node applications for more information and Command-line reference for the command reference.

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

`npm install -g generator-loopback.`

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

ACL generator

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

Before running this generator, you must create an application using `slc loopback`, the application generator.

You cannot modify built-in models using the ACL generator.

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

The tool will prompt you for the necessary information and then modify the Model definition JSON file accordingly.

The generator prompts for:

- Name of the model to which you want to apply access control or all models.
- Scope of access control: All methods and properties or a specific method.
- If you choose a specific method, the method's name.
- Access type: read, write, execute, or all.
- Role: all users, any unauthenticated user, any authenticated user, the object owner.
- Permission to apply: explicitly grant access or explicitly deny access.

For general information about setting up ACLs, see Controlling data access.

API definition generator

Use the LoopBack API definition generator to export a Swagger API definition for a LoopBack application.

> Before running this generator, you must create an application using `slc loopback`, the application generator.

```
$ cd <loopback-app-dir>
$ slc loopback:export-api-def
```

By default, the generator displays the API definition to the screen (stdout). Use the `--o <filename>` option to save the definition to a file instead.

By default, the generator exports the API definition in YAML format. Use the `--json` option to export in JSON format instead; or specify a file with a `.json` format.

Application generator

The LoopBack application generator creates a new LoopBack application:

```
$ slc loopback
```

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

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

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

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

Boot script generator

The LoopBack boot script generator adds a new boot script to the LoopBack application.

> Before running this generator, you must create an application using `slc loopback`, the application generator.

```
$ cd <loopback-app-dir>
$ slc loopback:boot-script [script_name]
```

The tool will prompt you for:

- The name of the boot script, if you didn't provide it on the command-line.
- Whether you want to created an asynchronous or synchronous boot script.

Then `slc` will create a JavaScript file with the specified name in the application's `server/boot` directory, depending on your selection.

Asynchronous:

```
module.exports = function(app, cb) {
  process.nextTick(cb);
};
```

Synchronous:

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

Data source generator

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

> ⚠ Before running this generator, you must create an application using `slc loopback`, the application generator.

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

The tool will prompt you:

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

For example:

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

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

Middleware generator

The LoopBack middleware generator adds a new middleware configuration to an existing LoopBack application.

See also:

- Defining middleware

Before running this generator, you must create an application using `slc loopback`, the application generator.

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

The tool will prompt you to:

- Enter the name of the new middleware. If you supplied a name on the command-line, just hit **Enter** to use it.
- Select the phase to use for the middleware.
- Select the sub-phase to use for the middleware.
- Add a list of paths.
- Add the JSON parameters.

You must install the middleware package for your application with `npm install -save <middleware-package>`.

Here is an example to add a middleware to an existing phase.

```
$ slc loopback:middleware

? Enter the middleware name: m1
? Select the phase for m1: 5. routes
? Select the sub phase for m1: 1. before
Specify paths for m1:
Enter an empty path name when done.
? Path uri: /x
Let's add another path.
Enter an empty path name when done.
? Path uri:
? Configuration parameters in JSON format: {"a": 1}
Middleware m1 is added to phase routes.
```

The following is an example to add a middleware to a custom phase.

```
$ slc loopback:middleware
? Enter the middleware name: m2
? Select the phase for m2: (custom phase)
? Enter the phase name: p1
? Select the phase before which the new one will be inserted: 4. parse
? Select the sub phase for m2: 2. regular
Specify paths for m2:
Enter an empty path name when done.
? Path uri: /a
Let's add another path.
? Configuration parameters in JSON format: {"x": "2"}
Middleware m2 is added to phase p1.
```

Model generator

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

> Before running this generator, you must create an application using `slc loopback`, the application generator.

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

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

The tool will prompt you for:

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

Depending your response to the last prompt, the tool will create a new file defining the model; either `/common/models/`*`model-name`*`.json` (for use by client and server) or `/server/models/`*`model-name`*`.json` (server only). See Model definition JSON file for details.

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

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

Property generator

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

> Before running this generator, you must create an application using `slc loopback`, the application generator.

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

The tool will then prompt you to:

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

For example:

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

Relation generator

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

Before running this generator, you must create an application using `slc loopback`, the application generator.

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

Then `slc` will prompt you for:

- Name of the model to create the relationship from.
- Relation type (HasMany, BelongsTo, HasAndBelongsToMany, or HasOne).
- Name of the model to create a relationship with.
- Name for the relation (property name).

  The name of the relation must be different than a property it references.
- Custom foreign key (optional)
- Whether a "through" model is required. Repy "Y" to create a HasManyThrough relations.
  - Name of the "through" model, if appropriate.

Swagger generator

The LoopBack Swagger generator creates a fully-functional application with an API defined using the Swagger 2.0 specification.

See also: Swagger RESTful API Documentation Specification (version 2.0).

Before running this generator, you must create an application using `slc loopback`, the application generator.

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

The tool will prompt you for the location of the Swagger spec file (in JSON):

```
[?] Enter the swagger spec url or file path:
```

Enter a URL or a relative file path. Based on the REST API defined in this file, the tool will then prompt you for the models to generate. For example, if you enter the Swagger simple petstore example URL:

```
https://raw.githubusercontent.com/wordnik/swagger-spec/master/examples/v2.0/json/petst
ore-simple.json
```

The tool will display:

```
[?] Select models to be generated:
 swagger_api
  pet
  petInput
  errorModel
```

Move the cursor with the arrow keys, and press the space bar to de-select the model next to the cursor. Then press Return to generate all the selected models.

The tool will prompt you for the data source to use then display information on what it's doing; for example:

```
[?] Select the data-source to attach models to: db (memory)
Creating model definition for swagger_api...
Creating model definition for pet...
Creating model definition for petInput...
Creating model definition for errorModel...
...
```

Project layout reference

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

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

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

All your model JSON and JavaScript files go in the `/common/models` directory.

File or directory	Description	How to access in code
	Top-level application directory	
`/node-modules` directory	Contains Node packages as specified as dependencies in `package.json`. Update with `npm install`.	N/A
`package.json`	Standard npm package specification. See package.json.	N/A
`README.md`	Stub file for internal documentation.	N/A
	/server directory - Node application files	

`/boot` directory	Add scripts to perform initialization and setup. See boot scripts.	Scripts are automatically executed in alphabetical order.
`component-config.json`	Specifies LoopBack components to load.	
`config.json`	Application settings. See config.json.	`app.get('setting-name')`
`datasources.json`	Data source configuration file. See datasources.json. For an example, see Create new data source.	`app.datasources['datasource-name']`
`middleware.json`	Middleware definition file. For more information, see Defining middleware.	N/A
`middleware.production.json`	Middleware definition file with production configuration. See Preparing for deployment.	
`model-config.json`	Model configuration file. See model-config.json. For more information, see Connecting models to data sources.	N/A
`server.js`	Main application program file.	N/A
	/client directory - Client application files	
README.md	LoopBack generators create empty `README.md` file.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
	/common directory - shared application files	
`/models` directory	Custom model files: • Model definition JSON files, by convention named *`model-name`*`.json`; for example `customer.json`. • Custom model scripts by convention named *`model-name`*`.js`; for example, `customer.js`. For more information, see Model definition JSON file and Customizing models. See note below.	Node: `myModel = app.models.myModelName`

The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

package.json

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

For more information, see the package.json documentation.

For example:

```
{
  "name": "myapp",
  "version": "1.0.0",
  "main": "server/server.js",
  "scripts": {
    "start": "node .",
    "pretest": "jshint ."
  },
  "dependencies": {
    "compression": "^1.0.3",
    "cors": "^2.5.2",
    "loopback": "^2.22.0",
    "loopback-boot": "^2.6.5",
    "loopback-component-explorer": "^2.1.0",
    "loopback-datasource-juggler": "^2.39.0",
    "serve-favicon": "^2.0.1"
  },
  "devDependencies": {
    "jshint": "^2.5.6"
  },
  "repository": {
    "type": "",
    "url": ""
  },
  "description": "myapp"
}
```

server directory

The `/server` directory contains files that create the backend LoopBack (Node) application.

The standard files in this directory are:

- component-config.json
- config.json
- datasources.json
- middleware.json
- model-config.json
- policy-config.json
- server.js

The `boot` sub-directory contains the following boot scripts by default:

- `authentication.js`
- `root.js`

component-config.json

Use the `component-config.json` file to configure and load components of a LoopBack application. The file contains a top-level key for each component to load, where the key is the **module name** of a npm package or the relative path of a local component. For a component to be loaded, the value must be a JavaScript object that does not evaluate to false.

The default file that the Application generator creates is:

```
{
  "loopback-component-explorer": {
    "mountPath": "/explorer"
  }
}
```

When configuring a local component, the `.js` extension is optional.

Following is an example of a `component-config.json` file that loads a component installed using `npm`, and two local components.

```
{
  "loopback-component-explorer": {
    "mountPath": "/explorer"
  },
  "./components/my-component.js": {
    "path": "/my-component"
  },
  "./components/new-component": "myApp"
}
```

As seen in the above example, the configuration properties of components depend on the individual components. Refer to the component's documentation for details.

config.json

Overview

Define application server-side settings in `/server/config.json`. For example here are the default settings:

config.json

```
{
  "restApiRoot": "/api",
  "host": "0.0.0.0",
  "port": 3000,
  "remoting": {
    ... // See below
  },
  "legacyExplorer": false
}
```

Top-level properties

The following table describes the properties you can configure.

Property	Description	Default

aclErrorStatus	When an authenticated user is denied access because of an ACL, by default the application returns HTTP error status code **401 unauthorized**. If you want instead to return 403 (forbidden) set the value here. This may be required, for example, when using an AngularJS interceptor to differentiate between the need to show an access denied/request access page instead of a login dialog. Can also set this in the Model definition JSON file to define per-model.	401
host	Host or IP address used to create the Node HTTP server. If a request comes to a different host or IP address, then the application won't accept the connection. See server.listen() for more information.	0.0.0.0
legacyExplorer	Set to `false` to disable old routes `/models` and `/routes` that are exposed, but no longer used by API Explorer; use `true` or omit the option to keep them enabled. When upgrading to `v2.14.0`, set `"legacyExplorer": false`	false
port	TCP port to use.	3000
remoting	See Remoting properties below.	N/A
restApiRoot	Root URI of REST API	`/api`

To access the settings in application code, use `app.get('`*`property`*`')`.

You can also retrieve Express `app` object properties with this method. See `app.get()` in Express documentation for more information.

Remoting properties

Properties under the `remoting` top-level property determine how the application performs remote operations using strong-remoting; for example:

```
...
  "remoting": {
    "context": {
      "enableHttpContext": false
    },
    "rest": {
      "normalizeHttpPath": false,
      "xml": false
    },
    "json": {
      "strict": false,
      "limit": "100kb"
    },
    "urlencoded": {
      "extended": true,
      "limit": "100kb"
    },
    "cors": false,
    "errorHandler": {
      "disableStackTrace": false
    }
  },
...
```

The following table describes the remoting properties.

The full names of the properties below are prefixed by "remoting," for example, `remoting.json.limit`.

Property	Type	Description	Default
cors	Boolean	If false, use the CORS settings in middleware.json.	false

context.enableHttpContext	Boolean	Advanced feature. For more information, see Using current context.	false
errorHandler.disableStackTrace	Boolean	Set to true to disable stack traces; removes the `stack` property from the Error object. Ignored when NODE_ENV is "production", when stack traces are always disabled.	false
json.limit	String	Maximum request body size. You can set other JSON propertis as well; see body-parser.json().	100kb
json.strict	Boolean	Parse only objects and arrays. You can set other JSON propertis as well; see body-parser.json().	false
rest.handleErrors	Boolean	If true (the default), then the REST adapter handles all errors by sending back a JSON-formatted error response. If false, then errors are passed to the top-level application error-handler.	true
rest.handleUnknownPaths	Boolean	If true (the default), then the REST adapter emits a 404 error for unknown paths. The REST error handler or the application error handler then handle this error; rest.handleErrors above. If false, then the REST adapter delegates handling unknown paths to the top-level application by calling `next()`.	true
rest.normalizeHttpPath	Boolean	If `true`, in HTTP paths, converts: • Uppercase letters to lowercase. • Underscores (_) to dashes (-). • CamelCase to dash-delimited. Does not affect placeholders (for example ":id"). For example, "MyClass" or "My_class" becomes "my-class".	false
rest.supportedTypes	Array	**List of content types that the API supports in HTTP responses.** The response type will match that specfied in the HTTP request "accepts" header, if it is in this list of supported types. If this property is set, then `rest.xml` is ignored. NOTE: 'application/vnd.api+json' is supported, but is not one of the default types.	'application/json' 'application/javascript' 'application/xml' 'text/javascript' 'text/xml' 'json' 'xml'
rest.xml	Boolean	If true, then 'xml' is added to the supported content types. Then, the API will then respond with XML when the HTTP request "accepts" type contains 'xml'.	false
urlencoded.extended	Boolean	Parse extended syntax with the qs module. For more information, see bodyParser.urlencoded().	true
urlencoded.limit	String	Maximum request body size. For more information, see bodyParser.urlencoded().	100kb

Environment-specific settings

You can override values that are set in `config.json` in:

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

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

For example:

config.production.js

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

For more information, see Environment-specific configuration.

datasources.json

- Overview
- Standard properties
- Properties for database connectors
- Environment-specific configuration

Overview

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

For example:

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

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

Standard properties

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

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

Properties for database connectors

To connect a model to a data source, follow these steps:

1. Use the data source generator, `slc loopback:datasource`, to create a new data source. For example:

```
$ slc loopback:datasource
? Enter the data-source name: mysql-corp
? Select the connector for mysql: MySQL (supported by StrongLoop)
```

   Follow the prompts to name the datasource and select the connector to use. See Connecting models to data sources for more information. This adds the new data source to `datasources.json`.
2. Edit `server/datasources.json` to add the necessary authentication credentials: typically hostname, username, password, and database name. For example:

server/datasources.json

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

   For information on the specific properties that each connector supports, see:
   - Cloudant connector
   - DB2 connector
   - Memory connector
   - MongoDB connector
   - MySQL connector
   - Oracle connector
   - PostgreSQL connector
   - Redis connector
   - SQL Server connector

3. Install the corresponding connector as a dependency of your app with `npm`, for example:

```
$ cd <your-app>
$ npm install --save loopback-connector-mysql
```

   See Connectors for the list of connectors.
4. Use the model generator, `slc loopback:model`, to create a model. When prompted for the data source to attach to, select the one you just created.

> The model generator lists the memory connector, "no data source," and data sources listed in `datasources.json`. That's why you created the data source first in step 1.

```
$ slc loopback:model
? Enter the model name: myModel
? Select the data-source to attach myModel to: mysql (mysql)
? Select model's base class: PersistedModel
? Expose myModel via the REST API? Yes
? Custom plural form (used to build REST URL):
Let's add some test2 properties now.
...
```

You can also create models from an existing database; see Creating models for more information.

Environment-specific configuration

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

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

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

Example data sources:

datasources.json

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

datasources.production.json

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

middleware.json

- Overview
- Phases
- CORS settings

Overview

Set up middleware in `middleware.json`. Here is the default version created by the Application generator:

```
{
  "initial:before": {
    "loopback#favicon": {}
  },
  "initial": {
    "compression": {},
    "cors": {
      "params": {
        "origin": true,
        "credentials": true,
        "maxAge": 86400
      }
    }
  },
  "session": {},
  "auth": {},
  "parse": {},
  "routes": {
    "loopback#rest": {
      "paths": [
        "${restApiRoot}"
      ]
    }
  },
  "files": {
    "loopback#static": {
      "params": "$!../client"
    }
  },
  "final": {
    "loopback#urlNotFound": {}
  },
  "final:after": {
    "loopback#errorHandler": {}
  }
}
```

Phases

Each top-level property in `middleware.json` corresponds to one of the following middleware phases:

1. **`initial`** - The first point at which middleware can run.
2. **`session`** - Prepare the session object.
3. **`auth`** - Handle authentication and authorization.
4. **`parse`** - Parse the request body.
5. **`routes`** - HTTP routes implementing your application logic. Middleware registered via the Express API `app.use`, `app.route`, `app.get` (and other HTTP verbs) runs at the beginning of this phase. Use this phase also for sub-apps like `loopback/server/middleware/rest` or `loopback-explorer`.
6. **`files`** - Serve static assets (requests hit the file system here).
7. **`final`** - Deal with errors and requests for unknown URLs.

Each phase has "before" and "after" subphases in addition to the main phase, encoded following the phase name, separated by a colon. For example, for the "initial" phase, middleware executes in this order:

1. `initial:before`
2. `initial`
3. `initial:after`

Middleware within a single subphase executes in the order in which it is registered. However, you should not rely on such order. Always explicitly order the middleware using appropriate phases when order matters.

In general, each phase has the following syntax:

```
phase[:sub-phase] : {
 middlewarePath : {
 [ enabled: [true | false] ]
 [, name: nameString ]
 [, params : paramSpec ]
 [, methods: methodSpec ]
 [ paths : routeSpec ]
 }
}
```

Where:

- *phase* is one of the predefined phases listed above (initial, session, auth, and so on) or a custom phase; see Adding a custom phase.
- *sub-phase* (optional) can be `before` or `after`.
- name: optional middleware name.
- *middlewarePath*: path to the middleware function.
- *paramSpec*: value of the middleware parameters, typically a JSON object.
- *methodSpec*: An array containing HTTP methods for which the middleware is triggered; for example: `"methods" : ["GET", "POST"]`. If not present, applies to all methods.
- *routeSpec*: REST endpoint(s) that trigger the middleware.

For more information, see Defining middleware.

CORS settings

Set Cross-origin resource sharing (CORS) settings as `cors.params` properties in the initial phase.

You can set other CORS properties as well. For more information, see cors.

Property	Type	Description	Default
cors.params.origin	Boolean	Configures the **Access-Control-Allow-Origin** CORS header. Expects a string (ex: "http://example.com/"). Set to `true` to reflect the request origin, as defined by `req.header('Origin')`. Set to `false` to disable CORS. Can also be set to a function, which takes the request origin as the first parameter and a callback (which expects the signature `err [object], allow [bool]`) as the second.	true
cors.params.credentials	Boolean	Configures the **Access-Control-Allow-Credentials** CORS header. Set to `true` to pass the header, otherwise it is omitted. You can set other cors properties as well. For more information, see cors.	true
cors.params.maxAge	Number	Configures the **Access-Control-Allow-Max-Age** CORS header. Set to an integer to pass the header, otherwise it is omitted.	86400

model-config.json

- Overview
- Top-level properties
    - Model properties

Overview

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

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

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

model-config.json

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

Top-level properties

Property	Type	Description
_meta.sources	Array	Array of relative paths to custom model definitions. By default, LoopBack applications load models from `/common/models` subdirectory. To specify a different location (or even multiple locations) use the `_meta.sources` property, whose value is an array of directory paths.
_meta.mixins	Array	Array of relative paths to custom mixin definitions. See Defining mixins for more information.
modelName	String	Name of a model, either a built-in model or a custom model defined in the `common/models/` folder.

Model properties

Each JSON key is the name of a model and an object with the following properties.

Property	Type	Description

datasource	String	Name of the data source to which the model is connected. Must correspond to a data source defined in datasources.json.
public	Boolean	Whether the model API is exposed. If true, then the model is exposed over REST. Does not affect accessibility of Node API.

policy-config.json

Overview

Use `policy-config.json` to configure StrongLoop API Gateway policies, pipelines, and mappings. The Application generator does not create this file: you have to add it manually.
This file contains three top-level keys:

- **mappings**: an array of objects, each of which defines a mapping between a pipeline and a REST endpoint (a URL path and an HTTP method).
- pipelines: an array of objects, each containing a name property and a `policyIds` property that is an array of policies.
- **policies**: an array of objects, each of which defines a policy.

```
{
  "mappings": [ ... ],
  "pipelines": [ ... ],
  "policies": [ ... ]
}
```

Mappings

The `mappings` key contains an array of objects, each of which defines a mapping between a pipeline and a REST endpoint (a URL path and an HTTP method). For example:

Example excerpt from policy-config.json showing mappings

```
"mappings": [
    {
      "name": "all-apis",
      "verb": "ALL",
      "endpoint": "^/api/(.*)$",
      "pipelineId": "default-api-pipeline"
    },
    {
      "name": "local-protected-resource",
      "verb": "ALL",
      "endpoint": "^/protected|me|_internal/(.*)$",
      "pipelineId": "local-protected-resource-pipeline"
    },
    ...
  ]
```

Property	Description	Allowed values
name	Name of the mapping	Any string.

verb	HTTP method of the REST endpoint (or "ALL")	GET, POST, PUT, DELETE, or ALL (if the mapping applies to all HTTP methods).
endpoint	Regular expression defining the endpoint URI.	
pipelineId	Name of the pipeline to which the mapping applies.	Must match the name property of one of the objects in the `pipelines` array.

Pipelines

The `pipelines` key contains an array of objects, each containing a `name` property and a `policyIds` property that is an array of policies. For example:

```
"pipelines": [
    {
      "name": "default-api-pipeline",
      "policyIds": [
        "default-metrics-policy",
        "default-auth-policy",
        "default-rate-limiting-policy",
        "default-api-proxy-policy"
      ]
    },
    {
      "name": "note-api-pipeline",
      "policyIds": [
        "default-metrics-policy",
        "auth-note-policy",
        "auth-demo-policy",
        "default-rate-limiting-policy",
        "default-api-proxy-policy"
      ]
    },
    ...
]
```

Property	Description	Allowed values
name	Name of the pipeline	Any string.
policyIds	Array of policy names (strings)	Each policy name much match the `name` property of a policy defined in the `policies` key.

Policies

The `policies` key contains an array of objects, each of which defines a policy. For example:

```
"policies": [
    {
      "name": "default-metrics-policy",
      "type": "metrics",
      "phase": "initial:before"
    },
    {
      "name": "default-auth-policy",
      "type": "auth",
      "phase": "auth",
      "scopes": [
      ]
    },
    {
      "name": "default-api-proxy-policy",
      "type": "reverseProxy",
      "phase": "proxies",
      "targetURL": "http://localhost:3002/api/$1"
    },
    {
      "name": "default-rate-limiting-policy",
      "type": "rateLimiting",
      "phase": "routes:after",
      "limit": 100,
      "interval": 60000,
      "keys": {
        "app": {
          "template": "app-${app.id}",
          "limit": 1000
        },
        "ip": 500,
        "url": {
          "template": "url-${urlPaths[0]}/${urlPaths[1]}",
          "limit": 1000
        },
        "user": {
          "template": "user-${user.id}",
          "limit": 1000
        },
        "app,user": {
          "template": "app-${app.id}-user-${user.id}",
          "limit": 1000
        }
      }
    },
    ...
]
```

All policies have `name`, `type`, and `phase` properties. Additional properties are applicable only to certain types of policies.

Property	Description	Allowed values
name	Name of the policy	Any string.

type	Type of policy	One of: • auth • metrics • rateLimiting • reverseProxy
phase	The middleware phase where the policy is applied.	One of: • initial • session • auth • parse • routes • proxies (this is a custom phase) • files • final Append ":before" or ":after" to specify that the policy applies before or after the specified phase.
scopes	For an auth policy, scopes that enables access to protected resources, such as user data.	Array of strings.
targetURL	For a reverseProxy policy...	Regular expression (string)

Rate limiting policy

The following table summarizes additional properties for a rate limiting policy. For more information, see Configuring policies.

Property	Description	Allowed values
limit	Maximum number of requests the policy allows.	Positive integer
interval	Period of time (in ms) over which the limit applies.	Positive integer (represents milliseconds)
keys.ip	Identifies a specific IP address	
keys.app.template	Regular expression that identifies the app making a request.	
keys.app.limit	Maximum number of requests the policy allows from the apps identified by the correspo nding template property.	Positive integer
keys.url.template	Regular expression that identifies the URL making a request.	
keys.url.limit	Maximum number of requests the policy allows from the URLs specified by the corresp onding template property.	
keys.user.template	Regular expression that identifies a user; for example: `"user-${user.id}”`	
keys.user.limit	Maximum number of requests the policy allows from the users specified by the corresponding template property.	

ReverseProxy policy

For proxy, to use the "proxies" middleware phase, you must add the following to middleware.json after `routes:after` phase and before the `fil es` phase. For more information, see Configuring policies.

```
...
  "proxies": {
    "./middleware/proxy": []
  },
 "files": {
...
```

server.js

This is the main application script in the standard scaffolded application, as created by `slc loopback`.

1 - 3: Require LoopBack modules and set up standard objects `loopback`, `app`, and `boot`.

4: Start the web server.

7: Emit the 'started' event.

10 - 13: Start API Explorer.

18: Initialize (boot) the application.

```
var loopback = require('loopback');
var boot = require('loopback-boot');
var app = module.exports = loopback();
app.start = function() {
  // start the web server
  return app.listen(function() {
    app.emit('started');
    var baseUrl = app.get('url').replace(/\/$/, '');
    console.log('Web server listening at: %s', baseUrl);
    if (app.get('loopback-component-explorer')) {
      var explorerPath =
app.get('loopback-component-explorer').mountPath;
      console.log('Browse your REST API at %s%s',
baseUrl, explorerPath);
    }
  });
};
// Bootstrap the application, configure models,
datasources and middleware.
// Sub-apps like REST API are mounted via boot scripts.
boot(app, __dirname, function(err) {
  if (err) throw err;
  // start the server if `$ node server.js`
  if (require.main === module)
    app.start();
});
```

common directory

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

> If you want to be able to share models between client and server parts of your app, put your model JSON and JavaScript files in the `/common/models` directory.
>
> If you want to have separate client and server models, then put your model JSON and JavaScript files in the `/server/models` directory.

Model definition JSON file

- Overview
- Top-level properties
- Options

Related articles

- Creating models
- Customizing models
- Creating model relations
- Querying data
- Model definition JSON file
- PersistedModel REST API

Overview

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

> The LoopBack model generator, `slc loopback:model`, automatically converts camel-case model names (for example MyModel) to lowercase dashed names (my-model). For example, if you create a model named "FooBar" with the model generator, it creates files `foo-bar.json` and `foo-bar.js` in `common/models`. However, the model name ("FooBar") will be preserved via the model's name property.

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

customer.json

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

Top-level properties

Properties are required unless otherwise designated.

Property	Type	Description
name	String	Name of the model.

<table>
<tr><td>description</td><td>String or Array</td><td>Optional description of the model.

You can split long descriptions into arrays of strings (lines) to keep line lengths manageable.

<pre>[
"Lorem ipsum dolor sit amet, consectetur adipiscing elit",
"sed do eiusmod tempor incididunt ut labore et dolore",
"magna aliqua."
]</pre></td></tr>
<tr><td>plural</td><td>String</td><td>Plural form of the model name.

Optional: Defaults to plural of name property using standard English conventions.</td></tr>
<tr><td>base</td><td>String</td><td>Name of another model that this model extends. The model will "inherit" properties and methods of the base model.</td></tr>
<tr><td>idInjection</td><td>Boolean</td><td>Whether to automatically add an id property to the model:
<code>true</code>: <code>id</code> property is added to the model automatically. This is the default.
<code>false</code>: <code>id</code> property is not added to the model
See ID properties for more information.

Optional; default is <code>true</code>. If present, the <code>idInjection</code> propery in <code>options</code> takes precedence.</td></tr>
<tr><td>http.path</td><td>String</td><td>Customized HTTP path for REST endpoints of this model.</td></tr>
<tr><td>strict</td><td>Boolean</td><td>Specifies whether the model accepts only predefined properties or not. One of:
<code>true</code>: Only properties defined in the model are accepted. Use if you want to ensure the model accepts only predefined properties.
<code>false</code>: The model is an open model and accepts all properties, including ones not predefined in the model. This mode is useful to store free-form JSON data to a schema-less database such as MongoDB.
"validate": The unknown properties will be reported as validation errors.
"throw": Throws an exception if properties not defined for the model are used in an operation.
Undefined: Defaults to false unless the data source is backed by a relational database such as Oracle or MySQL.</td></tr>
<tr><td>options</td><td>Object</td><td>JSON object that specifies model options. See Options below.</td></tr>
<tr><td>properties</td><td>Object</td><td>JSON object that specifies the properties in the model. See Properties below.</td></tr>
<tr><td>relations</td><td>Object</td><td>Object containing relation names and relation definitions.

See Relations below.</td></tr>
<tr><td>acls</td><td>Array</td><td>Set of <code>ACL</code> specifications that describes access control for the model.

See ACLs below.</td></tr>
<tr><td>scopes</td><td>Object</td><td>See Scopes below.</td></tr>
</table>

Options

The `options` key specifies advanced options, for example data source-specific options.

> You can set `idInjection` here in `options` or at the top-level. The value set here takes precedence over the top-level value of `idInjection`.

Advanced options

Property	Type	Description

`validateUpsert`	`Boolean`	By default, the `upsert()` method (also known as `updateOrCreate()`) does not enforce valid model data. Instead, it logs validation errors to the console. This is not optimal, but necessary to preserve backwards compatibility with older 2.x versions. Set this property to `true` to ensure that `upsert()` returns an error when validation fails. The next major version of LoopBack will enable this option (set the property to true) by default. Set this property to `false` to prevent `upsert()` from calling any validators at all. By default, `upsert()` calls all validators and reports any validation errors to the console log.

Data source-specific options

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

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

Properties

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

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

General property properties

Each model property can have the properties described in the following table. Only the `type` property is required; for properties with only a `type`, you can use the following shorthand:

"*propertyName*": "*type*"

For example:

```
...
"emailVerified": "boolean",
"status": "string",
...
```

Key	Required?	Type	Description
default	No	Any*	Default value for the property. The type must match that specified by `type`.
defaultFn	No	String	A name of the function to call to set the default value for the property. Must be one of: • `"guid"` or `"uuid"`: generate a new globally unique identifier (universally unique identifier) using the computer MAC address and the current time (UUID version 1). • `"now"`: use the current date and time as returned by `new Date()` NOTE: For discussion of providing additional options, see LoopBack issue 292 on GitHub.

description	No	String or Array	Documentation for the property.
			You can split long descriptions into arrays of strings (lines) to keep line lengths manageable.
			`[`
			`"Lorem ipsum dolor sit amet, consectetur adipiscing elit",`
			`"sed do eiusmod tempor incididunt ut labore et dolore",`
			`"magna aliqua."`
			`]`

doc	No	String	Documentation for the property. **Deprecated, use "description" instead.**
id	No	Boolean	Whether the property is a unique identifier. Default is false. See Id property below.
required	No	Boolean	Whether a value for the property is required. Default is false.
type	Yes	String	Property type. Can be any type described in LoopBack types.
*	No	Any	See below.

ID properties

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

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

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

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

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

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

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

For example,

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

Then:

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

Composite IDs

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

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

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

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

Data mapping properties

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

Property	Type	Description
columnName	String	Column name
dataType	String	Data type as defined in the database
dataLength	Number	Data length
dataPrecision	Number	Numeric data precision
dataScale	Number	Numeric data scale
nullable	Boolean	If true, data can be null

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

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

Exclude properties from base model

By default, a model inherits all properties from the base. To exclude some base properties from being visible, you need to set the property to `false` or `null`. For example,

common/models/customer.json

```
...
"base": "User",
"properties": {
  "lastUpdated": false,
  "credentials": null,
  "challenges": null,
  "modified": "date"
}
...
```

Hidden properties

A hidden property is not sent in the JSON data in the application's HTTP response. The property value is an array of strings, and each string in the array must match a property name defined for the model.

An example of a hidden property is User.password:

common/models/user.json

```
...
  "properties": {
    ...
    "password": {
      "type": "string",
      "required": true
    },
...
   "hidden": ["password"],
...
```

If you want to white-list the fields returned instead of black-listing them, consider:

- Applying the `fields` of the model's default `scope`. This will operate at the database response layer so limiting your ability to check a

field in the database that you otherwise would not wish exposed to the outside world (a private flag, for example).
- Overriding your model's `toJSON` method

See discussion of white-listing on GitHub.

Protected properties

A protected property is not sent in the JSON data in the application's HTTP response if the object is nested inside another object. For instance if you have an Author object and a Book object. A book has a relation to with Author, and book is public API. Author will have personal information such as social security number etc, and they can now be "protected" such that anyone looking up the author of the book will not get those information back (from GitHub pull request). The property value is an array of strings, and each string in the array must match a property name defined for the model.

An example of a hidden property is User.email:

common/models/user.json

```
...
  "properties": {
    ...
    "email": {
      "type": "string",
      "required": true
    },
...
   "protected": ["email"],
...
```

Validations

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

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

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

For example:

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

Relations

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

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

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

Example of hasManyThrough:

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

ACLs

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

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

Key	Type	Description
accessType	String	The type of access to apply. One of: • READ • WRITE • EXECUTE • * (default)
permission	String	Type of permission granted. Required. One of: • **ALLOW** - Explicitly grants access to the resource. • **DENY** - Explicitly denies access to the resource.

principalId	String	Principal identifier. Required.
		The value must be one of:
		• A user ID (String|number|any)
		• One of the following predefined dynamic roles:
		  ◦ `$everyone` - Everyone
		  ◦ `$owner` - Owner of the object
		  ◦ `$related` - Any user with a relationship to the object
		  ◦ `$authenticated` - Authenticated user
		  ◦ `$unauthenticated` - Unauthenticated user
		• A static role name
		⚠ $related principalId is not yet implemented.

principalType	String	Type of the principal. Required. One of: • Application • User • Role
property	String Array of Strings	Specifies a property/method/relation on a given model. It further constrains where the ACL applies. Can be: • A string, for example: `"create"` • An array of strings, for example: `["create", "update"]`

Scopes

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

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

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

The snippet above defines two named queries for the model:

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

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

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

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

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

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

Default scope

If you wish for a scope to be applied across all queries to the model, you can use the default scope for the model itself.

For example:

```
{
  "name": "Product",
  "properties": {
    ...
  }
  "scope": {
    "order": "name",
    "limit": 100
    "where": {
      "deleted": false
    }
  }
}
```

Now, any CRUD operation with a query parameter runs in the default scope will be applied; for example, assuming the above scope, a find opearation such as

```
Product.find({offset: 0}, cb);
```

Becomes the equivalent of this:

```
Product.find({order: "name", offset: 0, limit: 100, where: {deleted: false}}, cb)
```

Methods

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

This feature is not yet implemented.

Indexes

Declare indexes for a model with the `indexes` property, for example:

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

The snippet above creates two indexes for the declaring model:

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

The full syntax for an individual index is:

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

A key value of 1 specifies ascending order, and -1 specifies descending order.

If you don't need to specify any options, you can use a shortened form:

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

You can specify indexes at the model property level too, for example:

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

This example creates two indexes: one for the `name` key and another one for the `email` key. The `email` index is unique.

client directory

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

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

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

Basic model object

Overview

By default, the basic LoopBack Model object has properties and methods "mixed in" from:

- Inclusion object - Enables you to load relations of several objects and optimize numbers of requests.
- Validateable object - provides validation methods; see Validating model data.

When you define relations between models, the RelationMixin object object also gets mixed in to the model object.

Events

> The following events are deprecated in favor of operation hooks:
>
> - changed
> - deleted
> - deletedAll

The following table summarizes the events that LoopBack models can emit.

Event	Emitted when...	Arguments	Argument type	Class methods that emit	Instance methods that emit
'attached'	Model is attached to an app.	Model class	Object	app.model(*modelName*)	
'dataSourceAttached'	Model is attached to a Data source.	Model class	Object		• DataSource.prototype.createModel • DataSource.prototype.define
'set'	Model property is set.	Model instance	Object		Model.prototype.setAttributes()

Connected model object

In addition to the methods of the Basic model object, the following are mixed in when a model is connected to a data source:

- RelationMixin class
- PersistedModel class

LoopBack middleware

- loopback.context()
- loopback.favicon()
- loopback.rest()
- loopback.static()
- loopback.status()
- loopback.token()
- loopback.urlNotFound()

See also: Defining middleware

LoopBack types

- Overview
- Array types
  - Array of objects
- Object types

Overview

Various LoopBack methods accept type descriptions, for example remote methods and dataSource.createModel(). The following is a list of supported types.

Type	Description	Example
null	JSON null	`null`
Boolean	JSON Boolean	`true`
Number	JSON number	`3.1415`
String	JSON string	`"StrongLoop"`
Object	JSON object or any type See Object types below.	`{ "firstName": "John", "lastName": "Smith", "age": 25 }`
Array	JSON array See Array types below.	`[ "one", 2, true ]`
Date	JavaScript Date object	`new Date("December 17, 2003 03:24:00");`
Buffer	Node.js Buffer object	`new Buffer(42);`
GeoPoint	LoopBack GeoPoint object	`new GeoPoint({lat: 10.32424, lng: 5.84978});`

In general, a property will have `` `undefined` `` value if no explicit or default value is provided.

The type name is case-insensitive; so for example you can use either "Number" or "number."

Array types

LoopBack supports array types as follows:

- `{emails: [String]}`
- `{"emails": ["String"]}`
- `{"emails": [{"type": "String", "length": 64}]}`
- `{"emails": "array"}` (a shorthand notation for `{"emails": ["any"]}`)

Array of objects

Define an array of objects as follows (for example):

```
...
"Address": {
  "type": [
    "object"
  ],
  "required": true
}
...
```

Object types

Use the Object type when you need to be able to accept values of different types, for example a string or an array.

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`. LoopBack 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 an "anonymous" model.

If you intend to reuse the address model, 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'`.

Valid names in LoopBack

In general, names of names of LoopBack applications, models, data sources, and so on must at a minimum be valid JavaScript identifiers, with the additional restriction that the dollar sign ($) is not allowed.

The rules for JavaScript identifiers are fairly broad (for example, you can use Unicode characters). However, as a best practice, follow these more restrictive guidelines:

- Begin names with a letter or underscore (_). Don't begin a name with a number.
- Other characters can be letters, numbers, or underscore (_).
- Application names can contain the dash, that is, the minus-sign character (-).

In general, LoopBack allows other characters in names, including other non-alphanumeric characters and Unicode characters, but using them is not recommended as a best practice. If you have questions, inquire on the LoopBack Google Group.

It is also a good idea to avoid using JavaScript reserved words as names, since doing so may be problematic.

Also, using a predefined property name of a JavaScript Object, such as `constructor`, as a property name may cause issues.

Release notes

- Current open issues

For an exhaustive list of all updates to StrongLoop repos, see http://strongloop.github.io/changelog/.

LoopBack 2.23

LoopBack dependencies

LoopBack dependencies on other StrongLoop modules, from package.json:

```
"dependencies": {
"loopback-connector-remote": "^1.0.3",
"loopback-phase": "^1.2.0",
"strong-remoting": "^2.21.0",
…
}
```

```
"devDependencies": {
"loopback-boot": "^2.7.0",
"loopback-datasource-juggler": "^2.19.1",
"strong-task-emitter": "^0.0.6",
…
}
```

StrongLoop dependencies

StrongLoop dependencies on other StrongLoop modules, from package.json:

```
"dependencies": {
"generator-loopback": "1.x",
"loopback-sdk-angular-cli": "1.x",
"strong-agent": "^2.0.0",
"strong-arc": "^1.2.0",
"strong-build": "^2.0.0",
"strong-deploy": "^3.x",
"strong-mesh-models": "^8.0.0",
"strong-pm": "^5.0.0",
"strong-registry": "^1.0.0",
"strong-start": "^1.1.0",
"strong-supervisor": "^3.0.0",
...
}
```

LoopBack 2.22 release notes

loopback core

All methods of the Application and User models now support promise-based invocation as well as callback-based calls. Replication-related methods like `PersistedModel.replicate` support promises too. We've also got an initial start on documentation for promises in LoopBack, with much more planned as promise support is expanded.

As of PR#1561, you can specify a method-based filter for middleware. For example, the following limits the specified middleware to GET requests only:

server/middleware.json

```
{
  "auth": {
    "my-middleware": {
       "methods": ["GET"],
       "params": ["arg1", "arg2"],
       "enabled": true
    }
  }
}
```

PR#1502 implemented a wrapper around express' middleware error handler. The wrapper adds a configuration option to exclude error stack traces from HTTP responses. Here is the `middleware.json` from a standard scaffolded LoopBack app:

server/middleware.json

```
{
  "final:after": {
    "loopback#errorHandler": {}
  }
}
```

And here is a typical configuration that disables stack traces when running in production. To make your app use this, set NODE_ENV to "production".

server/middleware.production.json

```
{
  "final:after": {
    "loopback#errorHandler": {
      "params": {
        "includeStack": false
      }
    }
  }
}
```

loopback-boot

PR#149 and PR#151 added a post-processing step of `middleware.json` and `component-config.json` where all string values in the format `${var}` are resolved to the value provided by `app.get(var)`.

loopback-datasource-juggler

PR#697 and PR#710 added support for querying embedded models and array values à la MongoDB when using the memory connector.

Let’s illustrate the new querying capabilities on the following example model:

common/models/person.json

```
{
  "properties": {
    "name": "string",
    "children": ["string"],
    "friends": [{ "name": "string" }]
  }
}
```

You can use the following query in order to find all persons who have a child called “Jane”:

```
Person.find({ where: { children: "Jane" } })
```

Similarly, to find all persons who have a friend called “John”:

```
Person.find({ where: { "friends.name": "John" } })
```

PR#665 implemented support for RegExp operator in queries. For example, using the model from above, the following query finds all persons with a name starting with the letter J:

```
Person.find({ where: { regexp: /^J/ } })
```

In addition to the memory connector, the following connectors support the `RegExp`operator:

- MongoDB
- MySQL
- PostgreSQL
- MS SQL

For more information, see the where filter documentation.

Last but not least, PR#647 adds support for promises to all model-discovery methods.

loopback-component-explorer

We renamed loopback-explorer to loopback-component-explorer, since it is now a full-fledged LoopBack component. And we released version 2.0.0 of the module with two breaking changes.

First, as a LoopBack component, you can load and configure it via`server/component-config.json`:

```
// server/component-config.json
{
  "loopback-component-explorer": {
    "mountPath": "/explorer"
  }
}
```

Second, to simplify migration from the 1.x version range, we provide middleware you can mount in any Express application in the same way as before:

```
var explorer = require('loopback-component-explorer');

// v1.x - does not work anymore
app.use('/explorer', explorer(app, options));
// v2.x
app.use('/explorer', explorer.routes(app, options));
```

Loading components with `component-config.json` is a new feature, and we're in the process of adding documentation for it.

Connectors

loopback-connector-mongodb

@6332dcd and @fdb3448 implemented the flag `allowExtendedOperators`. When enabled, data for update operations can include MongoDB operators like `$rename`. There are two ways to enable this flag:

In your model definition file (for example `common/models/my-model.json`), set the property `settings.mongodb.allowExtendedOperators` to true; for example:

```
{
  "name": "MyModel",
  "settings": {
    "mongodb": {
      "allowExtendedOperators": true
```

When calling the `updateAll()` method from code, set the flag in the options object.

```
User.updateAll(
  { name: 'Al' },
  { '$rename': { name: 'firstname' }},
  { allowExtendedOperators: true });
```

loopback-connector-postgresql

The connector can discover and create indexes now. See @eddc5a2 for more details.

loopback-connector-mssql

PR#52 added support for unicode string values in SQL queries.

LoopBack 2.19 release notes

Core loopback module

New enableAuth() method

We changed `app.enableAuth` to automatically setup all required models not attached to the app nor a datasource. The purpose of this change is to make it easier to set up authentication from code that does not use slc loopback project scaffolding, such as unit tests.

To use this new option, just provide the name of the data source to use for these models. For example:

```
var app = loopback();
app.dataSource('db', { connector: 'memory' });
app.enableAuth({ dataSource: 'db' });

app.use(loopback.rest());
app.listen(3000);
```

Observer API

We made several improvements to the hook API. Including the ability to skip other observers by calling `context.end()`. Now you can also notify multiple observers by passing an array of operation names (for example, `notifyObserversOf(['op1', 'op2'])`).

loopback-boot

- Released 2.8.0
- PR#130 Port can't be number checked to support iisnode
- @44f733f Support iisnode using named pipes as PORT value (Jonathan Sheely)
- Released 2.8.1
- PR#133 Better debug output when loading complex configurations

loopback-datasource-juggler

Persist hook

We added a new "persist" hook. Observers are notified during operations that persist data to the datasource (for example, create, updateAttributes). Don't confuse this hook with the existing "before save" hook:

- **before save** – Use this hook to observe (and operate on) model instances that are about to be saved (for example, when the country code is set and the country name not, fill in the country name).
- **persist** – Use this hook to observe (and operate on) data just before it is going to be persisted into a data source (for example, encrypt the values in the database).

Loaded hook

We have also just added a new "loaded" hook. Observers are notified right after raw data is loaded or returned from the connector and datasource. This allows you to do things like decrypt database values before they are used to create a model instance.

Other changes

Here is a filtered changelog including important changes to the juggler.

- PR#611 De-dupe ids args of inq for include

- Released 2.29.1
- PR#584 add test suite for scope – dynamic function
- PR#588 Fix pagination on collections with many-to-many relationships
- PR#604 Fix destroyById not removing instance from cache
- PR#609 Don't silently swallow db errors on validation
- Released 2.29.0
- PR#602 Enhance the apis and add more tests
    - End observer notification early
    - Allow multiple notifications in a single call: notifyObserversOf(['event1, 'event2])
- PR#600 Fix toJSON() for level 3 inclusions
- PR#598 Mixin observer apis to the connector
- PR#597 Enhance fieldsToArray to consider strict mode
- Released 2.28.1
- @cbb8d7c Remove dep on sinon
- PR#586 Add new hook persist
- Released 2.30.1
- @8302b24 Pin async to version ~1.0.0 to work around context propagation
- Released 2.30.0
- PR#618 Allow 0 as the FK for relationships
- PR#626 Fix for issues #622 & #623
- PR#630 Promisify 'automigrate'

strong-remoting

We added a couple of new features to strong-remoting and remote methods in LoopBack.

Default status codes

Now you can define both a status and errorStatus in the HTTP options of your remote method.

Header and status argument targets

To set a header or the status code for an HTTP response, now you can specify the target of your callback argument to either header or status.

Here is an example of both default status codes and argument targets.

```
// define a default status code
Person.remoteMethod(
  'greet',
  {
    accepts: {arg: 'msg', type: 'string'},
    returns: {arg: 'greeting', type: 'string'},
    http: {
      status: 201,
      errorStatus: 504
    }
  }
);

// define a callback arg that will set the status or a header
Person.remoteMethod(
  'greet',
  {
    accepts: {arg: 'msg', type: 'string'},
    returns: [
      // this callback arg will set the status
      {arg: 'status', type: 'number', http: { target: 'status' }}
      // this callback arg will set a header "X-My-Header"
      {arg: 'myHeader', type: 'string', http: { target: 'header', header:
'X-My-Header' }}
    ]
  }
);
```

loopback-filter

We extracted the implementation of data filtering from the memory connector to a new module called loopback-filters. Using this module, you can now filter arrays of objects using the same filter syntax supported by MyModel.find(filter). We'll soon be converting all LoopBack modules to use loopback-filter, so it will become the common "built-in" filtering mechanism.

LoopBack supports a specific filter syntax: it's a lot like SQL, but designed specifically to serialize safely without injection and to be native to JavaScript. Previously, only the PersistedModel.find() method (and related methods) supported this syntax.

Here is a basic example using the new module.

```
var data = [{n: 1}, {n: 2}, {n: 3, id: 123}];
var filter = {where: {n: {gt: 1}}, skip: 1, fields: ['n']};
var filtered = require('loopback-filters')(data, filter);
console.log(filtered); // => [{n: 3}]
```

For a bit more detail, let's say you are parsing a comma-separated value (CSV) file, and you need to output all values where the price column is between 10 and 100. To use the LoopBack filter syntax you would need to either create your own CSV connector or use the memory connector, both of which require some extra work not related to your actual goal.

Once you've parsed the CSV (with some module like node-csv) you will have an array of objects like this, for example (but with, say, 10,000 unique items):

```
[
{price: 85, id: 79},
{price: 10, id: 380},
...
]
```

To filter the rows you could use generic JavaScript like this:

```
data.filter(function(item) {
return item.price < 100 && item.price >= 10
});
```

This is pretty simple for filtering, but sorting, field selection, and more advanced operations become a bit tricky. On top of that, you are usually accepting the parameters as input; for example:

```
var userInput = {min: 10, max: 100}

data.filter(function(item) {
return item.price < userInput.min && item.price >= userInput.max
});
```

You can rewrite this easily as a LoopBack filter:

```
filter(data, {where: {input: {gt: userInput.min, lt: userInput.max}}})
```

Or if you just adopt the filter object syntax as user input:

```
filter(data, userInput)
```

But `loopback-filters` supports more than just excluding and including. It supports field selection (including / excluding fields), sorting, geo/distance sorting, limiting and skipping. All in a declarative syntax that is easily created from user input.

As a LoopBack user this is a pretty powerful thing. Typically, you will have learned how to write some complex queries using the `find()` filter syntax; before you would need to figure out how to do the same thing in JavaScript (perhaps using a library such as underscore). Now with the `loopback-filters` module, in your client application you can re-use the same exact filter object you were sending to the server to filter the database without having to interact with a LoopBack server at all.

generator-loopback

The new LoopBack middleware generator adds a middleware configuration to an existing application.

```
$ slc loopback:middleware [name]
```

The tool will prompt you to:

- Select the phase to use for the middleware.
- Add a list of paths.
- Add parameters.

See more about the new middleware generator in the documentation.

loopback-workspace

The loopback-workspace module provides the backend functionality for Arc API composer and `slc loopback`.

The default template for generating LoopBack projects now includes default lookup paths for mixins. The defaults are the following:

- loopback/common/mixins
- loopback/server/mixins – the first two will include loopback core mixins.
- ../common/mixins relative to your project.
- ./mixins relative to your project.

Note that the last two paths are relative to your project (just like `modelSources`).

We also added support for middleware.json. This lays the groundwork for support for generating middleware.json files with Arc API Composer and with slc loopback.

Connectors

loopback-connector

We added a new “execute” hook to the connector API. It allows you to observe the low-level `connector.execute()` method. See the documentation for more information.

- Released 2.2.0
- @2fc9258 Update deps
- PR#18 Add before/after hooks for connector native operations
- Released 2.1.2
- @a5f11ac Put request with non existent properties no longer results in error
- Released 2.1.1
- @a62e06d Improved query support for Date

loopback-connector-soap

- @b40f92b Add before and after “execute” hooks for the underlying soap invocation
- PR#20 bump(soap) bump node-soap version from 0.8 to 0.9

loopback-connector-rest

- Released 1.9.0
- PR#33 Add before and after “execute” hooks

loopback-connector-mongodb

- @84a1ab0 Add before and after “execute” hooks
- Released 1.9.2
- @8a55f92 Update to memwatch-next for node 0.12 compatibility
- Released 1.9.1

- @5d907cf Update deps
- Released 1.9.0
- @331e158 Replaced ensureIndex() with createIndex()
- Released 1.11.0
- PR#142 Add a workaround for auth with multiple mongos servers
- PR#141 Autoupdate and automigrate now respect settings.mongodb.collection

loopback-connector-postgresql

- @111f8c2 Add better support for the Date type
- Released 2.2.0
- PR#88 Make sure UTC is used for date

loopback-component-oauth2

- Released 2.2.0
- @83635b0 Tidy up the models to work with MySQL
- Released 2.1.1
- @a32e213 Allow models to be customized via options
- Released 2.1.0
- @32fcab9 Clean up oAuth2 client app attributes
- Released 2.3.0
- @807762d Remove auth code after 1st use
- @f613a20 Allow options.scopes to be a custom function
- Released 2.2.1
- @79a7df8 Allow options.userModel/applicationModel to be strings
- Released 2.0.0
- @e5da21e Change license to StrongLoop

Other changes

- loopback-component-storage
  - PR#74 Bugfix: Cannot read property ‘forEach’ of undefined
  - Released 1.5.0
  - PR#70 Add missing finish event when uploading to s3
- loopback-testing
  - PR#47 Add withUserModel to extend user related helpers
  - PR#51 use findorCreate to create roles
  - PR#45 Update helpers.js
- loopback-component-passport
  - Released 1.4.0
  - PR#70 feature: Make email optional
- loopback-gateway
  - This module is now deprecated and superseded by StrongLoop API Gateway.
- loopback-sdk-angular
  - Released 1.4.0
  - PR#138 Add createMany method
- loopback-component-push
  - PR#88 Pass contentAvailable through to APNS
  - @e9022a0 Forward “contentAvailable” and “urlArgs” to APNS
  - @08e73ee Update deps

LoopBack 2.0 release notes

> The current version of loopback is .
>
> The latest version is available on npm.
>
> For a list of the latest commits, see the change log.

- LoopBack generators
- Express 4.x
- New project structure
  - Support for external configuration
- Bootstrapper was moved to its own module
- Email connector uses nodemailer 1.x
- The method loopback.getModel throws for unknown model names
- Remote methods
- New loopback-component modules

See also:

- What's New in LoopBack: July-Aug 2015
- What’s New in the LoopBack Node.js Framework – May 2015
- Current open issues

LoopBack generators

For LoopBack 2.0, the Command-line tool, `slc loopback`, now uses Yeoman generators to create and scaffold applications. The `slc lb` and `slc example` commands are no longer available.

Express 4.x

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

New project structure

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

Support for external configuration

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

New object PersistedModel

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

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

Bootstrapper was moved to its own module

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

Email connector uses nodemailer 1.x

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

The method loopback.getModel throws for unknown model names

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

Remote methods

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

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

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

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

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

New loopback-component modules

Several modules used by LoopBack have been renamed.

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

Migrating apps to version 2.0

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

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

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

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

Runtime

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

Update package.json

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

package.json

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

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

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

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

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

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

Use Express 4.x

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

Middleware

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

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

An example error message

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

Routing

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

app.js (loopback 1.x)

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

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

app.js (loopback 2.x)

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

routes/index.js (loopback 2.x)

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

// define your custom routes on the router object

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

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

Model definitions

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

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

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

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

   model.json (database-backed model)

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

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

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

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

Remote methods

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

- LoopBack 2.0 release notes
- Remote methods

Not exposing methods over REST

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

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

In version 2.0, you do, for example:

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

For more information, see Exposing models over a REST API

Troubleshooting

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

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

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

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

Project layout

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

package.json

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

App settings

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

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

Data sources

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

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

Models

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

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

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

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

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

   models/car.json

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

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

   model-config.json

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

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

models/car.js

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

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

models.json

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

Accessing the application object and other models

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

common/models/my-model.js

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

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

common/models/my-model.js

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

Attaching built-in models

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

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

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

Boot scripts

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

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

server/boot/authentication.js

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

LoopBack Definition Language (LDL)

Use LoopBack Definition Language (LDL) to define LoopBack data models in JSON or JavaScript.

Use `slc loopback:model` command to crete a JSON definition of a model. For more information, see Model generator.

The simplest form of a property definition in JSON has a `propertyName: type` element for each property. The key is the name of the property and the value is the type of the property. For example:

```
{
    "id": "number",
    "firstName": "string",
    "lastName": "string"
}
```

This example defines a `user` model with three properties:

- id - The user id, a number.
- firstName - The first name, a string.
- lastName - The last name, a string.

Each key in the JSON object defines a property in the model that is cast to its associated type. See LoopBack types for more information.

You can also describe the same model in JavaScript code:

```
 var UserDefinition = {
    id: Number,
    firstName: String,
    lastName: String
}
```

The JavaScript version is less verbose, since it doesn't require quotes for property names. The types are described using JavaScript constructors, for example, `Number` for `"Number"`. String literals are also supported.

To use the model in code is easy, because LoopBack builds a JavaScript constructor (or class) for you.

Error object

By convention, LoopBack passes an `Error` object to callback functions as the `err` parameter.

For more information, see

- JavaScript Error object (Mozilla)
- Error Handling in Node.js (Joyent)
- What is the error object? (Nodejitsu)

The following table describes the properties of the error object.

Property	Type	Description
name	String	Name of the error.
status	String	When the error occurs during an HTTP request, the HTTP status code.
message	String	The error message.

> Any other properties of the error object are copied to the error output.