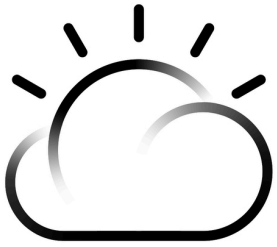
LoopBack

Quick Lab: REST APIs in minutes with LoopBack 4

Diana Lau, IBM LoopBack Development, dhmlau@ca.ibm.com
Biniam Admikew, IBM LoopBack Development, biniam@ca.ibm.com



IBM Cloud

REST APIs in minutes with LoopBack 4

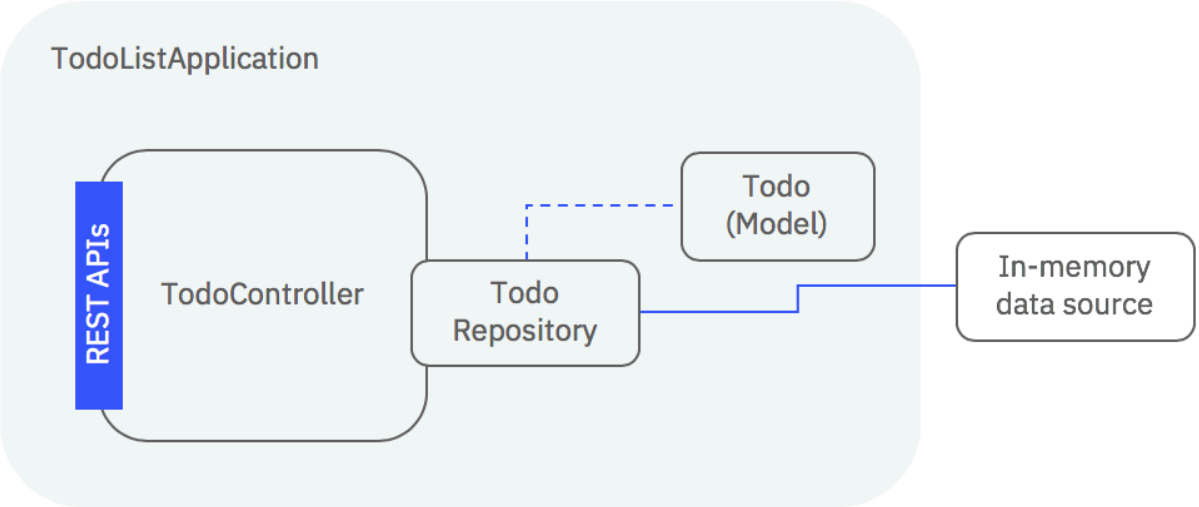
1. Introduction

LoopBack is a Node.js framework that allows users to create REST APIs quickly while interacting with the backend resources, such as databases and web services. In the latest version of LoopBack (LoopBack 4), we have rewritten the framework with:

- Updated technology stacks, e.g. TypeScript, ES2016/2017
- Better extensibility
- Better alignment with cloud native experience for microservices

This lab walks you through the steps required to create, build and run a simple LoopBack 4 application with model relations. There is a bonus section for exposing GraphQL APIs from the same LoopBack application.

For this lab, we are going to build a ToDo application that keeps track of to-do items using an in-memory database. For more tutorials and documentation, please refer to our web site: http://v4.loopback.io.



This lab walks you through the steps required to scaffold a LoopBack 4 application and create various LoopBack artifacts.

2. Setting Up

Before starting this lab, there are a few things to setup:

1. **Node.js**
   Install Node.js (version 8.9 or higher): https://nodejs.org/en/download/

2. **LoopBack 4 CLI**
   The LoopBack 4 CLI is a command-line interface that scaffolds a project or an extension by generating the basic code. The CLI provides the fastest way to get started with a LoopBack 4 project that adheres to best practices.

   Install the CLI globally by running

```
npm i -g @loopback/cli
```

3. Create Application Scaffolding

In this section, you will use the LoopBack Command-line Interface toolkit (CLI) to scaffold your application.

The LoopBack 4 CLI toolkit comes with templates that generate whole applications, as well as artifacts (for example, controllers, models, and repositories) for existing applications.

To generate your application using the toolkit, run the `lb4 app` command and fill out the on-screen prompts:

```
$ lb4 app
? Project name: todo-list
? Project description: A todo list API made with LoopBack 4.
? Project root directory: (todo-list)
? Application class name: (TodoListApplication)
? Select features to enable in the project:
❯◉ Enable tslint
 ◉ Enable prettier
 ◉ Enable mocha
 ◉ Enable loopbackBuild
 ◉ Enable vscode
 ◉ Enable repositories
 ◉ Enable services
# npm will install dependencies now
Application todo-list was created in todo-list.
```

For this tutorial, when prompted with the options for enabling certain project features (loopback's build, tslint, mocha, etc.), leave them all enabled.

Structure

After your application is generated, you will have a folder structure similar to the following:

```
src/
 controllers/
   README.md
   ping.controller.ts
```

```
  datasources/
    README.md
  models/
    README.md
  repositories/
    README.md
  application.ts
  index.ts
  sequence.ts
test/
  README.md
  mocha.opts
  acceptance/
    ping.controller.acceptance.ts
node_modules/
  ***
LICENSE
README.md
index.js
index.ts
package.json
tsconfig.json
tslint.build.json
tslint.json
```

4. Create the ToDo Model

In this section, you will use the LoopBack CLI to create a ToDo model.
Now we can begin working on the representation of our data for use with LoopBack 4. To that end, we're going to create a Todo model that can represent instances of a task for our Todo list. The Todo model will serve both as a Data Transfer Object (also known as a DTO) for representing incoming Todo instances on requests, as well as our data structure for use with loopback-datasource-juggler.

A model describes business domain objects and defines a list of properties with name, type, and other constraints.

A todo list is all about tracking tasks. For this to be useful, it will need to let you label tasks so that you can distinguish between them, add extra information to describe those tasks, and finally, provide a way of tracking whether or not they're complete.

For our Todo model to represent our Todo instances for our business domain, it will need:

- a unique id
- a title
- a description that details what the todo is all about
- a boolean flag for whether or not we've completed the task

We can use the `lb4 model` command and answer the prompts to generate the model for us. Note that this command should be run within the newly created 'todo-list' directory. Press `return` with an empty property name to generate the model. Follow these steps:

```
lb4 model
? Model class name: todo

Let's add a property to Todo
Enter an empty property name when done

? Enter the property name: id
? Property type: number
? Is id the ID property? Yes
? Is it required?: No
? Default value [leave blank for none]:
```

```
Let's add another property to Todo
Enter an empty property name when done

? Enter the property name: title
? Property type: string
? Is it required?: Yes
? Default value [leave blank for none]:

Let's add another property to Todo
Enter an empty property name when done

? Enter the property name: desc
? Property type: string
? Is it required?: No
? Default value [leave blank for none]:

Let's add another property to Todo
Enter an empty property name when done

? Enter the property name: isComplete
? Property type: boolean
? Is it required?: No
? Default value [leave blank for none]:

Let's add another property to Todo
Enter an empty property name when done

? Enter the property name:

  create src/models/test.model.ts
  update src/models/index.ts

Model todo was created in src/models/
```

And that's it! We've created the 'Todo' model and described it in TypeScript in 'src/models/todo.model.ts'. The update in 'index.ts' adds the todo.model.ts to the list of exported models.

5. Add a Datasource

Datasources are LoopBack's way of connecting to various sources of data, such as databases, APIs, message queues and more. A `DataSource` in LoopBack 4 is a named configuration for a Connector instance that represents data in an external system. The Connector is used by `legacy-juggler-bridge` to power LoopBack 4 Repositories for Data operations.

In LoopBack 4, datasources can be represented as strongly-typed objects and freely made available for injection throughout the application. Typically, in LoopBack 4, datasources are used in conjunction with Repositories to provide access to data.

For more information about datasources in LoopBack, see DataSources.

Since our Todo API will need to persist instances of Todo items, we'll need to create a datasource definition to make this possible.

From inside the project folder, we'll run the `lb4 datasource` command to create a DataSource. For the purposes of this tutorial, we'll be using the memory connector provided with the Juggler.

```
lb4 datasource
? Datasource name: db
? Select the connector for db: In-memory db (supported by StrongLoop)
? window.localStorage key to use for persistence (browser only):
? Full path to file for persistence (server only): ./data/db.json

  create src/datasources/db.datasource.json
  create src/datasources/db.datasource.ts
  update src/datasources/index.ts

Datasource db was created in src/datasources/
```

Since we have specified the file `./data/db.json` to persist the data for the in-memory connector, let's create the directory and file and add in some values.

```
| => mkdir data
_____________________   | ~/todo-list @ biniams-mbp (badmike)
| => touch data/db.json
```

data/db.json


```
{
 "ids": {
  "Todo": 5
 },
 "models": {
  "Todo": {
   "1": "{\"title\":\"Take over the galaxy\",\"desc\":\"MWAHAHAHAHAHAHAHAHAHAHAHAHAMWAHAHAHAHAHAHAHAHAHAHAHA\",\"id\":1}",
   "2": "{\"title\":\"destroy alderaan\",\"desc\":\"Make sure there are no survivors left!\",\"id\":2}",
   "3": "{\"title\":\"terrorize senate\",\"desc\":\"Tell them they're getting a budget cut.\",\"id\":3}",
   "4": "{\"title\":\"crush rebel scum\",\"desc\":\"Every.Last.One.\",\"id\":4}"
  }
 }
}
```


Once you're ready, we'll move onto adding a repository for the datasource.

6. Add a Repository

The repository pattern is one of the more fundamental differences between LoopBack 3 and 4. In LoopBack 3, you would use the model class definitions themselves to perform CRUD operations. In LoopBack 4, the layer responsible for this has been separated from the definition of the model itself, into the repository layer.

A Repository represents a specialized Service interface that provides strong-typed data access (for example, CRUD) operations of a domain model against the underlying database or service.

For more information about Repositories, see Repositories.

From inside the project folder, we'll run the lb4 repository command to create a repository for our todo model using `db` datasource from the previous step which will show up by its class name`DbDataSource` from the list of available datasources.

```
| => lb4 repository
? Please select the datasource DbDatasource
? Select the model(s) you want to generate a repository Todo
  create src/repositories/todo.repository.ts
  update src/repositories/index.ts

Repository Todo was created in src/repositories/
```

The `src/repositories/index.ts` file makes exporting artifacts central and also easier to import as shown below:

```
// in src/models/index.ts
export * from './foo.model';
export * from './bar.model';
export * from './baz.model';

// elsewhere...

// with index.ts
import {Foo, Bar, Baz} from './models';
// ...and without index.ts
import {Foo} from './models/foo.model';
import {Bar} from './models/bar.model';
import {Baz} from './models/baz.model';
// Using an index.ts in your artifact folders really helps keep
// things tidy and succinct!
```

The newly created `todo.repository.ts` class has the necessary connections that are needed to perform CRUD operations for our Todo model. It leverages the Todo model definition and 'db' datasource configuration and retrieves the datasource using Dependency Injection. Next, we'll need to build a controller to handle our incoming requests.

6. Add a Controller

In LoopBack 4, controllers handle the request-response lifecycle for your API. Each function on a controller can be addressed individually to handle an incoming request (like a POST request to `/todos`), to perform business logic and to return a response.

`Controller` is a class that implements operations defined by an application's API. It implements an application's business logic and acts as a bridge between the HTTP/REST API and domain/database models.

In this respect, controllers are the regions *where most of your business logic will live*!

For more information about Controllers, see Controllers.

So, let's create a controller to handle our Todo routes. You can create an empty Controller using the CLI as follows:

```
lb4 controller
? Controller class name: todo
? What kind of controller would you like to generate? Empty Controller
```

In addition to creating the handler functions themselves, we'll also be adding decorators that set up the routing and the expected parameters of incoming requests.

First, we need to define our basic controller class as well as connect our repository, which is needed to perform operations against the datasource.

src/controllers/todo.controller.ts

```
import {repository} from '@loopback/repository';
import {TodoRepository} from '../repositories';

export class TodoController {
  constructor(@repository(TodoRepository) protected todoRepo: TodoRepository) {}
}
```

Now that the repository is connected, let's create our first controller function.

src/controllers/todo.controller.ts

```
import {repository} from '@loopback/repository';
import {TodoRepository} from '../repositories';
import {Todo} from '../models';
```

```
import {HttpErrors, post, param, requestBody} from '@loopback/rest';

export class TodoController {
 constructor(@repository(TodoRepository) protected todoRepo: TodoRepository) {}

 @post('/todos', {
  responses: {
   '200': {
    description: 'Todo model instance',
    content: {'application/json': {'x-ts-type': Todo}},
   },
  },
 })
 async createTodo(@requestBody() todo: Todo) {
  if (!todo.title) {
   throw new HttpErrors.BadRequest('title is required');
  }
  return await this.todoRepo.create(todo);
 }
}
```

In this example, we're using two new decorators to provide LoopBack with metadata about the route, verb and the format of the incoming request body:

- `@post('/todos')` creates metadata for `@loopback/rest` so that it can redirect requests to this function when the path and verb match.
- `@requestBody()` associates the OpenAPI schema for a Todo with the body of the request so that LoopBack can validate the format of an incoming request.
- We've also added our own validation logic to ensure that a user will receive an error if they fail to provide a `title` property with their `POST` request.

Finally, we are using the functions provided by our `TodoRepository` instance to perform a create operation against the datasource.

You can use these and other decorators to create a REST API for a full set of verbs:

src/controllers/todo.controller.ts

```
import {repository} from '@loopback/repository';
import {TodoRepository} from '../repositories';
import {Todo} from '../models';
import {
 HttpErrors,
 post,
 param,
 requestBody,
 get,
 put,
```

```
  patch,
  del,
} from '@loopback/rest';

export class TodoController {
  constructor(@repository(TodoRepository) protected todoRepo: TodoRepository) {}

  @post('/todos', {
    responses: {
      '200': {
        description: 'Todo model instance',
        content: {'application/json': {'x-ts-type': Todo}},
      },
    },
  })
  async createTodo(@requestBody() todo: Todo) {
    if (!todo.title) {
      throw new HttpErrors.BadRequest('title is required');
    }
    return await this.todoRepo.create(todo);
  }

  @get('/todos/{id}', {
    responses: {
      '200': {
        description: 'Todo model instance',
        content: {'application/json': {'x-ts-type': Todo}},
      },
    },
  }
)
  async findTodoById(@param.path.number('id') id: number): Promise<Todo> {
    return await this.todoRepo.findById(id);
  }

  @get('/todos', {
    responses: {
      '200': {
        description: 'Array of Todo model instances',
        content: {
          'application/json': {
            schema: {type: 'array', items: {'x-ts-type': Todo}},
          },
        },
      },
    },
  })
  async findTodos(): Promise<Todo[]> {
    return await this.todoRepo.find();
```

```
  }

  @put('/todos/{id}', {
    responses: {
      '204': {
        description: 'Todo PUT success',
      },
    },
  })
  async replaceTodo(
    @param.path.number('id') id: number,
    @requestBody() todo: Todo,
  ): Promise<void> {
    return await this.todoRepo.replaceById(id, todo);
  }

  @patch('/todos/{id}', {
    responses: {
      '204': {
        description: 'Todo PATCH success',
      },
    },
  })
  async updateTodo(
    @param.path.number('id') id: number,
    @requestBody() todo: Todo,
  ): Promise<void> {
    return await this.todoRepo.updateById(id, todo);
  }

  @del('/todos/{id}', {
    responses: {
      '204': {
        description: 'Todo DELETE success',
      },
    },
  })
  async deleteTodo(@param.path.number('id') id: number): Promise<void> {
    return await this.todoRepo.deleteById(id);
  }
}
```

Some additional things to note about this example:

- Routes like `@get('/todos/{id}')` can be paired with the `@param.path` decorators to inject those values at request time into the handler function.
- LoopBack's `@param` decorator also contains a namespace full of other "sub-decorators" like `@param.path`, `@param.query`, and `@param.header` that allow specification of metadata for those parts of a REST request.
- LoopBack's `@param.path` and `@param.query` also provide subdecorators for specifying the type of certain value primitives, such as `@param.path.number('id')`.

Now that we've connected the controller, our last step is to tie it all into the Application!

7. Putting it all together

We've got all of our artifacts now, and all that's left is to bind them to our Application so that LoopBack's Dependency injection system can tie it all together for us!

LoopBack's boot module will automatically discover our controllers, repositories, data-sources and other artifacts and inject them into our application.

src/application.ts

```
import {BootMixin} from '@loopback/boot';
import {ApplicationConfig} from '@loopback/core';
import {RepositoryMixin} from '@loopback/repository';
import {RestApplication, RestServer} from '@loopback/rest';
import {MySequence} from './sequence';

export class TodoListApplication extends BootMixin(
  RepositoryMixin(RestApplication),
) {
  constructor(options?: ApplicationConfig) {
    options = Object.assign({}, options, {
      rest: {
        openApiSpec: {
          servers: [{url: 'http://localhost:3000'}],
        },
      },
    });
    super(options);

    // Set up the custom sequence
    this.sequence(MySequence);

    this.projectRoot = __dirname;
    // Customize @loopback/boot Booter Conventions here
    this.bootOptions = {
      controllers: {
        // Customize ControllerBooter Conventions here
        dirs: ['controllers'],
        extensions: ['.controller.js'],
        nested: true,
      },
    };
  }

}
```

Try it out

Let's try out our application! First, you'll want to start the app.

```
$ npm start
Server is running on port 3000
```

Next, you can use the API Explorer to browse your API and make requests!

Here are some requests you can try:

- `POST /todos` with a body of `{ "title": "buy milk" }`
- `GET /todos/{id}` using the ID you received from your `POST`, and see if you get your Todo object back.
- `PATCH /todos/{id}` with a body of {"title": "buy milk", "desc": "need milk for cereal"}
- `GET /todos/{id}` using the ID from the previous steps to see that your changes are persisted

That's it! You've just created your first LoopBack 4 application. But it's not over yet. Let's explore the concepts of relatios in LoopBack 4 in the following sections!

8. Exposing GraphQL APIs in LoopBack

The OASGraph module is a new TypeScript module that creates a GraphQL wrapper for existing REST APIs which are described by the OpenAPI specification. With the collaboration of IBM Research and LoopBack teams, it is available at https://github.com/strongloop/oasgraph.

OASGraph requires additional node modules, so we'll need to install these modules in our application by running the following command:

```
| => npm i --save oasgraph && npm i –save-dev express-graphql
```

Next, we'll need to have our LoopBack Application running. Make sure the LoopBack application is running. If not, run `npm start`.

From the browser, go to: http://localhost:3000/openapi.json, copy the openapi.json file, and save it under the root of the LoopBack application. Name the file as todo-openapi.json. Alternatively, you can use cURL to download it:

```
| => curl localhost:3000/openapi.json -o todo-openapi.json
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 13012  100 13012    0     0  1236k      0 --:--:-- --:--:-- --:--:-- 1270k
```

Next, we will use the `oasgraph` CLI to set up a GraphQL HTTP Server backed by express on port '3001'. Start up the server by running the following command:

```
node_modules/.bin/oasgraph todo-openapi.json
```

That's it! We are now ready to try out some tests and requests in the browser at http://localhost:3001/graphql. Here are some new requests you can try out with the expected values:

Get all Todo instances:

```
query{
  todos {
    id
    title
    desc
  }
}
```

Expected output:

```
{
  "data": {
    "todos": [
      {
        "id": 1,
        "title": "Take over the galaxy",
        "desc": "MWAHAHAHAHAHAHAHAHAHAHAHAMWAHAHAHAHAHAHAHAHAHAHAHA"
      },
      {
        "id": 2,
        "title": "destroy alderaan",
        "desc": "Make sure there are no survivors left!"
      },
      {
        "id": 3,
        "title": "terrorize senate",
        "desc": "Tell them they're getting a budget cut."
      },
      {
        "id": 4,
        "title": "crush rebel scum",
        "desc": "Every.Last.One."
      }
    ]
  }
}
```

Create a Todo instance and retrieve its id and title in the response object

```
mutation {
  postTodos(todoInput: {
    title: "Take over the universe"
  }) {
    id
    title
```

```
  }
}
```

Expected output:

```
{
  "data": {
    "postTodos": {
      "id": 5,
      "title": "Take over the universe"
    }
  }
}
```

Retrieve the Todo instance created in the step above:

```
query{
  todo(id: 5) {
    title
  }
}
```

Expected output:

```
{
  "data": {
    "todo": {
      "title": "Take over the universe"
    }
  }
}
```

9. Conclusion

Congratulations! You have completed the lab.

You have successfully built and run a LoopBack Microservice that is cloud-ready and can be deployed to IBM Cloud and IBM Cloud Private! Plus, in the bonus stage, you were able to expose LoopBack's REST APIs in GraphQL using the new OASGraph module and Express!

10. Resources

1. LoopBack 4 website, http://v4.loopback.io/
2. LoopBack 4 GitHub repo, https://github.com/strongloop/loopback-next
3. LoopBack blog, https://strongloop.com/strongblog/tag_LoopBack.html
4. OpenAPI specification, https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md
5. CloudNative JS, https://www.cloudnativejs.io/
6. Node@IBM Developer Center: http://developer.ibm.com/cloud/node

Acknowledgements and Disclaimers (v8)