connect-controller

Introduced in new curricular plan for Internet Programming in 2017, regarding an MVC (model view controller) approach for Node JS express based web applications.

connect-controller allows you to create **Plain Controller Objects** with NO express boilerplate code. **Plain Controller Objects** do not require any additional configuration, nor annotations, nor a specific base class, nor `req`, `res` or `next` arguments, etc. The connect-controller suppresses all the express web server verbosity from a web controller, such as: `router.get(...)`; paths specification e.g. `/path/subPath/:routeParam`; arguments lookup on `res.params`; rendering views `res.render(<viewPath>, <context>)`; specifying views paths; etc.
For instance, given a domain service `footballDb` with a promises based API, **compare** the two approaches of building a `football` router with, and without `connect-controller`, in listing 1 and listing 2 respectively. Both cases build and bind a single route to the path `/leagues/:id/table` which uses the `getLeaguesIdTable(id)` method of `footballDb`. (**NOTE** that connect-controller is also able to parse methods conforming to the node.js callback convention)

Listing 1 - Build and bind an express route with connect-controller (example/lib/controllers/footasync.js):

```
const connectCtr = require('connect-controller')
const getLeaguesIdTable = footballDb.getLeaguesIdTable
const controller = { getLeaguesIdTable }
app.use('footasync', connectCtr(controller))
```

Listing 2 - Build and bind an express route (example/lib/routes/football.js):

```
const router = express.Router()
router.get('/leagues/:id/table', (req, res, next) => {
    const id = req.params.id
    footballDb
      .getLeaguesIdTable(id)
      .then(league => {
        res.render('football/leagues/table', league)
      })
      .catch(err => next(err))
})
app.use('football', router)
```

Note that in former example, the `connect-controller` overwhelms all verbosity:

1. NO need of `router.get(...)`. Methods bind to http GET, by default. For different verbs just prefix `<verb>_`to method's name.
2. NO path definition `/leagues/:id/table`. Router paths are mapped to methods names.

3. NO need of `req`, `res`, `next` arguments.
4. NO arguments lookup, such as `req.params.id`. Just add `id` as a method parameter.
5. NO explicit renderization. `res.render(...)` is implicit.
6. NO view path specification. By default, the view path is `/controllerName/actionName`.
7. NO error handler.

The connect-controller builds a connect/express Middleware from a Plain Controller Object. By default, every controller method (*Action*) is mapped to a route with the path `/controllerName/actionName`, following the server-side controller conventions (according to the Controller definition of Rails)
Put it simply, for each action method:

- the `connect-controller` searches for a matching argument in `req.params`;
- to bind action parameters to *route parameters* you just need to include the parameters names in the method's name interleaved by _ or different case;
- if you want to handle an HTTP method (i.e. verb) different from GET you just need to prefix the name of the method with `verb_`;
- the `resp.render(viewPath, context)` is just a continuation appended to an action method, where:
    - the `context` is just the method result, or the content of the returned `Promise`,
    - the `viewPath` corresponds to `controllerName/actionName`, which is located inside the `views` folder by default.
- to take a different response other than `res.render(data)` you just need to add the `res` parameter to the action method and do whatever you want. In this case the connect-controller gets out of the way and delegates to the action method the responsibility of sending the response.

There are additional keywords that can be used to parametrize *Actions* following additional conventions, such as:

- prefix HTTP method, e.g. `get_<action name>`, `post_<action name>`, etc;
- `req`, `res` and `next` are reserved parameters names (optional in action arguments list) binding to Middleware `req`, `res` and `next`.
- whenever an action receives the `res` parameter, the connect-controller gets out of the way and delegates on that action the responsibility of sending the response.
- `index` reserved method name, which maps to a route corresponding to the Controller's name

Finally you may configure the `connect-controller` behavior with additional parameters passed in an optional `Object` to the default function

(e.g. `connectCtr('./controllers', { redirectOnStringResult: true })`).
This `Object` can be parameterized with the following properties:

- `name` - the name of controller when it is loaded as a single controller instance (default: `''`).
- `redirectOnStringResult` - set this property to `true` when an action method returns a string as the path to redirect (default: `false`).
- `resultHandler` - `(res, ctx) => void` function that will handle the result of the action methods, instead of the default `res.render(...)` behavior.

Installation

```
$ npm install connnect-controller
```

Usage

Given for example a controller `football.js` located in application root `/controllers` folder you may add all `football.js` actions as routes of an express app just taking the following steps. In this example we are adding 4 routers: one to render views (the default behavior of `connect-controller`) for `football.js` (callback based) and other router for`footasync.js` (promise based) and two more routers to serialize the context objects to json. The latter routes with prefix `/api`. Note that we are using exactly the same controller module to build all router objects. The only difference is in the options object which includes a `resultHandler` for the latter.

```
const express = require('express')
const connectCtr = require('connect-controller')
const app = express()
app.use(connectCtr(
  './controllers', // contains football.js and footasync.js
  { redirectOnStringResult: true }
))
app.use('/api', connectCtr(
  './controllers',
  { resultHandler: (res, ctx) => res.json(ctx) }
))
/**
 * Alternatives:
 * app.use(connectCtr())                                          // loads
all controllers located in controllers folder
 * app.use(connectCtr(require('./controllers/footasync.js')))  // loads
a single controller object
 * app.use(connectCtr(                                           // loads
a single controller object with name soccer
 *   require('./controllers/footasync.js'),
 *   { name: 'soccer'}
 * ))
```

```
 */
```
In this case `footasync.js` could be for example:
```
const footballDb = require('./../db/footballDb')

/**
 * connect-controller supports action methods names in both conventions
 * underscores and lower camel case.
 * In this sample we are using underscores, but it will work too if you
replace
 * underscores by different case.
 */
module.exports = {
    leagues_id_table, // binds to /footasync/leagues/:id/table
    leagues,          // binds to /footasync/leagues
    index,            // binds to /footasync/
    index_id          // binds to /footasync/:id
}

/**
 * Every action parameter (e.g. id) taking part of method's name (e.g.
_id_)
 * is bound to the corresponding argument of req.params (e.g.
req.params.id).
 * In this case this function is useless and we could simply bound
 * property 'leagues_id_table' to method footballDb.leagueTable.
 */
function leagues_id_table(id){
    return footballDb.getLeaguesIdTable(id)
}

/**
 * Every action parameter (e.g. name) that is NOT part of the method's
name
 * will be searched on req.query, req.body, req, res.locals and
req.app.locals.
 */
function leagues(name) {
    return footballDb
        .getLeagues()
        .then(leagues => leagues
            .filter(l => !name || l.caption.indexOf(name) >= 0)
}

/**
 * Whenever an action receives the `res` parameter, the connect-
controller
 * gets out of the way and delegates on that action the responsibility
of
 * sending the response.
 * So whenever you want to do something different from the default
behavior
 * you just have to append res to your parameters.
 */
function index(res) {
    /**
     * Once this controller is loaded with an options object set with
```

```
     * the property `redirectOnStringResult` then this is equivalent
     * to removing the `res` parameter and just return the destination
     * string path '/footasync/leagues'.
     */
    res.redirect('/footasync/leagues')
}

/**
 * If this controller is loaded with an options object set with the
property
 * `redirectOnStringResult` then this action method redirects to
 * `/footasync/leagues/:id/table`.
 */
function index_id(id) {
    return '/footasync/leagues/' + id + '/table'
}
```

Changelog

2.0.1 (May 18, 2017)

Add support for Plain Controller Objects with methods conforming to node.js callback convention. Remove the automatic binding of action arguments to the properties of `req`, `req.query`, `req.body`, `res.locals`, `app.locals`. Now you have to receive a `req` or `res` and look for desired properties.

1.3.0 (February 8, 2017)

- `connectCtr` function may me configured with an additional options `Object` with the following optional properties:
    - `name` - the name of controller when it is loading a single controller instance.
    - `redirectOnStringResult` - set this property to `true` when an action method returns a string as the path to redirect.
    - `resultHandler` - `(res, ctx) => void` function that will handle the result of the action methods, instead of the default `res.render(...)` behavior.

1.2.0 (January 13, 2017)

- Action methods (i.e. methods of a controller instance) can be defined in lowerCamelCase and not only with underscores. The `connect-controller` automatically bind action methods in one of those formats: lowerCamelCase or underscores.

License

MIT