Create a Node module

You can create local modules for use in your Node app to clean up your code and better organize it (recall how crowded the main ***index.js*** file got when we added MongoDB to the mix). Creating a module for Node follows the module design pattern we tried out previously.

Recall, when we created a module using Javascript, we needed to export the functions/properties we wanted available:

```
//file named: loggerModule.js
export function getMessage() {
  //getMessage code here…
}();
```

In the other script file, we would then need to add the following line to import:

```
//import from a file named loggerModule.js from within the same directory
import {getMessage} from './loggerModule.js';
```

Since we exported the one function/method we can then access that function/method from outside. In the import statement above the green text is the name of the method we want to import.

Also recall: we can also set a function or property to be exported by default. When doing that, we can import without the curly brackets. In that case, we can use an alias for the import since we aren't specifying a specific function or property (so we get the default export in that case).

Finally, we need to run this in a server environment. The above example is not a Node app, so you can run this using Apache or some other server. You will also need to modify the HTML file to add **`type="module"`** to the <script> tags for **both** files.

In Node, reusable code is written using exports as well (to use, use **`require`** instead of import).

Later, if you want, you can even publish your modules to npm. We'll focus on local site-specific modules first.

Create a local module

We'll create a routing module for all site paths to neaten up our code a little (at least move it out of the main file). If you recall, previously we did everything in *index.js*, but this is messy. We can break up our code into local modules for use in our app.

1. Using our previous example, we need to move all the MongoDB-related code and app routes into new files.
    a. Create a folder named ***modules/menuLinks*** and within it, create two files named ***func.js*** and ***router.js***.
2. Fill out ***func.js*** to hold the various functions using MongoDB (these were our async functions we listed at the bottom of *index.js* before).

```
const { MongoClient, ObjectId } = require("mongodb");

//Mongo stuff
```

```
const dbUrl = "mongodb://127.0.0.1:27017/";
const client = new MongoClient(dbUrl);

//MONGO FUNCTIONS
async function connection() {
  db = client.db("testdb"); //select testdb database
  return db;
}
/* Async function to retrieve all links documents from menuLinks collection.
*/
async function getLinks() {
  db = await connection(); //await result of connection() and store the
returned db
  var results = db.collection("menuLinks").find({}); //{} as the query means
no filter, so select all
  res = await results.toArray();
  return res;
}
/* Async function to insert one document into menuLinks. */
async function addLink(link) {
  db = await connection();
  let status = await db.collection("menuLinks").insertOne(link);
  console.log("link added");
}
/* Async function to delete one document by _id. */
async function deleteLink(id) {
  db = await connection();
  const deleteIdFilter = { _id: new ObjectId(id) };
  const result = await db.collection("menuLinks").deleteOne(deleteIdFilter);
  if (result.deletedCount === 1)
    console.log("delete successful");
}
/* Async function to select one document by _id. */
async function getSingleLink(id) {
  db = await connection();
  const editIdFilter = { _id: new ObjectId(id) };
  const result = db.collection("menuLinks").findOne(editIdFilter);
  return result;
}
/* Async function to edit one document. */
async function editLink(filter, link) {
  //fill this out
  //https://www.mongodb.com/docs/drivers/node/current/usage-
examples/updateOne/
}
```

3. Add an export statement to the bottom of ***func.js*** to export all of the functions which will be used elsewhere.

```
module.exports = {
  getLinks,
  addLink,
  deleteLink,
  getSingleLink,
  editLink
};
```

4. Now work on the ***router.js*** file.
    a. Make sure to require the Express and Express's Router modules.

```
var express = require("express");
var router = express.Router();
```

    b. Move the page routes and parsing settings (because these are needed when retrieving form values) from ***index.js*** into ***router.js*** and rename the **app** object to **`router`** (in ***router.js***). We need to do this because we've used router as our Express Router object variable name in step (b). We are also going to use the functions from ***func.js*** so we have to import it and call the functions by name (see the **`model`** and function names prepended with **`model`**). The code in ***router.js*** should now be:

```
const express = require("express");
const router = express.Router();
const model = require("./func");

//In order to parse POST body data as JSON, do the following.
//The following lines will convert the form data from query
//string format to JSON format.
router.use(express.urlencoded({ extended: true }));
router.use(express.json());

//PAGE ROUTES
router.get("/", async (request, response) => {
  links = await model.getLinks();
  response.render("index", { title: "Home", menu: links });
});
router.get("/about", async (request, response) => {
  links = await model.getLinks();
  response.render("about", { title: "About", menu: links });
});
router.get("/admin/menu", async (request, response) => {
```

```
  links = await model.getLinks();
  //view path has "admin/" because the "menu-list" view was placed in a
subdirectory (views/admin/)
  response.render("admin/menu-list", { title: "Menu links admin", menu:
links });

});
router.get("/admin/menu/add", async (request, response) => {
  links = await model.getLinks();
  response.render("admin/menu-add", { title: "Add link", menu: links });
});
router.get("/admin/menu/edit", async (request, response) => {
  if (request.query.linkId) {
    let id = request.query.linkId;
    let linkToEdit = await model.getSingleLink(id);
    links = await model.getLinks();

    response.render("admin/menu-edit", { title: "Edit link", editLink:
linkToEdit, menu: links });
  }
});

//FORM PROCESSING PATHS
router.post("/admin/menu/add/submit", async (request, response) => {
  //for a POST form, field values are passed in request.body.<field_name>
  //we can do this because of the setting to convert the urlencoded data to
JSON
  let newLink = {
    weight: request.body.wgt,
    path: request.body.path,
    name: request.body.text
  };
  await model.addLink(newLink);
  response.redirect("/admin/menu");
});
router.get("/admin/menu/delete", async (request, response) => {
  //for a GET form, field values are passed in request.query.<field_name>
because we're retrieving from a query string
  let id = request.query.linkId;
  await model.deleteLink(id);
  response.redirect("/admin/menu");
});
router.post("/admin/menu/edit/submit", async (request, response) => {
  //fill out this code
  //get the _id to use this as a filter
```

```
  let id = "<fill_out>";
  //get weight/path/name values and build this is your updated document
  let link = {
    weight: "<replace>",
    path: "<replace>",
    name: "<replace>"
  };
  //run editLink()
});
```

5. Now we need to make the code in ***router.js*** available as a module to ***index.js*** because we want to use it. Add the following line to the **end of *router.js***.

   ```
   module.exports = router;
   ```

6. To use our routes module in ***index.js*** we need to import it with a require statement. Add the following line with the rest of the require statements in ***index.js***:

   ```
   const router = require("./modules/menuLinks/router");
   ```

   Notice that the require statement is a path to the ***router.js*** file from the current directory **minus the *.js***!
7. Now to make sure the app uses the routes, add the following line in ***index.js***.

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

   This tells Express to use the **`router`** (variable name for our module) for routing all paths (since just a "/" was specified). Remember that our module contains all the app paths.

Models and controllers

Rather than just dumping the routes and code in one module file, you can further break things down into models and controllers.

Use models to hold the properties and methods for the different data types. For example, using the menu link example, you can create a model which holds the link properties and methods for adding, deleting and updating the link. This way there's better data encapsulation and less messy DB code in the main file.

When apps are complex, it's easier to maintain code by breaking down the code into particular features/functions (components) rather than having one ***models***, ***controllers*** folder.

MVC vs component-based patterns

In a smaller project, MVC is a popular pattern (i.e. splitting models, views and controllers), but for larger projects you may find it more popular to split by component (e.g. each component in a subfolder such as /src/my-component-name) with the code files in each component's folder. This is easier because when there are very many components, organizing by component makes it easier to find code related to that component.

Publishing a Node module to npm

If you're curious about creating and publishing your own modules to npm, take a look at the following resources:

- https://docs.npmjs.com/creating-and-publishing-unscoped-public-packages
- https://www.guru99.com/node-js-modules-create-publish.html
- https://initialcommit.com/blog/nodejs-module

Note that—of course—modules you want to publish would have more standalone function and would not be so entwined like our basic module example was. It was just done to show how you would use code modules within your Node app so that not everything is in your main file.