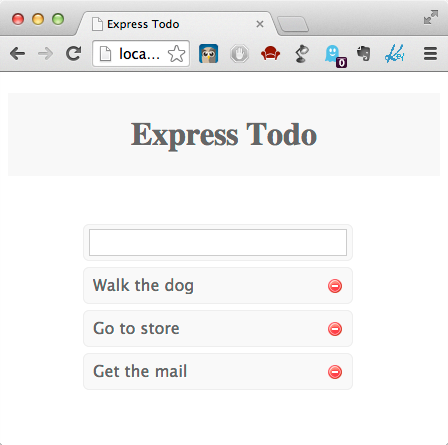
Express

In this lab, you will create a simple “todo” application using the Express web application framework.



# Objectives

In this lab, you will learn to

* define CRUD routes in Express, and
* implement behavior for each route.

# Set things up

You have been provided a working Express application to start the lab but first we need to install the required modules.

## Install dependencies

1. Run npm install in the lab directory.

This will install the dependencies for our application from the package.json file. In this case, our only dependencies are express, an MVC web framework, and jade, a popular template engine that will render our views. The views and basic CSS styling have been provided for you in the views and public directories.

While not required, you probably want to install nodemon (npm install –g nodemon). nodemon automatically restarts your application when a file system change is detected and will save time so you don’t have to manually restart Node to view your edits.

# Initialize the application

1. Open app.js in a text editor and take note of the fs.readFile invocation. Normally, a web application would use a database to store data, but working with a database is out of scope for this lab. Instead, we will be using a JSON file, todo\_db.json, to load sample data into memory. After todo\_db.json is parsed, we simply store its data in the app.locals.todos variable.

Express provides the variable app.locals to store app-level data that is available throughout the application.

## Specify the root route

1. Open routes/index.js in your editor and define a HTTP GET route for the root path (/) by using app.get.

The first argument to app.get is the URL path for the matching route, and the second argument is the callback function that will handle the request. In the spirit of a layered architecture, our controller layer concerning “todos” will be placed in module todo-controller.js located in the same directory.

The service layer will be placed in the services directory. You can see that as your application grows, this allows your routes/index.js to remain easy to read and keeps the controllers separate. Use todoController.findAll, which we'll write in the next step, as the second argument for your route.

## Create the controller function for the root route

1. Open routes/todo-controller.js and provide an implementation for the function findAll(req, res, next).

In your implementation, delegate to the service.findAll function, and pass to it a callback of the form function onFound(err, todos).

In the callback, use the res.render function to display the index.jade view that has already been created for you in the views directory.

As you can see, views/index.js expects a variable called todos to be passed containing all of the “todo” items, which is used to render edit & delete links for each todo in the browser. The value of todos will come from the service layer call.

Express is a model-view-controller framework. Controllers are functions (routes) that handle requests, delegate to services, produce models, and select views. Views render models. A model is simply a map of data whose entries are used by the view to get to the data. You can think of the model as the "data contract" between the controller and the view: the view expects certain data to be present in the view, and the controller must ensure that it provides that data.

## Implement findAll in the service layer

1. Open services/todo-service.js and provide an implementation for the function findAll(callback). In a normal application, this would be a database lookup, but in our case, we only need to return todos, which has already been passed into the service layer constructor. The implementation should simply be callback(null, todos).

## Start the application

1. Run node app.js (or nodemon app.js) and, if everything is working correctly, you should be able to navigate to <http://localhost:3000> and see three sample “todo” items already in your application!

You may notice that while our data renders correctly, the rest of the functionality to edit, create, and delete todos is broken because we need to define more routes.

## Create the ‘/create’ route & controller function

1. Open routes/index.js and define a HTTP POST route for path /create and corresponding function addTodo(req, res, next) in routes/todo-controller.js.

According to REST conventions, creation routes are typically defined with a HTTP POST, and our application is no different. The addTodo function in routes/todo-controller.js should call the service layer's addTodo function and pass req.body.content, along with a callback of the form function onAdded(err, todo). In the callback, we need to terminate the response, so add res.redirect(‘/’) to render the main index view after we've processed the business logic.

## Create the addTodo service function

1. Open services/todo-service.js and provide an implementation for the addTodo function.

If you examine the todo\_db.json structure, you will see that we are storing only an id and content:

[{

"id":"f47b6329-e3db-4af2-e2a1-6fd818ab08ac",

"content":"Walk the dog"},

…

}]

1. In services/todo-service.js, a function to generate unique IDs, called uuid(), has been provided for you. Create a new todo inline object with property id equal to uuid() and property content equal to the content parameter.

Use the JavaScript push() method to add a new entry to todos, the array that represents our backing store, simulating database persistence.

Finally, call the callback function with parameters null, indicating no error, and the newly created todo object.

At this point, you should be able to verify that new items are correctly added in your application. Run your application again via node (or nodemon) and confirm everything is working ok.

## Create the ‘/edit’ route & controller function

In order to render our edit form, we need to define an edit route & corresponding controller function.

1. Open routes/index.js and define a HTTP GET route for /edit and corresponding controller function editTodo in routes/todo-controller.js.

The edit route should be in the form /edit/:id, which allows the id parameter to be accessed via request.params.id.

The edit view expects a model parameter called todos, so the editTodo function needs to call service.findAll and render the edit view in the callback, passing the model map.

The view also expects a model parameter called current, which corresponds to the id of the “todo” item to be edited. This should be the same as the input parameter from the URI.

The model map should have two parameters, todos (an array of todo items) and current (the index of the todo item to be edited)

If everything works correctly, you should now be able to edit the selected “todo” item. Now, in order to save our changes, we need to define an update route.

## Create the ‘/update’ route & controller function

1. Open routes/index.js and define a HTTP POST route for /update and corresponding controller function in routes/todo-controller.js. Like the /edit route, /update should use a named parameter for the "todo" id.

The updateTodo function should call the service layer's updateTodo function and pass the updated todo's id and content for the "todo" item being changed, and a callback of the form function onUpdated(err).

In the callback, use res.redirect('/') to render the root view.

## Create the updateTodo service function

1. Open services/todo-service.js and provide an implementation for updateTodo.

Loop through all of the todos; if the id parameter matches the current “todo” item's id, update the current todo's content property with the content parameter, then call return callback() to break the loop & return immediately, indicating success.

If the id is not found, then our loop will exit; after the loop, then, invoke the callback with an error.

## Create the ‘/delete’ route & controller function

1. Open routes/index.js and define a HTTP GET route for /delete and corresponding controller function deleteTodo in routes/todo-controller.js.

Like the edit and update actions, we need to use a named parameter for the id corresponding to the specific “todo” item we wish to remove.

Call the service layer deleteTodo function and pass in the id of the item to be deleted, and a callback of the form function onDeleted(err). Then use res.redirect(‘/’) to render the root view in the callback.



The HTML specification, as implemented in browsers, does not support the DELETE verb in <form> elements, so make sure you use a GET instead. Express supports all HTTP verbs, including GET, POST, PUT and DELETE for RESTful clients.

## Create the deleteTodo service function

1. Open services/todo-service.js and provide implementation for deleteTodo.

First, initialize a temporary variable to keep track of an index into the todos array. Loop through the todos. if the given id matches the current element's id, use todos.splice(index, 1) to remove it, then invoke return callback() to break the loop and return immediately.

If the id is not found, we will exit the loop; at that point, invoke the callback with an error.

Once everything is working properly, this lab is now complete!