Web Servers

In this lab, you will use most of the key components of the node.js http module. We will create an application that handles a login.

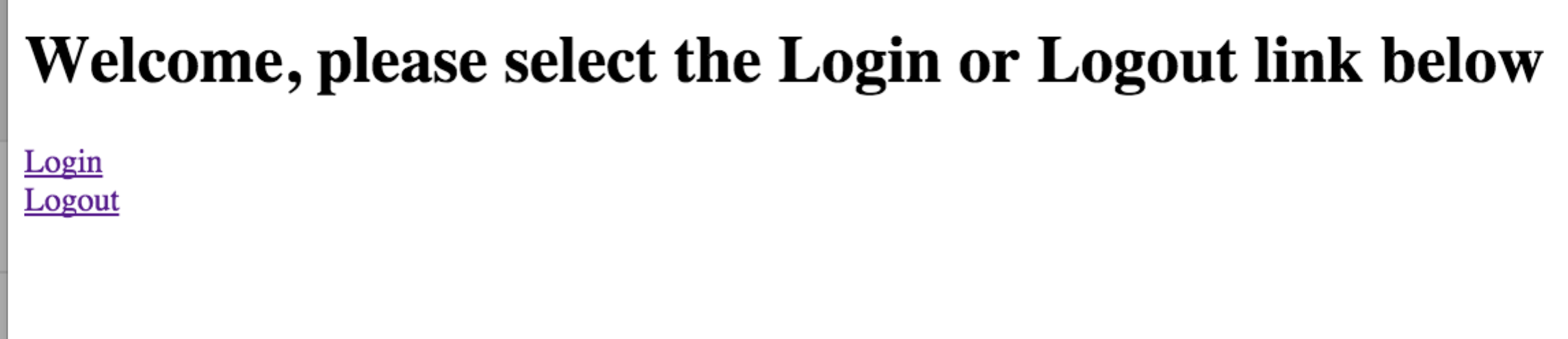
# Objectives

In this lab, you will learn to

* create HTTP servers and make client requests,
* read and write data to and from the HTTP server,
* render pages to the browser,
* implement the login function.

# Start your webserver

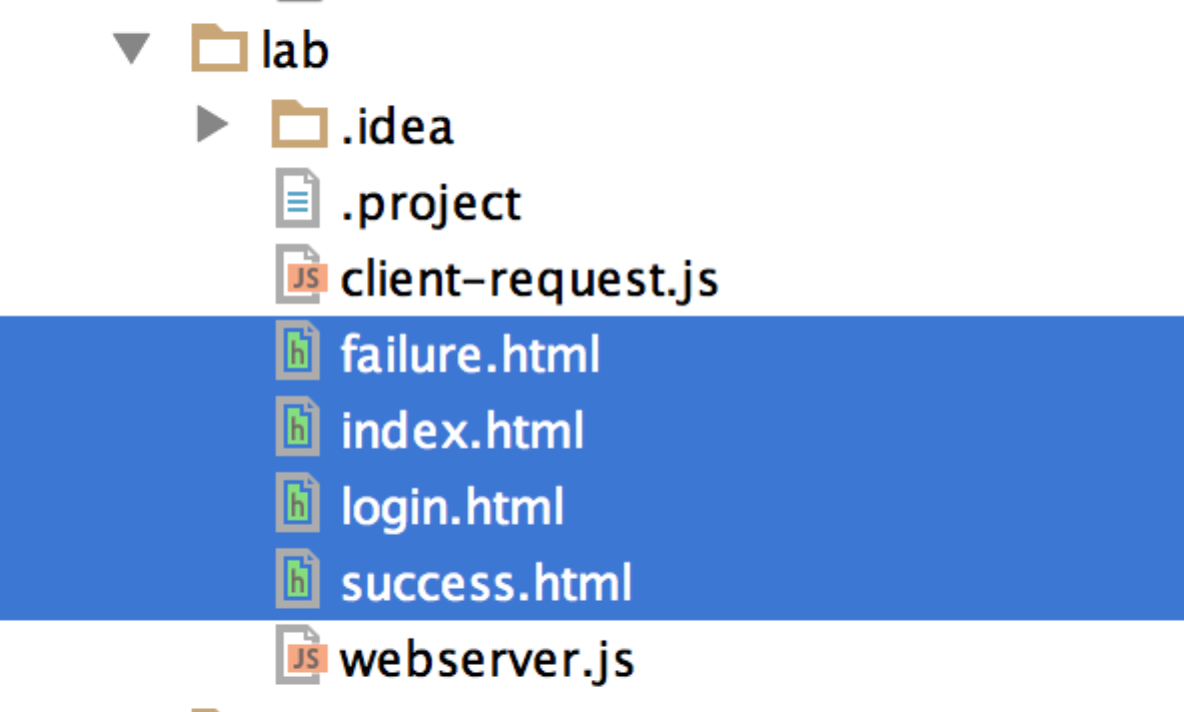
1. Open webserver.js in your IDE. This prints the index.html page to the browser. Run this file from your IDE or from the command line. Navigate to <http://localhost:3000> with your browser and you should see the following in your browser.



The application has four pages defined.

1. Index.html – This is the main page as shown above. This page displays when the browser sends a GET /
2. Login.html – This contains the login form with data entry fields for the username and password. This page displays when the browser does a GET /login
3. Success.html – This contains a success message and a link back to the index page. This page displays when the browser does a POST /login and the username and password fields are NOT empty.
4. Failure.html – This contains a login failure message and a link back to the index page. This page displays when the browser does a POST /login and the username or password field is empty.

The lab folder shown below contains all four html files as well as the webserver.js application that we will modify.

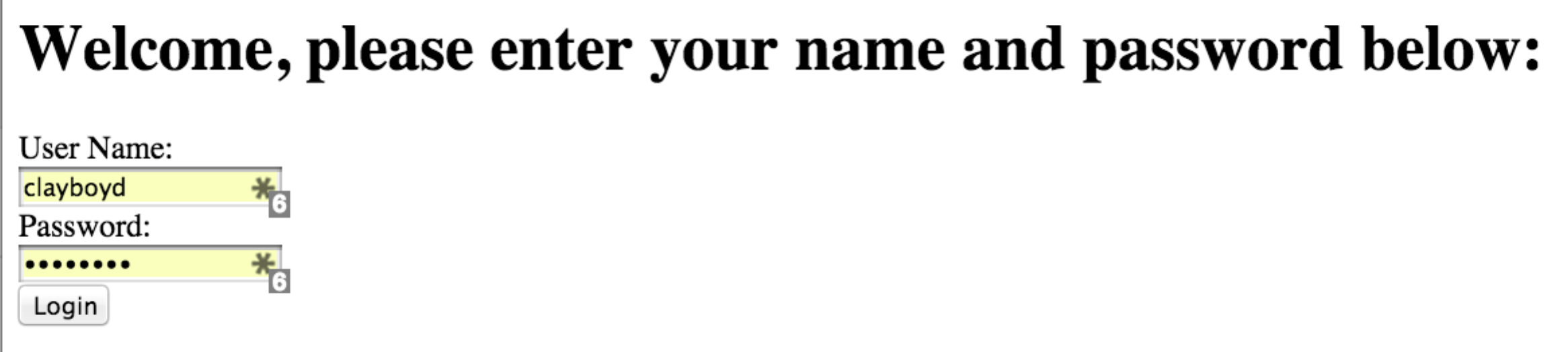


# Start the solution

First, install nodemon using npm install –g nodemon. This will listen for file changes in the exercise and automatically restart the server.

Change to the solution folder and start the solution server with nodemon webserver.js. In a browser display the first page with <http://localhost:3000/>. This should display the index.html page as shown earlier.

Click on the Login link to show the login form (login.html).



Enter any name and password, the application verifies that the fields are not empty. It does not check for specific data. The next page should appear (success.html).



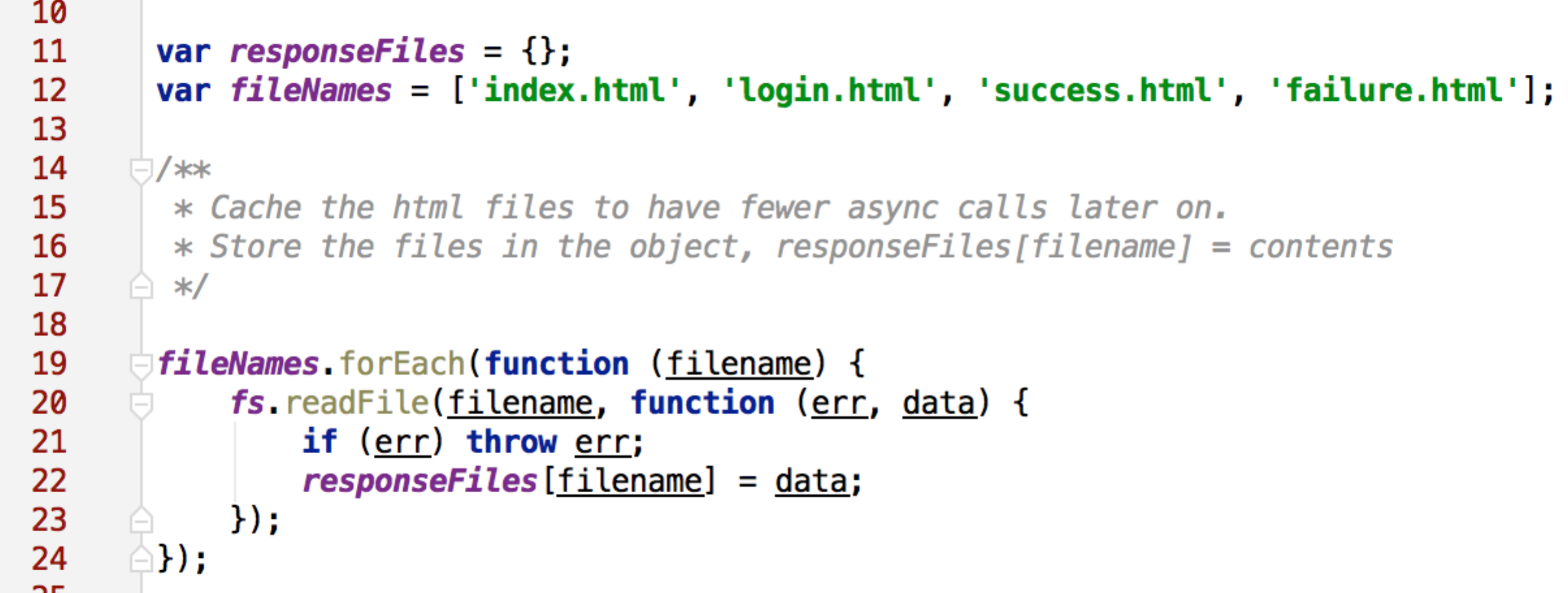
Go back to the login page and enter an empty password and click Login again. This time the login should fail (failure.html).



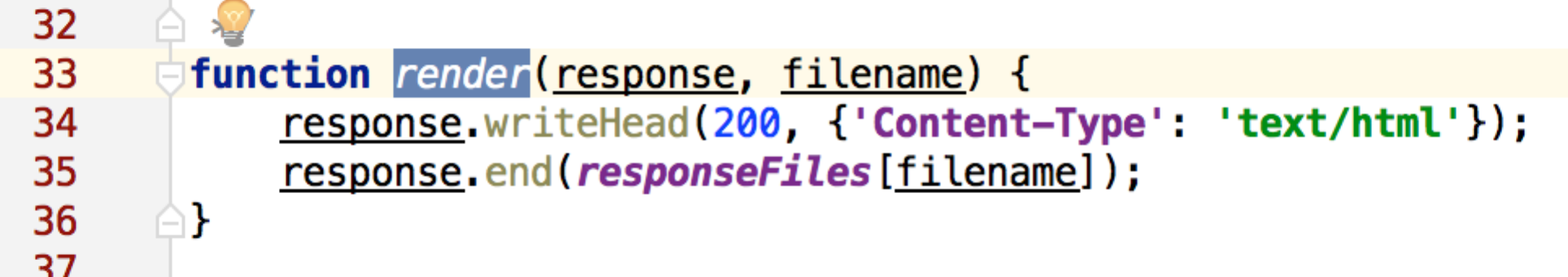
Stop the solution server, change to the lab folder and start the lab server with nodemon webserver.js. This application displays the index.html page regardless of the input URL. Let’s fix it.

# Examine the lab server

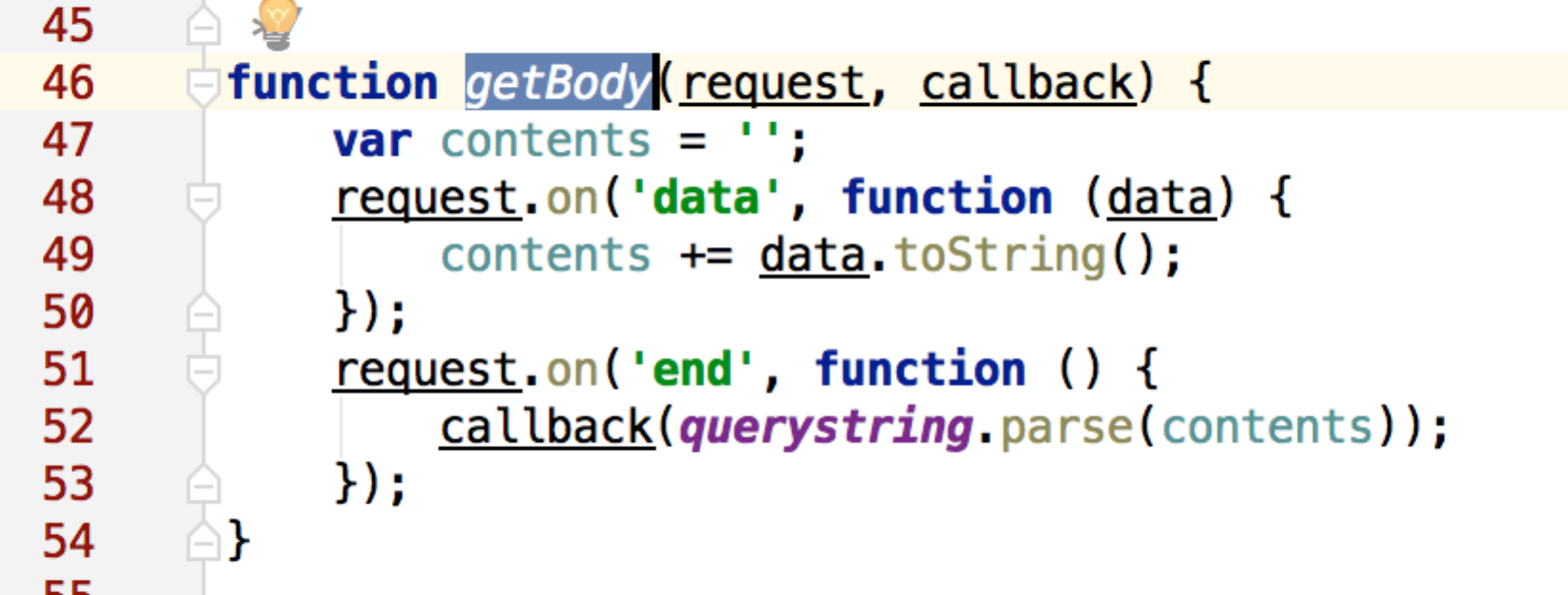
The lab server has several parts of the application already implemented as discussed below.



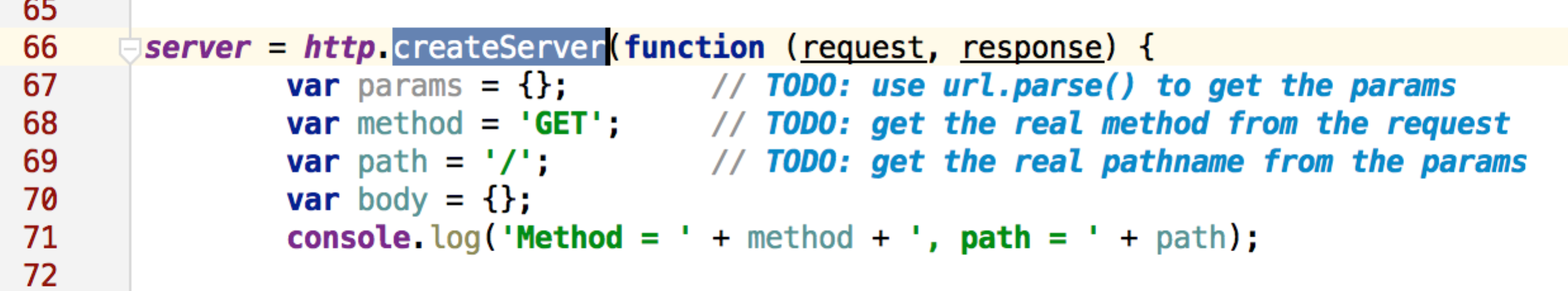
The above caches the four html files to avoid loading them on demand.



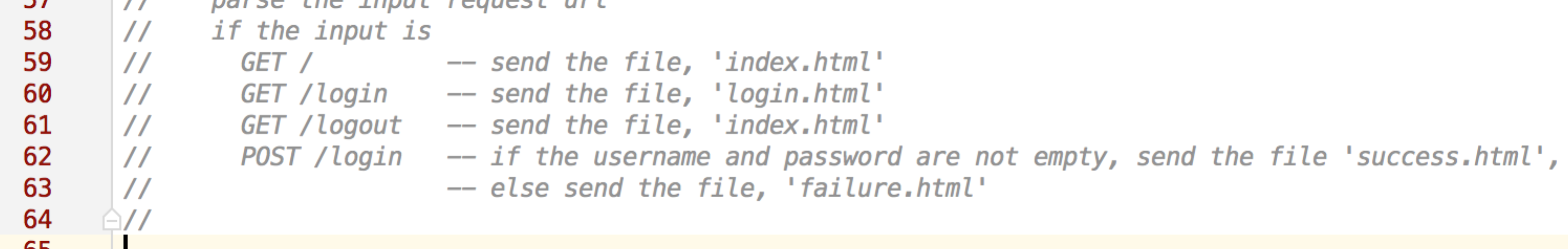
The above renders the html page from the cache on demand. Note the Content-Type value.



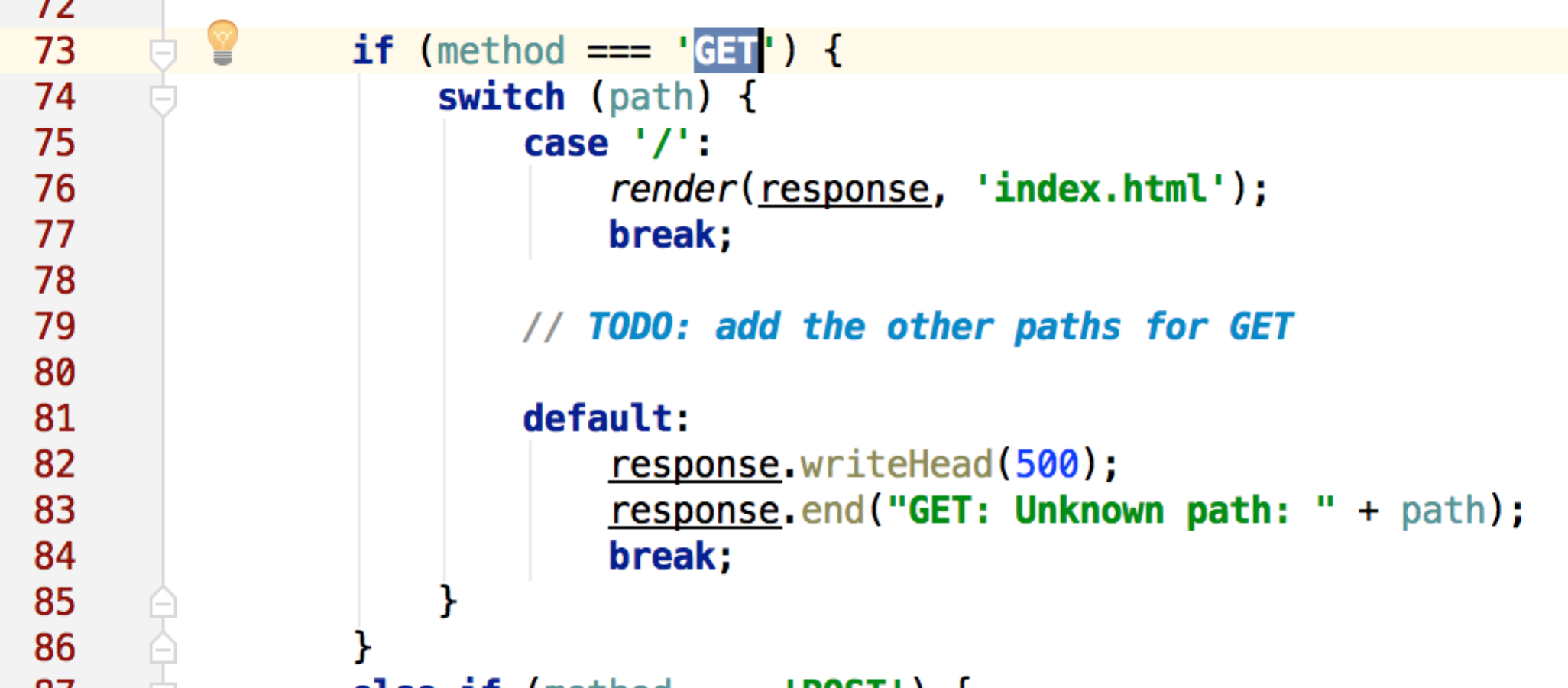
The above retrieves the body of the request. Remember that a POST method sends the parameters as attribute / value pairs in the body. The querystring.parse() converts this string into an object array making access easier. The ‘data’ listener accumulates the body data onto the contents variable. The ‘end’ listener sends the data to the callback.



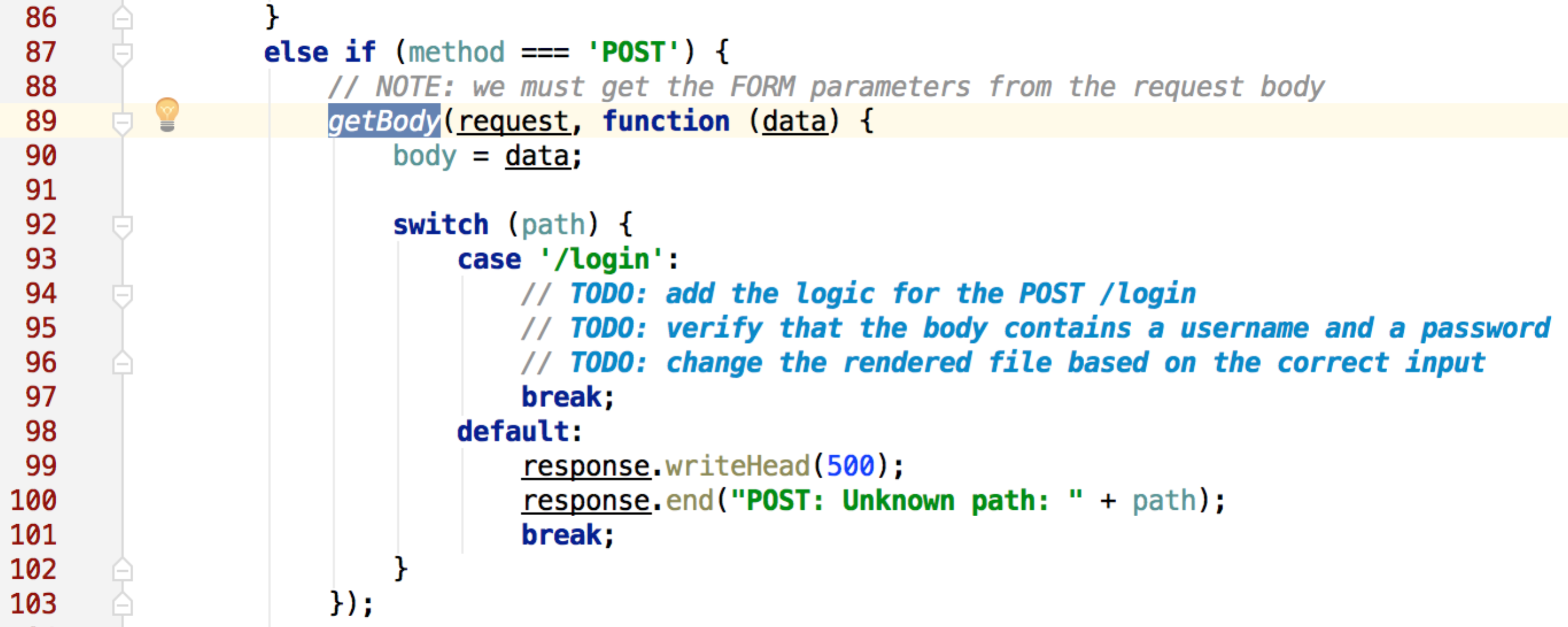
TODO: The above shows the creation of the server. NOTE: the parameters used in the application are all constants. We need to set the values based on the request. Do this now.



The above shows the business requirements. There are three paths associated with the GET method and one path associated with the POST method.



todo: The above shows the business logic. If the method is GET, then select each path and perform the business logic. NOTE: this shows the logic for the first page (when the path is ‘/’). Complete the logic for the other GET paths.



todo: The above shows the logic for handling the POST method. The getBody() function described earlier extracts the body of the request and returns an object whose indices are the attribute names and whose values return the data entered by the user on the form. Our task is to examine the username and password and render either the success.html or the failure.html file.

Congratulations! You are almost finished… Just prepare to answer the questions below.

# Food for thought

* If we wanted dynamic pages. For example, to place the username in the success.html file. How would we do this?
* If our application had 30 pages, would it be easy to maintain with a very large switch statement?
* Would it make sense to separate the dynamic pages from the static pages? After all, static pages don’t change, they just get sent to the browser. There might be thousands of images, CSS files, JS files, and other static files.