Web Technologies

Coursework 2

SET08101
Lecturer: Simon Wells

Xavier Sala
40290125
10/04/2019

Contents

1. Introduction

This assignment has consisted in the implementation of the previously developed static website, *Cryptoforfun*, as a dynamic website served by the Node.js application webserver with several added features; users register, log in sessions and the option to send and receive messages between users. In order to accomplish such task, HTML, CSS and JavaScript languages were used, as well as a MongoDB database installed locally.

The previously developed website consisted in several static HTML files and a single CSS file for describing their presentation. On this occasion, the several pages are rendered dynamically by using just a couple of different templates thanks to Handlebars, a library used as a View Engine in Node.js.

The previously implemented ciphers can be used now as methods to cipher a message sent to any other user of the website. At the same time, the inbox provides the method to decrypt the received messages.

The reading of The Web Application Hacker's Handbook 2nd Edition (D. Stuttard, M. Pinto, 2011), has been of use as it has helped a lot to understand how a web application works and where the website built for this project is most vulnerable.

2. Software design

Node.js is a programming environment which allows the execution of JavaScript code as server-side, offering the possibility to produce dynamic web page content before sending it to the client's browser. This project has been developed using the Express web application framework for Node.js.

Handlebars is a library which can be used as the Node.js view engine, like Pug. The main difference is that Handlebars requires the use of HTML code. Not only I feel more comfortable using HTML rather than the own Pug language, but it was much easier just to copy the code from the previously developed HTML static files, which is why it was the library of choice.

Two layouts were developed as templates to render the HTML contents of the different pages. The first one consists in the default template and, the other one, was used just for the ciphers from the new mailer section. Both templates keep the same original

The first challenge about to carry out during the project consisted in developing a user register and authorization system. Several approaches were considered in order to achieve it.

In first place, users would need to register to obtain an account. A new page presenting a form with input fields such as name, username, email and password and rendered using the default template would submit a POST request to the server. The server would then read the data and store it into the database according to the user schema.

Initially, the plan consisted in keeping a collection of usernames and passwords in a database stored locally, in the same device as the web server. The log in functionality would check whether the password provided by the user corresponds to the one stored in the database for such username thanks to a form submitting the user input in a POST request to the server. In case that the credentials were correct, a cookie simply containing a specific value could be

generated and installed in the client's browser as a token for keeping the user's session. In order to log out, such value could be deleted or changed.

Despite its simplicity and clear insecurity, this method would work and a session could be kept for the logged user, granting them access to restricted pages within the website, such the mailer and the inbox. The navigation diagram of Figure 2.1 shows how pages are organized in relation to each other.

The second approach to achieve the user login sessions, which was the method of choice, consisted in the use of the Passport authentication middleware. Middleware are the functions operating between a request and its respective response. Passport provides a bunch of methods and functions which can be used to authenticate users in the Express web application framework. It was the library of choice thanks to the amount of documentation available and the ease of use.

For security reasons, a policy enforcement would prevent the user to register a password shorter than eight characters long and they could be asked to input the same password twice when registering it. Alert messages could warn the user of errors or successful events.

The other challenge compressed within the assignment consisted in the development of a message system which could allow users to send and receive encoded messages and provide them with a method to decrypt them. No special libraries were needed to achieve this, other than Body-Parser to read body content from the requests and Mongoose to store the users input data in the MongoDB database, the locally installed database of choice. Mongo provides a simplistic and integrated way to work with Node.js, as well as the fact that there is plenty of documentation and support online.

In order to send the encrypted messages, a similar page based on the ciphers pages was developed for each cipher and rendered using a second Handlebars layout. However, the input fields would be now part of a form which would be sent to the server in the form of a POST request and stored according to an appropriate schema in the same database, in a separate collection. The cipher algorithm of choice as well as the parameters used for encryption would be recorded as well and sent to the datastore along with the encrypted message. The same client-side encryption scripts would be used to perform such operations.

The inbox would consist in the default template with almost no body, as it should be populated with the user incoming messages. They could be retrieved filtered as per username recipient as a response to a GET request.

3. Implementation

The requests to the server are handled by the three different routers defined in the app.js file as shown in Figure 3.1, which are stored within the routes directory. Index.js is the router for the guest users' pages. Users.js is the router used to login, logout and register new users. Dashboard.js routes all the pages for the logged in users.

The models directory contains the two models (Figure 3.2) that define the schemas of how the data objects are going to be stored within the two collections, user and messages, in the same database.

The views folder contains all the files for the view engine as defined in the app.js in Figure 3.3. The directory contains all the templates used to display the HTML contents and a subdirectory containing the two layouts.

One of the main advantages of using a dynamic website versus using a static one, is that HTML templates can contain variables so they are populated with data and may personalize the pages as shown in Figures 3.4 and 3.5. The same example applies when accessing to the Inbox page when there are no messages or when there are messages available to read (Figures 3.6 and 3.7).

The very same concept is applied in order to pass some information from the datastore to the client across the server. The current logged user's username is stored in a variable and passed to the client's browser in a hidden form. Then, this hidden form is submitted again along the user's ciphered message to register the sender's username (Figures 3.8 and 3.9). It can be appreciated that in the validation function there is also a message passed to the view layouts using Flash in case that a non-logged user tries to access a restricted page and the navigation bar buttons change as well depending on whether the user is logged in or not (Figure 3.10).

Several error messages are shown to the user when they do not supply valid data, for instance, when registering a new user or when leaving required fields blank in the mailer page (Figures 3.11 and 3.12).

4. Critical evaluation of the implementation

The approach carried out for this assignment achieved its minimum requirements; the implementation of user register registration, an authentication system and a messaging system between users able to cipher and decrypt the messages using the client-side scripts.

There have been a few extra details implemented along the requirements such as the error and success messages after user input and submission of data to the server, validation of such data and security policies such as a password minimum length and retype it at registration or the storage of salted hashed passwords in the database.

There is a total of fifteen pages using just two view layouts, which means that it really takes advantage of the dynamic aspect of the website.

I would have liked to add the possibility to delete messages from the inbox as well as the chance to edit the user's data in their profiles as should not be really difficult to achieve and seems quite basic functionality, despite that it was not a requirement.

5. Personal evaluation

As I understand, the main objective of this assignment pretended that the student built the basics of the communication between a client and an application server, starting with the requests and responses, manipulating cookies... I tried to build the authorization system this way, with no libraries and just by inserting a "session token" with a value stored in plain text in a cookie. However, it was taking me too long, I was stuck and the deadline was close. Unfortunately, I could not take advantage of the deadline extension due to personal reasons. Because of that, I decided to implement the user authorization using the Passport library, which

facilitates such implementation and there is a lot of documentation about it online and allowed me to progress a bit faster.

On the other hand, I must add that it might have been really useful to me to use a few extra libraries such as Passport or Handlebars for this project. The reason for that is that the approach of doing everything “by hand”, with no libraries, has been used in every programming module I have coursed and it is because of that fact that I did not really know how to properly search libraries and their documentation. This project pretended to put quite a lot of different technologies together and that has required a lot of research and documentation reading to learn to use new libraries.

I have also understood much better how the Express framework works with Node.js, how the requests are routed by the HTTP request method, how the responses dynamically render HTML code and how the server interacts with the database.

6. References

Node.js used libraries and packages

bcryptjs	https://www.npmjs.com/package/bcrypt
body-parser	https://www.npmjs.com/package/body-parser
express	https://expressjs.com/
flash	https://www.npmjs.com/package/flash
handlebars	https://handlebarsjs.com/
mongoose	https://www.npmjs.com/package/mongoose
passport	http://www.passportjs.org/

Other material used:

Figure: “About us”. Tashatuvango – Fotolia. Retrieved from: https://goo.gl/images/JraQNb [Accessed: 5/03/2019]

Figure: “Bacon’s Cipher”. Retrieved from: http://bamcneil.blogspot.com/2011/11/national-geographic-aryan-brotherhood.html [Accessed: 4/03/2019]

Figure: “Caesar’s Cipher”. Retrieved from: http://www.stmp.camden.sch.uk/blog/caesar-shift-code/ [Accessed: 3/03/2019]

Figure: “Cryptography”. Retrieved from: https://searchsecurity.techtarget.com/definition/cryptography [Accessed: 5/03/2019]

Figure: “ROT13”. Retrieved from: https://en.wikipedia.org/wiki/ROT13#/media/File:ROT13_table_with_example.svg [Accessed: 3/03/2019]

Traversy Media, *Node.js Login System With Passport,*Youtube. Available at: https://www.youtube.com/watch?v=iX8UhDOmkPE [Accessed: 07/4/2019]

Simon Wells. (2018). SET08101 Web Technologies Lab 5: Design. Available at: https://github.com/siwells/set08101/blob/master/labs/lab05_design.tex [Accessed: 3/03/2019]

Appendices

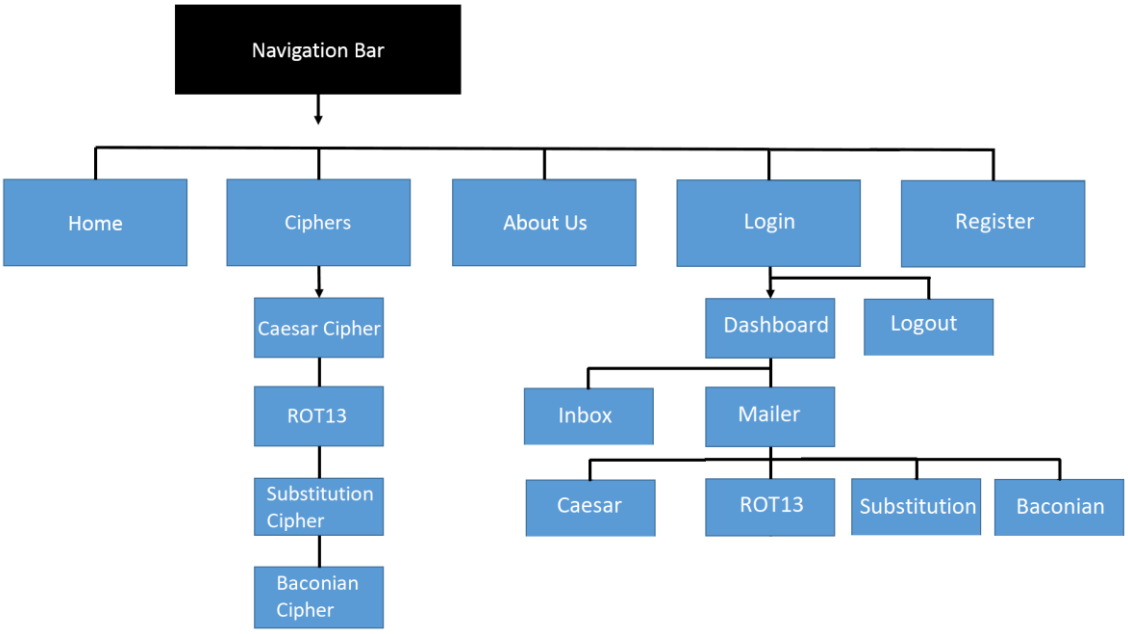

Figure 2.1: Navigation diagram.

```
85  //Routers
86  app.use('/', routes);
87  app.use('/users', users);
88  app.use('/dashboard', dashboard);
```

Figure 3.1: Routers defined in app.js.

```
4   //User Schema
5   var UserSchema = mongoose.Schema({
6     username: {
7       type: String,
8       index:true
9     },
10    password: {
11      type: String
12    },
13    email: {
14      type: String
15    },
16    name: {
17      type: String
18    }
19  });
```

Figure 3.2: User schema from the user.js model.

```
28  //View Engine
29  app.set('views', path.join(__dirname, 'views'));
30  app.engine('handlebars', exphbs({defaultLayout:'layout'}));
31  app.set('view engine', 'handlebars');
```

Figure 3.3: View Engine defined in app.js.

```
<div class="cos">
  <h1 id="ciphers_font">Dashboard </h1>
  <p id="pic"><img src="images/messages.png" width="300" height="auto"></p>
  <div id="about_cipher">
    <div class="container">
    <p>Welcome to your Dashboard, {{user_name}}!</p><br>
    <p>Check your <input type="button" value="Inbox" onclick="location.href = 'dashboard/inbox';"> for new messages!</p>
    </div>
</div><br><br>
```

Figure 3.4: The user's name is passed to the HTML file when it's dynamically generated.

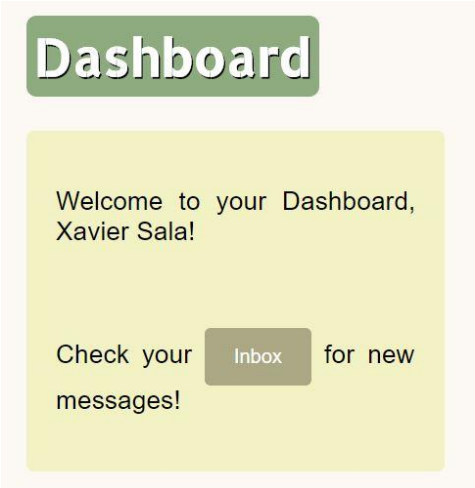

Figure 3.5: Dynamically rendered HTML data.

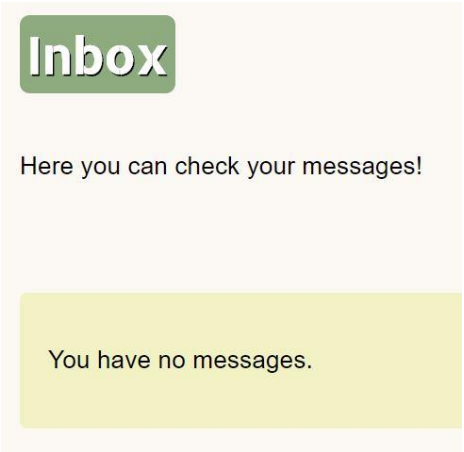

Figure 3.6: Empty Inbox.

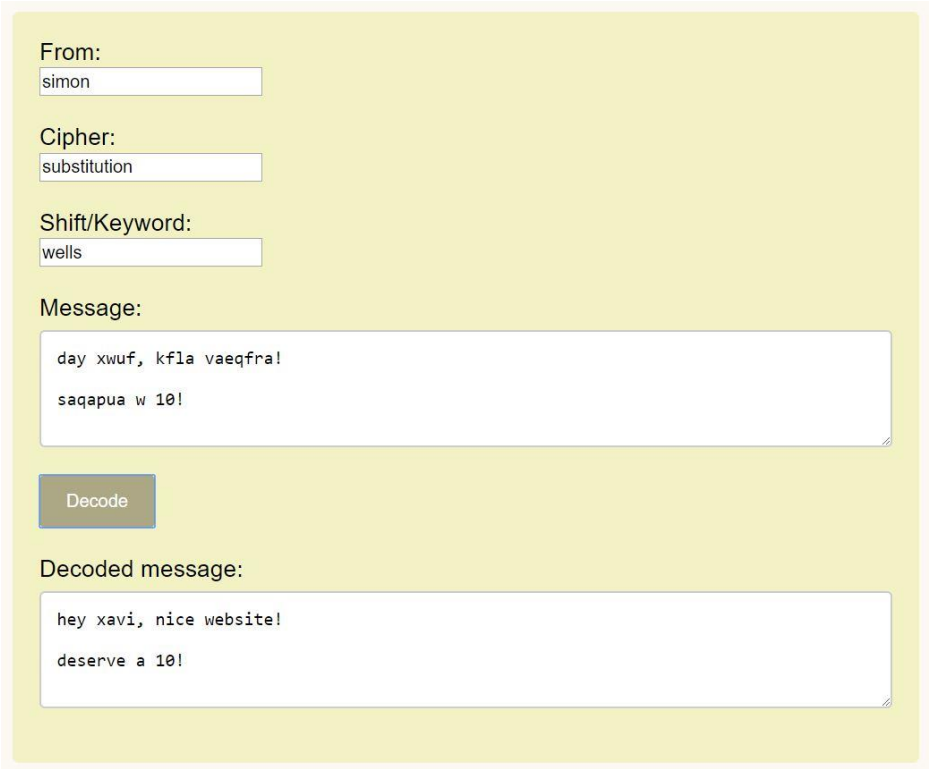

Figure 3.7: A received message in the inbox with the decode function.

```
//Validation function
function ensureAuthenticated(req, res, next){
  if(req.isAuthenticated()){
    return next();
  } else {
    req.flash('error_msg', 'You must log in.');
    res.redirect('/users/login');
  }
}


//GET ciphers routes
router.get('/ciphers/caesar', ensureAuthenticated, function(req, res) {
  var curr_user = req.user.username;
  res.render('ciphers/caesar', {
    layout: 'dashb_layout.handlebars',
    from_user: curr_user
  });
});
```

Figure 3.8: Router with validation function. This route applies authentication, specifies the view engine layout to use when rendering and passes a variable to the rendered data.

```
<div class="container">
  <form method="post" action="/dashboard/ciphers/substitution" id="substitution">
    <input type="hidden" name="cipher" value="substitution">
    <input type="hidden" name="from_user" value="{{from_user}}">
    <label>To:</label><br>
    <input id="to_user" class="form-control" placeholder="Username" name="to_user"></input><br><br>
    <label>Your message</label><br>
    <textarea id="message" placeholder="Type something..."></textarea><br>

    <label>Keyword</label>
    <input id="key" name="param" form="substitution"></input><br><br>

    <input type="button" value="Encode" onclick ="encode()"> <input type="button" value="Decode" onclick ="decode()">

    <label id="output_name">Encoded or decoded message</label><br>
    <textarea id="output" readonly class="form-control" name="message_ciphered" form="substitution"></textarea><br>
    <input type="submit" value="Send"><br><br>
  </form>
</div>


<script type ="text/javascript" src ="../../scripts/substitution.js"></script>
</body><br><br>
</div>
```

Figure 3.9: Form which receives data when rendered and sends it back to the server when submitting the POST request of the form.

```
    {{#if user}}
    <a href="/dashboard">Dashboard</a>
    <a href="/dashboard/mailer">Mailer</a>
    <a href="/dashboard/inbox">Inbox</a>
    <a href="/users/logout">Logout</a>
    {{else}}
    <a href="/users/login">Login</a>
    <a href="/users/register">Register</a>
    {{/if}}
  </div>

  <!--Alert messages -->
  {{#if success_msg}}
    <div class="alert-success">{{success_msg}}</div>
  {{/if}}

  {{#if error_msg}}
    <div class="alert">{{error_msg}}</div>
  {{/if}}
```

Figure 3.10: Flash error and success messages.

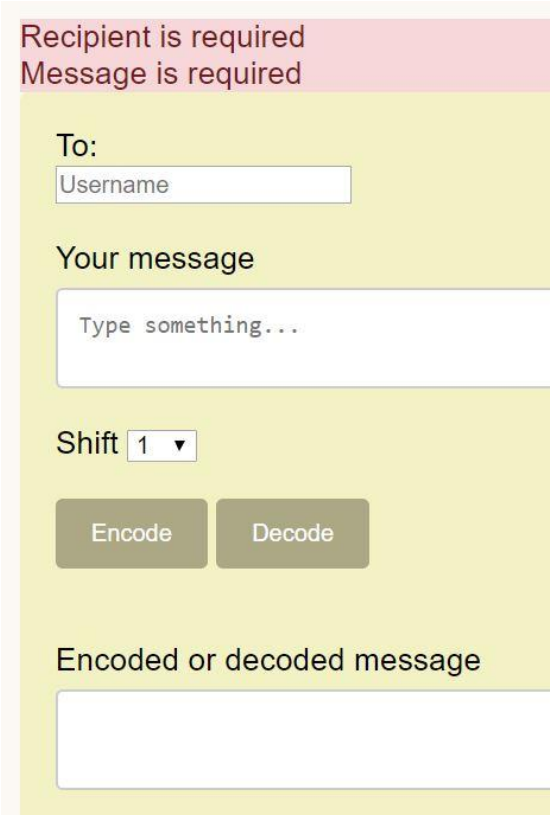

Figure 3.11: Flash error messages.

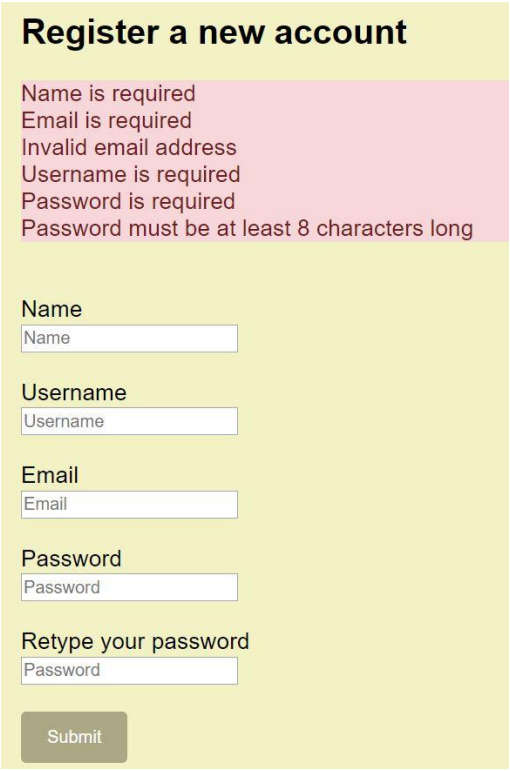

Figure 3.12: Flash error messages.

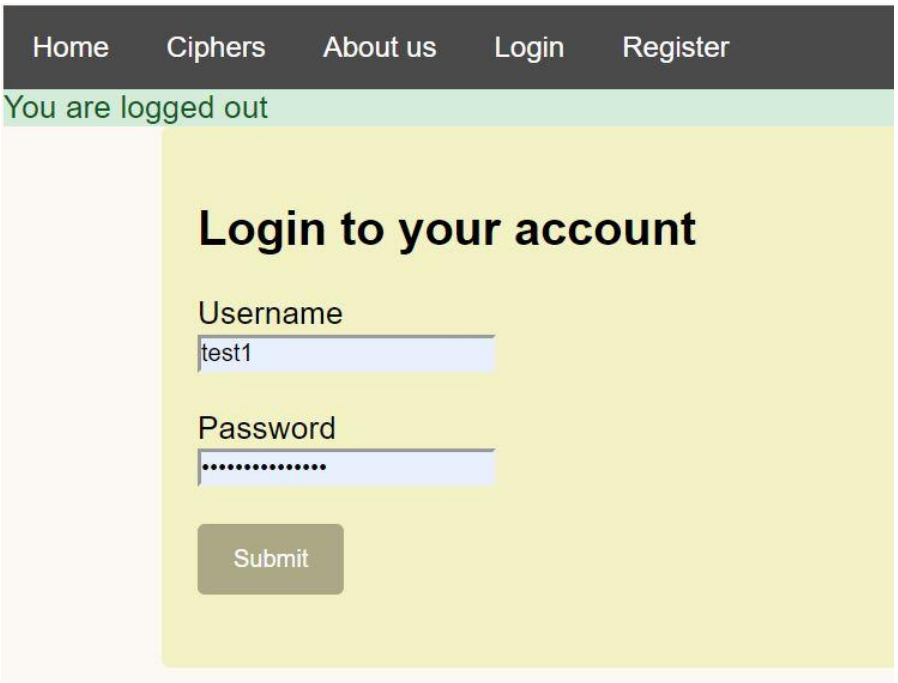

Figure 3.13: Flash success message.