RepuTech

"Reputable technology services"

Design

RepuTech is a Node.js application that exhibits a 3-tier web architecture.

First, the presentation layer consists of the front-end user interface, which is generated by a templating engine. Once the engine renders a markup template, it is then served to the user via an HTTP request. Static files (stylesheets, scripts) are also part of this, and are entirely separate from the other layers. These are served to the user as-is, as the user requests each one of them separately.

Second, the logic layer consists of the middleware in the application. Including Express and a majority of the plugins used, including plugins for session management, routing, APIs, and database connections. Some examples of this layer's functionality also include authentication of users, the handling of file uploads, security, and communication with external OAUTH APIs.

Finally, the data layer consists of a PostgreSQL server and a Redis server. Both of these are separate from the actual application, and these maintain most of the data storage needed by the application. The PostgreSQL server stores the user and post data, and its schema description (in the form of DDL statements) is included. The Redis server, on the other hand, is NoSQL, and as such does not have a well-defined schema. It changes whenever needed, and is used for session management. Both of these servers cannot do anything on their own; the data layer must interact with them to retrieve their stored information.

The logic layer facilitates the communication between the data and presentation layers, which never interact with each other. The logic layer maintains pooled connections to interact with the data layer, and provides any necessary information to the templating engine to aid in the presentation of pages. Furthermore, the logic layer handles optimizations for this facilitation, and this is detailed in its relevant section.

We designed our application this way as this allows for reusable components (as pooled connections can persist between the logic and data layers, and templates can be cached and/or reused between the logic and presentation layers) which helps performance and allows for ease of implementation. Furthermore, this allowed for greater logical organization and independence, resulting in easier task allocation and development.

Security

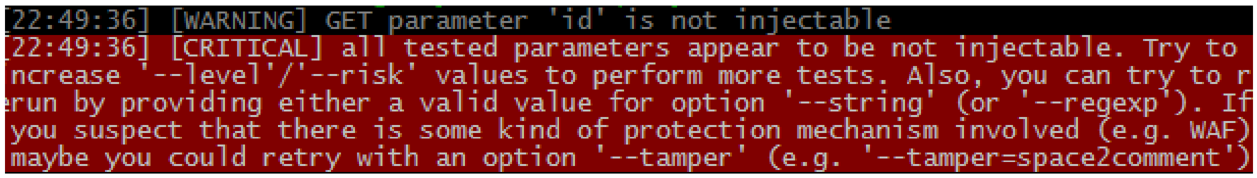
1. HTTP headers are carefully set by the "helmet" plugin. This includes: Content-Security-Policy which allows resources to be loaded only from certain specified URLs, X-Powered-By to hide the fact that the server is powered by Express (which attackers could take advantage of), headers that disallow the page from being embedded in frames (preventing clickjacking exploits), DNS Prefetching Control (disallowing prefetching of DNS which could

be used to load resources from unwanted sources), MIME type sniffing protection and HTTP Strict Transport Security which ensures HTTPS connections (on our Heroku server). We used Firefox's Firebug and Chrome's Inspector to view these headers, and found that all of these exploits were patched.

2. CSRF protection is enabled via the "csurf" plugin. On every POST form, we include a CSRF token which protects against malicious attacks, as attackers cannot replicate this token. We tested our app by altering the CSRF token in various POST forms. If an invalid request is found, the server will simply throw out an error. Users must go through our own application legitimately to make these requests.

3. Cross site scripting is mostly dealt with, as all inputs are sanitized before being dealt with, with the use of the "express-validator" plugin. All API calls have inputs validated. We tried to break our application many times (e.g. in form data) but to no avail. The server simply throws out an error to the API call when confronted with a validation error.

4. SQL injection is more-or-less impossible, as every single input is validated (as above), and every single SQL query is parameterized with the help of "pg" and "pg-pool" (PostgreSQL clients) plugins. Invalid input is just sanitized and discarded. We used "sqlmap" which is an automated Python tool to check for these vulnerabilities. Not even one vulnerability was found.



```
[22:49:36] [WARNING] GET parameter 'id' is not injectable
[22:49:36] [CRITICAL] all tested parameters appear to be not injectable. Try to
ncrease '--level'/'--risk' values to perform more tests. Also, you can try to r
run by providing either a valid value for option '--string' (or '--regexp'). If
you suspect that there is some kind of protection mechanism involved (e.g. WAF)
maybe you could retry with an option '--tamper' (e.g. '--tamper=space2comment')
```

5. File uploads are checked, as a maximum size of 2MB is enforced, and the MIME type of uploads is checked (we only allow image uploads!). This was done with the help of "formidable". They are also renamed and stored in specific, secured folders.

6. Cookies do not store any information other than a temporary CSRF token and a session ID. Sessions are managed on the server, making it so cookies cannot be manipulated to great effect. This was achieved with the "express-session" plugin. All IDs are also non-enumerable as they are random base-64 hashes. Cookies are also signed by the "cookie-parser" plugin, making them even harder to manipulate, as invalid cookies will not be recognized by the server.

7. It is harder for bots to register accounts and post spam, as an e-mail verification system is implemented. The plugin "nodemailer" was used for this, as well as a few changes to the database schema. E-mail verification is required for new registrations.

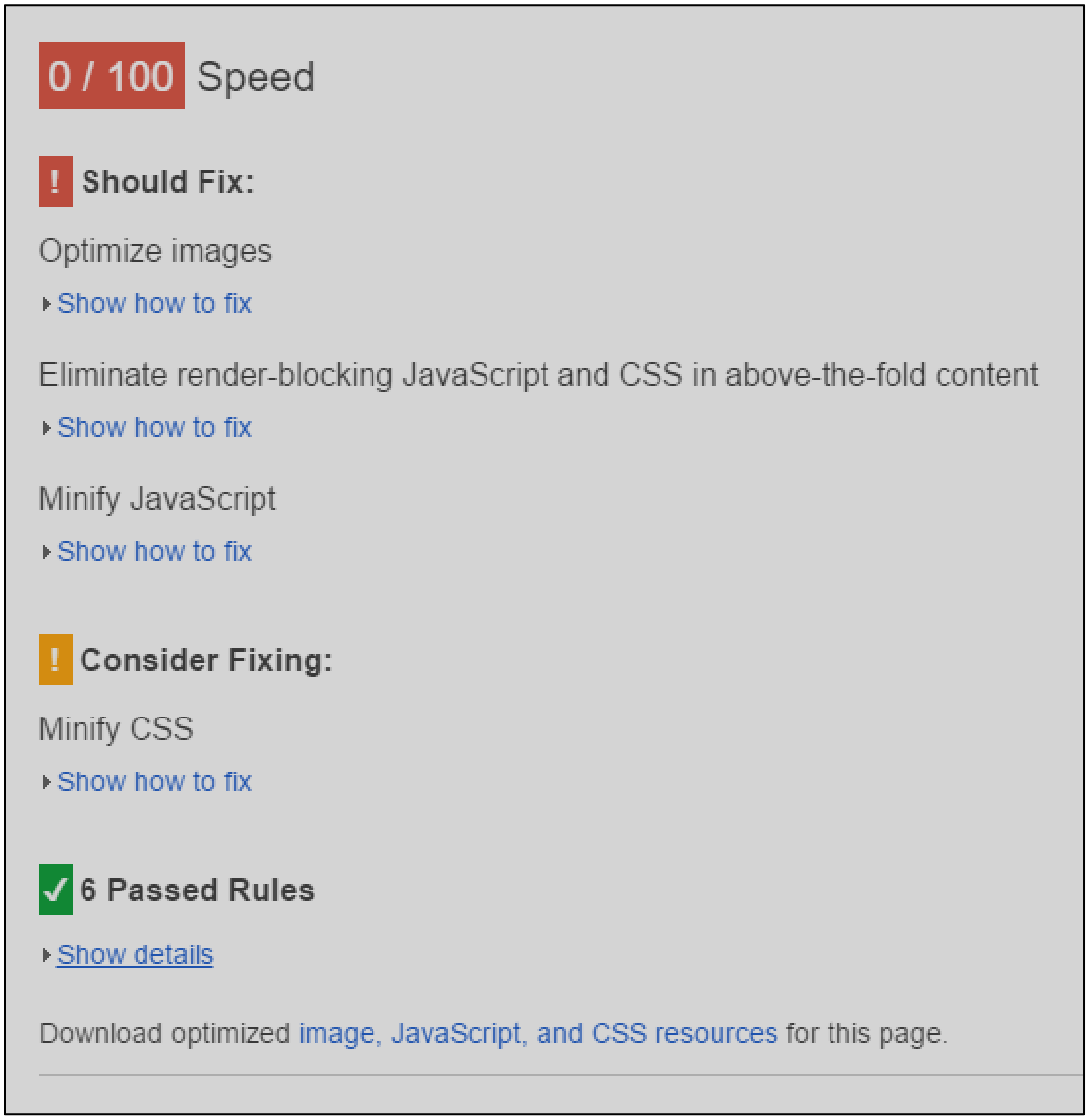
8. Finally, logins were also protected from bruteforce attacks. We have a 30-second timeout when one tries to log in too frequently and too fast. This part was not done via a plugin. Furthermore, we use Bcrypt (with the "bcrypt-nodejs" plugin) which ensures a long time

(~2 seconds) used for computing hashes, which takes a significant amount of time between request and response.

Benchmarks

We used Google's PageSpeed Insights to benchmark the site, including load times.
Link: https://developers.google.com/speed/pagespeed/insights

After our first iteration of the project, we tested our site with this analyzer and got a score of "0/100" for performance (but "100/100" for user interface). It gave us tips:



0 / 100 Speed

! Should Fix:

Optimize images

▸ Show how to fix

Eliminate render-blocking JavaScript and CSS in above-the-fold content

▸ Show how to fix

Minify JavaScript

▸ Show how to fix

! Consider Fixing:

Minify CSS

▸ Show how to fix

✓ 6 Passed Rules

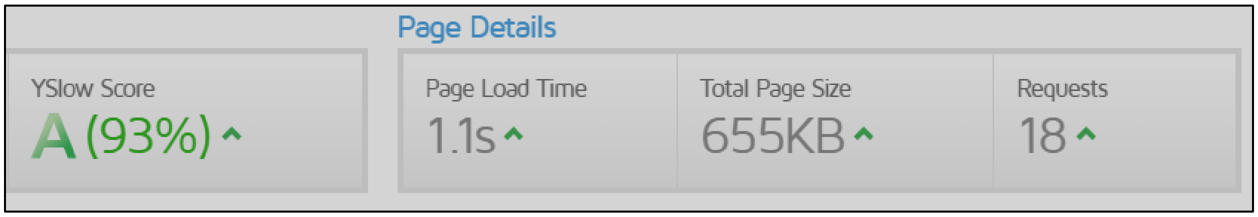
▸ Show details

Download optimized image, JavaScript, and CSS resources for this page.

We then did what it said: optimized our static resources. With the use of Gulp, we used plugins that would automatically optimize and minify our content. We used plugins "cssnano" to minify our CSS, "uglify" to minify our JavaScript and "imagemin" to optimize our images whenever our server was deployed. We also added “Expires” headers to allow clients to cache our resources.

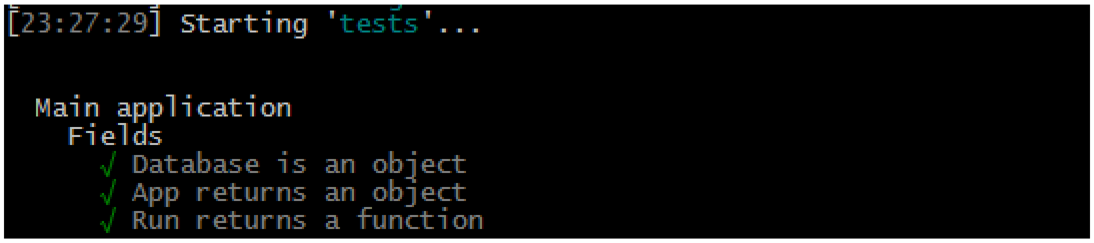
Doing this earned us a modest score of “68/100” as some tips were not possible to do.

We believe we have optimized the performance, as other tools give significantly higher scores. (e.g. Pingdom at https://tools.pingdom.com gives a 100/100 score, GTmetrix’s YSlow score at https://gtmetrix.com is 93%).



Testing

As mentioned previously, we use Gulp to preprocess our server. This means that Gulp also runs automated testing with Mocha. Every time that the server deploys, Gulp automatically runs the given unit tests in the “tests” folder. The server only runs when the tests pass.



```
[23:27:29] Starting 'tests'...


  Main application
    Fields
      √ Database is an object
      √ App returns an object
      √ Run returns a function
```

We looked at each function in our main file and tested whether it worked or not. Some database queries were also tested. So far, we have only tested the back-end.

In the future, we do wish to implement client-side testing (on the browser and its functions) as well as testing of an active server (i.e. running tests after a server is already up) through simulating browser requests and checking status codes.

Enhancements

1. Additional features which were not specified in the handout. This includes: the ability to associate avatars with users and image carousels with posts through file uploads, privacy settings for posts, a system to reset passwords through e-mail verification, and the ability to filter search results by various criteria.

2. Ease of deployment. Gulp is used, which handles dependencies and streamlines the deployment process through the use of tasks and daemons to monitor the environment. Tests are automatically run before starting the server.

3. Responsive design of the website in all pages. Professional user interface.

4. High attention to security: passwords are encrypted with Bcrypt, the SQL schema is separated into sensitive and public information, an e-mail verification system is implemented, and many types of exploits were considered and prevented.

5. High attention to performance: all static files are minified, concatenated, cached and compressed, CDNs are used, data connections are pooled, and multiple threads are used (using cluster).

6. The next generation of code (ES6, CSS3) is used in both the back-end and front-end, as Babel and CSS preprocessors are used to improve compatibility with older browsers.

7. Use of frameworks for the front-end, including jQuery UI to create image sliders, and React to generate templates of forms, improving performance and user-friendliness.

8. High logical organization of the Node.js back-end. All files having to do with different functions (e.g. SQL queries, or REST calls, or user authentication) were separated, and the use of Node.js exports were utilized to a great extent. Things like REST calls, third-party authentication and input validation are very easily added through just adding to an array or JavaScript object.

Additional Remarks

Some features from the handout are still present, though they are not exactly as described. Firstly, there is no function to "delete" a user completely from the database, but there is an option for an administrator to ban a user. This effectively hides the user from all views and disallows them from ever logging in again. An administrator must also visit a user's profile in order to edit or ban them.

While there are no features that are missing from the handout, there were some additional features (i.e. enhancements) that we did not implement due to the time constraints. Some of these were detailed in Phase I, but did not come to fruition. These are listed below.

One feature that was considered for Phase I was an extended administrator panel. The original mockup included user profile searching, database statistics and the handling of user reports. All of these were additional features that were scrapped. User profile searching would work similarly to post searching, but only by first and last name and/or email address. It would list users that matched the given query. Database statistics were simply some numbers, such as a count of the number of users currently registered, or a count of the number of posts made. These would be easily pulled from the database. Finally, user reports would come in handy for a site such as RepuTech that values reputation, as private reports could enable shady activity to be reported to administrators.

These would come up as a panel in the administrator panel, and an administrator would see information of both the accused and the reporter, and decide to ban the accused or not.

Another feature that was originally planned was a news feed. When users followed each other, it would also result in a news feed tab that would post updates from a followed user to each of their followers. This would be in real-time and would update automatically. We planned to use Socket.io (with its 1.0 update) for this, associating sessions and WebSockets together to achieve real-time updating.

Some performance features were also planned. For example, we wanted to implement a CAPTCHA when one tried to access a particular resource (such as logging in) too frequently. This would be implemented by accessing the relevant API.

Miscellaneous features that we wanted to implement include: the ability to un-like a post after liking it already, the ability to un-follow a user after following them already, and the ability to delete post images (as the owner of the post). These all have similar functionality implemented already, and these would be implemented similarly.

The Team

GitHub URL: https://github.com/akshay-nair/RepuTech
Heroku URL: https://reputech.herokuapp.com/
Instructions on how to use the app: Found in readme.txt

Team Members:

Name: Akshay Nair
CDF: c6nairak
E-mail: akshay.nair@mail.utoronto.ca

Name: Seyed Hossein Fazeli
CDF: c5fazeli
E-mail: hossein_free@yahoo.com

Name: Kevin Thich
CDF: g5thichk
E-mail: kevin.thich@mail.utoronto.ca

Name: Anson Chen
CDF: g5chen
E-mail: ans.chen@mail.utoronto.ca