ezkomment

Orbital 2022 Milestone 2 README

Vu Van Dung Nguyen Viet Anh

25 June 2022

Contents

1 Team Information

- **Application/team name:** ezkomment
- **Proposed level of achievement:** Artemis

2 Project Scope

ezkomment aims to produce a PaaS to provide easy comments handling and moderation for the web.

Static sites have always been struggling to support commenting, because unlike posts or other parts of the sites which are static, comments are dynamic data that can change at any time. Existing solutions are also not flexible and lack several necessary features. Hence, we aim to produce a PaaS to deal with these issues: an easy and straightforward comment solution for the web, with built-in moderation features and full customisability.

3 Motivation

3.1 Problems with Handling Comments on the Web

Many programmers start learning web development by writing *static* sites. It can be for various purposes: self-introduction, product showcases or blogs. And we all want to get feedback on the products we built. However, since the sites we built, whether by pure HTML, with static generators such as Jekyll, or even more complex frameworks such as React, are all static, we cannot host comments on them since they are dynamic contents and do require a back-end with at least a database to store those contents.

A member of the team, Dung was also once in the same position: he would like to add a form to the end of his blog for readers to add comments to it, so that he (and other readers if he wanted to) could know what people think of his contents. However, he could only work on front-end development at the time, so that was not possible. He found a tutorial on the Internet on making some tweaks to his Google account and the form such that whenever the reader submits the form, client-side JavaScript would update a Google Sheets document, which would consequently trigger his Google account to send an email to himself. He did make it work, but of course not only was it terribly complicated, it was also impractical: other readers could not read those comments, and he would like to host the comments elsewhere so that his inbox does not get filled.

And not only do static sites face these problems: since implementing the whole system to handle comments to posts may be unnecessarily complex while not contributing much to the main application logic, many modern full-stack applications also did not implement a comment section at all, despite having a fully functional back-end and database.

Therefore, many websites and applications opted to use more easy-to-use services.

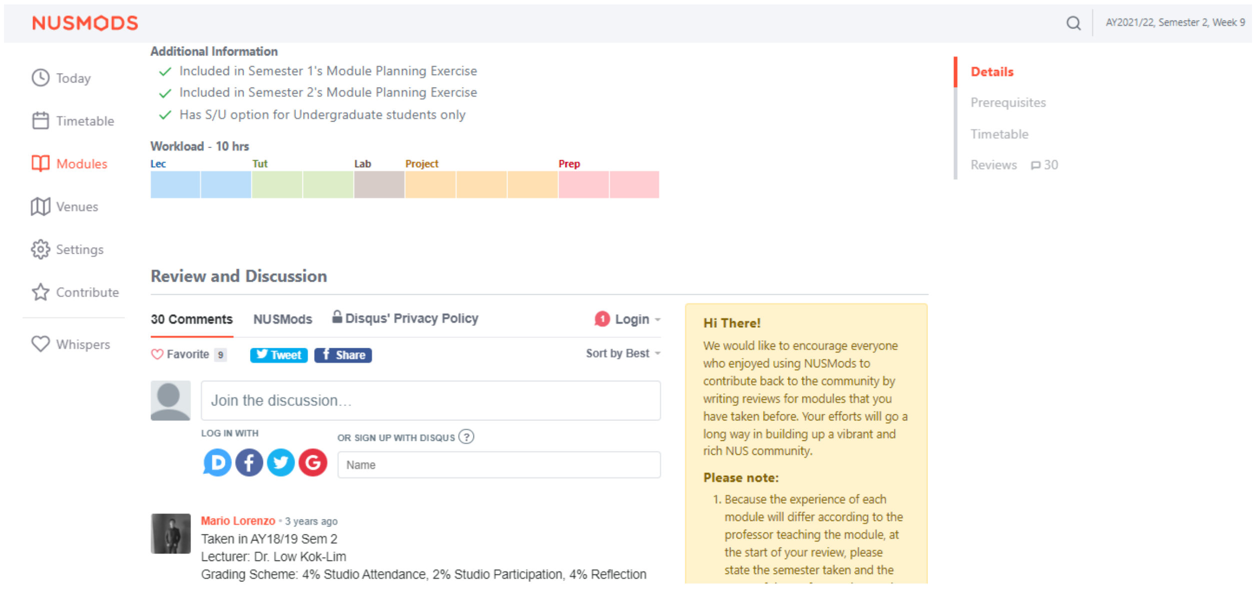
3.2 Existing Solutions

Disqus is arguably the most popular option on the web as of now (2022): whenever you go to a site not by WordPress or similar tools (which provide comments as a built-in feature), there is a high chance the comment section is managed by Disqus. A website that all NUS students use frequently, NUSMods, use it for module feedback management.

There is also fastfeedback.io. Its main goal is not to provide a service but to teach a course, but it still does the work and can be used on any website. Dung got to know of this site after taking that very course.

3.3 Problems with Existing Solutions

However, there still remain many problems. Firstly, these services do not allow custom design for the entire section. Their own design is nice, but it is not customisable enough to be consistent with the main site design. For example, we do not think Disqus' design below can go along with NUSMods' design well. Some of the consistencies that anyone can notice with the naked eye without inspecting the source, written by a person who has not even taken an official course in digital and UI/UX design:



Additional Information

✓ Included in Semester 1's Module Planning Exercise

✓ Included in Semester 2's Module Planning Exercise

✓ Has S/U option for Undergraduate students only

Workload - 10 hrs

Review and Discussion

Mario Lorenzo • 3 years ago

Taken in AY18/19 Sem 2

Lecturer: Dr. Low Kok-Lim

Grading Scheme: 4% Studio Attendance, 2% Studio Participation, 4% Reflection

Hi There!

We would like to encourage everyone who enjoyed using NUSMods to contribute back to the community by writing reviews for modules that you have taken before. Your efforts will go a long way in building up a vibrant and rich NUS community.

Please note:

1. Because the experience of each module will differ according to the professor teaching the module, at the start of your review, please state the semester taken and the

- Fonts: NUSMods uses the standard system font stack (Segoe UI on Windows, San Francisco on macOS, etc.), while Disqus uses Helvetica Neue.
- Colours: While Disqus allows customisation of the primary colour (with the `--publisher-color` CSS variable), other colours do not have the same privilege. A naked eye can tell the body text colour of NUSMods and Disqus are completely different (NUSMods' is "greyer"). A source inspection shows NUSMods' body colour is #60707a, while Disqus uses all sorts of grey shades: #2a2e2e, #687a86, #657c7a, etc. Furthermore, the dropdown by Disqus does not follow `--publisher-color` at all, and still uses normal blue (#2e9fff).
- Spacing: While NUSMods buttons are quite spacious and have large paddings, Disqus buttons (such as the social share buttons or dropdown items) are narrower.
- Borders: NUSMods' borders are normally 1px wide and are pretty faded, Disqus borders are 2px wide and filled with a far stronger colour.
- etc. and etc.

As a person making static sites myself, we would like a service which is fully customisable and contains all the features we think a commenting service should have. We decided to build this service in Orbital, leading to the project scope and aim described above.

4 Features

4.1 Some Vocabulary Definitions

A user can have a lot of websites under different domains, each website needs several different comment sections (each for a blog post, for example). Therefore, we will define specific terms for the application, which will be used throughout the project.

A *site* is a collection of *pages*. Each page may correspond to a blog post, an announcement, etc. and will have one comment section dedicated to it. Each site is identified from its domain, and all of its pages have to be under that domain.

For example, assume that a user has two websites at `foo.com` and `bar.com`. `foo.com` hosts his 20 blog posts and `bar.com` hosts his portfolio. Then

- He will have two *sites* configured, one for `foo.com` and one for `bar.com`.
- For `foo.com`, he will have 20 *pages*, each for one of his posts.
- For `bar.com`, he will have only one page for his portfolio.

4.2 Zero-configuration HTML Embed for Comments

In its simplest form, the service should be able to produce a URL for each page so that users can embed them into their websites, with as few clicks required as possible. No matter how extensible the service may become, it is absolutely crucial that it can do the most basic form of its job and does it well.

The embed is done by `<iframe>` – it has been supported by all browsers since basically the birth of the Internet, so as long as the user is using web technologies, the embed will work. We plan to render the HTML on the server before serving it (instead of performing data fetching on the client), and cache the HTML served as much as possible to optimise performance.

4.2.1 User Stories

From the goal of this feature, we plan the typical user workflow for this feature to be:

- As the user logs in, he is redirected to a dashboard where all of his sites are listed. In the dashboard, there is a button to add a new site.
- Upon clicking that button, the user is asked for the site name and the site domain. They are for identification only, so the user knows which ezkomment *site* corresponds to each of his websites.
- After submitting, the user is redirected to the dashboard of that site. There is another button to add a new page.

- The user will then be asked for a page ”title” and page URL, these are also for identification purposes only. Typically, the page title is the blog title.
- The user is then redirected to the dashboard of that specific page, where he has a `<iframe>` tag with an embed URL ready to be used.

4.2.2 Programmatically Creating New Pages

Power users may want the process to be automated, e.g. whenever a new page is added in their CMS. For creating new pages, we do plan to support this. However it is not among our core features, and we will discuss more on this in later versions of this README.

However, we do not plan to support programmatically creating new sites. Since a new site typically means a new website or web application, we do not think this feature is practical. We expect that users have to create new sites by visiting the web application (or reverse-engineer the app to find a way to do this).

4.3 How the Comment Section Works

4.3.1 Everything is Public

We plan to make these comment sections public. In other words, everyone who can see that comment section can write comments on it. Both the `GET` and the `POST` methods for the comments will not require any authentication.

If a user wants the comment section to be limited to a particular set of people (e.g. only people from his team can comment), he can put it inside a protected page. Although the endpoints to access the content are public, the URL is randomised enough that it cannot be guessed.

Users may find this not secure enough, as their teammates can always leak the URL to an outsider. However, we do not plan to address this issue, since even if the URL is protected, the insider can still leak by other methods (e.g. screenshotting). Adding authentication would mean unnecessary JavaScript sent to everyone, which is against the principle of this application.

However, if a user *truly*, *really* want to integrate an authentication system, he can always use the API routes. Instead of exposing those routes directly, he can hide them inside his own protected internal API. By doing so, he will also hide the randomised URL from everyone, thereby resolving the issue of leakage to outsiders.

4.3.2 Sending Comments (updated)

We think that although the comments are all public and authentication is not needed to post new ones, there should still be a way to identify the commenter. It will be optional: we ask for the name of the commenter along with the required comment body.

If the name is provided, it will be shown to the ezkomment user, who may use that information to decide whether the comment should be approved (see the section on Moderation below). If the name is provided, it will be displayed publicly instead of the placeholder ”Anonymous”.

4.3.3 Comment Formatting

We plan to support Markdown for all comments from embedded URLs. In other words, we compile the comment content as Markdown to HTML on the server before serving. Since it is likely most comments will not need Markdown formatting features (real-life example: Reddit), we will not add a Markdown editor since it will only increase the unnecessary JavaScript footprint, but only add a little help text ”Markdown is supported”.

Many users will probably not be happy with that, however, they can always introduce their own editor by customising the comment section (see the customisation section below). We restrict the amount of JavaScript introduced by us but never impose any restrictions on custom JavaScript by users.

Some users may find that they need custom Markdown features/plugins or use their own Markdown alternative. In that case, the API route will need to be used; we plan to send the comment body *verbatim* without any formatting, so users can process that string in any way they want. However, using this method, users need to be aware of and guard against XSS attacks, since the comment body is not escaped. See the API section for more details.

4.3.4 Replies to Comments (updated)

In the previous iteration of the README, we specified that we did plan to support replies to comments on a number of occasions. However, after thinking more about this, We find that there are many reply-less discussions on the Internet that work well (one notable example is GitHub issues/PR), and implementing replies will take a lot of effort. Therefore, this feature has been dropped.

4.3.5 Editing (updated)

After thinking about this feature, we found that it is possible to implement it in a way that resolves the current issues. In a nutshell, after submitting, the user is issued a JWT, which expires after 5 minutes. This token is used to identify the commenter, hence can be used to support editing. If during that 5 minutes, the comment has not been approved, the user can edit the comment.

However, we found it to introduce extra complexities to the app, and considering the current situation, we do not find it to have high priority. Therefore, we do not plan to implement this feature, but we will revisit the decision if we have more time.

4.3.6 Pagination

We will not support pagination, again for the reason of optimising client-side JavaScript. We consider that the majority of comment sections will only have up to 10 comments, so pagination is not necessary for them. Hence, supporting paginations is like writing complicated JavaScript workarounds to support dinosaur browsers such as Internet Explorer: increasing the page load by a large amount just to support a small number of use cases.

However, in the API route, we do plan to support pagination, since client-side JavaScript no longer makes sense in this context. See the API section for more details.

4.4 Moderation Features

When a comment section is public, if it is on a popular website (e.g. in a newspaper), undoubtedly there will be spam and hate speech. To mitigate this issue, we plan to support moderation features, where the site owner acts as a moderator to decide which comment to keep and which to remove.

If this feature is turned on, whenever a new comment is posted, it does not show up in the public comment section right away, but instead, it is directed to a ”pending” section in the ezkomment app. The site owner can then approve or remove any comments in the pending queue, and only after a comment is approved will it appear in the public feed.

Even after approval, the site owner can still remove comments. Removed comments are gone forever and not recoverable, therefore we will add a big warning to the UI to prevent the user from doing so by accident.

By default, this feature is turned *off* (i.e. auto-approve is turned *on*), since that is how most current solutions work. To ensure that the user knows of this feature, we will display a big ”info” banner in the UI if the feature is turned off. Also, even when auto-approve is on, we will still allow the deletion of ”auto-approved” comments.

4.4.1 User Stories (updated)

In the page dashboard, there are two sections, one for pending comments and one for approved comments. If auto-approve is off, a typical workflow is as follows:

- User checks the notifications of any pending comments.
- User goes to the page dashboard of the said comment section.
- There are two buttons, approve and disapprove (delete). The user reads the name of the commenter (if provided) and clicks the appropriate button. If the comment is approved, it will be displayed publicly, otherwise, it is forever lost.
- For approved comments, the deletion button is also visible for the user to use. Again, if the user clicks this, the comment is forever lost.

4.4.2 Notifications

We will display a notification pane in the navigation bar of the app, so the user always knows the latest updates to their comment sections, whether this feature is enabled or not. Especially when the feature is enabled, it is necessary for the user to check the pending queue once in a while so as not to effectively prevent all new comments from ever being displayed publicly.

4.4.3 Scope

Users can set this feature per page. Pages that may involve heated discussions may need this feature, while more ”neutral” pages may not need it, hence we do not think this feature should be set site-wide.

4.4.4 Integration with the API

We do not plan to integrate this with the API at the moment. The API is not designed to replace the UI, instead, it is designed for users to post and get comments and metadata for any particular comment section, so as to transform and customise it in any way they want.

4.5 Customisation

A default styling for the comment sections will be provided, which will follow the overall design principles of the ezkomment front-end application. However, obviously those design principles cannot fit many websites, for the UI/UX reasons that we have described above. Therefore we will allow the user to completely customise the HTML and CSS of the comment section.

4.5.1 Customisation is for Sites, not Pages

By UI design principles, all pages under the same website should share the same design system for consistency. To encourage this, we only allow customisations to be done site-wide, i.e. for all pages under a site.

If a user *really* wants to hack the system and have different styling for each of his blog posts, he can always have two different ezkomment ”sites”. However, it will require many clicks and keystrokes, and the bad UX of that is intentional: we want to discourage users from doing so.

4.5.2 Customise Everything (updated)

The specification of this feature is still under discussion and consideration. Therefore, it might be modified in the next Milestone README. However, we expect it to be quite stable by now and it should not change as much as from Milestone 1 until now.

We plan to allow users to customise the app by the whole code of the HTML sent to clients, even from the `<!DOCTYPE html>` tag. In that, users can decide how and where to put stylesheets in, how to add custom JavaScript libraries, everything.

In essence, the user will be allowed to modify a HTML file, in its entirety, from the usual opening `<!DOCTYPE html>` to the closing `</html>` tag. Of course, the user is also allowed to add any scripts and styles he wants, just like how we typically developed the web from the HTML-CSS-JS days.

However, the users need to add the `data-ezk` attribute to a number of elements. This attribute is used to identify each part of the comment section, so that the app can populate it with the appropriate information, or handle form submission correctly.

Currently planned `data-ezk` attribute values:

- `comments`: The wrapper of the comments. Whatever HTML inside this wrapper is treated as *one comment*. Therefore, if a page has 10 comments, the `innerHTML` is replicated 10 times inside this wrapper. That may sound confusing, see the example below.
- `comment-author`: The author of the comment. When the comment section is populated with data, the element with this `data-ezk` value is populated with the author's name (`element.textContent = authorName`).
- `comment-content`: The content of the comment. When the comment section is populated with data, the element with this `data-ezk` value is populated with the content of the comment (already compiled from Markdown to XSS-safe HTML).
- `comment-date`: The date of the comment. When the comment section is populated with data, the element with this `data-ezk` value is populated with the date of the comment (`element.textContent = date`). The date format is not yet decided, but we will enforce one universal format for everyone. If the user wants to change the format he can always use custom JavaScript to do it.
- `form`: The form to submit a new comment. When submitted, this form will send a request to the back-end to submit the comment. **Required to have descendants with the `data-ezk` value `form-author` and `form-content`.**
- `form-author`: The author field of the form. When the form is submitted, the value of this field is used as the author name of the comment.
- `form-content`: The content field of the form. When the form is submitted, the value of this field is used as the content of the comment. Support Markdown.

An example: Assume the following is the user-provided HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <!-- custom CSS, JS, etc. -->
  </head>
  <body>
    <div class="container">
      <div data-ezk="comments">
        <div class="comment">
          <div class="metadata">
            <div class="author" data-ezk="comment-author"></div>
            <div class="time" data-ezk="comment-date"></div>
          </div>
          <div class="text" data-ezk="comment-content"></div>
        </div>
      </div>
      <form data-ezk="form">
        <input placeholder="Display name" name="name" required data-ezk="form-author" />
        <textarea placeholder="Content" name="content" required data-ezk="form-content"></textarea>
        <div class="form-bottom">
          <span>Styling with Markdown is supported</span>
          <button type="submit">Post</button>
        </div>
      </form>
    </div>
  </body>
</html>
```

with the following comments:

```
[
  {
    "author": "John Doe",
    "content": "This is a comment",
    "date": "2018-01-01"
  },
  {
    "author": "Jane Doe",
    "content": "This is another comment *with markdown*",
    "date": "2018-01-02"
  }
]
```

Then, after the HTML is sent to the browser and the necessary scripts (automatically added) are loaded, the resulting HTML is

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="UTF-8" />
    <!-- custom CSS, JS, etc. -->
  </head>
  <body>
    <div class="container">
      <div data-ezk="comments">
        <div class="comment">
          <div class="metadata">
            <div class="author" data-ezk="comment-author">John Doe</div>
            <div class="time" data-ezk="comment-date">2018-01-01</div>
          </div>
          <div class="text" data-ezk="comment-content">
            <p>This is a comment</p>
          </div>
        </div>
        <div class="comment">
          <div class="metadata">
            <div class="author" data-ezk="comment-author">Jane Doe</div>
            <div class="time" data-ezk="comment-date">2018-01-02</div>
          </div>
          <div class="text" data-ezk="comment-content">
            <p>This is another comment <em>with markdown</em></p>
          </div>
        </div>
      </div>
      <form data-ezk="form">
        <input placeholder="Display name" name="name" required data-ezk="form-author" />
        <textarea placeholder="Content" name="content" required data-ezk="form-content"></textarea>
        <div class="form-bottom">
          <span>Styling with Markdown is supported</span>
          <button type="submit">Post</button>
        </div>
      </form>
    </div>
    <!-- and additional JavaScript added by ezkomment -->
  </body>
</html>
```

We need the said additional JavaScript to support the following:

- Data fetching: We need a data fetching strategy close to how SWR works so that the UX is enhanced.
- Data parsing and populating: We are not doing React here, so we need to manipulate the HTML with pure, vanilla JavaScript.
- Send a message to the `<iframe>` parent about the document height, so that the `<iframe>` can be rendered correctly (we will not go into details, but you can see more related info).

As of 25 June, this is the entirety of that additional JavaScript.

```
const apiURL = "/api/temp"; // change this URL according to site ID and page ID
let hasFocus = false;
let isVisible = false;
let blockValidate = false;
let COMMENTDIVCONTENT = "";

function handler() {
  validate();
  blockValidate = true;
  setTimeout(() => (blockValidate = false), 1000);
}

function initialise() {
  document.addEventListener("visibilitychange", () => {
    isVisible = document.visibilityState === "visible";
    if (hasFocus && isVisible && !blockValidate) handler();
  });
  window.addEventListener("focus", () => {
    hasFocus = true;
    if (hasFocus && isVisible && !blockValidate) handler();
  });
  /** @see {@link https://stackoverflow.com/a/42308842} */
  window.addEventListener("message", event => {
    if (event.data === "FrameHeight") {
      const height = Math.max(
        document.body.scrollHeight,
        document.body.offsetHeight,
        document.documentElement.clientHeight,
        document.documentElement.scrollHeight,
        document.documentElement.offsetHeight
      );
      event.source.postMessage({ EzkFrameHeight: height }, "*");
    }
  });
  document.querySelector("[data-ezk=form]").addEventListener("submit", onFormSubmit);
}

async function validate() {
  const comments = await fetch(apiURL).then(res => res.json());
  const commentsDiv = document.querySelector("[data-ezk=comments]");
  if (COMMENTDIVCONTENT === "") COMMENTDIVCONTENT = commentsDiv.innerHTML;
  commentsDiv.innerHTML = "";
  comments.forEach(({ author, text, date }) => {
    const commentDocument = new DOMParser().parseFromString(COMMENTDIVCONTENT, "text/html");
    const name = commentDocument.querySelector("[data-ezk='comment-author']");
    const content = commentDocument.querySelector("[data-ezk='comment-content']");
    const time = commentDocument.querySelector("[data-ezk='comment-date']");
    if (name) name.textContent = author;
    if (content) content.innerHTML = text; // already rendered safely on server side
    if (time) time.textContent = date;
    commentsDiv.innerHTML += commentDocument.body.innerHTML;
  });
}

async function onFormSubmit(event) {
  event.preventDefault();
  const authorField = document.querySelector("[data-ezk='form-author']");
  const commentField = document.querySelector("[data-ezk='form-content']");
  const comment = { author: authorField.value, text: commentField.value };
  authorField.value = "";
  commentField.value = "";
  await fetch(apiURL, {
    method: "POST",
    body: JSON.stringify(comment),
    headers: { "Content-Type": "application/json" },
  });
  await validate();
}

window.addEventListener("load", () => {
  initialise();
  validate();
});
```

This script will probably need some work to ensure it does not clash with any possible user-provided scripts, to capture errors, etc. But that is how it basically works.

With this, we believe the user already has very good control of how the whole thing looks and works. In fact, he can add custom scripts to render a customised Markdown editor, he can even add scripts to mine cryptocurrency in his users' browsers etc (not that we support it).

4.5.3 Dark Mode (updated)

The embed URL will have a query parameter `isDark`. If it is used, the comment is delivered in dark mode. The `<html>` element would have a class `.dark`, and of course, the user can control how their comment section looks in dark mode by modifying his `<style>` accordingly.

```
<!-- /embed/:siteId/:pageId -->
<html>
  <!-- everything -->
</html>
```

```
<!-- /embed/:siteId/:pageId?isDark=1 -->
<html class="dark">
  <!-- everything -->
</html>
```

4.6 Application Programming Interface

Power users can also ignore the entire embed and customisation thing and use the API to handle comments, for the ultimate™ customisation capability. The API is designed for two purposes: fetching comments on a page and posting comments to it (for it to be sent to the pending queue).

We plan to make the API backwards compatible with version-based endpoints (if we ever make it to version 2). Therefore, the currently planned API endpoint is

```
https://ezkomment.joulev.dev/api/:version/comments/:siteId/:pageId
```

This URI may be changed in later revisions of README.

4.6.1 Fetching Comments for a Page

The `GET` method can be used to fetch all or part of all comments for a particular page.

```
const res = await fetch("url");
const comments = res.ok ? (await res.json()) : [];
console.log(comments);
```

The output is a JSON which includes all comments and their metadata (name, timestamp).

```
// This may change when we actually implement it
{
  "siteName": "somesitename",
  "pageId": "somepageid",
  "commentCount": 12,
  "comments": [
    // { name, content, time }...
  ]
}
```

The comment content is sent back as-is, i.e. there are absolutely no transformation processes happening in between. Although attackers can use this to attack XSS-vulnerable websites, we are determined not to escape anything in the content string, so that *truly* power users can transform it with whatever library and/or method they want, completely and absolutely freely. It is the job of the user not to fall to XSS attacks (for example, just do not blindly pass the string to `eval()` or put it to `dangerouslySetInnerHTML`).

Extra options can be passed to the URL as queries. Invalid queries and queries with invalid values are ignored. For now, the following options are planned:

- `limit` and `page` as numbers, for **pagination**. For example `url?limit=10&page=2` means the API will return the 11th to the 20th comment, sorted by newest first.

4.6.2 Posting New Comments to the Pending Queue

The `POST` method can be used to post new comments. Depending on the page configuration, it may go to the pending queue or get straight to the approved comment list.

```
const name = document.getElementById("form_name").value;
const content = document.getElementById("form_content").value;
const res = await fetch("url", {
  method: "POST",
  body: JSON.stringify({ name, content })
});
if (res.ok) showSuccessNotice();
```

Requests with invalid body are ignored. Currently we do not plan any extra options for this method.

5 Application Design and Implementation

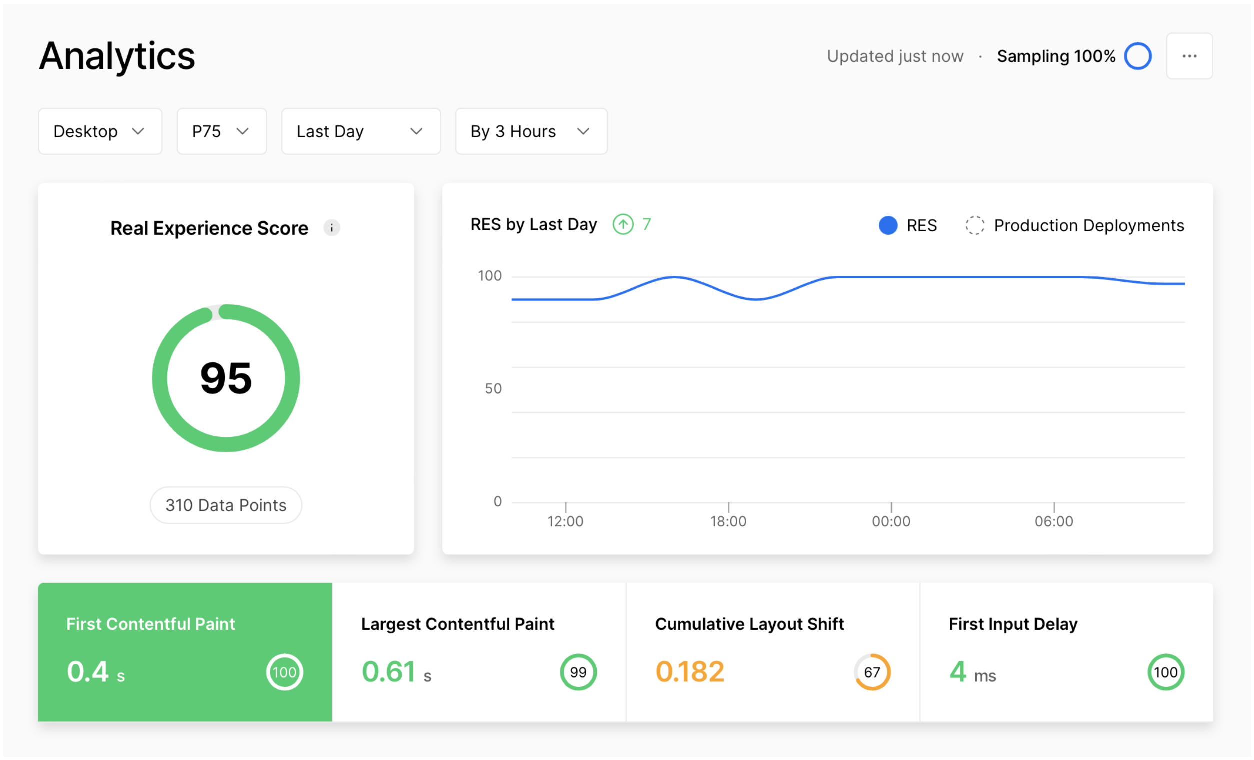
5.1 How the App Works, In a Nutshell (updated)

5.1.1 Technology Stack

The front-end is built using Next.js, which is a popular React framework. Especially after version 12 was released last year, it is the most popular React-based framework right now in the medium and is almost as popular as Vue and Angular (source). After trying it out, we found it to be very powerful and have excellent DX.

While Next.js is a primarily front-end-based framework, its serverless API routes are well enough for our use case for a back-end, therefore we decided that the entirety of the app will be implemented in Next.js.

Vercel is used for hosting, because you simply cannot say Next.js without Vercel. While the front-end is very performant with image optimisations working exceptionally well, the back-end is also very fast, and we have seen no cold-boot time at all. Moreover, Vercel offers insights into app performance, which helps us a lot in improving the app.


Analytics
Updated just now · Sampling 100%
Desktop
P75
Last Day
By 3 Hours
Real Experience Score
95
310 Data Points
RES by Last Day 7
RES
Production Deployments
100
50
0
12:00
18:00
00:00
06:00
First Contentful Paint 0.4 s 100
Largest Contentful Paint 0.61 s 99
Cumulative Layout Shift 0.182 67
First Input Delay 4 ms 100

We decided to use TypeScript for the entire project for easier management. After using TypeScript, we simply cannot imagine how we would use pure JavaScript for any serious projects ever again. Typescript's type system ensures correctness and provides IntelliSense when using Visual Studio Code, which is absolutely vital for our development speed.

On the back-end side, Firebase Admin SDK is used. It is relatively easy to use, and we think it is a good choice for the project. Naturally, we use Cloud Firestore as the database and Firebase Storage for storage. However, we do not implement the whole authentication system with Firebase Admin, instead, we outsource most of it to the Firebase Client Library to have the best security.

On front-end, we use Tailwind CSS for styling. Compared to the CSS/Sass mess, we prefer the HTML mess. Also, since Tailwind CSS has a very low bundle size on production compared to, say, Bootstrap, it helped make the app very performant (in the analytics screenshot above, you can see FCP and LCP scores are very good).

For data fetching on the client-side, we decided to use SWR instead of the normally used Fetch API or Axios. The first reason is that SWR is like a part of the Vercel family, and integrates very well with Next.js. Another reason is that it has a pretty clever revalidation technique that makes the user experience on the front-end very good.

5.1.2 Communication between Front-end and Back-end

As usual for web applications, the communication between two ends of the app is done by RESTful API routes. The data format is JSON whenever possible.

To authenticate, we retrieve the user's ID token from the Firebase client library. This ID token is sent in the request header to identify the user.

5.1.3 Authentication Providers

We will only support authentication with third-party OAuth providers (currently GitHub and Google), since we think password-based authentication is unnecessarily complex and is now a thing of the past. Passwordless ”email link” authentication was also planned, but we no longer plan to support it.

5.1.4 Front-end Page Structure

- `/`: The landing page for the whole app, with introduction and core features showcase
- `/orbital`: Everything related to Orbital 2022, including all versions of the poster, video and README
- `/docs`: The documentation pages for the application
- `/auth`: The unified page for all authentication-based actions
- `/app`: The main application pages
- `/app/dashboard`: The dashboard, listing all sites
- `/app/new`: Adding a new site
- `/app/account`: Account settings
- `/app/site/:siteName`: The dashboard for site with name `siteName`
- `/app/site/:siteName/customise`: The page to customise all comment sections for the site
- `/app/site/:siteName/settings`: Settings for the site with name `siteName`
- `/app/site/:siteName/:pageId`: The dashboard for page with ID `pageId`
- `/app/site/:siteName/:pageId/settings`: Settings for the page with ID `pageId`

5.1.5 Back-end API Endpoint Structure

- `/users/:uid`: The endpoint to get, update and delete user using their ID
- `users/:uid/photo`: The endpoint to upload user's photo

- `/sites/:siteId`: The endpoint to get, create, update and delete site, using its ID
- `site/:siteId/icon`: The endpoint to upload site's icon.
- `/pages/:pageId`: The endpoint to get, create, update and delete page, using its ID

5.2 User Interface Design

The overall design is heavily inspired by Vercel. @joulev is very impressed by the UI design of the Vercel website, and decided to implement that himself. Of course, dark mode is fully supported.

The logo was drawn with Figma. SVG versions can be found in the repository.

We decided to use the default Tailwind CSS colour scheme, since it is sufficient for our need. `indigo` variant is the primary colour, with `indigo-500` (#6366f1) being the main colour and features in the logo. The background/foreground colours are based on the `neutral` scheme. Other accent colours are `red` (for danger actions), `emerald` (for success status), `amber` (for warnings) and `sky` (for information banners).

We also use the default Tailwind CSS spacing scale, and pick all spacing entries whose values are divisible by 3. Hence, most spaces in the application are a multiplier of 0.75rem.

5.3 User Experience (updated)

We take a *lot* of effort to improve the user experience on this site.

We do not have an existing ”user base” to ask for opinions, like how UX designers typically do. A/B Testing and similar approaches are also out of the question for this project. However, Dung is fairly critical of the user experience of every app he has ever used, hence he criticises this app to the best of his abilities, then improves the app based on that ”self-feedback”.

We also asked for feedback from Zhu Hanming, the UI/UX workshop tutor, and improved the app accordingly.

As expected, Firebase is much slower compared to a self-hosted system, so the authentication section is relatively slow compared to a typical company website and we cannot really do anything against it. However, for the sections where we can improve, we have pretty much done everything we can.

Some steps we have taken:

5.3.1 Everything Is Static

All pages are statically generated and cached to the CDN around the world. Yes, *including protected pages* (although the generated pages only have loading screens and no sensitive information). Not only does this massively improve load time, Next.js's prefetch strategy makes the router super fast too, and navigating between pages feels instant.

5.3.2 Everything Is Responsive

Needless to say, responsive design is absolutely crucial for a modern web application. We take that seriously and always test our application on all viewports, from super small iPhone SE's to ultra-wide screen sizes.

5.3.3 Strategies for Fetching and Storing Data

The back-end is designed in such a way that it is possible to only fetch the data that the front-end needs. That optimises the speed of the HTTP connections between the front-end and the back-end.

In the front-end, we use SWR and the React Context API to store the fetched data. This has a lot of advantages:

- The React Context makes it possible to reuse data across different pages, therefore if you navigate from `/app/site/mysite` to `/app/site/mysite/settings`, you will not see a loading screen.
- SWR's nice revalidation strategy makes the front-end data always up-to-date without the user even knowing.

We try to reduce the loading screens as much as possible (no one wants to see spinners). Right now, you will likely only see the loading screen when you are not yet authenticated or when you visit a site/page for the first time. All other data fetching is done in the background.

5.3.4 Handling Router When Changing Site Name

Since the site name is used in the URL

```
https://ezkomment.joulev.dev/app/site/:siteName
```

whenever a user changes the site name in the settings, we need to update the URL. Normally this is not easy, since the fast revalidation by SWR makes the page redirect to a 404 page before the redirect even has a chance to happen. You can solve it by manually updating the URL with `window.location`, but since it does not use the Next.js router, it is slower and the website feels 1990s again.

Therefore, we had a look at how some popular applications handle this. Vercel changes the URL without even giving the impression that the page has been reloaded, and that is what we aim for. In the end, we use a temporary URL query to handle this (similar to how Vercel does it) and remove that URL query when it is no longer necessary. It works and although you will see a loading screen again (since you are visiting the new site again), the overall user experience is enhanced a lot.

Interested people can see how we implemented it in here, here and here.

5.4 Search Engine Optimisations

In pull request #66, we have added the necessary SEO tags to every page that needs them, and also set up a process to

- automatically generate new `og:image` image URL for new pages based on a common skeleton, at build time, and
- easily add SEO tags to new pages, as well as new page types.

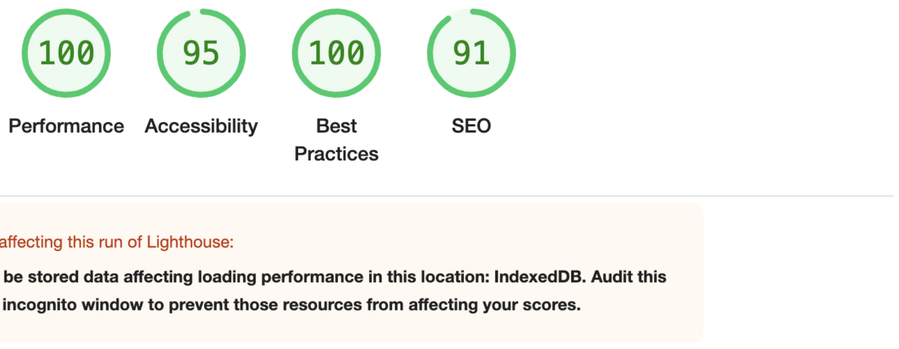
Although we have not enabled the site to be crawled for search engines as of now, the tags are now effective when sharing the pages over social media.

5.5 Lighthouse Audit (updated)

Lighthouse is a popular tool for evaluating the performance of a website. We conduct Lighthouse score checks frequently and also use Vercel analytics to monitor, from there we can make suitable improvements to improve the whole application.

As of right now, the website has a very good score for these metrics. Aside from the photo above of real-time Vercel analytics, the following are the Lighthouse scores as of 24 June 2022:

- Performance: 100 for desktop, 99 for mobile
- Accessibility: 95
- Best Practices: 100
- SEO: 91


100 Performance

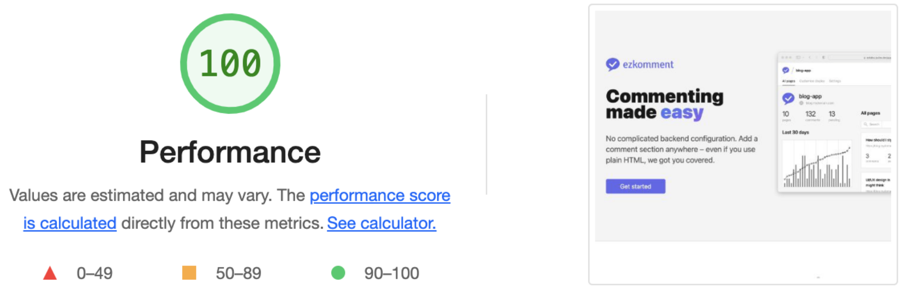
95 Accessibility

100 Best Practices

91 SEO

There were issues affecting this run of Lighthouse:

- **There may be stored data affecting loading performance in this location: IndexedDB. Audit this page in an incognito window to prevent those resources from affecting your scores.**


100

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

ezkomment
Commenting made easy
No complicated backend configuration. Add a comment section anywhere – even if you use plain HTML, we got you covered.
Get started

5.6 Security (updated)

Authentication workflow is mainly done by the Firebase client library. With it being one of the most popular libraries of its type, with about 1.3 million downloads per week, I think it is safe to say this step is as secure as we can possibly get. Especially compared to self-made authentication patterns which may fall to many well-known attacks, such as Cross-Site Request Forgery attacks if cookie-based authentication is used.

For other interactions between the front-end and the back-end, the front-end authenticates by attaching the short-lived user ID token to the request header, and the back-end would decode this token to verify its authenticity. This is secure, and given that the token is short-lived, there are few chances for any potential attackers.

That said, security issues may still show up, a notable example being issue #137. We will continue to closely monitor the whole application to ensure such vulnerabilities are addressed quickly.

5.7 Software Engineering Practices

5.7.1 Repository Structure (updated)

Since the back-end has been decided to be based on Next.js' API routes, we have ditched the monorepo system. Now everything is developed in a single JavaScript package.

Even so, we still organise the repository in such a way that clearly differentiates back-end code and front-end code. This is the current repository structure:

```
.
├── __tests__              # Jest tests
│   ├── client
│   ├── misc
│   └── server
├── client                 # front-end source code
├── config                 # Configuration files (if any)
├── constants              # Some misc constants
├── docs                   # Documentation for users, rendered to /docs
├── firebase-rules         # Rules for Firebase products (required to start the Firebase Emulator)
├── misc                   # Used in both front-end and back-end
├── pages                  # All pages, mandatory by Next.js
│   ├── api                # API endpoints, mandatory name, for back-end
│   └── ...others          # All other pages are for front-end
├── public                 # Static files, mandatory by Next.js
├── sample                 # Testing mock data (if any)
├── server                 # back-end source code
└── types                  # TypeScript typings
    ├── client
    └── server
```

5.7.2 Contributions and Feature Branching

The repository is developed mainly on two branches, `main` and `prod`.

`prod` is the production branch. Currently, the main domain https://ezkomment.joulev.dev is deployed from this branch. It is reserved for production-ready code only, and developing directly on this branch is not allowed.

`main` is the development branch, also the default branch of the repository. The front-end is deployed to https://ezkdev.joulev.dev. This is where we normally push our code or make pull requests, during the development process. Dependabot updates are also configured to be made against this branch.

We also follow the practice of feature branching. For most work, it will be done on a separate branch with a descriptive name, and then a pull request is made to merge the branch to `main`. A typical workflow is as follows:

- Developer creates a new branch
- Developer commits to it
- Developer opens a pull request
- Developer fixes any merge conflicts

- Now, if the code changed includes the code that another person is in charge (i.e. @joulev for front-end and @VietAnh1010 for back-end), it is necessary to request a code review from that person. Pull requests should only be merged when that person has agreed to.
- In any case, the developer should not merge the pull request immediately, even when code reviews are not required. There should be a cool-down period of a few hours, before the developer self-reviews the whole pull request, makes any further changes and finally merges the pull request.

We will use the ”Rebase and merge” method as the merging method for all pull requests.

However, feature branching is not strictly enforced. In cases where there are few chances for bugs, separate branch and pull requests are not required. We have seen in repositories where feature branching is strictly enforced, a developer needs a separate branch and pull request just to update the copyright year value in the footer. Such pull requests would typically be merged immediately without reviewing code and waiting for CI pipelines to pass, hence losing the purpose of pull requests. We consider it to be way too redundant and will only spam the notification of all developers watching the repository (similar to how `+1` messages are discouraged in GitHub issues).

Therefore, for smaller commits, typically about less than 50 lines changed (excluding machine-generated files such as lock files), feature branching is not needed and the developer can always push directly to `main`. After all, `main` is not production-ready and if anything goes wrong, `git revert` and `git reset` are available.

The contribution process is described in slightly more detail in the CONTRIBUTING file.

5.7.3 Continuous Integration and Continuous Deployment (updated)

Thankfully, Vercel automatically takes care of the deployment process for every commit in the repository. Therefore, the front-end section has always had CD since it was first deployed.

As we decided to mitigate from Express.js to Next.js for back-end implementation, the back-end section also has CD.

CI pipelines are done by GitHub Actions, and currently it involves running ESLint and unit tests for the whole application.

5.7.4 Testing (updated)

Testing is done by Jest whenever possible, due to its simplicity, ease of use, high speed and good developer experience. Test results are then uploaded to Codecov.

Back-end tests also involve running Firebase emulator.

As of 24 June 2022, the code coverage stands at 94%, which we think is acceptable.

6 Timeline (updated)

6.1 Currently Already Implemented Features

- [MS1] The main interfaces of the front-end app
- [MS1] Public pages of the front-end
- [MS1] The front-end has been deployed
- [MS1] Firebase (client-based) authentication for the front-end app
- [MS2] Integrate some back-end routes to support authentication
- [MS2] Create-Read-Update-Delete for sites
- [MS2] Create-Read-Update-Delete for pages
- [MS2] Basic demo for how a comment section looks like in a plain HTML page

6.2 Looking Back to Milestone 1

This is the plan we specified in the Milestone 1 README:

- By 3 June: set up a basic deployable and usable back-end with cookie-based authentication features, as well as deploy it
- By 7 June: finish integrating authentication between front-end and back-end
- By 14 June: implement site management features (users can add new sites, can edit existing sites, can delete existing sites)
- By 20 June: implement page management features (users can add new pages to sites, can edit and delete existing pages, have a embeddable `<iframe>` to use with placeholder comments)
- By 27 June (Milestone 2): make the comment sections *commentable*, with updated statistics metrics in the site dashboard.
- Also by 27 June: find resolutions for all known issues in the app design mentioned above.

As you can see, we have completed the first four items and the last item on the list. The fifth item is not yet implemented at all. So we did not meet the deadline.

That said, we have almost fully implemented all application pages in the process, from `/app/dashboard` to `/app/site/:siteName/:pageId/settings`. They are now fully functional and we do not expect to spend much time on them from now onwards. One can say we have laid out the foundation of the application, and with the most critical infrastructure now completed, adding more features to it should not take too much time. Therefore, although we could not finish all planned items in time, we are optimistic that the application will be completed as planned.

6.3 Going Forwards to Milestone 3 (25 July)

- By 30 June: front-end design the default styling for the comment section; back-end implement the routes needed to serve the comment HTML
- By 7 July: implement the moderation features
- By 17 July: implement the customisation features
- By 20 July: implement the public API (this is entirely the back-end, and I actually think we can get this done within a day since the tools are already there)
- From 20 July onwards: polish the system, add minor features such as search bar/filtering
- Incrementally from now onwards (also past Milestone 3): populate the public documentation at `/docs` with actual documentation for users on how to use the app

7 Problems Encountered (updated)

In the technical aspect, the project progresses fairly smoothly (to a surprising extent). In fact, we have not actually asked our mentor for any technical assistance since we have been able to resolve all issues by ourselves.

However, in the first half of June, we had quite a few problems with miscommunication between the two members, thereby leading to a significant delay in the project:

- Basic backend: by 3 June, actual date of completion **9 June**
- Auth integration: by 7 June, actual date of completion **18 June**
- Site management: by 14 June, actual date of completion **21 June**
- Page management: by 20 June, actual date of completion **23 June**
- Comment *commentable*: by 27 June, **not completed**

Although in the end, we were not too much behind schedule, from 18 June to 24 June both members had to work for many hours per day to bring the massive delay of 11 days down to how it is currently.

Thankfully, the workload for the next phase turns out to not be that packed, thus this delay did not cause too much damage to the whole project. That said, we are determined to resolve any issues and questions early so that this problem will not appear again.