ezkomment

Orbital 2022 Milestone 1 README

Vu Van Dung Nguyen Viet Anh

27 May 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 in 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 it 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 on 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 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 in any websites. 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 naked eye without

inspecting the source, written by a person who has not even taken an official course in digital and UI/UX design:

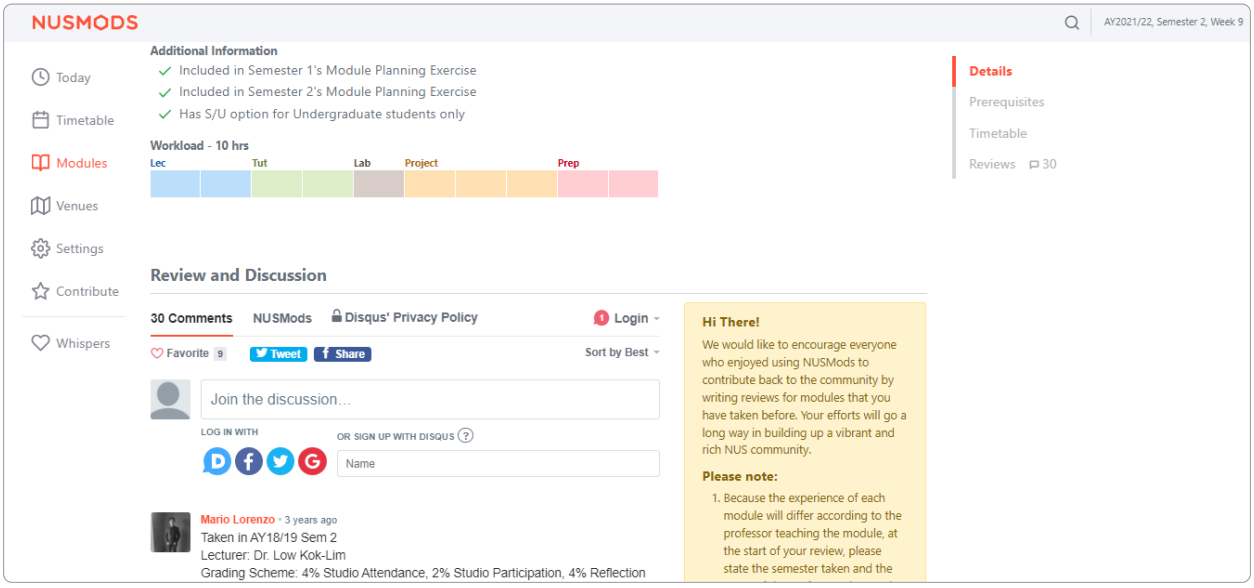

Figure 1: NUSMods screenshot as of 14 March 2022, with Disqus loaded.

- 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 Planned 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.

A way to perform this optimisation may be to use On-demand Revalidation, which has been supported by Next.js since version 12.1. Every time a comment is sent to the server, it triggers a revalidation for the page, hence the server only actually renders the page when the whole app is built or during the few occasions that someone submits a comment. The page that everyone receives is completely static, making it is very performant.

Since the HTML is rendered on the server, it is (probably) possible that JavaScript is not needed for the page to function (although Next.js is a React framework, it can still function well with JavaScript disabled). Therefore, we will try to remove JavaScript entirely if possible, and the comment form submission is handled by the default HTML form submission handler. That can guarantee that the embedded frames are completely free of any potential trackers and third-party scripts, the problems which many people complain about Disqus and others. However, depending on how the page will look like when it is implemented, this might not be possible at all, therefore

removing dependency on JavaScript is not among our top priorities. Even so, we plan to reduce the JavaScript sent to clients as much as possible for performance optimisation.

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 his teammate 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. screenshoting). 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 it 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

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

If these optional fields are filled, they will be shown to the ezkomment user, who may use those 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". Emails are never exposed publicly.

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 ways they want. However, using this method, users need to be aware of and guard against XSS attacks, since the comment body are not escaped. See the API section for more details.

4.3.4 Replies to Comments

In a lot of occasions, the comments serve as questions to the post author. Therefore, the feature for authors to reply to comments is absolutely necessary, and we do plan to support that.

For the public to reply to other users' comments, we are still undecided as in whether to allow this. The design for the comment section and the page dashboard will be a lot cleaner without this, and the JavaScript footprint will also be reduced significantly. However, that might make it hard to have a discussion in the comment section.

Even if we do allow everyone to reply to any comment, it is likely that we will only support one level deep (so Reddit-like threads are not supported).

4.3.5 Editing

We do not plan to support editing comments at the moment, since we are unable to find a solution to address the following issues at the same time to our satisfaction:

- Comments should not be edited after a time frame (5 minutes)
- What happens if a comment is edited after it is approved? (See the moderation features section below)
- How should pending comments be displayed?
- How should this integrate with the API?

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 workaround 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 in 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. 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, 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 deletion of "auto-approved" comments.

4.4.1 User Stories

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). User reads the name and email of the commenter (if any) and click 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 to ever be displayed publicly.

4.4.3 Scope

Users can set this feature per page. Pages that may involve heated discussions may need this features, 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 that design principles cannot fit to 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 are 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

> This section may change significantly in later revisions of the README.

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 details, we will give the user two HTML files to edit, `index.html` for the overall HTML content and `comment.html` for each individual comment. In `index.html`, there is a special placeholder `<COMMENT>` which works as a placeholder for the comments. In `<comment.html>` there are placeholder "HTML tags" for the commenter name and comment body.

For example, with the following `index.html`

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <!-- (i) include spreadsheets, js libraries, etc -->
  <body>
  <COMMENT>
  <!-- (ii) include submission form, etc -->
  </body>
</html>
```

and the following `comment.html`

```
<!-- comment.html -->
<div class="comment">
  <div class="comment__name"><NAME></div>
  <div class="comment__body"><CONTENT></div>
  <div class="comment__time"><TIME></div>
</div>
```

then with the following (approved) comments (in JSON format)

```
[
  {
    "name": "John Doe",
    "content": "Hello world",
    "time": "2022-01-01T00:00:00.000Z"
  },
  {
    "name": "Jane Doe",
    "content": "Bye world",
    "time": "2022-01-01T00:00:00.000Z"
  }
]
```

then the compiled HTML sent to clients is

```
<!DOCTYPE html>
<html>
  <!-- (i) include spreadsheets, js libraries, etc -->
  <body>
    <div class="comment">
      <div class="comment__name">John Doe</div>
      <div class="comment__body">Hello world</div>
      <div class="comment__time">2022-01-01T00:00:00.000Z</div>
    </div>
    <div class="comment">
      <div class="comment__name">Jane Doe</div>
      <div class="comment__body">Bye world</div>
      <div class="comment__time">2022-01-01T00:00:00.000Z</div>
    </div>
    <!-- (ii) include submission form, etc -->
  </body>
</html>
```

We are still undecided as in whether to compile the time to more readable formats from the server, or send the code to convert the timestamp as JavaScript to client. Since it is just the time, the JavaScript should be tiny enough; it may even be simply

```
const timestamp = new Date("2022-01-01T00:00:00.000Z");
console.log(time.toLocaleDateString()); // "01/01/2022"
```

but since doing so will put an end to our ”no-JavaScript” mission, we are still undecided about this matter.

Also, there are still many unresolved issues about this feature.

- How should it handle replies to comments?
- How should the ”Reply” button for top-level comments be displayed?
- How should custom logic be handled?
- etc.

These are logic issues that can be handled on the server completely, but is not doable for the current spec of this feature. In fact, we think that simple bare HTML may not be sufficient for this problem, and we may need to let users write in more complex languages, such as EJS or JSX. For example, with JSX, users can do

```
const convertTime = time => new Date(time).toLocaleDateString();

module.exports = (
  <section>
    {comments.map(({ name, content, time }, index) => (
      <div key={index} className="comment">
        <div className="comment__name">{name.toUpperCase()}</div>
        <div className="comment__body">{content}</div>
        <div className="comment__time">{convertTime(time)}</div>
      </div>
    ))}
  </section>
);
```

Due to the complexity of the issue, this will be decided when we implement it.

4.5.3 Export and Import Customisation Files

Since the customisation process involves text files, we can also pack them together and use it as a ”theme” file. Users can then share these theme files between each other to have a good-looking design for their comment sections.

Therefore, on the long term, we think it may be good to support exporting current customisation configuration to a theme file that can be saved and shared; as well as importing user-uploaded theme files.

This opens the door to a marketplace-like feature for the application. However we do not plan to implement it at the moment, since the application design is complex enough (just look at the length of this document and it is just Milestone 1). We may consider it when the core features of the app is complete.

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 the two purposes: fetching comments in a page and posting comments to it (for it to be sent to the pending queue).

The currently planned URI for the API is

```
https://api.ezkomment.joulev.dev/comments/{siteName}/{pageId}
```

This URI may be changed later revisions of README, depending on how the back-end API is implemented.

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 include all comments and their metadatas (name, timestamp). Since the API is public, emails are not exposed.

```
// 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 user 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 email = document.getElementById("form_email").value;
const content = document.getElementById("form_content").value;
const res = await fetch("url", {
  method: "POST",
  body: JSON.stringify({ name, email, 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

5.1 How the App Works, In a Nutshell

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, therefore we decided that, for the front-end section at least, the app will be implemented in Next.js.

We will host the front-end in Vercel. Since Next.js is made by Vercel, it is literally the best platform to host Next.js apps. We also notice that Vercel can build our front-end app significantly faster than other services, such as Netlify.

The back-end is currently written with Express.js. Currently, we are planning to deploy our back-end API using Firebase Hosting – however, we are still considering other options for suitable hosting cost and deployment process.

Since Next.js also offers API routes to act as a back-end, it can also be used for back-end development. However, currently we do not plan to mitigate our back-end to Next.js.

We decided to use TypeScript for the entire project for easier management.

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.

5.1.3 Authentication

Currently authentication is done with Firebase. We have implemented a client-based authentication system successfully as of pull request #57, and the app can be accessed at https://ezkdev.joulev.dev.

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 Route Structure

- `/users/:uid`: The route to get, update and delete user information, given the user ID
- `/sites/:id`: The route to get, create, update and delete sites, given the site ID
- `/pages/:id`: The route to get, create, update and delete pages, given the pages ID

The Route Structure will be further improved as we progress. Nested routes will be used to show the hierarchy relation between the entities.

5.2 User Interface Design

The front-end is styled with Tailwind CSS. We found it to speed up the styling process even more than we ever expected.

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.

Dark mode is fully supported.

5.2.1 Logo

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

5.2.2 Colour

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) is 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).

5.2.3 Spacing

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

5.3 Search Engine Optimisations

In pull request #66, we have added the necessary SEO tags to every page that need 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 enable the site to be crawled for search engines as of now, the tags are now effective when sharing the pages over social media.

5.4 Software Engineering Practices

5.4.1 Repository Structure

Since both the front-end and the back-end are JavaScript packages, we decided to host it in the same repository as a *monorepo*, supported by NPM workspaces.

```
└── packages
    ├── client # Front-end, @ezkomment/client
    └── server # Back-end, @ezkomment/server
```

5.4.2 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 to, 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 request should only be merged when that person has agreed to.
- In any case, 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 that developer self-reviews the whole pull request, makes any further changes and finally merging the pull request.

We will use the "Merge and rebase" 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 repository 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 details in the repository wiki.

5.4.3 Continuous Integration and Continuous Deployment

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.

CI pipelines are done by GitHub Actions, and currently it involves running ESLint and unit tests for the front-end section.

When the codebase of the back-end is ready for deployment, we will also add CI and CD for the back-end.

5.4.4 Testing

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.

However, for front-end, Jest is likely to not be enough. We are considering also using more complex testing tools, such as Cypress for integration tests and end-to-end tests, however these tests are not yet implemented.

6 Timeline

6.1 Currently Already Implemented Features

- The main interfaces of the front-end app
- Public pages of the front-end
- The front-end has been deployed
- Firebase (client-based) authentication for the front-end app

6.2 Going Forwards to Milestone 2 (27 June)

- 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 resolution for all known issues in the app design mentioned above.

6.3 After Milestone 2

These do not have a planned timestamp yet, since (at least) @joulev does not fully understand how fast we will be moving yet.

- Implement moderation features
- Implement customisation features
- Implement the public API

7 Conclusion

The above describes the design and planning of the appplication we are planning to build for Orbital 2022. Please note that while the design is quite detailed and has covered all core features that we plan, there are still many issues with the design for which we have not yet found a resolution, and most of these issues are also mentioned above.

As we move along the implementation process, we expect some of the planned features above to change, as it may become too complicated, too bug-prone, too unnecessary to support, among other reasons. Therefore, while the design above describes how we currently intend the app to be, it may be changed considerably in later versions of the README.