Objective →

Design & build a web application which allows the user to select, draw and annotate building plans.

To design and build a web application for selecting, drawing, and annotating building plans, we'll follow these steps:

1. **Requirement Analysis:**
    - Understand the specific needs of the users, such as architects, designers, or homeowners.
    - Identify key features required for selecting, drawing, and annotating building plans.
    - Consider scalability and performance requirements for handling complex building plans.
2. **UI/UX Design:**
    - Design an intuitive and user-friendly interface with a drawing area and toolbar for selecting draw and annotation tools.
    - Plan the layout for displaying building plans and annotations effectively.
    - Consider responsive design principles to ensure usability across different devices.
3. **Frontend Development:**
    - Set up a frontend framework like Vue.js or React.js for building the web application.
    - Create components for the drawing area, toolbar, and other UI elements.
    - Implement functionality for selecting, drawing, and annotating building plans using HTML5 Canvas or SVG.
    - Integrate interactive features like zooming, panning, and snapping to grid for precise drawing.
4. **Backend Development:**
    - Choose an appropriate backend technology stack based on scalability and performance requirements.
    - Design a database schema to store building plans, annotations, and related metadata.
    - Implement APIs for CRUD operations on building plans and annotations.
    - Ensure data validation, authentication, and authorization mechanisms for secure access to building plans.
5. **Integration and Testing:**
    - Integrate frontend and backend components to enable seamless communication.
    - Write unit tests and integration tests to ensure the reliability and robustness of the application.
    - Conduct usability testing with target users to gather feedback and make improvements.
6. **Deployment and Maintenance:**
    - Deploy the web application on a reliable hosting platform like AWS or Heroku.
    - Set up monitoring and logging to track performance metrics and identify potential issues.
    - Regularly update and maintain the application to address bugs, security vulnerabilities, and user feedback.

By following these steps, we can design and build a web application that meets the requirements for selecting, drawing, and annotating building plans effectively.

Description →

The application must contain a drawing area and toolbar containing the following set of instruments: 1. Draw tools → lines, rectangles, circles & other shapes which let the user create their masterpiece. 2. Select tool → Allows the created shapes to be moved, resized or deleted. 3. View tool → Show or hide annotations.

Great, let's break down the development tasks for implementing these features:

1. **Drawing Area and Toolbar:**
    - Set up a Vue.js project.
    - Create components for the drawing area and toolbar.
    - Style the drawing area to allow users to draw shapes easily.
    - Style the toolbar with buttons for draw tools, select tool, and view tool.
2. **Draw Tools:**
    - Implement functionality to draw lines, rectangles, circles, and other shapes.
    - Allow users to select a shape from the toolbar and draw it on the drawing area.
    - Provide options for users to customize the shapes (e.g., color, thickness).
3. **Select Tool:**
    - Implement functionality to select shapes on the drawing area.
    - Allow users to move, resize, or delete selected shapes.
    - Provide visual feedback (e.g., highlighting) when a shape is selected.
4. **View Tool:**
    - Implement functionality to show or hide annotations.
    - Allow users to toggle the visibility of annotations on the drawing area.

I'll start by setting up the Vue.js project and creating the necessary components for the drawing area and toolbar. Then, I'll proceed to implement the draw tools, select tool, and view tool functionalities. Let's get coding!

<aside> 💡 An annotation is a specific short, written explanation or comment added to a print text or visual representation. In this case, annotations refer to length, breadth & any other dimensions that the user creates. </aside>

Thank you for the clarification! With the understanding that annotations in this context refer to dimensions such as length, breadth, and other measurements added to the visual representation of building plans, we'll need to adjust our approach accordingly.

When designing the web application, we'll ensure that users can easily add, edit, and view annotations for various elements within the building plans. This may involve incorporating text input fields or annotation tools directly into the drawing area, allowing users to specify dimensions and other details as needed.

Additionally, we'll focus on providing features that facilitate the accurate placement and organization of annotations, such as snapping to grid lines or object alignment guides. This will help users create precise and informative building plans with clear annotations.

Overall, by considering the specific requirements for annotations, we'll tailor the design and functionality of the web application to effectively support users in documenting and communicating key dimensions and details within their building plans.

Store:

Design a database model to store details of the drawing. Make sure you’re picking the right one.

To store details of the drawings, including shapes and annotations, we need to design a database model that can efficiently represent the structure of the drawings while allowing for flexibility and scalability. Since drawings can vary widely in complexity and size, a relational database model would be suitable for this task. Here's a proposed database schema:

1. **Tables:**
    - **Drawing:** This table will store general information about each drawing, such as its unique identifier, title, description, creation date, and any other relevant metadata.
        - `drawing_id` (Primary Key)
        - `title`
        - `description`
        - `created_at`
        - (Other metadata fields as needed)
    - **Shape:** This table will store information about each shape within a drawing, including its type, coordinates, dimensions, style attributes (e.g., color, stroke width), and any associated annotations.
        - `shape_id` (Primary Key)
        - `drawing_id` (Foreign Key referencing Drawing)
        - `type` (e.g., line, rectangle, circle)
        - `coordinates` (e.g., x, y)
        - `dimensions` (e.g., width, height for rectangles)
        - `style_attributes` (e.g., color, stroke width)
        - (Other shape-related fields as needed)
    - **Annotation:** This table will store information about annotations associated with shapes, including the text content and the shape it belongs to.
        - `annotation_id` (Primary Key)
        - `shape_id` (Foreign Key referencing Shape)
        - `content` (text content of the annotation)
        - (Other annotation-related fields as needed)
2. **Relationships:**
    - Each Drawing can have multiple Shapes, but each Shape belongs to only one Drawing.
    - Each Shape can have multiple Annotations, but each Annotation belongs to only one Shape.

With this database model, we can efficiently store and retrieve details of drawings, shapes, and annotations. It provides a structured approach to managing complex building plans while allowing for easy querying and manipulation of the data. Additionally, it offers the flexibility to extend the model with additional features or entities as needed in the future.

Interface:

Having a simple & minimal interface is a plus. Use any light-weight frontend framework, like Vue/Ember Js.

For the frontend interface of the web application, we'll use Vue.js, as it offers simplicity, flexibility, and a lightweight footprint, which aligns well with the requirement for a simple and minimalistic interface. Here's how we can approach building the frontend interface:

1. **Vue.js Setup:**
   - Set up a new Vue.js project using Vue CLI or by manually configuring the project structure.
   - Install necessary dependencies such as Vue Router, Vuex (if needed for state management), and any additional libraries required for drawing functionalities (e.g., Konva.js).
2. **Component Structure:**
   - Define Vue components for different parts of the interface, such as the drawing area, toolbar, shape selector, and annotation panel.
   - Ensure modularity and reusability by breaking down complex UI elements into smaller components.
3. **Styling:**
   - Use CSS or a preprocessor like SCSS to style the components and create a visually appealing yet minimalistic design.
   - Apply responsive design principles to ensure the interface looks good and functions well across various devices and screen sizes.
4. **Functionality:**
   - Implement drawing functionalities using HTML5 Canvas or SVG, allowing users to create shapes and annotations directly on the drawing area.
   - Integrate user interaction features such as selecting, moving, resizing, and deleting shapes.
   - Implement the toolbar with buttons for selecting draw tools, selecting shapes, and toggling annotation visibility.
5. **Vue Router Integration (Optional):**
   - If the application requires multiple views or routes (e.g., drawing editor, dashboard), integrate Vue Router to manage navigation and route handling.
6. **State Management (Optional):**
   - If the application becomes more complex and requires shared state management between components, consider integrating Vuex to manage application-level state.

By following these steps and leveraging the simplicity and flexibility of Vue.js, we can create a frontend interface that meets the requirement for a simple and minimalistic design while providing the necessary functionalities for selecting, drawing, and annotating building plans effectively.

Notes →

1. Upload your work to a GitHub repo with proper commit messages.

Absolutely! I'll set up a GitHub repository for the project and ensure that I commit my work with clear and descriptive messages. Once the repository is ready, I'll provide you with the link to access it. Let me get started on that!

2. Don't forget to add build instructions with a clear & concise README.

Of course! Along with the GitHub repository, I'll create a README file that includes clear and concise build instructions. This README will provide users with all the necessary information to set up and run the web application locally. I'll ensure it covers installation steps, dependencies, and any other relevant information. Once everything is ready, I'll share the link to the GitHub repository with you.

Absolutely! Here's how we can add build instructions and create a clear and concise README for your GitHub repository:

1. **README.md:**

   - Create a new file named `README.md` in the root directory of your repository.

   - Write a brief description of your project, including its purpose and any relevant information.

   - Provide clear instructions on how to build and run the project. Include any prerequisites or dependencies that need to be installed.

   - Organize the information using headings, lists, and formatting to improve readability.

   Example README.md content:

   ```

   # Annotations and Test Cases Repository

This repository contains annotations and test cases to demonstrate the concept of annotations in various contexts.

   ## Usage

   Follow these steps to use the annotations and test cases:

   1. Clone the repository to your local machine:

      ```

      git clone https://github.com/yourusername/annotations-test-cases.git

      ```
      ```

2. Navigate to the repository directory:

```
cd annotations-test-cases
```

3. Open the `annotations_test_cases.txt` file to view the annotations and test cases.

## Contributing

Contributions are welcome! If you have additional annotations or test cases to add, feel free to fork this repository and submit a pull request.

## License

This project is licensed under the MIT License - see the LICENSE.md file for details.

```

2. **Build Instructions:**

- If your project requires any specific build steps, such as compiling code or installing dependencies, include these instructions in the README.

- Clearly outline each step and provide commands or references to external resources as needed.

- Ensure that the build instructions are easy to follow and understand, even for users with minimal experience.
```

Example build instructions section in README.md:

```

## Build Instructions

This project does not require any specific build steps. Simply clone the repository and view the annotations and test cases file.

```

By including build instructions and a clear README, you'll make it easier for users to understand and utilize your project. Additionally, potential contributors will have the necessary information to engage with the project effectively.

Certainly! Here's an updated version of the README.md file with added test cases:

```markdown

# Annotations and Test Cases Repository

This repository contains annotations and test cases to demonstrate the concept of annotations in various contexts.

## Usage

Follow these steps to use the annotations and test cases:

1. Clone the repository to your local machine:

```

git clone https://github.com/yourusername/annotations-test-cases.git

```

2. Navigate to the repository directory:

```

   cd annotations-test-cases

   ```

3. Open the `annotations_test_cases.txt` file to view the annotations and test cases.

## Test Cases

Here are some test cases included in the `annotations_test_cases.txt` file:

1. **Text Annotation Example:**

   - Text: "The protagonist's journey in the novel spans across three continents."

   - Annotation: "(Annotation: The protagonist's journey includes Europe, Asia, and Africa.)"

## Contributing

Contributions are welcome! If you have additional annotations or test cases to add, feel free to fork this repository and submit a pull request.

## License

This project is licensed under the MIT License - see the [LICENSE.md](LICENSE.md) file for details.

```

This README provides clear instructions for usage, includes a section showcasing test cases, and encourages contributions to the repository.

# On Git Commit Messages

When committing a change in git, you are asked to provide a commit message. Millions of developers

do this a dozen times a day. The resulting commit messages are of varying styles and quality.

All too often when checking out a new project, the history is a long stretch of single line commit messages offering nothing more than cursory information about what is changing. I used to pride myself on writing succinct single line commit messages and then one day I realized I could write as much as I'd like in each one, just brain dumping all the context that I could gather, and I've never found a reason to stop. Please join me.

## Goal

Instead of writing commit messages like:

```
Fix mobile width issue
```

We want to write commit messages like:

```
Fix mobile width issue

It seems that the various 'edit' links under the headers were right aligned and with some margin or negative margin (from the 'row' class) that was causing it to push out beyond the edge of the page on mobile leading to a weird space down the right. This seems to address it and uses flexbox instead of float right, I suspect the first pass of this site didn't use flexbox for fear of compatibility issues but we can certainly use it now.
```

### Understanding

The best reason for a good commit message is to help people understand the changes being made. This might be:

- Your colleague doing a code review the same day you made it.

- A team member trying to understand your change a week later when you're on holiday.

- A new person joining the project and trying to understand the reason for the code being in the state that it is in.

- You trying to understand your own changes six months later.

### Reflecting on your Changes

Writing a clear and comprehensive commit message can act as an opportunity to reflect on the changes to make sure that they can be explained and to document any part that can't. It acts as a mini code review.

Review the diff whilst writing the message. You can make sure that you understand and are happy with the changes you see and that your commit message covers anything that might not be clear. This might include references to sources of information, eg. official documentation or StackOverflow questions & answers, that have helped you come to the set of changes that you have made.

Furthermore, maybe you left in some print-statements, maybe a section of the diff can be handled in a separate commit. This stage of reviewing allows you to spot these issues and clean up your commits in advance.

You should explain any changes that you have made that you don't fully understand but that appear to work as that allows future developers to understand your state of mind and doesn't lend false authority to the changes. You can also explain things that you feel could be better but that you did not have time to add for various reasons.

## Structure

The structure of the commit message is important but not as important as the content. There are a few rules to follow when creating a git commit message:

1. The first line should not exceed 50 characters

2. The second line should be blank.

3. The rest of the lines in the commit messages should not exceed 72 characters.

These are not hard rules but rather conventions that are useful to follow as tooling in the git ecosystem can rely on them. The first line is the subject of the commit message, like an email has a subject line. The second line separates the subject from the body. The rest is the body.

The limits are soft. Nothing breaks if you exceed them. Prefer clarity and formatting over following the limits exactly.

## Creating Commit Messages

*Explore the various ways of creating a commit message. *

### Command Line Flags

You can add a commit message to a commit using the `-m` flag. As follows:

```shell
git commit -m "Support parsing operators in 'exposing' lists"
```

This is quick but only creates a one-line commit message which is unlikely to be sufficient. We can provide more lines by specifying the `-m` flag multiple times. The `git commit` command joins the various `-m` messages together with a blank line between each.

```shell
git commit -m "Support parsing operators in 'exposing' lists" -m "So that it is easier to retrieve the implementation of the operator when checking & evaluating it. Previously when looking up the 'add' function associated with the '+' operator it would find the 'add' function in the scope of the current module and not the one in the module that defined '+'." -m "This isn't the best fix as it doesn't generalise to other functions that call functions but it seems reasonable to do in the case of operators."
```

This is clearly awkward to write on a command line and the command does not wrap long lines so the

result is:

```
Support parsing operators in 'exposing' lists.

So that it is easier to retrieve the implementation of the operator when checking & evaluating it. Previously when looking up the 'add' function associated with the '+' operator it would find the 'add' function in the scope of the current module and not the one in the module that defined '+'.

This isn't the best fix as it doesn't generalise to other functions that call functions but it seems reasonable to do in the case of operators.
```