Concordia UNIVERSITY | GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

SOEN 341 Software Process

Team Project, Winter 2025

Project Title:

ChatHaven, a seamless communication application

The goal of this sprint is to perform maintenance on your system by fixing bugs and refactoring poor design choices. You will use a static analysis tool to identify bad design and bad code practices. You also must complete the implementation and testing of your new proposed features, which stand out from the other teams' features. Your new features must show some novelty and reflect a good understanding of your app's domain and the best software development practices.

1. User Authentication & Management:

- Implement User's login (with roles). *Already completed in Sprint 1.*
- Functionality for Admin to create ~~teams~~ channels and assign users to specific channels. Ensure channels are visible to both admins and normal users. *Already completed in Sprint 1.*

Sprint 2 (At least 2 of 4 features from the list below).

2. Text Channels for Group Communication:

- Implement functionality to send messages visible to all users in a channel.
- Enable Admins to moderate messages (delete inappropriate ones)

3. Direct Messaging (DM):

- Enable users to message each other.
- Ensure conversations are private and only visible to participants.

For Sprint 3, you worked on the following features and must have implemented at least the **Text Channels enhancements**, and 4. User presence & status.

Text Channels enhancements

- There would be a set of default channels generated by the administrator, (e.g., all-general) where everybody registered can participate, and private channels that the users will create. When users create a private channel, they can invite registered people on the platform to join the channel.
- Users can leave channels or request the creator to join.

4. User Presence & Status

- On the dashboard, there will be a list of registered users.
- Online, offline, away indicators
- Last seen timestamp
- Users can send direct messages to users, even if they are not logged in

5. Message Enhancements

- Emoji support.
- Quoting

6. Your unique team features

You need to propose and implement, by Sprint 4 at the latest, unique features to set your team apart from the rest. These features must have been accepted by the TA responsible for your team.

Here are some examples, but these are only suggestions, and this is not an exhaustive list.

- Implement a chatbot in your system (e.g., Welcome Bot, Translation Bot)
- Threads to keep related messages organized in conversations
- File sharing and Multimedia with file previews for supported formats.
- Mentions & Notifications: notify users with @username or @all

Sprint 4 delivery instructions.

1. Plan for this sprint the same way that you did for Sprint 3.
2. Finalize the user stories of all features: label user stories and link them to tasks, and acceptance tests.
3. Finalize the development and implementation of all features.
4. Create unit tests for all your code and ensure they run automatically on your continuous integration pipeline (to be demonstrated during lab sessions).
5. Upload the meetings' minutes
6. Each team member must commit a detailed log of their contribution including time spent on each activity and commit it to their repository.
7. The team must demo the features implemented during the sprint to the TA assigned to their team one day before the due date (i.e., during the lab session). **After the due date, no changes made to the repo will be considered for evaluation of this sprint, without exceptions.**
8. (Re)Organize the file structure of your repository in GitHub. The project files must be organized in packages/directories according to their intent. For example, tests are separated from production code, business classes are separated from UI, Utility classes, etc. See Appendix A.
9. Perform code review on the new features implemented in this Sprint by at least one team member (not the author). See Appendix B and https://github.com/features/code-review for further information.
10. Search and repair 5 code bugs on your repository reported by an automatic tool, e.g., FindBugs, SonarQube, etc. The TA must approve the tool during tutorials. Here is a link with some linters that integrate with GitHub. You must document your maintenance effort with code commits and a detailed description, including references to the linter reports that triggered the fixes. See Appendix C.

A detailed grading rubric can be accessed here. This will be used by the markers.

Appendix A. Sprint 4 Repository organization guidelines.

Organizing files in folders for a web app is crucial for maintaining a clean and manageable project structure. Here are some general guidelines to help you organize files effectively:

1. **Separation of Concerns:**
   - Group files based on their concerns and responsibilities. For example, separate HTML, CSS, and JavaScript files.
   - Use the MVC (Model-View-Controller) or a similar architecture to divide your application into logical components.

2. **Use a Standard Directory Structure:**
   - Follow a standard directory structure. Many web frameworks and project generators have predefined structures. For example:

```
/project
  /public
    /css
    /js
    /images
  /src
    /controllers
    /models
    /views
  /templates
```

3. **Assets and Static Files:**
   - Place static files such as images, stylesheets, and client-side scripts in a dedicated `public` or `static` folder.
   - Organize these folders based on file type or module.

4. **Components or Modules:**
   - Group related files together, especially if they form a component or module.
   - For example, if you have a user authentication module, you might have folders like `auth` or `user`.

5. **Vendor Libraries:**
   - Keep third-party libraries and dependencies in a separate folder, often named `vendor` or `node_modules`.
   - This makes it clear which files are yours and which are external.

6. **Configuration Files:**
   - Place configuration files at the root level of your project or in a dedicated `config` folder.
   - Keep configuration files separate from your source code.

7. **Server-Side Code:**
   - Organize server-side code based on functionality (e.g., routes, controllers, middleware).
   - For Node.js projects, you might have folders like `routes`, `controllers`, `models`, and `middleware`.

8. **Client-Side Code:**
   - Organize client-side code based on the type of content (e.g., pages, components, utilities).
   - Use a consistent naming convention for your files.

9. **Naming Conventions:**
   - Follow a consistent naming convention for files and folders.
   - Use clear and descriptive names that convey the purpose of the file or folder.

10. **Testing:**
    - If you have automated tests, organize them in a separate `tests` or `spec` folder.
    - Mirror the structure of your source code to make it easy to find corresponding tests.

11. **Documentation:**
    - Consider having a `docs` folder for documentation, including README files, API documentation, and any other relevant documentation.

12. **Build and Deployment Scripts:**
    - Keep build scripts, deployment configurations, and other tools in a dedicated folder, such as `scripts` or `tools`.

13. **Version Control:**
    - Add configuration files for version control systems (e.g., `.gitignore`, `.gitattributes`) at the root level.

14. **Keep It Simple:**
    - Don't overcomplicate the structure. Aim for a balance between simplicity and organization.
    - If your project is small, a flat structure might be sufficient.

Remember that these guidelines are general, and the specific needs of your project or the framework you're using might lead to variations.

Real web apps regularly review and refine their project structure as the application grows and evolves.

Appendix B. Sprint 4 Repository Code review guidelines.

Performing code reviews is an essential aspect of maintaining code quality and ensuring that best practices are followed in a web application. Here are some guidelines for conducting effective code reviews for a web app stored in a GitHub repository:

Before the Code Review:

1. **Understand the Purpose**: - Before starting a code review, understand the purpose of the changes. Know whether it's a bug fix, a new feature, or a refactoring.

2. **Familiarize Yourself**: - Review the pull request description, associated issues, and any relevant documentation to familiarize yourself with the context of the changes.

3. **Environment Setup:** - If applicable, test the changes locally to ensure they work as intended and don't introduce new issues.

During the Code Review:

4. **Review Code in Manageable Chunks:**
- Break down the review into manageable chunks. Don't attempt to review the entire codebase in one go.

5. **Coding Standards:**
- Ensure the code follows coding standards and guidelines established for the project.

6. **Readability and Maintainability:**
- Check if the code is readable and maintainable. Are variable/method names clear? Is the code logically structured?

7. **Functionality and Requirements:**
- Verify that the code meets the functional requirements specified in the associated issues or pull request description.

8. **Error Handling:**
- Check for proper error handling and edge case scenarios. Ensure the code gracefully handles unexpected situations.

9. **Performance Considerations:**
- Look for potential performance bottlenecks or inefficiencies in the code.

10. **Security:**
    - Assess the code for security vulnerabilities. Ensure sensitive information is handled securely and that the code is not susceptible to common security issues.

11. **Testing:**
    - Verify that the code includes tests for new functionality or bug fixes. Check if existing tests still pass.

12. **Comments and Documentation:**
    - Ensure the code includes comments where necessary, and that documentation (such as README updates) is provided.

13. **Code Duplication:**
    - Check for code duplication. Encourage the developer to refactor common functionality into reusable components or functions.

14. **Version Control Best Practices:**
    - Verify that the commit history is clean, with each commit representing a logical and atomic change. Check for meaningful commit messages.

15. **Dependencies:**
    - If new dependencies are introduced, ensure they are necessary and their versions are appropriate.

Providing Feedback:

16. **Be Constructive:**
    - Frame feedback in a constructive manner. Instead of saying, "This is wrong," suggest improvements or ask questions to understand the reasoning.

17. **Focus on High-Impact Issues:**
    - Prioritize feedback on high-impact issues. Address critical problems first before moving on to less critical ones.

18. **Use Code Review Tools:**
    - Leverage code review tools provided by GitHub or third-party tools to comment directly on code lines. This makes it easier to provide specific feedback.

19. **Encourage Discussion:**
    - Encourage open communication. If there's uncertainty or a difference in opinion, have a discussion to reach a consensus.

20. **Recognize Positive Contributions:**
    - Acknowledge positive aspects of the code. Positive reinforcement motivates developers and fosters a positive team culture.

After the Code Review:

21. **Follow-Up:**
    - After feedback is provided, follow up to ensure the necessary changes are made and address any outstanding issues.

22. **Merge Approval:**
    - Once the code meets the required standards and all feedback is addressed, provide merge approval.

23. **Celebrate Success:**
    - Celebrate successful code contributions and improvements. Recognize the effort and collaboration involved.

General Best Practices:

24. **Regular Code Reviews:**
    - Conduct regular code reviews to maintain a continuous feedback loop and improve code quality over time.

25. **Rotate Reviewers:**
    - Rotate reviewers to distribute knowledge across the team and avoid bottlenecks.

26. **Automated Checks:**
    - Integrate automated tools and checks (linters, static code analyzers) into the review process to catch common issues automatically.

Remember that effective code reviews contribute not only to code quality but also to knowledge sharing and collaboration within the development team. It's a collaborative process aimed at improving the overall quality of the software.
These guidelines are general and exhaustive. You need to show a good understanding and application of your code review to your TA.

Appendix C. Sprint 4 static analysis guidelines.

Applying static analysis tools in a web app stored in a GitHub repository can significantly improve code quality and adherence to coding standards. Here are guidelines to effectively use static analysis tools like linters, SonarQube, or FindBugs:

1. **Select Appropriate Tools:**
   - Choose static analysis tools that are well-suited for your technology stack and programming languages. For a web app, common tools include ESLint for JavaScript, Stylelint for CSS, and Pylint for Python.

2. **Integrate into Build Process:**
   - Integrate static analysis tools into your build process to automatically analyze code with each build. This ensures consistent application of coding standards.

3. **Establish Baseline:**
   - Run static analysis on the existing codebase to establish a baseline. This helps identify and address existing issues.

4. **Define Coding Standards:**
   - Clearly define coding standards and rules for your project. Configure the static analysis tools to enforce these standards.

5. **Customize Rules:**
   - Tailor the rules of static analysis tools to match your project's specific requirements. Adjust severity levels based on your team's preferences and project needs.

6. **Regularly Update Rules:**
   - Regularly update the ruleset of your static analysis tools to include improvements and updates from the tool developers or the community.

7. **Use Pre-commit Hooks:**
   - Implement pre-commit hooks to run static analysis locally before allowing a commit. This prevents the introduction of new issues into the codebase.

8. **Continuous Integration (CI) Integration:**
   - Integrate static analysis tools into your CI pipeline. This ensures that every pull request is automatically checked for code quality and standards adherence.

9. **Set Quality Gates:**

- Establish quality gates in your CI/CD pipeline to prevent the merging of code that doesn't meet the predefined quality criteria.

10. **Utilize Git Hooks:**
    - Use Git hooks to run static analysis before certain Git actions (e.g., pre-push or pre-commit). This provides an additional layer of enforcement.

11. **Configure Exclusions:**
    - Configure the static analysis tools to exclude certain files or directories if necessary. This is useful for ignoring third-party libraries or autogenerated code.

12. **Leverage Editor Integration:**
    - Integrate static analysis tools into your code editor to provide real-time feedback to developers during the coding process.

13. **Address False Positives:**
    - Regularly review and address false positives reported by static analysis tools. This ensures that developers don't waste time fixing non-issues.

14. **Educate the Team:**
    - Educate the development team on the purpose and benefits of static analysis. Encourage a culture of continuous improvement.

15. **Automated Code Reviews:**
    - Implement automated code review tools that can perform static analysis and provide feedback directly in your pull requests.

16. **Monitor Trends:**
    - Monitor trends in static analysis reports over time. Track improvements and address any emerging patterns of deteriorating code quality.

17. **Include Security Scans:**
    - Integrate security scanning tools into your static analysis process to identify and address potential security vulnerabilities.

18. **Documentation:**
    - Document the static analysis process and guidelines for developers. **Use the wiki for this purpose**. Ensure that new team members are familiar with the tools and their configurations.

19. **Periodic Reviews and Adjustments:**
    - Conduct periodic reviews of the static analysis setup. Adjust rules and configurations based on evolving project requirements and changing coding standards.

Implementing these guidelines will help you leverage static analysis tools effectively, leading to improved code quality, consistent coding standards, and a more maintainable web application.