Documentation Guidelines

Table of Documentation

Necessary Components for Building a Table of Documentation:

- **Centralized Document**: Create a single document containing links to all essential documentation.
- **Organized by Areas**:
  - **Getting Started**: Machine setup, dependencies.
  - **Contribution Guide**: Git guidelines, PR process, development workflow.
  - **Troubleshooting Guide**: FAQs, common issues.
  - **Additional Resources**: Links to other relevant documents.

Purpose of the Table:

- **Single Reference Point**: Serves as a comprehensive list of links to all key documentation.
- **Streamlined Onboarding**: Helps new team members quickly access relevant materials.
- **Maintenance**: Ensures that redundant or fragmented documents are consolidated.

Technical Glossary

Necessary Components for Building a Technical Glossary:

- **Three-Column Table**:
  - **Term**: The acronym or term used.
  - **Definition**: Detailed explanation.
  - **Examples/Context**: Real-life examples or links for context.

Purpose of the Glossary:

- **Clarify Team-Specific Jargon**: Provides clear definitions for internal terms, acronyms, and jargon.
- **Enhance Communication**: Helps new and existing team members understand terminology without needing to ask.

- **Encourage Collaboration**: Involve both experienced and new hires in contributing to the glossary.
- **Accessible Location**: Should be prominently linked in the Table of Documentation.

Table of Expertise

Necessary Components for Building a Table of Expertise:

- **Areas of Expertise**: Define high-level (problem domain) and detailed (specific features) expertise.
- **Team Member Assignments**: List the specialists or developers responsible for each area.
- **Collaborators**: Include designers, product team members, and external collaborators (e.g., back-end, infrastructure specialists).
- **Contact Information**: Provide contacts for easy reference.

Purpose of the Table:

- **Identify Expertise**: Helps team members find the right person to consult for specific areas or features.
- **Avoid Inefficiencies**: Reduces the need for unnecessary redirections when looking for expertise.
- **Ensure Accountability**: Assigns ownership of areas to developers when team members leave to maintain support.

FAQs

Necessary Components for Building a Frequently Asked Questions (FAQ) Document:

- **Common Questions**: Identify frequent questions from team members related to processes, tools, or code.
- **Answers**: Provide clear, concise responses to each question.
- **Organization by Areas**: Categorize questions by relevant topics (e.g., coding standards, deployment issues).
- **Link to Documentation**: Ensure answers are linked in the Table of Documentation for quick access.

Purpose of the FAQ Document:

- **Reduce Repetition**: Minimizes the need to repeatedly address the same questions.
- **Improve Efficiency**: Provides team members with a quick reference for common issues.
- **Centralized Access**: Links to the FAQ in the Table of Documentation for easy retrieval.

Important Questions

Contribution Guide

- What dependencies and tools are required to install on local machines?
- What environment variables need to be configured for the application to function correctly?
- Which branches are used as the main branches? Which branch should be used for new work?
- Should repositories be forked, or should branches be pushed directly to the origin?
- What Git flow (if any) is followed by the team?
- What naming conventions are used for branches?
- What conventions should be followed for Git commit messages?
- What continuous integration (CI) pipeline is in use?
- How can a pull request (PR) build be previewed? Are there deploy previews for PRs?
- Where can details of PR builds be viewed, and how can build failures be diagnosed?
- How can PR builds be restarted? Are there specific commands, or should force-push be used?
- Is there a preference for rebasing vs. merging when updating local branches?
- Should PRs be squash-and-rebase before merging?
- Who should be tagged for PR reviews, and how many reviewers are required?
- Is there a PR checklist available? (Consider using issue or PR templates for this.)
- Are signed commits required?
- What patterns and tools are used for software testing?
- What is the expected code coverage for different parts of the application?
- Where can test account credentials be found, created, or mocked?

Troubleshooting Guide

- What does the error message "(obscure error)" indicate?
- What are the common causes of the error "(error)," and what steps can be taken to resolve it?
- Who should be contacted for assistance if a specific problem arises?

Process Guide

- What is the end-to-end process for feature development? Are kick-off meetings part of this process?
- What regular meetings are scheduled, and how frequently do they occur? (Scheduling recurring calendar invites may be useful.)
- Which HR charge codes should be used for various types of work?
- What is the process for requesting time off?
- Are there any special team events, such as workshops or team social gatherings?

Document Code

Key Points for Documenting Code Effectively

1. **Importance of Documentation**:
   - Documentation helps others (and yourself in the future) understand the code.
   - Skipping documentation leads to confusion and inefficiencies over time.
2. **Common Reasons for Neglecting Documentation**:
   - **Time pressure**: No time allocated for documentation.
   - **Bad habits**: Coming from environments where documentation is not prioritized.
   - **Ego**: Believing the code is self-explanatory.
3. **Effective Documentation**:

   - Documentation should explain **why** code was implemented, not just **what** it does.

   Example of ineffective comment:

   ```
   // Get all offline users
   const offlineUsers = users.filter(user => !user.isOnline);
   ```

4. **Useful Comments**:

   - **Provide context**, reasoning, or future considerations that are not immediately obvious.

   JavaScript Example (Why a fix is applied):

   ```
   /* NOTE: Temporarily fixes a race condition to give the browser time
   to render the element before we do something with it. */
   setTimeout(doSomething, 0);
   ```

   TypeScript Example (Documenting Prop Type):

```
type Props = {
  /** The user's locale, pulled from their browser settings.
   * Example values: `en-US`, `ar`.
   */
  locale?: string;
};
```

CSS Example (Explaining property use):

```
.parent button {
  /* NOTE: Using opacity here instead of visibility/display to ensure
  that keyboard users can still tab over to this button. */
  opacity: 0;
}
```

5. **Reviewing for Documentation**:
    - Add clarifying comments before submitting pull requests to answer potential questions upfront.

Document App End-to-End Data Flow

Key Points for Documenting App End-to-End Data Flow

1. **High-Level Overview**:
    - **Data Types**: Identify key types of data stored (use relational diagrams or JSON schemas).
    - **Storage Locations**: Clarify where data is stored (database, local storage, cache).
2. **Data Caching**:
    - **Caching Details**: Specify where, when, and for how long data is cached.
3. **Data Entry Points**:
    - **Front-End Entry**: Identify the starting point for data on the front end.
    - **Global Store vs. Component Management**: Determine if a global store is used or if each component handles its own queries.
4. **Data Flow**:
    - **Flow Between Components**: Explain how data travels from the entry point through various components.
    - **Handling Concurrent Access**: Describe how concurrent data read/write operations are managed.
5. **Data Operations**:
    - **Querying/Updating Data**: Ensure there’s a standardized approach for queries and updates.

- **Update Frequency**: Specify how often data is updated and whether updates are batched.

6. **Offline Data**:
   - **Offline Storage**: Document how data is stored offline and synced when the connection is restored.
7. **Real-Time Data Updates**:
   - **UI Synchronization**: Detail how the app ensures real-time data updates and avoids stale data.
   - **Libraries/Patterns**: Mention any specific libraries or patterns used to manage data synchronization.
8. **Supporting Diagrams**:
   - Use diagrams and high-level explanations, linking to more detailed documentation when needed.

Document How to Document

Key Points for Documenting How to Document

1. **Where to Place Documentation**:
   - Use designated platforms such as GitHub wikis, a `/docs` directory, or other chosen tools.
2. **Collaboration on Documentation**:
   - Use tools like Google Docs, GitHub issues (edit comments), or Slack/Teams channels to collaborate.
3. **Approval Process**:
   - Define how documentation should be reviewed and approved before submission.
4. **Maintaining Documentation**:
   - Establish processes for keeping documentation up to date.
5. **Tips for Effective Documentation**:
   - **Examples**: Provide clear, simple examples or code samples.
   - **Linking**: Include links to external resources (documentation, StackOverflow, GitHub issues).
   - **Comments**: Use TODO, NOTE, or FIXME comments to flag issues.
   - **Explain Unconventional Choices**: Document reasoning behind non-standard approaches.
   - **Reference Files and PRs**: Link to related files, pull requests, or issues for more context (use GitHub permalinks to avoid dead links).
6. **Encourage Contributions from New Hires**:
   - New team members should contribute improvements to documentation while onboarding. Their fresh perspective helps identify gaps.

Documenting Features

Key Points for Documenting Features

1. **Feature Description**:
   - **High-Level Overview**: Provide a clear explanation of the feature's purpose, functionality, and how it fits into the system.
   - **Example**:

     ```
     Feature: This feature allows the application to upload or update
     file details (file metadata and paths) in the system. It handles
     both individual files and batch uploads, ensures correct folder
     and file naming conventions, and integrates with database models
     for consistency.
     ```

2. **Code Description**:
   - **Detailed Breakdown**: Explain how the code works, focusing on key functions, their inputs, outputs, and any business logic.
   - **Example**:

     ```
     // Function: upsertFileDetails(data: any, isIndividual: boolean)
     // Handles file upsertion by either updating metadata or creating
     a new entry.

     Parameters:
     - data: Contains folder, file details, and upload path.
     - isIndividual: Boolean to distinguish between individual and
     batch uploads.

     Process:
     1. Retrieve the base directory for PDF files from environment
     variables.
     2. Extract folder and file names from the data parameter.
     3. Generate the filename depending on the upload type (individual
     vs. batch).
     4. Query the database to check if the file exists; update
     metadata or create a new entry accordingly.
     5. Handle related whitelabel forecast files for consistency
     across groups.
     ```

3. **Comments and Code Documentation**:
   - **Environment Variables**: Explain the use of variables like `process.env.PDFDIR` to clarify defaults.
   - **File Naming Logic**: Comment on the logic for generating file names for individual vs. batch uploads.

- **Database Queries**: Explain database lookups and the rationale for updates or insertions.
- **Error Handling**: Add comments or TODO notes for error handling or missing data.

4. **Example Comments in Code**:

```
// Get the base directory for PDF files from environment variables
const pdfDir = process.env.PDFDIR || '/default/path';

// Extract folder and file names from the data
const { folder, fileName } = data;

// Generate the filename based on whether it's an individual or batch
upload
const fullFileName = isIndividual ? `${fileName}.pdf` :
`${fileName}_batch.pdf`;

// Query the database to check if the file exists
const existingFile = await db.findFile(folder, fullFileName);
if (existingFile) {
  // Update the metadata for the existing file
  await db.updateFileMetadata(existingFile.id, data);
} else {
  // Insert a new file entry into the database
  await db.insertNewFile({ folder, fileName: fullFileName, ...data
});
}

// TODO: Handle cases where the database query fails or data is
missing.
```