ChatGPT

Unified Agent Guidelines for ERP Projects

These guidelines apply to both the back-end (**erp-system**) and the front-end (**erp-client**) projects that are generated with JHipster. They are intended to harmonise how we write code, document our work and verify that the system meets the expectations of both developers and end-users.

Purpose

We want to provide future developers and users with enough context to understand why a change was made and how to use the resulting feature. There are two complementary kinds of documentation:

1. **Technical analysis and man-pages** — case-study style write-ups of the underlying workflows and design decisions. These capture the model's reasoning when adding or modifying code, and may include excerpts from logs. They live in the `man_pages` folder of the current module and in the root project's `man_pages` so that a change in either module is visible at the top level.
2. **User stories and manuals** — narrative descriptions of what a user wants to accomplish and how they achieve it. These do not delve into internal class names or file structures; instead they focus on the sequence of pages, buttons and outcomes that a human interacts with. They live in the `user-stories` folder of the current module and also in the root project's `user-stories` folder for discoverability. The user manual (in the `user-pages` folder) should be updated or created to reflect any feature that was modified or introduced.

For every significant feature, especially those related to the report-submission service, you should provide both types of documentation where appropriate. If a workflow can be expressed as user stories then include them; if not, a technical analysis alone is acceptable. The goal is to be thorough but realistic: avoid duplicating obvious behaviours or copying code comments verbatim, and focus on explaining the *why* and *how* behind your changes.

Commit Messages

Write commit messages using a short imperative summary (no more than 72 characters) followed by an optional body. The summary should convey what the change does (“Add audit log to report submission” rather than “Added…”). If the change is complex, the body should explain the rationale and reference any relevant documentation you have added.

Documentation Tasks

When you implement or modify a feature, perform the following documentation steps:

1. **Technical write-up** — Draft a detailed analysis of the workflow(s) affected by the change. Explain the business purpose, the key services or components involved, and why the change was necessary. Include notes on any decisions taken during development. Organise technical documents by subject: within each `man_pages` directory, create subfolders based on the topic, theme or JHipster module that the document covers (for example, `report-submission/` or `lease-liability/`). Save each write-up as a Markdown file in:
2. `man_pages/` at the root of the repository.

3. The corresponding `man_pages/` folder in the module you are working in (`erp-system/man_pages/`, `erp-client/man_pages/` or the deployment project). Create the folder if it does not exist.
4. **User stories** (if applicable) — Describe the scenario from an end-user's perspective. Within each `user-stories` directory, group stories into subfolders that correspond to the feature or module they relate to (for example, `report-submission/` for report submission user journeys). For each story, outline:
5. The persona or role.
6. The steps they take through the UI (pages visited, buttons pressed).
7. The expected result or outcome. Save these in `user-stories/` at the root and in the module's own `user-stories/` subfolder.
8. **User manual** — Update or create a page in the `user-pages/` folder that explains how to use the feature in practice. This can reference the user stories but should be written as a how-to guide. Ensure there is a section for every feature you have modified or added.
9. **Entity definitions** — If you add, remove or rename fields on a JHipster entity, update the corresponding `/.jhipster/*.json` file so that future code generation is aware of the change.
10. **Integrated tests** — Where feasible, write integration or end-to-end tests that follow the user stories. These tests should cover the full workflow from navigating the appropriate front-end component through to the services involved in report submission. This helps ensure that the documented behaviour is verified by the system.
11. **Queries** — If a workflow involves custom SQL, place a PostgreSQL script in `erp-system/queries/` showing the relevant query. These scripts are for review and manual testing; they will not be executed automatically.
12. **Sensitive files** — Update the top-level `.gitignore` to exclude any configuration, credentials or generated artefacts introduced by your change.

While comprehensive documentation is important, it should not impede development. Use your judgement to balance detail with practicality, and avoid labelling the requirements themselves as unrealistic. If a change does not materially alter the user experience or business logic, a brief note may suffice.

Project Structure and Custom Code

Adhere to the standard JHipster layout and naming conventions. Avoid modifying generated service classes directly; instead, implement custom logic in classes suffixed with `Extension` (e.g. `ReportSubmissionServiceExtension`) within the appropriate subpackages (`service`, `repository`, `web.rest`, etc.) under the `io.github.erp.erp` namespace. To apply your enhancements in production without altering the generated resource, create a copy of the generated resource class and append a `Prod` suffix to its name. This `Prod` resource should inject your `Extension` class and serve as the entry point used at runtime. This approach keeps your custom logic separate from the generated code while still being invoked in production.

Keep your code changes scoped to the appropriate module (server, client or deployment) and ensure that any cross-module impacts are documented. For example, a new API endpoint in the back-end should be accompanied by the corresponding UI changes in the front-end and both should be described in the relevant user stories.

Quality Assurance

Create unit tests and run them locally before creating a pull request. Ensure that formatting and linting checks pass. Integration tests should cover the main workflows described in your user stories and verify that the system behaves as documented. If there are limitations that prevent full test coverage (for example, complex UI interactions), document these limitations alongside the user stories so reviewers understand what has been verified manually.

By following these unified guidelines, both the ERP back-end and front-end projects will maintain consistent documentation and efficient development practices.
