Documenting SW projects

One approach – Use this one, or define and follow a better one

8.2.2024

Juhani Välimäki

Principles or goals for good SW documentation

- Maintainable
- => Generate what you can generate automatically
- Document only to the needed level
- Avoid documentation that can be made unnecessary by other means
- School methods are often different ones, as they often introduce phases of some step-wise learning process
  - Nothing bad in that, and while learning even necessary. But leads often to unmaintained and bloated documentation
- In real project add only the documentation and visualization that is necessary
- Just link if the information is available elsewhere
- Provide Table of Contents where each developer can find just the interesting parts
  - Divide the content to shorter modules for easier learning but also for selection based on need and interest
- Optimize understanding and project reading speed, ’never’ project writing speed
  - (Exception: There might be elegant shorter solutions available)

Parts of project documentation

- Environment, tool and project information
- Architecture introduction
- Data model - database design and visualization
- Program code comments
- API documentation

Environment, Tool and Project information

- **Handover** is important in all projects. We never know who might continue with the project
- README.md is our project information de-facto standard.
  - Some Markdown markup examples given on the course. Check them out. Test whether works on GitHub pages.
- Make your project installation and configuration clear to the reader. An average IT professional has to be able to setup everything without further assistance!
- Don’t write redundant information. Thus, no instructions on how to e.g. install Docker. Just list it as pre-requisites and possibly give link to elsewhere.
- Remember to explain the git-ignored secrets config! (But no real values to the git repo (history)!)
  - E.g. .env or .env.local file location and model structure with fake values
- Be modular in your explanations, link to the other .md files in the project.

Architecture introduction

- Give just the big picture, put the reader on the map
- It’s a lot easier to study the project folders and code when one has some kind of idea what to look for
- Maybe some rough visualization of the architecture and very brief explanations of each part or module?
- Thus: Less detail than e.g. in exam question explaining the architecture, which is proof of learning.
- Possible just on this level: **Frontend**: React, MaterialUI, AJAX with Axios, react-router-dom (v6 routing contexts used).
    - Would it be possible to link to e.g. the library list in package.json of a Node project?
- Keep this simple and as short as possible.
- Keep this so generic that there should not be much need for changes later.

Database design and visualization

- In school you have learned good long processes for database design. From conceptual level ER diagrams, to logical level design, normalization, database diagrams, etc.
- Those are to some extent for learning the database design
- Some developers just do the database design and implementation at once (database diagram or just SQL DDL scripts). This of course requires some expertise and experience.
- Many tools offer generation of diagrams based on SQL DDL Create table statements.
  - E.g. DBeaver offers adding more diagrams to the project and selecting which tables you want to include in there. DBeaver calls them ER diagrams, but they are actually **logical level *database diagrams***, table diagrams.
- In addition to generated database diagrams, we need some ***data dictionary*** for:
  - a) avoided aliases/synonyms in project documentation, code and UI (customer, ~~client, buyer, consumer, lead~~)
  - b) agreeing on the units/limits etc. flightHeight: ft? m? km? max? min? accuracy?
  - c) general understanding of some complicated business case concept.
- Many databases offer the **COMMENT ON** feature of the SQL standard. Comments on tables and columns.
- Then we could avoid having separate database documents at all? All generated from scripts?

Program code comments

- First rule: Avoid need for the code comments. Rather try to make your code clear with naming conventions and folder structure
  - Folder **structure**
  - **Naming**: Folders, files, classes, modules, functions, variables, attributes of objects
  - Simple tricks help: e.g. ProductList.js and ProductListStyle.xyz stay alphabetically close in folder listing.
- Then, if still needed, explain the confusing, irregular/unconventional/ or complicated parts only
- Less is more. Quality over quantity. Think from reader’s point of view and starting point, not yours.
- Try to understand things incorrectly, if possible, improve.
- Sometimes writing longer code helps, optimize reading speed, never the writing speed.
- E.g. changing from the **a ? b : c** ternary operator to **if-else** might help the readability of the code and e.g. allow using explanatory variable names and comments next to lines

API documentation

- Libraries exist for generating API documentation based on the API (the interface)
- We just need to add possible comments as some kind of annotation or javadoc-kind of comments
  - (Javadoc: Write comments ín certain way and they go to the Javadoc-tool-generated HTML etc. documentation)
    - Someting like /** */ instead of /* */
    - With parameter etc. annotations with @
  - Microsoft has a similar thing called ”XML comments”
- Thus, maybe use some library or language doc feature instead of a non-updating Word document.

Didn’t we agree through this presentation mostly that we can almost totally remove non-generated, non-code/script-linked documentation? ☺