Quality Software Documentation

Author: Jon Fox

Course: CSC 640

Semester: Fall 2025

Quality Software

Quality software isn't exactly something I lose sleep over until I'm actually reviewing a PR or working with poorly designed software.

Security matters. Is the input being validated? Is auth actually protecting sensitive operations?

Simplicity is huge. High complexity becomes impossible to maintain. Simple implementations are often the most effective for future extension and iteration.

Documentation saves time. Can someone new actually get this running without external assistance.

This Documentation

Throughout this documentation I use the REST API PHP Server Project as an example along with other callouts from previous frameworks and docs I've written.

- **Documentation.md** - Documentation standards and practices
- **Frameworks.md** - Power of frameworks with examples
- **Principles.md** - General principles from experience
- **Testing.md** - Testing principles and approaches
- **Report.md** - Weekly activities from semester work

General Principles

Some general principles I follow up

Fail Fast

It's important to not over analyze everything and get to writing code. I do believe architecting solutions is a good pre requisite, but am also cautious of over architecting.

Often times when architecting there is a gap between the high level overview and the actual implementation limitations that are learned during development.

Don't Repeat Yourself

This one is pretty cliche these days, but I think its a useful thing to keep in mind when writing software.

It's important to reuse functionality. This way it can be extended and maintained in one place.

Separation of Concerns

Microservicing can be a good approach but there is definitely a spectrum. Over microservicing has its own problems with maintenance of many separate services along with their deployments. While massive monoliths can be difficult to maintain for reasons in the inverse.

Authentication scattered across 14 endpoint files means updating auth touches 14 files. Authentication in middleware means one place, one change.

If changing the database requires touching API route handlers, your concerns aren't separated. Good separation means swapping the database only touches the database layer.

Finding the right balance whether microservices vs monolith or code organization depends on team size and complexity. Start simple and split when you need to.

Environment Isolation

Dev, staging, and prod should be as similar as possible but completely isolated.

All of the environmental differences we have should be configurable. So we have ymls or the equivalent that contain the configs we need for separate environments.

In the case of cloud environments we can store the configurations in the cloud environment and pull from there during run/build time.

Automate Repetitive Tasks

If it becomes a repetitive task then we should automate it, like homework 4.

Manual process:

- SSH into server
- Pull latest code
- Run migrations
- Restart service
- Check logs

Scripted:

```
./deploy.sh staging
```

This way we can kick off the script whenever we make changes or deploy, and can mostly

Iterate and optimize over time

This is a big one for me. Don't try to solve every problem at the outset or over optimize.

First we need to actually develop and deploy an MVP like application. From there we can iterate and optimize based on user feedback and metrics.

It's important to collect good metrics like API calls, CPU usage, etc. From there we have a benchmark to compare iterations against.

Keep It Simple

Probably one of the most important to me.

High complexity becomes so difficult to maintain and iterate on. It also becomes difficult for other developers to understand and maintain later.

Developing low complexity applications can actually be difficult. I've found this especially true with AI. AI so often wants to implement the most complex solution possible. Understanding code and what you are doing helps immensely when guiding AI solutions.

The Power of Frameworks

Frameworks solve common problems so we don't have to. Security, validation, routing, and auth are common features - frameworks provide well tested implementations so we can focus on business logic/use cases.

Key benefits: Speed to production, security by default, community support, less code to maintain.

Trade-offs: Learning curve, abstraction complexity, dependency on maintainers, potential bloat. Learning curve is definitely a consideration, some frameworks are like an extension of an existing programming language.

Frameworks are great for enterprise applications that need security, validation, auth, etc.

FastAPI - Python Web Framework

FastAPI is ubiquitous in enterprises using Python for APIs. Teams often switch to Python specifically to use FastAPI.

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Agent(BaseModel):
    name: str
    status: str

@app.post("/agents")
async def create_agent(agent: Agent):
    return {"id": 1, "name": agent.name, "status": agent.status}
```

What you get:

OpenAI Python SDK - Client Library Design

This python SDK is awesome for LLM api applications.

```
from openai import OpenAI

client = OpenAI(api_key=api_key)
response = client.chat.completions.create(
    model="gpt-5",
    ...
)
```

Handles automatically:

- Authentication, headers, API key management
- Retries with exponential backoff
- Rate limiting and quota tracking
- Type-safe responses

FastMCP - MCP Server Framework

FastMCP handles MCP protocol details, is becoming the standard along with the low level MCP SDK by anthropic.

```python
from fastmcp import FastMCP

mcp = FastMCP("Database Tools")

# decorator for tool creation
@mcp.tool()
def search_records(query: str, limit: int = 10) -> str:
    """Search database records by query string"""
    return f"Found results for: {query}"
```

Framework handles: Protocol implementation, tool registration, serialization, validation, error handling.

This framework is awesome for standing up and building an mcp server.

Why Frameworks are Important

Security: Auth, SQL injection protection, secure password hashing. Easier than writing a custom solution in many cases, or we can extend the framework for our use case.

Developer Experience: Clear patterns, excellent docs, active communities, IDE support, generated API documentation.

Constant Iteration: This is especially true for popular open source frameworks. The communities are comprised of many developers that contribute updates and features constantly.

Well Tested: Frameworks are often well tested and have been used for many production use cases.

Maintenance: Community security patches, bug fixes, performance optimizations, backward compatibility.

Real-World Impact

FastAPI's `/docs` endpoint saves hours. Developers test APIs interactively without separate documentation or curl commands.

OpenAI SDK handles rate limiting automatically. Implementing exponential backoff, quota tracking, and retry logic yourself takes days and still misses edge cases.

Frameworks are force multipliers. They handle solved problems correctly so you spend time on what makes your project unique.

They give us the scaffolding we can build and iterate on for future development.

Testing and Quality Assurance

Why Testing Matters

Untested code is likely to contain issues

Cost of Bugs

- Find it while coding: Fix it immediately, minimal cost
- Find it during QA: Someone files a ticket, you context-switch back, big cost
- Find it in production: Users affected, emergency fix, reputation hit, max cost

> Testing can prevent bugs from reaching produciton. Although it won't catch every edge case or environmental difference, it provides a good line of defense against bugs be introduced during initial deployment or further iterations.

Testing Approaches in my HW4

Automated API Testing

Built `test_api.sh` that hits all 14 endpoints. Tests auth, validates parameters, checks error handling.

Runs the entire suite in seconds. Catches regressions immediately after changes. Also doubles as API documentation through examples.

Swagger docs for apis are awesome for local dev and even higher envs. It provides an easy interface for developers and other users to leverage when interacting/testing the endpoints for APIs.

Browser-Based Testing

Created `test.html` with forms for each endpoint and live response display.

Good for visual confirmation and testing the actual user experience.

What Good Tests Cover

Happy path - normal usage with valid data produces expected results

Error cases - invalid inputs return useful error messages instead of crashing

Edge cases - empty strings, null values, max lengths, special characters all handled

Security - auth actually prevents unauthorized access, SQL injection attempts fail safely

Breaking Changes - arguably the most important to me. Breaking changes being caught before being deployed to production is a big saver of time and prod issues.

Documentation

Documentation Basics

Something super close to my own heart is documentation. This was especially critical pre AI, maybe not so much now. Although I still run into a good number of gotchas.

What makes documentation useful is a good title so other devs can easily find it when searching for relevant docs for their use case.

So something like 'API Setup' probably isn't as useful as 'API Gateway Infrastructure setup with Authentication'. The latter title gives far more context and is specific to AWS APIs with the callout of Authentication.

Step by step and Screenshots are the biggest. The best documentation is written while walking through the implementation step by step with plenty of screenshots along the way.

Here is an example of something I've published to be publically available (can't show

Documentation as Code

Code should be self documenting is a common cliche that often gets repeated.

This is valid for certain languages and less so for others. So for example, python is super easy to read and for the most part, understand what is going on. Java, however, is not. It can be difficult to discern what is happening in Java because of the layers of abstractions being used.

Anyone using AI (which is most folks these days) knows that most models include comments in the code they write. I often find these comments too verbose for a lot of these cases and they can be really noisy.

Comments in code are awesome, but they should be used when something isn't clear or could be confusing. Especially with multithreaded py functions, it is good to have a solid doc string to say what this function does and what it interacts with. Making use of keywords to catch a devs eye is important. So for python we have NOTE, HACK, etc. Those