Develop

SOLID Principles:

- Single-responsibility principle:
  We write methods that have discreet purposes and avoid side-effects.
- Open-closed principle:
  We recognise that both composition and inheritance are just tools in our arsenal and use each where appropriate.
- Liskov substitution principle:
  Object polymorphism is inherent in our choices of programming language.
- Interface segregation principle:
  We prefer many topic-specific interfaces over larger more general ones.
  We avoid implementing interfaces we cannot completely fulfil.
- Dependency inversion principle:
  We integrate components with other via interfaces where there is the possibility of that implementation ever changing, including potential changes of implementation to support testing.

YAGNI, ‘You ain’t gonna need it’:

Good developers will often:

- Look for the generic solutions to the specific problem they are solving.
- Plan for future variance and produce highly configurable code.

There are many excellent development practises that contribute to clean and dry code (see below) but that can also be time-consuming and may not ultimately be needed. We aim to balance these against the YAGNI principle.

DRY, “Don’t repeat yourself”:

We look for opportunities to reduce the amount of code we write by encapsulating functionality into discreet, reusable functions. We balance this against the YAGNI principle.

Documenting / Commenting

Informative comments provide help to the next developer to work on an area of code, even when that developer is the original author at some point in the future. We write comments because we recognise that reading code is harder than writing it.

Complex areas of code are especially important to comment, often noting the paths not taken. Public APIs are specified using Open API (swagger).

TDD

Test driven development front-loads the overhead of testing and lowers the overall cost of development. We identify errors early and at a time then the code has been freshly written.

Unit tests provide evidence that a single unit of code works, typically in isolation. Integration tests, sometimes called “sociable unit tests”, provide evidence that a whole sequence of code works together.

When bugs are found we write new automated tests that prove the bug exists before we fix it. Although our initial testing is included in the cost of the story or task in Jira, bug reports that have been promoted beyond a developer environment are triaged via a new ticket in Jira.

Our pipeline collects test-coverage metrics when it executes our tests so that we can monitor and improve on this.

GIT Source Control

Semantic Versioning:

We use git commit tagging to designate versions according to semver principles.

Feature Branching:

Developers create feature branches named after their ticket

Avoiding ‘fox-trot’ Merges:

We rebase our feature branches on top of the master branch and then merge the feature branch back into the master. This ensures that origin/master is always the 1 parent in every merge.

Pull Requests and Peer Review:

Github supports a peer review process that our developers use to validate each other’s code before we merge it into the master branch. This process help raise the quality of the code written and can prevent errors and misunderstandings from entering the main work stream.

Security

Common Principles

- We adhere to the principle of least privilege and deny by default.
- We prefer guides as identifiers to guard against URL manipulation with direct object referencing.
- We exercise judicious CORS configuration
- We issue short-lived JWTs
- For data travelling to outside of our trusted environment, we always use SSL
- We rely on well recognised encryption algorithms, avoiding older techniques, and never try to write our own.
- We prefer not to store sensitive information where possible, including in a cache.
- We recognise the dangers of user-supplied content where it can lead to injection hacking. We sanitise input appropriately, including the use of parameters for database queries.
- Our pipeline includes a static analyser that scans for vulnerabilities, both in our code and in the third-party packages we use.
- We review the third-party packages we include with our software to make sure only those which are necessary are included.
- Included packages are pinned to specific versions that we have tested against, rather than being bound to the latest major/minor versions.
- We employ a third party to perform penetration testing. We redo this with every major release or at least once a year.
- We implement strict user password requirements on length and complexity and prevent common passwords from being used.
- Users are locked out, or delayed from attempting to log in, or presented with a Captcha when a given number of log in attempts have been made. This guards against password-stuffing.

Revision #2
Created 21 September 2023 10:58:29 by James Hall
Updated 21 September 2023 12:32:37 by James Hall