SDLC

- Diligence
- Develop
- Deploy
- Design

Diligence

User Interface Testing

We transform our user stories in Jira into a test plan. The test plan is then scripted to provide an automated, system-level testing of the user interface and overall customer experience. This can be run headlessly in an automated pipeline.

As with the code-level tests above, system-level tests are written to prove the existence of bugs before they are fixed. In this way we build a library of tests that grow over time and provide ever more reassurance.

User Acceptance Testing

We test the system in real-world scenarios with real-world users, to validate that it meets their requirements and expectations.

Non-functional Testing

Some or all of the below may be necessary

- Stress Testing
  Assess the system's performance and stability under extreme workloads.
- Penetration Testing
  Simulate a cyber-attack against the system to identify vulnerabilities.
- Failover Testing
  Deliberately cause components of a system to fail, ensuring that backup components automatically take over (with minimal disruption to usage).
- Chaos Engineering
  Introduce random failures into the system to assess its resilience and fault tolerance.

Bug reporting

Bug reports should include the following attributes to assist in understanding and speedy resolution:

- Title.

- Description
  Including what the testing objective was, what the tester expected to happen and what happened
- Steps to Reproduce
  A step-by-step guide to reproduce the issue.
- Environment
  Where the bug was encountered.
- Severity and Priority
- Any relevant attachments
  Screenshots, error messages, logs, etc

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.

Deploy

CI/CD

Branch Protection

The master branch of the project is projected from direct changes. Changes can only be merged into the master branch, and then only when certain conditions are met.

Feature Branch Triggers

Code can be pushed to a feature branch without consequence. Raising a pull request should do the following:

The PR name is linted against a Jira ticket.

The code is compiled.

The code is subject to static analysis for security purposes.

The unit and integration tests are run.We only run in-process tests here. Some integration tests may be run later.

Code coverage is collected

A new artifact is created for this build.

*A new resource group is created in the cloud and the artifact + any requisite data is deployed there using IaC.

*The system tests are run against the newly created environment.Integration tests that require an environment will be run here.

*Code coverage is collected

* Branch-level deployment is an advanced strategy and a good target to aim for.

Pushing to a feature branch with an outstanding Pull Request re-runs the above checks.If branch-level development is present the testing environment can be safely torn down.

Master Branch Triggers

The code can be merged to the master branch with at least 1 acceptance in the PR process. Merging into master triggers the same actions as above but with deployment to UAT

Release Management

Creating a new tag for a commit on master that matches the semver pattern triggers several sequential actions:

The Staging environment for this project is updated with the infrastructure code from the tagged commit.

The artifact that was created for the tagged commit is deployed to Staging. This process can be triggered manually by specifying the commit to deploy, which facilitates a rollback if we deploy malfunctioning code.

Infrastructure as Code

Our infrastructure is written in declarative yaml, it is executed as part of our pipeline and stored in source control.

Monitoring

Logging in the Cloud

In a cloud environment it can be very challenging to debug software. We should implement detailed logging across the system, both to monitor healthy activity and to shine a light on errors. We will utilise the cloud provider's native logging services for this

Health Checks

Health checks ensure the system is up and responding, and may be used to verify specific system functionalities.

Alerting

If an error occurs, or a health check fails, an alert should be triggered to notify the relevant team members as soon as possible

Design

Baseline Technical Specifications

Where an existing solution is already in place, analyse its implementation, assess the strengths and weaknesses of the implementation, and decide if any parts can be reused. Do this from a technical and user-oriented perspective.

Identify the architecture patterns, technologies used, data flows, security measures and so on.Identify 3 party interactions and their mechanisms.

rd

Target Technical Specifications:

Provide technical solutions that fulfil the agreed requirements. Identify software languages, service interactions, database choices, cloud providers, third party suppliers and other strategic choices.

APIs may also be defined here with the understanding that these are prone to change as the project progresses.

Security is planned from the outset, including where data will be stored and encrypted, defensive network topologies and both authorisation and authentication are considered.

Robustness is planned from the outset, with redundancy in storage and infrastructure across multiple zones or regions. Traffic management, load balancing and DNS records all contribute to this.

Scalability is planned from the outset, either with serverless infrastructure, VM scale-sets or containers and container orchestration. Database partitioning should be considered.

Where there is more than one viable solution, a list of the pros and cons should be collated.

Engage the development team early in this process, their input will ensure the design is technically feasible and utilises their skills effectively. Defining the technical architecture should be a collaborative process, with developers having the opportunity to propose their own solutions and clarify their understanding.Technical architecture is a guide for development, not a rigid plan. Allow for flexibility of implementation but ensure the overall structure and principles are adhered to.

Gather Team and Stakeholder Feedback

Technical assistance is offered to the wider team and key stakeholders to guide them through the implications of the choices they make at this level. The pros and cons of the viable solutions must be discussed and a strategy chosen that is aligned to achieving both technical feasibility and financial sustainability.