ICT Project Guidance

Generic Service Development View

Version:

0.1

Author:

Sky Sigal

## Purpose

The purpose of this document is to summarise expectations for the development of a new system in a reusable manner such that it can be distributed as is to relevant stakeholders, while being referenced from a project’s SAD.

## Synopsis

The development of systems are expected to follow practices, patterns and principles as defined in this document, or adhere to previously agreed best practices.

## Contents

[Description 1](#_Toc154927865)

[Synopsis 1](#_Toc154927866)

[Contents 2](#_Toc154927867)

[Purpose 3](#_Toc154927868)

[Background 3](#_Toc154927869)

[Outcomes [Objectives] 3](#_Toc154927870)

[Options [Considered & Selected] 3](#_Toc154927871)

[Constraints 3](#_Toc154927872)

[Assumptions 3](#_Toc154927873)

[Dependencies 3](#_Toc154927874)

[Decisions 3](#_Toc154927875)

[Deliverables/Outputs 3](#_Toc154927876)

[Heading Level 3 3](#_Toc154927877)

[Heading Level 4 4](#_Toc154927878)

[Appendices 5](#_Toc154927879)

[Appendix A - Document Information 5](#_Toc154927880)

[Versions 5](#_Toc154927881)

[Images 5](#_Toc154927882)

[Tables 5](#_Toc154927883)

[References 5](#_Toc154927884)

[Review Distribution 5](#_Toc154927885)

[Audience 5](#_Toc154927886)

[Structure 5](#_Toc154927887)

[Diagrams 6](#_Toc154927888)

[Terms 6](#_Toc154927889)

# Introduction

Developing, deploying, and integrating a system is more than just writing code. It requires delivering both functional and non-functional requirements while meeting transitional project objectives and obligations, all within the constraints imposed by the environment, sector, organisation, and project.

A system’s functional requirements must support the needs of a wide range of stakeholders beyond the business sponsor, including service consumers, customer support teams, service providers, operations, monitoring, maintenance, and development teams.

At the same time, the system must meet expected qualities of service, assessed against ISO-25010-defined metrics such as security, availability, performance, usability, and accessibility.

Beyond functionality and quality, the service must also be validated, accredited, deployed, integrated, and provisioned according to agreed processes.

Possibly less well understood by developers is that all three of these aspects—functionality, quality, and deployment—as well as the structuring and infrastructure— are shaped by external constraints, including legislation, policies, and procedures specific to the countries, environments, sectors, organisations, and projects in which the system operates.

Without a clear understanding of these obligations and their impact on system design, along with a structured approach to balancing delivery to competing requirements, development efforts risk inefficiencies, vulnerabilities, and costly rework before it can be made available to end users.

This document outlines key development considerations to mitigate these risks and support successful system implementation.

## Legal, Regulatory and Policy based Obligations

All service providing systems must adhere to a foundational set of legal, regulatory, and policy obligations that ensure compliance, security, and accountability.

Legal obligations fall into several broad categories. General laws apply to all persons and services, regardless of their sector. Examples include privacy laws. There are also regulations specific to certain domains, particularly for government agencies, which mandate additional compliance measures. Examples include long-term record keeping, accessibility, and security minimums. Some industries impose trade-specific requirements that, while not laws, are necessary to participate in those markets—for example, compliance with payment processing standards for handling credit card transactions.

Additionally, government agencies are mandated to follow certain local and internationally defined standards when they are available. Examples include NZ Government's Digital Service Standards and the use of international ISO/IEEE standards.

It is important to recognise that while local variations exist, the core principles behind these obligations remain largely consistent across jurisdictions, largely revolving around transparency, privacy, security, accessibility, and maintainability.

The following sections will outline these obligations in greater detail, beginning with those specific to New Zealand.

### New Zealand Laws and Regulations

In New Zealand, all services, irrespective of purpose or industry, must adhere to common legal and regulatory frameworks, as well as mandated policies that govern their design, security, and accessibility.

#### Privacy Act 2020

Obligates obtaining prior consent for the collection, protection, and correction of personal data. Systems must be designed with mechanisms for containing and recording user consent, secure storage, controlled data sharing, and provide means for users to apply for record correction and/or deletion.

#### Copyright Act 1994

obligates systems obtain prior permission and provide attribution as required when disseminating copyrighted material. Systems must track content licensing, enforce copyright rules, and ensure proper attribution where necessary.

If the system is for use by or provided by the NZ Government, a number of additional laws and regulations must also be adhered, each having an impact on the functional or quality or transitional aspects of the system:

#### Public Records Act 2005

obligates government systems to manage data for long-term use by following structured information management practices. Systems must implement metadata tagging, categorisation, and access-controlled workflows for document state progression (e.g., Draft, For Review, Rejected, Approved, Published, Merged, Removed, Restored, Retired, Archived). They must also prevent physical deletion of records, using instead logical deletion, with appropriate anonymisation policies to comply with data retention policies.

#### Official Information Act 1982 & Local Government Official Information and Meetings Act 1987

obligates public sector systems to manage, store, and retrieve official information while ensuring appropriate accessibility to support requests for official information, withholding it only if there is a valid reason to (eg: impacts a person’s privacy). Systems must support structured data storage, discoverability, and controlled access to information, allowing retrieval while protecting sensitive data.

In addition to laws there are Government Directives and mandated policies. These include adhering to the following:

#### Protective Security Requirements (PSR) and NZISM

– Government systems are mandated with the Protective Security Requirements (PSR), which define overarching security policies for information, personnel, and physical security. The New Zealand Information Security Manual (NZISM) provides the detailed technical controls necessary to comply with the PSR. Systems must implement appropriate security measures based on their classification level (UNCLASSIFIED, IN-CONFIDENCE, or SENSITIVE), including authentication, encryption, logging, and intrusion detection. Developers must ensure compliance with these frameworks to meet the government's security expectations.

#### Government Cloud First Policy

– Obligates the adoption of cloud services over on-premise infrastructure unless an exemption is granted. System architecture must be designed for cloud scalability, security, and integration with cloud-native services [while minding long term cost considerations].

#### Data and Information Management Principles (DIMP)

– Establishes principles for effective government data management, ensuring openness, transparency, and accountability in how information is stored, accessed, and shared. This principle dictates system design to facilitate information be as much as possible made accessible (e.g. via APIs) to authenticated services without constraint.

#### Open Data Directives

– Supports making non-sensitive government data publicly available in accessible formats. Systems should be designed with open API capabilities and metadata structures that facilitate data reuse.

### New Zealand Government Standards

New Zealand government agencies are required by the New Zealand Digital Service Design Standard to adhere to established standards to ensure consistency, security, and accessibility in system design and operation.

Agencies are obliged to responsibly self-regulate and, where policy demands, must comply with a standard and provide evidence of compliance.

As such, many of the previously listed directives mandate the use of locally defined standards, which Government agencies must comply to ensure interoperability, security, and accessibility across government services.

These standards include:

* **NZ Government Web Standards** – obligates government services to meet accessibility standards by implementing WCAG compliance for visually and hearing-impaired users, and usability standards by ensuring Interfaces adhere to standardised government branding and usability requirements to maintain consistency and trust.
* **NZ Government Web API Standards** – Defines best practices for API design, security, and documentation to ensure consistent interoperability across government systems.
* **New Zealand Information Security Manual (NZISM)** – Provides technical controls for securing government information systems in alignment with Protective Security Requirements (PSR).
* **Data and Information Management Principles (DIMP)** – Defines principles for responsible government data management and sharing.
* **New Zealand Government Open Access and Licensing (NZGOAL) Framework** – Encourages the responsible release of public data while ensuring privacy, security, and appropriate licensing if required. Systems handling public datasets must support metadata management and controlled authenticated access.
* **Government Enterprise Architecture Framework (GEAF)** – Ensures system architecture aligns with government strategic ICT direction, promoting consistency across agencies.

## ****International Standards****

New Zealand government agencies are directed to adopt international standards as a first preference. If an international standard is unsuitable, agencies should consider developing a New Zealand-specific profile of an international standard before creating entirely new local standards. This approach ensures alignment with global best practices and improves interoperability and efficiency.

At one end of the spectrum, internationally recognised standards (e.g., ISO, IEEE) define formally agreed upon methods that industries and governments mandate. Further along, domain-specific standards (e.g., W3C, IETF, RFCs) provide structure within specialised fields, while industry-based standards (e.g., those used in banking or education) dictate sector-wide interoperability.

Some best practices, such as REST over HTTP or OAuth for authentication, have become so established that they are effectively treated as standards in most implementations. Lastly, certain conventions, such as internal code development guidelines and patterns, are sometimes referred to as "standards" but serve more as project-specific guidelines than enforceable global agreements.

In some scenarios there a several comparable standards to choose from. An example might be JSON versus XML, or OIDC versus SAML. These choices are made by following Best Practices.

Commonly used international standards and widely used protocols -- even if they have not yet been ratified as official standards -- include:

#### Service Quality

* **ISO 25010**  
  defines quality attributes for system and software evaluation, covering security, maintainability, and usability.  
  Provides a checklist of iteratively improvable characteristics of the *system* itself: Secure, Available, Performant, Maintainable, Etc.
* **ISO 25012**defines the quality attributes of the data *within* systems.  
  Provides a checklist of iteratively improvable characteristics of the *data* managed within a system: Accurate, Current, Complete, etc.
* **ISO 25022**defines the qualities of the user experience of systems.  
  Provides a checklist of desired characteristics of the *experience* of users have when using the system: trustability, understandable, safe (even joy of using), etc.

#### Localisation

* **ISO 639**  
  defines language codes for localisation and multilingual support.  
  Provides the basis for ensuring services are provided with language packs for the country’s national spoken languages (e.g., ‘en-NZ’ and ‘mi-NZ’).

#### Information Encoding

* **UTF-8**See Character Set (ISO-10646-2017). Supersedes earlier ISO-8859-1 which proved inappropriate for transmitting and persisting internationally sourced information without loss (i.e., no loss of diacritics, non-Latin characters, etc.).

#### Information Formatting

* **ISO-21778: JSON**Provides the means to use a relatively compact format to transmit structured information between the interface and presentation tiers.
* **XML**A profile of SGML (ISO-8879). Provides the means to transmitting structure information between components, as well transforming into other forms using XSLT
* **XSLT**Provides the means to transform stream of XML information into another shape (e.g. Text, XML, JSON, YML, etc.). While no longer used for UX reasons XML & XSLT continue to have an important part in long-lived services.

#### Data Representation

* **ISO 8601**  
  Standard for representing date, time, and duration formatting, ensuring consistency in time representation, providing portability.  
  Provides a means for creating and persisting events on servers that are in a different time zone than service consumers.
* **RFC 4122: UUID**  
  Defines the UUID (Universally Unique Identifier) format, ensuring uniqueness in distributed systems.  
  Provides the underpinnings for records being created in tiers closest to the end user (e.g.: the presentation tier) as well as facilitating horizontal scaling across multiple app server and/or database server devices.   
  Note: we highly recommend only using timestamp based random UUID (see UUID v7).

#### Data Code Sets

* **ISO 3166-1**  
  Defines country codes used for location-based data consistency.  
  Provides the underpinnings necessary for offering services in multiple locations (e.g.: New Zealand, Cook Islands, Australia, etc.)
* **ISO 4217**  
  Defines currency codes for financial transactions and reporting.  
  Provides the underpinnings necessary to offer services or copyright media usage in local currencies.
* **ISO-5218:2004: Sexes**   
  Provides the underpinnings necessary to record birth facts (DOB, Where, Gender, Names)   
  *Noting that Birth Sex is not to be conflated with Identity Gender claims which are a different, custom, code set*.

When working with finance in systems, refer to the following standards:

* **ISO 8583**: **Financial Transaction Card-Originated Messages — Interchange Message Specifications.**   
  Permits processing of credit card payments for services or the use of copyright media.

Rarely are the following needed for most apps, unless the system is tightly integrated to the organisation’s FMIS system.

* I**SO 20022**: **financial transaction messaging standard for interoperability in payments and banking**
* **ISO 17442**:   
  defines the Legal Entity Identifier (LEI) standard for uniquely identifying organisations.

#### Information Management

* **ISO/IEC 27001**: **International standard for information security management systems (ISMS), ensuring data protection compliance**.   
  Permits improvements to an agencies Data Maturity ranking, support its OIA obligations, support its Records Act obligations, support its Archiving obligations, support its Privacy Act obligations.  
  From a purely system development perspective, this includes enabling the following key outcomes:
  + being able to categorise data to improve discovery later (e.g.: Tag Management),
  + Copyright management (both attribution and controlled access)
  + never physical -- only logical – information deletion,
  + being able to de-identify it when requested or after a customisable duration (e.g. re-associating records to an anonymous user),
  + customisable workflows for managing the state of records from Draft thru Released to Retired.
  + Data being available by authorised consumers via APIs using international standard encoding and datasets (hence the requirement to know about standards that define universal types (e.g., UUIDs) and code sets (e.g., locales, currencies, sexes, etc.)

#### Interoperability

#### Diagnostics

* **SysLog: Diagnostics Standard**Permits an understanding on how to develop diagnostics logging such that it is easier to integrate with enterprise diagnostics and SEIM systems.

#### Interoperability

* **RFC-5322:** **Internet Message Format**  
  Permits invite, engage and notify, current and potential users.
* **RFC-2076:** **Common Internet Message Headers**   
  Permits browsers and other service agents how to return media (images, sound, etc.) (TEXT, JSON, YAML, etc.)
* **RFC-2045:** **Multipurpose Internet Mail Extension (MIME)**  
  Permits attaching email attachments correctly, as well as handling and persisting media uploaded by end users.
* **RFC-6265: Cookies**  
  Permits understanding how to ensure cookies are not transmitted over insecure channels (i.e., marked ‘secure’) as well not accessible via browser code (i.e., marked ‘http-only’).
* **Resource based Communication**: REST.  
  Permits providing simple (as compared to RPC based APIs) APIs for *external* service consumers to implement service clients for interface APIs.
* **ISO/IEC 20802-1: ODATA:** ***compliant* REST Queryability Extension**   
  Permits service consumers to Page, Order, Project, Select, Filter returned data to their needs without requiring the developing a specific endpoint just for that use case, nor sacrificing security or availability.
* **GraphQL:** ***non-compliant* REST queryability extension**   
  Permits service consumers to perform approximately the same as ODATA. The difference is GraphQL is more work -- but *also* able to query multiple data sources IF that a condition that needs solving for.
* **RPC-Based Communication**:   
  Permits cross-tier and cross-component communication within a system (ie, inside the LAN, VLAN, etc.) in communication protocols that are usually more flexible and higher performance than REST which is more suitable for 3rd party consumption.

#### Discoverability

* **RFC-792: Domain Name Standard (DNS)**The standard for naming services for easier discovery and routing on the internet.
* **RFC-1738:** **URLs**   
  The standard that defines web addresses for resources.  
  Permits providing links to resources managed by the service in a standard way.
* **Sitemaps Protocol**Defines a standard way for websites and services to inform search engines about pages available for crawling. Helps with content discoverability, structured URL exposure, and indexing efficiency.  
  Includes being usable to support better Search Engine Optimisation (SEO).

#### Authorisation

* **RFC 8252**  
  Security best practices for OAuth 2.0, ensuring secure authentication.  
  Includes being usable for using an external third pary Identity Provider Service rather than persisting credentials in system.

#### Authentication

* **RFC-8252**Open ID Connect (OIDC): An extension built on top of Auth 2.0, to authenticate and authorise users to systems.  
  Includes being usable for using an external third party Identity Provider Service rather than persisting credentials in system.

#### Security

* **TLS 1.3+ (RFC-8446)**use the latest TLS version for encrypting channels between devices.Includes being usable for secure communication between devices against interception, eavesdropping and manipulation.
* **JWT (RFC-7519)**the format used for encoding information with OIDC tokens exchanges.  
  Includes being usable to implement OIDC tokens.

#### Accessibility

* **ARIA: Accessible Rich Internet Applications Attributes**Required to understand how to meet Web Context Accessibility Guide (WCAG) 2.1+ Level AA baselines, as mandated.

#### Provisioning

* **RFC 764243/44: SCIM 2+**System for Cross Domain Identity Management:Includes being usable by service specialists to provision Users and Roles in systems that don’t have already have in place Application/Invitation/Acceptance self-service onboarding workflow.

#### Documentation

* **ISO/IEC 19501:2004:Unified Modelling Language**Use UML to develop diagrams to document system, integration, code and data design.

### Best Practices

Mature development stacks and frameworks typically implement many of these standards by default. However, it is critical not to assume that using a framework out of the box ensures compliance. Proper configuration and validation are necessary to ensure adherence to the required standards.

#### Code Management Patterns

* **Git-Based**  
  Source control and branching strategies follow Git workflows.
* **Git-Modules**Use multiple Modules rather than putting all the code for a project into a single module.   
  Put in some effort to isolate reusable core/base material from business/project specific code that relies on it. The benefit – beyond reusability which decreases bugs - produces a smaller code base that a support engineer has to understand to fix/extend a system.
* **Branching Per User Story**  
  Each unit of delivery (i.e., a user story) should have its own branch for feature isolation and traceability.
* **Protect The Main Branch**  
  At all times the main branch must remain releasable to production environments. To protect it, merge work branches, along with main, into a secondary branch to check, before merging that branch back to main.
* **Automated Branch Protection Before Merging**  
  Use automation in the pipeline to perform automated merge checks (e.g., static analysis, automated tests, peer review) before accepting changes into the main branch.

#### Development Quality Patterns

* **Strongly-Typed**prefer strongly-typed languages.
* **Strict**at all times develop with the strictest type validation setting.
* **High Warning**at all times develop with the compiler or interpreter set to the highest warning level to reduce issues from the very start rather than have too many to address at the end.
* **Prefer Compilation and Transpilation**Prefer using compilable languages over interpreted when there is choice, to leverage the compiler find errors before run-time.  
  if compiled is not an option, but transpiled is, prefer transpiled next.

#### System Design Patterns

* **Client-Server separation**To improve maintainability and longevity of the solution, prefer separating the interface code base from the service code base. This promotes the adherence to the API-First headless design pattern, but also ensures the interface – the quickest to be outdated part of a system – doesn’t require throwing out the backend service as well.

#### System Organisation Patterns

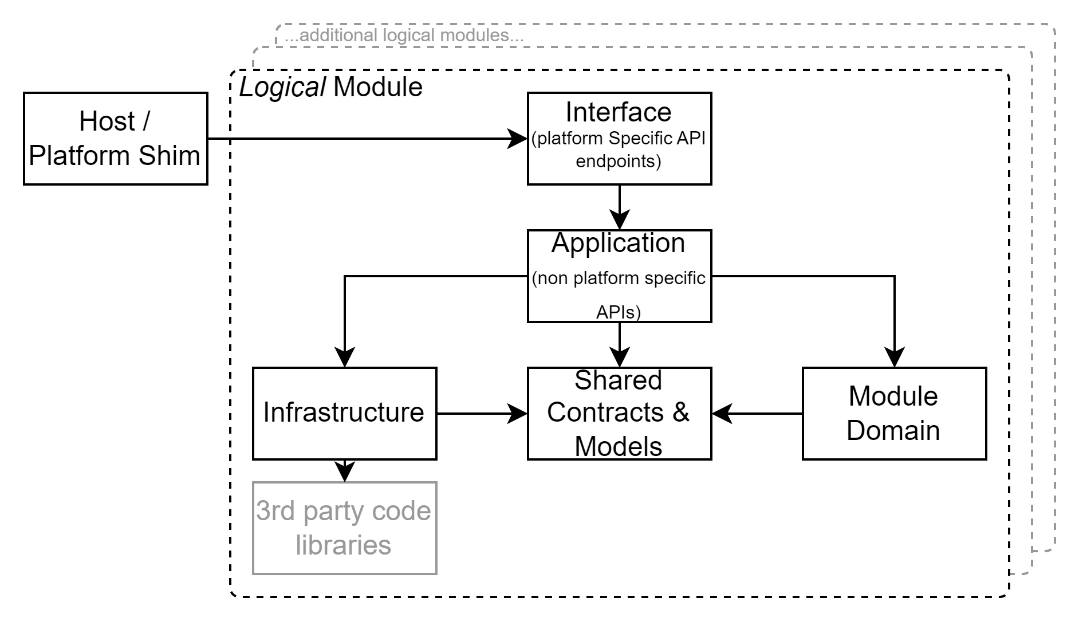
* **Domain Driven Design (DDD)**Follow Domain Driven Design guidance for develop code assemblies or at least areas: App.*Host*, App.*Interface*, App.*Application*, App.*Infrastructure*, App.*Domain*.   
  

Figure 1: Domain Driven Design (DDD) components layout

#### Code Maintainability Patterns

* **Code Comment**Commit profusely inline within the code as to why and how it was developed. Commit with the objective to assist and instruct a more junior developer (i.e. less expensive) developer, or a support developer who will have limited time to engage with the system design (amongst all the other services they must maintain).
* **Git Comments**Reserve Git commit messages to explain the areas changed.
* **ISO/IEC 19501:2004:UML**Use Unified Modelling Language to document development via diagrams.

#### Development Patterns

* **Object Oriented**Where technically possible, follow Object Oriented (OO) coding patterns.  
  Specifically, that Objects manage their own state, and Stateless Services are used to manage multiple objects.
* **Prefer Service Contracts to Singletons**Best practice is to prefer Inversion of Control (IoC) injectable Services to Singletons.
* **General Responsibility Assignment Software Patterns (GRASP)**Best practice is to follow GRASP’s 9 principles/patterns for developing code elements and their interactions: controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations, and pure fabrication.
* **SOLID**Adhere to SOLID development patterns: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle and dependency inversion principle.

#### Data Storage Schema Development Patterns

It is crucial to put effort into understanding the Information elements of a system design before the Technology to persist and manage it (it’s why the I is before the T in IT).   
A poor data schema often dooms a project’s evolvability: any subsequent automation to manipulate the state of the data within the schema will be impacted/impaired.

* **Code-First Approach**  
  Ensure the data storage schema and its updates are defined within the application code, ensuring consistency between application logic and data storage needs, both simplifying deployments and reducing unexpected behaviour later.
* **Avoid Model-First Design**  
  Discourage the use of graphical modelling tools that separate schema definition from the actual system logic implementation, reducing pointless bugs.
* **Avoid Manually coded Schema Development**  
  Schema should not be manually developed via raw Data Definition Language (DDL) statements and instead be managed through code-based automated migrations or schema management tooling (see Entity Framework’s *CodeFirst* Migrations feature for an example of this capability).
* **Avoid Stored Procs**  
  Avoid fragmenting or duplicating application layer logic into other layers or tiers, including the data tier.
* **Avoid Natural Keys  
  A**void conflating *domain* identifiers and *storage* identifiers. In other words, avoid natural keys for clustered indexes.
* **Avoid Conflating Non-Temporal and Temporal Concepts**Avoid conflating temporal and non-temporal entities in a data schema design. In other words, do NOT produce a Customer or Student or Teacher or Parent or table. Prefer instead the use of a Person who temporarily is a logical student due to having an Enrolment at a specific Service offered by an Organisation. Attach Student specific attributes to the [Student] Profile associated to the Enrolment table. These clarifications go a long way to improving the usability, flexibility and longevity of IT investments.
* **Plan multiple Identities and Identifiers:**Do NOT put identifiers on an entity – put it in a joined Value object. This permits the addition of secondary identifiers over time, promoting integration of information across multiple contexts and systems.  
  

#### Data Access Management Patterns

* **Avoid Triggering Save Operations**Avoid invoking save operations. Instead use the UnitOfWork pattern with request handling *middleware* submitting the changes at the *end* of the request. This both diminishes the chance of incomplete *partial* changes of state, as well as reduces performance costly communication between tiers.
* **Cache Reference Data**Consider iteratively working towards avoiding calls to the database to retrieve rarely changed sets of reference data, reducing unnecessary performance expensive traffic between tiers.

#### Operation Processing Patterns

* **Stateless Processing**  
  Ensures that each request is independent of other operations routes or devices, supporting scalability and fault tolerance.
* **Idempotency**  
  Guarantees that repeated requests for changing state are produce the same outcome, reducing unintended duplicate operations.
* **Undoable Processing**  
  Operations should be reversible where possible, using the Command Pattern and event-driven processing (e.g., queues) for failure recovery.

#### Scalability & Performance Programming Patterns

* **Asynchronous Processing**  
  Prefer over synchronous execution to reduce blocking and improve scalability.
* **Cache Often at the Edge**Cache as much as possible in the format closest to end use, on the device closes to the end user.

#### Testing Patterns

* **Automated Testing**  
  Testing is not manual but automated; exploratory testing may be conducted before automation.
* **No Direct Data Access**  
  Testing must not require direct database access, which always introduces a large security issue. All testing should instead be conducted by automation via application interfaces (e.g. APIs that support the UI) or be exploratory via the interface, to inform later automated testing.
* **Static Testing of the Application Layer APIs**Leverage mocking or other approaches to provide complete static test coverage of the Application Layer’s APIs. Essentially, this equates to 100% coverage of *used* code in lower layers, while not requiring testing of code that never comes into play, therefore not an issue.
* **Dynamic Tests of the Application Layer APIs**Deploy the service to a Build Test (BT) environment for automated testing against test data.
* **Test Data Management**  
  Develop data for testing; under NO circumstances should production data be used, even in anonymized or otherwise scrambled form.   
  Just providing direct access to production data to a maintenance person (or worse, a System Support Specialist or Business Analysts) to extract production data to make test data is a complete sidestep of security controls.

#### Infrastructure Development Patterns

* **Developed by Code**  
  Infrastructure should be provisioned and managed as code, avoiding manual configuration.
* **Prefer Cloud Resources**  
  Shared and dynamically horizontally scalable cloud resources should be used, avoiding legacy LAN-based defence models.   
  This often aligns with mandated obligations (see earlier).
* **Prefer Shared Resources**Develop services to be on infrastructure shared with other unknown organisations. Only when uncertain of its ability to secure the system fall back to more expensive dedicated resources.
* **Secure Integration Credentials**Do not embed integration credentials anywhere else than a secure store. All other options – e.g. In the code base itself, or in deployment document – it defeats security.
* **Secure Access to Secure Integration Credentials**Provide access to the secret store to only the deployment pipeline agent.  
  Any human access to integration credentials (to database servers, cache agents, identity providers, etc.) utterly defeats security, reducing it to reliance on unauditable trust.

#### Deployment Patterns

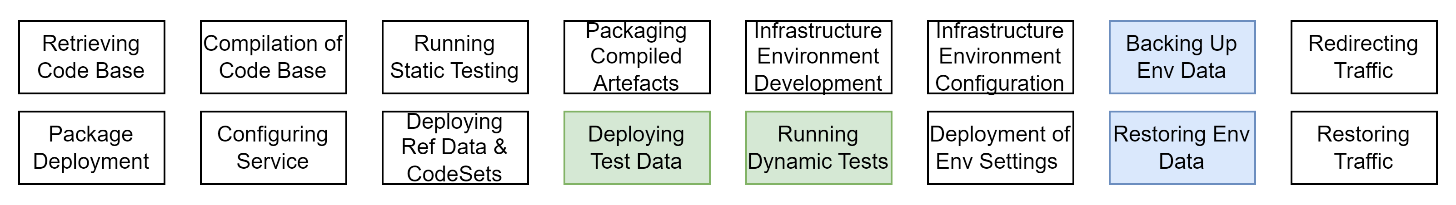
* **Managed Infrastructure Deployment Pipeline**:   
  

Figure 2: High level summary of automated deployment pipeline steps

Use an automated pipeline to manage code additions, testing, deployment.   
Essentially, the pipeline should iteratively be improved to address the following steps:

* + Code Compilation
  + Code packaging
  + Static analysis and automated testing
  + Infrastructure Development
  + Redirecting traffic
  + Package Deployment
  + Configuration,
  + Baseline data (i.e., required reference data & code sets) Provisioning
  + Optionally:
    - Test Data Provisioning
    - Dynamic testing
  + Environment data restoration from backups
  + Traffic Restoration

#### User Functionality

* **Feedback Collection**Systems don’t improve if feedback isn’t collected from end users.
* **Notifications**Keep users informed.

### Impacts

For several reasons, the obligatory application of these international, country specific and trade specific standards, protocols and best practices is less burdensome than it may initially appear. This is because, modern enterprise-grade development frameworks, such as .NET Core, already implement most of these international standards and best practices.

While some historical frameworks—such as Microsoft’s earlier .NET Framework — were not always aligned with best practices (see ASP.NET, WCF, etc.), within commercial constraints, contemporary enterprise frameworks generally adhere to them better.

Important:   
however, despite these improvements, it remains the responsibility of the vendor for the correct implementation of the standards – hence the obligation that the developers understand the standards to properly select a framework, validate and configure the framework if necessary, and demonstrate compliance.

At a high level, the gist can be summarised as:

* Choose a compiled language where possible.
* Choose an enterprise grade development framework.
* Develop with the strictest settings.
* Use DDD to lay out a system’s code components.
* Keep logic in the application tier, avoiding moving logic to an underlying data storage tier (i.e., avoid DB Stored Procs), or lower (the cloud infrastructure tier using Azure Functions or similar) or out of the application layer (use “thin presentation layer controllers”)
* Code for security, configurability, analysability, performance -- in that order.
* Use universal types (UUID, UTC, etc.).
* Follow OO, GRASP, SOLID, development patterns.
* Use international or sector code sets (locale, datetime, currency).
* UTF-8 is the standard for web pages now. Don't use anything else.
* Use the most precise Mime definition available every time (JSON is not TEXT, for example).
* Only transmit over encrypted channels (HTTP/S).   
  Redirect requests to insecure endpoints to secure endpoints.  
  Ensure cookies are marked “secure” so they are not transmitted over insecure channels. And marked “html-only” so they are not tamperable in the browser either.
* Be kind – and because it is the law – to visually and hearing impaired users. Use ARIA tags, meet as a a minimum WCAG AA+ compliance.
* Cache closest to the edge, in the form closest to its use.
* Use OIDC to authenticate Users who are service consumers.
* Use OAuth to authorise Systems that are service consumers.
* Accept all international characters (don’t strip diacritics, etc.)
* Limit modifying input to bare minimums for discoverability. Convert Windows smart quotes, apostrophes and dashes found in fields to simple quotes, apostrophes and dashes. These are introduced when users cut/paste from commonly used office tools (O365, etc.) making it more difficult to find text similar to “O’Brien”.
* Don’t be lazy and rely on firewalls for security. Transmit all information over secured channels. Validate all input, of any type, before persisting it, including uploaded files using an external anti-malware service. Encrypt datastores. Don’t give human access to them (only automation service agents).
* Use REST for Interface APIs. Augmented with OData over HTTPS for APIs. GraphQL. Is an acceptable duplication of OData (although not as a replacement).
* Enable users to invite people to become members as temporary roles in groups.
* But also ensure support staff can provision lots of users and groups, using SCIM.
* Keep refreshing dependency libraries so that delivered code is using the latest versions.

It’s not that difficult once you get into it.

### Common Areas of Difficulty

The following are common areas of difficulty for the delivery of services in government contexts.

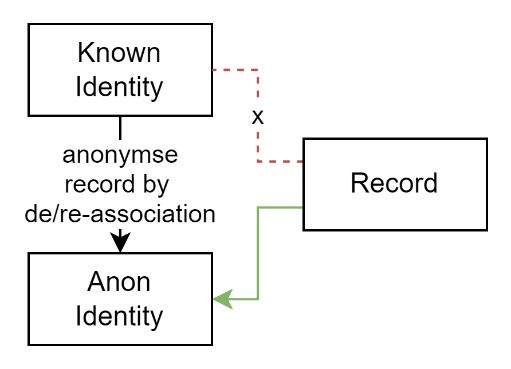
* **Logical Deletions**The service will require that official records are not physically deleted, only logically deleted.
* **Logical Archiving**The service will require to not necessitate separate archiving that needs securing, and a method to securely transfer data to it. Additionally, archived data is not removed from the system or depersonalised failing data retention obligations. Consider instead using a logical flag archiving.
* To meet data retention requirements, the data will require being not accessible. This doesn’t mean physical deletion, as this damages data integrity, but by using logical state changes that inform the system to not include it in responses.
* **Anonymisation**To meet PII data retention requirements data must be able to be anonymised after a duration. This again is not done by deleting records archiving records (the archive still contains the PII data) or deleting records (this damages data integrity) but by taking the *personal* information out of the equation. A method to use is reassociating records from an identified person to an anonymous person, effectively anonymising the data. Clearing or otherwise processing text note fields within the records may also be required.  
   

Figure 3: Anonymise Records

* **Multiple Localities & Cultures**Māori text requires being persisted and transmitted in a lossless manner.   
  This is the basis of the requirement for UTF-8.
* **Search Capabilities**NZ Māori pronunciation requires different solutions than the predominantly English based Semaphone, Soundex, Metaphone algorithms. Solutions that use AI and/or service based search agents have proven effective.
* **Multiple Roles**Government agencies manage records for a long time, often over a lifespan.  
  Examples include Lifelong Learning that can start at Early Learning and continue into Senior Education. During such long durations Persons have different Roles (Student, Training, Teacher, Parent, etc.) in different contexts (Schools, Families, Professions, etc.), sometimes concurrently. This is the basis of understanding that Roles must not be conflated with Persons, and Roles are Temporal (have a start/end date).
* **Security By controlling Role Duration**To ensure roles managed by an organisation are not left open ended in a way that can be used to gain unauthorised access after their role has expired should have an obligatory start and end date. End dates can always be reviewed and extended.
* **Secure Onboarding by Invitation**SCIM was developed for a single organisation, not a multi-organisation context so it’s use case is insufficient for sector use case. In such a case, a well trusted solution to this is to rely on the people closest to the situation, and instead implement a Request (to join), Invitation (to join as a Role), application Submission, review and Deny or Accept (and create a User & Role relationship) workflow. This permits both key workflows – a) where users (e.g.: admins) delegate other users access & permissions (e.g.: teachers) and b) where other users (e.g. parents) can apply to join and be accepted or denied as required. Note, this in turn requires the service to have workflow management capabilities.
* **Versioning Interoperability APIs**It is impossible to enforce service API consumers – other organisations - to invest in service clients upgrades to the same schedule. Versioning is required for continuity of service while being capable of incrementally improving the service.
* **Use Domain Transfer Objects (DTOs)**Never expose internal entities via API. Instead map internal only entities (e.g.: with Automapper) to external only DTOs and vice versa with responses returned.  
  Automapper’s “ProjectTo” extensions permits combining ODATA Queryability over DTOs that is seamlessly translated to internal system entity querying. Without exaggeration, this is frankly, an amazingly powerful combination of usability, extensibility and security.
* **Conflation of User Interface APIs and Integration APIs**While both are APIs, and even REST APIs, they are not the same. User Interfaces in general only permit viewing information from a single account (e.g.: one school’s data). Integration APIs in general produce data across all accounts (e.g.: all school’s), limited by permissions. Conflating the two, or using one for the other often leads to either security issues, or integration issues.
* **Portability**Portability has always been a desired quality. However, while portability of *system* has merit it has traditionally obfuscated where it is truly required: portability of *data*. Recognising that error in logic, Portability has been removed as an ISO-25010 (System) quality requirement, and added as an ISO-25012 (Data) quality requirement. Ensure the system’s integration APIs are sufficient to extract all data via APIs, for future import into the system’s replacement system.
* **Feedback Collection, Notifications, and Reporting**These three areas are often overlooked at design time or thought of as nice to haves, later becoming unavoidable, thereby extending completion timelines.   
  Reports – paper UI – are still required for use by decision makers.  
  Notifications are required to onboard and inform users.  
  User bases decrease instead of increase over time when users can’t inform of issues and suggest improvements.

#### Development Technologies

Organised by deployment Tier, the above is as follows:

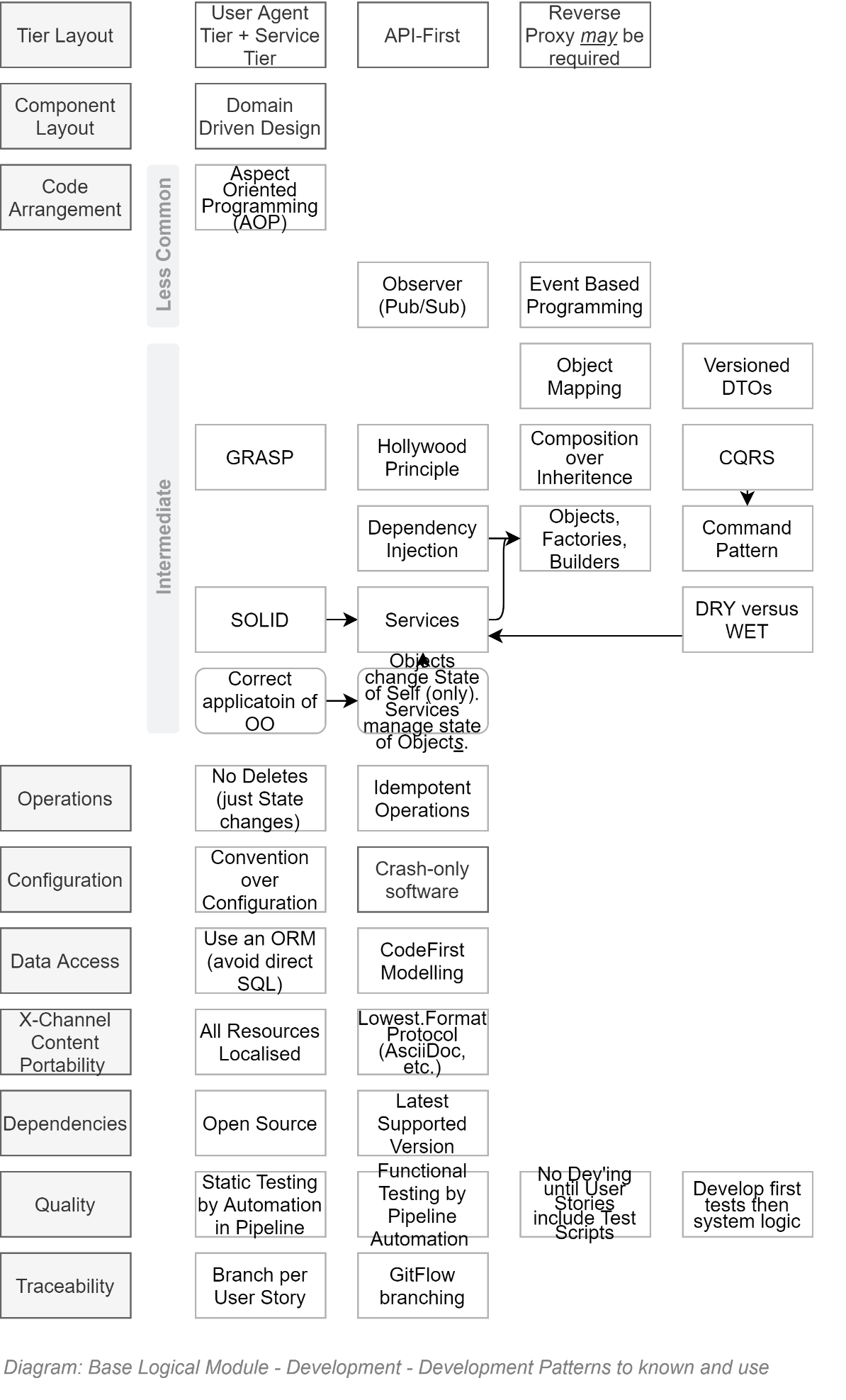


## Development Constraints

## Packaging Patterns

The use of a distributed Version control system is a key productivity driver.

Competent ALM Services – as referenced in the Delivery View –providing



### Pull-Requests

The Agile Manifesto was put together by top-notch developers. Even for them – and certainly for everyone else -- review of code submissions is a valuable process to protect the code base.

Pull-Requests for Review of changes must be issued before branches are merged into target branches.

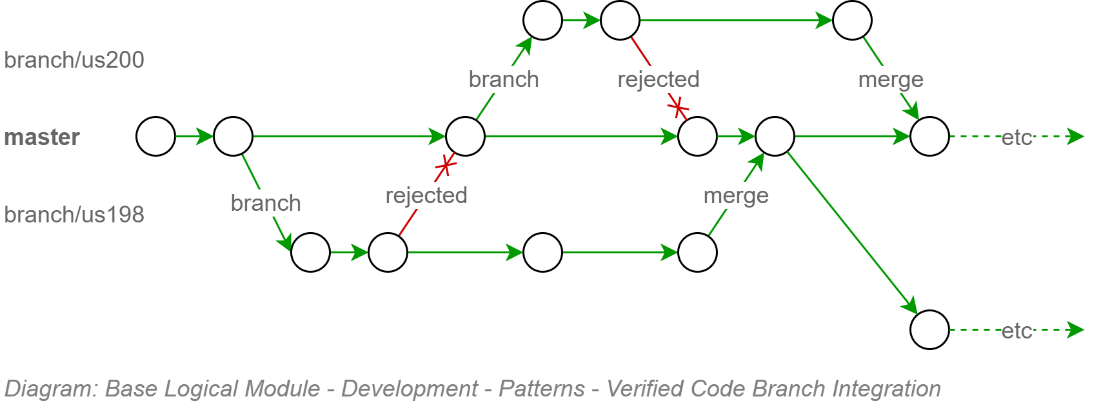
Mature code integration pipelines can be configured to automate reviews of code formatting, code complexity, unit testing, functional testing, etc – basically taking care of the minutia -- before the submitted code is reviewed by a human team member for opportunities to optimise the code.

**Note:**   
if the continuous pipeline has been configured to automate the review of format, complexity, unit testing, and functional testing, review by a human can often be skipped – most apps are not required to be race cars (they really just need to be functionally correct, while meeting quality expectations).

### Verified Code Branch Integration

The Version Control Service employed to deliver the solution performs \*Continuous Integration\* activities and verifies submitted code feature branches before integrating the code with the protected `master` branch.

If the Version Control Service rejects the code due to it failing tests -- or a `Pull Request` reviewer (see elsewhere in this View) has manually rejected the submission -- the developer has to fix it and try again before the Version Control Service will allow the submitted feature branch to be integrated with the protected `master` branch:



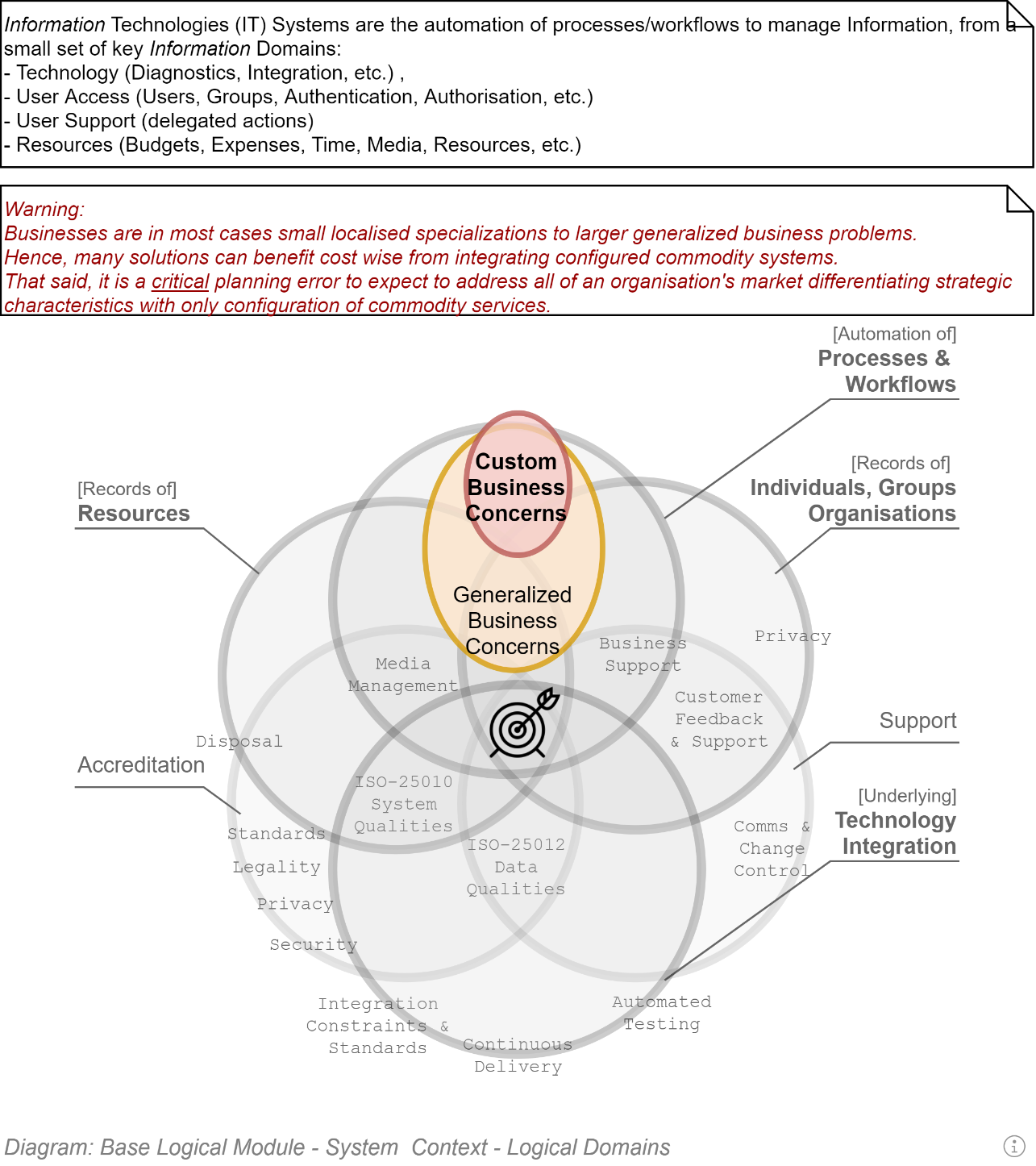
This upfront effort configuring the continuous integration pipeline to protect the code base from getting polluting, protects other developers from downloading poor code and working around it – only to have to remove the work arounds when the original poor code is fixed.

**Tip:**  
This is no different than the kinds of real-world safeguards that would be put in place if we were talking about a communal water reservoir, with everyone tasked with filling it up, and everyone drinking from it. Unless catching typhoid was an objective.

### Component Design Patterns

Domain Integration (DI) Service Design

Domain Integration is the recognition that Services are the composite result of integrating multiple Domains around their unique information.

Reality is a little more complex: the actual number of domains involved in an IT product is higher:  


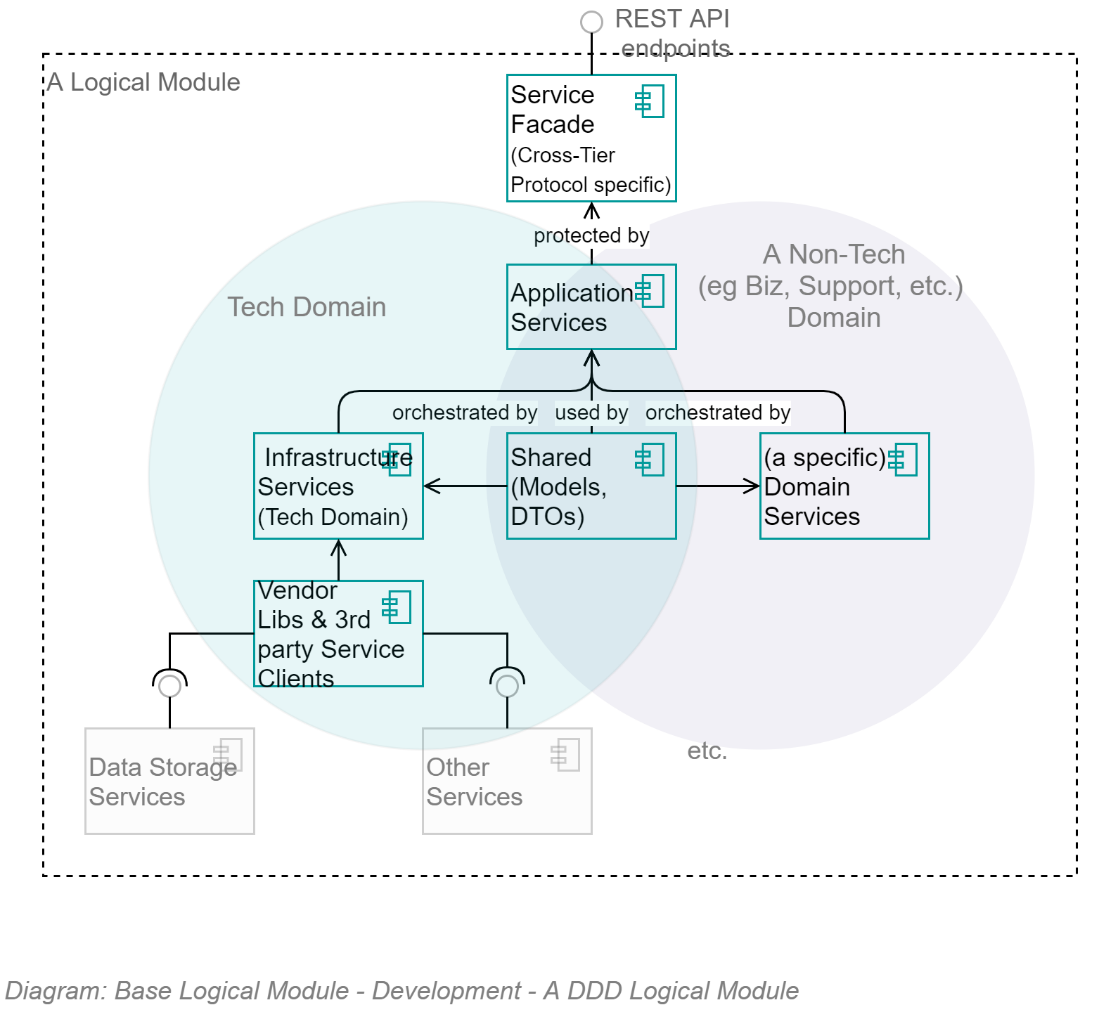
But that just means a couple more Logical Modules.

#### Domain Driven Development

Domain Driven Development is a well-established and successful approach to developing OO systems.

It recognises that there are disparate domains that MUST be kept distinct -- to not become one big unmanageable indistinct mess of logic across multiple domains – and handle the orchestration between them in a thin way.

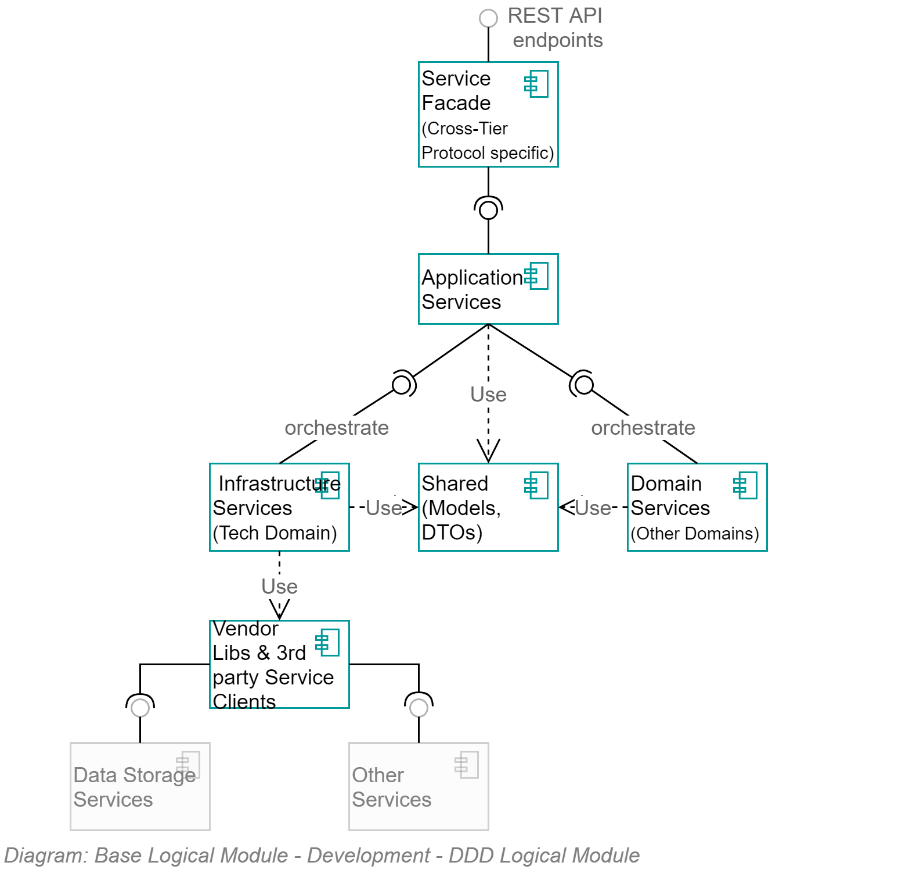
**Note:**  
Domain Driven Design is more holistic than the above short introduction to it and is well worth reading Eric Evan’s seminal work on the matter.



The above diagram demonstrates that, as per Domain Driven Development component composition pattern, all logical Modules are composed of 5 key parts:

* **Infrastructure Services:**   
  a set of infrastructure/technology domain services.  
  Note that a key benefit of the Infrastructure Assembly isolates the rest of the application from dependencies on 3rd party open source and vendor libraries by providing system specific wrappers that provided simplified, app-specific methods, returning to the rest of the system app-Specific messages.
* **[Non-Infrastructure] Domain Services:**  
  A set of (business or other non-infrastructure) domain services
* **Application Services:**  
  A set of Application Service over both of them, orchestrating calls between the two distinct domains, while keeping them separated
* **Shared Entities/Messages:**  
  A shared set of messages/entities that all three sets of services can be passed between them.
* **Service Facade Controllers & DTOs:**  
  A service facade to expose the Application Services to external users as APIs and DTOs.

The same information – this time in a straightforward classic UML Component diagram -- is as follows:



### Object Oriented Development

DDD builds upon and therefore has a dependency on developers applying correct Object Oriented (OO) development patterns.

Object-Oriented development means, in essence two things:

Object definitions are used to create Instances which have public and/or private properties and optional methods to manage their ***own*** properties. Objects do not invoke external services or change the properties of other Objects.

Vice versa, stateless singleton instances of Services definitions are used to manage multiple **other** objects (think of them as specialised helper classes).

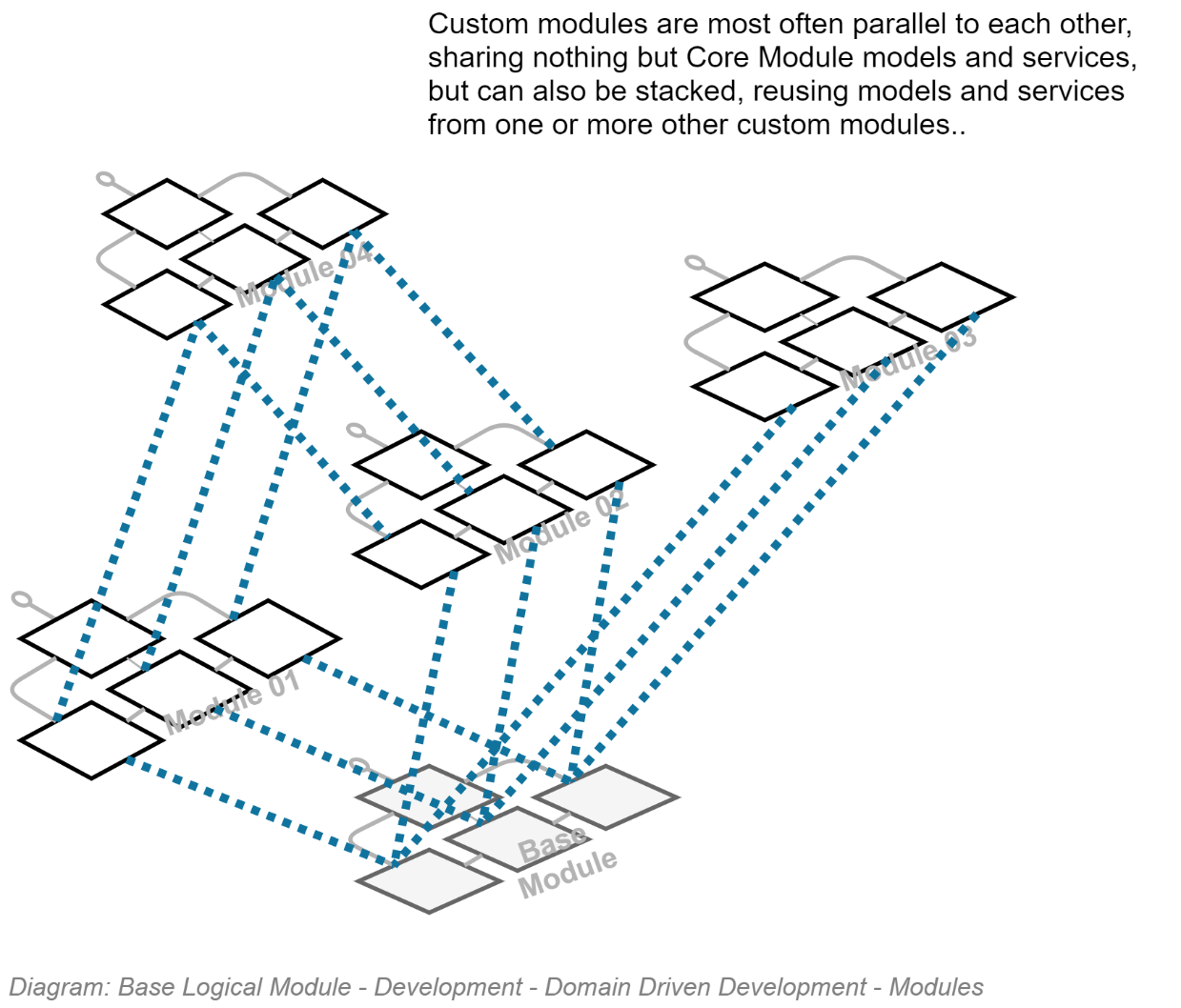
This clarity of each object type’s role is easily demonstratable as follows:

* A bank account object has a read-only Balance property and two methods: AddAmount(amount), RemoveAmount(amount) -- but no TransferTo(accountEntity) method as that would be meddling with another Entity. It also has no Save() method, as that would mean it is invoking/has a dependency on another class, in this case a Service.  
  Note: the ActiveRecord pattern is not conformant with OO theory.
* A BankingService would have no balance value, as Services are Stateless, but would have a Transfer(fromAccountEntity, toAccountEntity) to invoke the RemoveAmount(amount) on one and the AddAccount(amount) on the other.

### Packages

* Modules are a set of assemblies arranged in the classic DDD formation of the following:
* App.ModuleX.Application
* App. ModuleX.Infrastructure
* App. ModuleX.Shared
* App. ModuleX.Domain

**Tip:**  
The Base Module generally has lots happening in its Infrastructure Module and little in its Domain Assembly – whereas other Modules (Business Modules) have generally little to nothing in their Infrastructure Assembly, and more happening in their Business Domain Assembly.

* A solution is composed of a Base Module, on which multiple Business Modules can be developed in parallel:
* 

### Classes

## Security

### Configuration Information

### Service Integration Credentials & Key Vaults

### In Transit

As per the Design Principles and summarised in the Security View, all communication between internal and external devices must be secured by encryption. Therefore, all HTTP based traffic must over HTTPS, maintained by BAU to use the latest version of the TLS standard.

### OWASP

The Open Source Foundation for Application Security Project (OWASP) has developed the API Security Top 10[[1]](#footnote-2). The latest version at this point in time (2020) is the 2019 version, the IDs of the risks being:

* API1:2019 Broken Object Level Authorisation
* API2:2019 Broken User Authentication
* API3:2019 Excessive Data Exposure
* API4:2019 Lack of Resources & Rate Limiting
* API5:2019 Broken Function Level Authorisation
* API6:2019 Mass Assignment
* API7:2019 Security Misconfiguration
* API8:2019 Injection
* API9:2019 Improper Asset Management
* API10:2019 Insufficient Logging & Monitoring

## API Development

### API Versioning

API Version information is an essential element of a complete URI.

Requests using an URL that does not include an API Version number in the URL must return an error HTTP code.

Information  
**Note:**  
The use of a term such as “LATEST” [version] is **not** an acceptable solution, as client systems will stop working as expected upon breaking changes in APIs.

Bullseye**Requirement:**  
Older versions of APIs MUST be kept functioning in parallel for a published period (e.g.: 6 months) to allow service clients time to update their code to consume the latest version.

### Notification & Redirection

Bullseye  
**Quality Requirement:**  
Older versions of APIs must indicate to consuming service clients that they are not consuming the latest version of an API.   
The rationale is that although older versions of APIs are to be kept running in parallel for a duration, this state is not permanent.

Use HTTP headers to return:

* A non-200 HTTP response code.
* A single word enumeration value to indicate the state of the API (Draft, Latest, Superseded, Removed, Retired).
* The versioned URL of the latest version of the API (if Superseded).

Information  
**Note:**As usual when an international standard is not yet available to guide development, avoid re-inventing the wheel, research for the latest best practices, present and obtain approval from the lead architect.

## Services

### Base Infrastructure Services

* The core infrastructure services offered within the Base Module, for use by itself -- and Modules dependent on the Base module -- follow classic DDD recommendations and include (but not limited to):
* **ContextService:** a service to manage the current operation’s context.
* **ImmutableHostSettingsService:** a service to manage immutable settings of the host device (wrapping access to web.config).
* **DiagnosticsTracingService:** a service to trace rolling diagnostic exceptions.
* **ExceptionManagementService:** a service to manage and record exceptions.
* **LocalizedCachingService:** a service to cache resources closest to use, in the format closest to use, by the user agent’s culture-region code.
* **ObjectMappingService:** service to manage the mapping of application entities to versioned DTO objects, and back again.
* **ValidationService:** a service to manage the object and property validation of objects.
* **AuthorisationService: a service to manage access operations by principals.**
* **BlobStorageService:** provides managed control of private and public Blob Storage.
* **TenantManagementService:** a service to manage Tenants.
* **PrincipalManagementService:** a service to manage System Principal records.
* **PrincipalRoleManagementService:** a Setting to manage relationships between System Principals and System Roles.   
    
  Important:   
  May rely on external services (e.g.: EOI, Attributes, Relationships) to provide hints as to how to allocate roles – but role allocation is per system (not monolithic/centrally controlled).
* **SessionManagementService:** a service to manage one or more current active sessions per Principal.
* **SessionOperationService:** a service to persist untamperable auditable records of Principal initiated operations within a Session.
* **MutableSettingsService:** a setting to sync across hosts mutable Settings.
* **MediaUploadService:** orchestration service to manage the upload of media.
* **MediaMalwareVerificationService:** validates uploaded media.
* **MediaMetadataService:** develops a metadata object to describe uploaded media (mimetype, extensions, dates, whether contained malware, whether it was persisted, etc.).
* **RepositoryService:** provides managed access to relational storage.
* **UnitOfWorkService:** a service to manage batch persistence across tiers.
* **NotificationService:** a service to manage notifications to and among system users.
* **IUniversalDateTimeService:** a service to return UDT datetimes.
* **ISMTPService:** a service to deliver outward bound email notifications.
* **IDbContectPreCommitService:** a service to intercept database operations to ensure cleanup and logging is handled centrally.
* **IConversionService:** a service to convert values from one type to another.
* **ITelemetryService:** a service to manage the collection and recording of telemetry.

#### Service Design

* This is a living document: as time permits further technical definitions and design of the above services will be included.

## Service Façade

The Service Façade exposes the following appropriately queryable ODATA REST Controllers.

The following list of APIs mirrors what is expressed in the Service Interoperability View.

## Business Module's Service Facade

* The APIs that are exposed via a Business Specific Module Service Façade are addressed in a separate – thin -- SAD deliverable.

Appendices

Appendix A - Document Information

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Standards

ISO-25010

: …

ISO-25012

: …

ISO-25022

: …

### Acronyms

Refer to the project’s Glossary.

IT

: acronym for Information, using Technology to automate and facilitate its management.

##### ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

### Terms

Appendix B – Domain Driven Design

Appendix C – Core Logical Module Capabilities

The following are key capabilities of the core logical module of mature systems.

The term mature is deliberately used to imply that a developed system doesn’t need to have all these capabilities from the start – but it is important that system designers and developers know of them and have designed a place for them to be added into the correct place, when needed.

In a Domain Driven Design, they would be offered as the Core Module’s Services, such that other Modules can use them.

### Service

* **Diagnostic Message Handling Capabilities**Provide the means to record diagnostics log messages of various importance (Error, Information, Verbose, Debug). Usually stored as a rolling log to a remote, data store (cloud blob storage) that is accessible to authorised consumers.   
  Note that transient diagnostic logs are not to be confused with permanent auditing tool.
* **System Configuration**
* **Error Record Management Capabilities**Provides the capability to save and manage \*permanent\* error records.   
  Purposes include assisting with maintenance, as well as showing increasing qualities as the rate of errors descends.
* **Request Management Capabilities**Provides the means to pre-process and post-process requests.   
  Post processing is the means to ensure state changes are persisted to the data store.
* **Resource Routing Capabilities**Provides the capability to return to authorised users the current version of resources, as well as previous versions.   
  Required to serve different versions of a document as it progresses through states.
* **Request Authorising Capabilities**Provides the capability of securing resources so they are only accessible to authorised users. Consider using a pre-request Handler to return HTML error messages to unauthorised users.  
  Required to limit access to resources to authorised public or authenticated users.
* **Permissions Management Capabilities**Provides the capability of managing Permissions.
* **Notification Capabilities**A system must be able to signal asynchronously to an end user. This is a base requirement for inviting and accepting users to roles in groups, for async processing of workflows, to signal planned downtimes of the service, etc.   
  Not that even if signals can be sent to open browsers, SMTP based communication remains the required baseline.
* **Search Capabilities**  
  The menu, a staple of Windows, Icons, Menus, Pointer (WIMP) based systems has been replaced by spelling-forgiving & phonetic search that return a list of SearchItemSummary items to provide an more efficient manner of searching, discovering, navigating.
* **Data Categorisation Capabilities**Provides the capabilities to tag records for improved discovery later.
* **Data Classification Capabilities**Related by distinct form Data Categorisation, providing a means to tag data by data classification (UNCLASSIFIED, IN-CONFIDENCE, SENSITIVE, etc.
* **User or Principal Management Capabilities**Provides the ability to manage People who are currently using the system.  
  Required to manage current and past users of a system.   
  Do not conflate User/Principal and Person – they are distinct. A User is usually a “thin” object (It just has an ID, and Enabled flag) that is linked to a Person record, as well as one or more Roles within a system Group.
* **User Identity Management Capabilities**Provides the capability of associating 3rd party identities (e.g.: IdP issued identities) to internal system users.   
  Used to manage one or more user logon identities.
* **User System Profiles Management Capabilities**  
  Provides the capabilities for a system user to configure their personal system preferences. Often referred to inaccurately as “user settings”.
* **Queue Management Capabilities**Provides the capability of queuing operations to be processed asynchronously.  
  Queues are the basis of providing availability at lower hardware cost. They are also the basis of avoiding batch processing which is an anti-pattern.
* **Workflow Management Capabilities**Provides the capability of developing workflows to Invite Persons to become Users, to review documents before publishing, etc.
* **Scheduling Capabilities**Provides the capability to schedule tasks.  
  Enables the scheduling of reviews of expiring Roles.
* **Session Management Capabilities**Provides the capability of starting sessions for anonymous users and [optionally] keeping or transitioning to a new session after they authenticate themselves.
* **Session Operation Auditing Capabilities**Provides the capabilities of permanently recording operation requests and responses against the session that has been started for the user.   
  This is an essential capability required to audit and investigate activity, and provide transparency as required.
* **System Configuration Capabilities**Provides the capability of configuring system integrations (cache durations, potentially also integration credentials), system recognisability (logos, etc.).
* **Account Configuration Capabilities**Provides the capability of managing configuration settings for different Accounts.
* **Media & Media Metadata Management Capabilities**Users upload media (avatars, pictures, documents) to support profiles and records. Like anything that is submitted for storage for a service it requires first validation via a 3rd party media stream malware detection service, then storage, along with its metadata (mime type, dimensions, contents summary) to facilitate searching for later.

### People & Groups

Due in part to the tight relationship between User, Persons, Roles, the domain of People and Groups is often conflated with base System design, but remains a distinct logical domain:

* **Person Management Capabilities**Provides the capability of managing people, whether they are non-users, have the potential of being users (e.g.: invited people) , are current users, or ex users (their role within the system has expired).  
  Required to manage people who are not users (e.g., parents of a child) but could be invited to (e.g., to view a child’s progress report).
* **Group Management Capabilities**Provides the capability of developing Nested and Parallel Groups (Organisations, Departments, Schools, Classrooms, Teaching or Interest groups). An organisation is nothing more than a Group that has an Identifier between it and a Group representing a system (in this case the external company’s registry).
* **Person Group Role Management Capabilities**Provides the capability of organising Groups of System Permissions (and optionally Resource Permissions) as Roles associated to Groups.
* **Group Role Association Capabilities**Provides the capability of associating a Role within a Group to a Person.  
  Note that there are long term functionality benefits to being capable of assigning roles to persons who are not yet system Users (it allows setting up Persons before they are invited, for one).   
  Note that it is poor practice to Assign Roles. Best Practice is to Invite a Person to Accept the Responsibilities of a Role within a Group. Once accepted, the Role is assigned for a limited time (e.g., employment might be for 1 year) that triggers Reviews and Extensions in the future.

### Account Management

Account management is a requirement to provide a service to more than one organisation. An account is a specific implementation of a system Group.

* **Tenancy Management Capabilities.**Do NOT implement. See instead Account Management Capabilities.
* **Account Management Capabilities**Accounts are a more correct approach than Tenancy Management which does not permit the sharing of records among partner organisations.

Appendix D – Technologies

Refer to the organisation’s current technological registry.

1. <https://owasp.org/www-project-api-security/> [↑](#footnote-ref-2)