ICT Project

Development View (Services)

Version:

0.1

Author:

Sky Sigal, Solution Architect

## Purpose

The purpose of this document is to summarise baseline aspects of developing services, as opposed to infrastructure, automated testing, deployment pipelines, etc.

## Synopsis

Service development is distinct from other areas of development (infrastructure, data, integration, UIs, testing, deployment, etc.) with its own set of principles and best practices to optimise the value of what is delivered with the least need for costly redirection and rework.

## 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)

## Background

…

## Legal Regulatory & Policy based Obligations.

Refer to the regulatory & policy obligations already covered in the *Baseline Development View* document [1], which summarised the richer information compiled within the *Obligations, Agreements, Commitments, Standards & Principles* document. [2].

## International Standards

The standards that are expected to be adhered to are also defined within the preceding *Baseline Development View* document*.*

## Best Practices

### 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.

Note:  
While strongly typed development slows some developers, it permits the compiler to find errors before execution. For quality reasons, use it.

#### Strict

At all times develop with the strictest type validation setting.

Note:  
While strict development may slow coding, using the strictest type validation settings permits the compiler to find more errors before being found by end users. Use it.

#### 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.

Note:  
Transpilation is probably only really relevant to front-end development.

### 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.

**Note:**Specifically, avoid server-side user interface rendering technologies (ASP.NET, ASP.MVC, etc). Their vendors intentionally broke a long standing design pattern for commercial gain: specifically leading to requiring more servers earlier than if using the service client’s CPU -- in turn leading to more licenses (Operating System, VM, etc.) and more infrastructure needs (more CPU, more memory, more devices).

### System Organisation Patterns

#### Domain Driven Design (DDD)

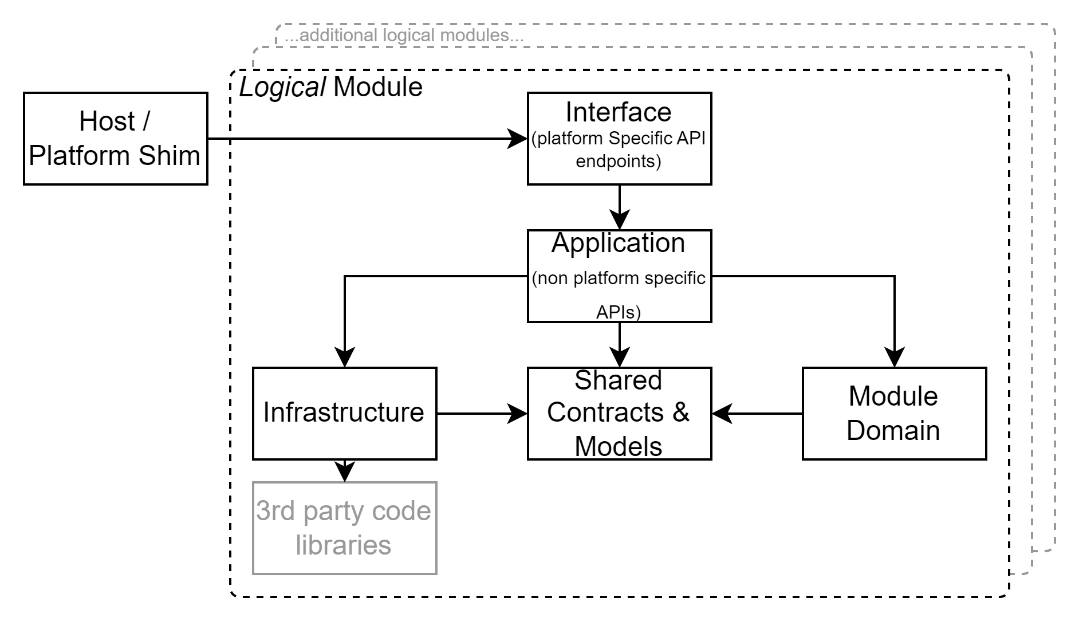
Follow Domain Driven Design guidance for develop code assemblies or at least areas: App.*Host*, App.*Interface*, App.*Application*, App.*Infrastructure*, App.*Domain*.   


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.

#### Diagrams

If drawings are worth a thousand words, diagrams are worth a thousand precise words. Use *ISO/IEC 19501:2004: Unified Modelling Language (UML)* to diagram components, composition, interactions, etc. Consider using C4 or similar diagram types *as well*.

### 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

The following is a succinct list of testing patterns sourced from *IT Project Guidance - Design - SAD - Baseline Development View (Testing)* [3]

### 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.

## Common Areas of Surprise

The following are common areas of surprise that lead to undesired extensions of delivery dates. These areas are especially true in the context of delivering government agency which have regulation obligations to meet before they can be Certified & Approved (C&A) for permission to be deployed to a production data environment.

#### Logical Deletions

The service will require that official records are not physically deleted, only logically deleted. The reasons are both regulatory and technical. For example, regulation exists in several industries and sectors – e.g., financial and government – to ensure data can be audited for context specific durations (e.g. 7 years, or longer). As for the technical aspect, removing records from a relational database is akin to knocking bricks out of a wall: while the wall may survive a couple of missing bricks, whole swaths of bricks being removed after a regulated duration (e.g., 7 years) has significant risk of destabilizing the whole structure, at a time when all original system developers are no longer available to provide insight.   
Finally, operational reporting is affected by removed records. An example may be running a comparison of a teacher-child ratio against historical data. The ratio will be 0 if the data is earlier than the date at which records were deleted.

Note:  
an appropriate method to meeting this objective is to use an Removed value of an information state flag enumeration (e.g., Draft, ForReview, Rejected, Approved, ForPublishing, Published, ForReapproval, Replaced, Merged, **Removed**, [Restored], Deleted).

#### Logical Archiving

Archiving must not require a separate system that introduces additional configuration, accreditation, or operational security overhead. Instead, data should remain within the system and be logically archived using a state flag. This logical flag must not impede depersonalisation tasks.

Note:  
while enterprise data warehouses are often proposed as a secure external archiving service, they typically lack an API by which to request the logical anonymisation of records (physical deletion of records at the data tier layer introduces data integrity issues just as much as at the operational tier).

#### Logical Anonymisation

It is unfortunately common for Privacy consultants to conflate anonymisation and deletion. They actions are distinct. Only the first is required by regulation.

To meet PII data retention requirements personal data must be removed from accessibility after a policy based duration. This requirement cannot be achieved by archiving records (the archive still contains the PII data) or physically deleting records (this damages data and reporting integrity) but by taking the *personal* aspect out of the information.

Note:  
The most appropriate method to harmoniously satisfy the above three competing requirements (i.e., delivering anonymisation outcomes while addressing both deletion and archiving objectives) is to reassociate records from an identified person to an anonymous person, effectively anonymising the data. Clearing or otherwise processing case text note fields within records may also be required in some cases.

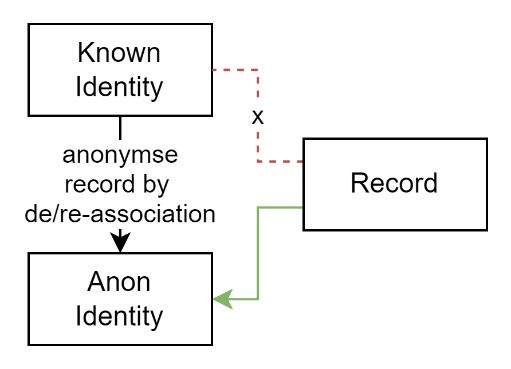


Figure 3: Anonymise Records

#### Multiple Localities & Cultures

Non Latin based characters (e.g.: Indonesian, Māori, Swedish, Indian, Chinese, etc.) characters requires being persisted and transmitted in a lossless manner.

Note:  
This is the basis of the quality requirement for the use of UTF-8 to permit the collection of data from multiple cultures (e.g., naïve, façade, fiancé, Al-Qur’ān, dividend, negociación, associação, Märkte, Корпорация, 資源, 国企, 銀行, 은행, etc.)

#### 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.

Note:  
An accepted best practice is to index and develop search token records in a dedicated search service before saving incoming new and updated records. System search capabilities later use the search service to find the Id of relevant records to develop SearchItemSummary record responses that encodes sufficient information (record type, record id) for an end user to navigate to the found record of any type.

#### 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

While SCIM may be sufficient for managing access within a single enterprise, a broader mechanism is required for multi-organisation environments, where users may be internal to any organisation or entirely external. SCIM is best suited for propagating identities across related systems but lacks the ability to support role invitations, responsibility review, and explicit acceptance of duties. Onboarding into a primary system requires a more structured approach that ensures users acknowledge and accept their roles before access is granted. This, in turn, necessitates workflow management capabilities within the service.

Note:  
A well-tested and trusted approach is to rely on the people closest to the situation, implementing a process that includes a request to join, an invitation to take on a role, an application submission, and a review process that results in either acceptance or denial. Acceptance establishes a user-role relationship, ensuring that responsibilities and privileges are acknowledged before access is granted. This supports both key scenarios: where users, such as administrators, delegate access and permissions to others, such as teachers, and where external users, such as parents, apply to join and are either accepted or denied. The use of invitations is preferred over direct role assignment, as it provides an opportunity to review and accept responsibilities and duties in exchange for role privileges. It also allows for a role to be explicitly associated with a person for a defined duration, such as a contractor’s engagement period, enabling periodic review and renewal. This richer approach ensures effective governance for both internal and external users, complementing or even replacing the need for SCIM in the onboarding process while allowing SCIM to focus on synchronising identities across related services.

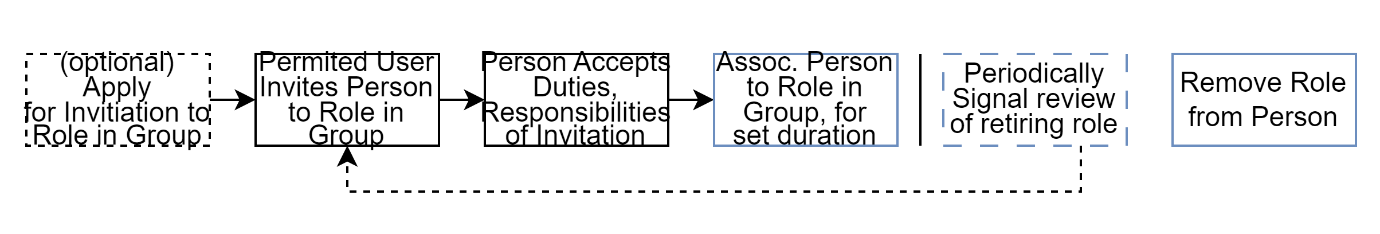


Figure 4: High Level summary of Onboarding by Invitation Workflow

#### Semantic Versioning Interoperability APIs

It is impossible to enforce service API consumers – other organisations - to invest in service client upgrades all at the same time according to schedule. Therefore, versioning is required for continuity of service while being capable of incrementally improving the service.

Note:  
This is the basis of the quality requirement for incorporating semantic versioning into the path (not metadata based!) to ensure link rot doesn’t occur.

#### 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.

Note:  
.NET Core 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, ensure the system’s integration APIs are sufficient to extract all data via APIs, for future import into the system’s replacement system. If proprietary, ensure the use of an industry standard data schema *as well* if one exists.

Note:  
It is common for even IT professionals to have not noticed that Portability was removed from ISO-25010 (System) quality requirements and added instead of ISO-25012 (Data) quality requirement, underlying the fact that it is a more valuable outcome for data to be portable than a system.

#### Messaging

Distinct from Notifications, Messaging uses persistent comms protocols (e.g.: SMTP) to inform users via persistent messages. For example, for onboarding users (welcome, tips, etc).

#### Notifications

Notifications is the alerting end users to status change. It’s one-way (system to user). It is often used to inform system users of system or account state changes (upcoming system downtimes, expiring subscriptions, pending tasks, etc).

Note:  
Notifications doesn’t have an agreed protocol or standard to follow, however [Web Push API](https://www.w3.org/TR/push-api/) could be a good starting point.

#### Feedback Collection, Notifications

User bases decrease instead of increase over time when users can’t inform of issues and suggest improvements to remove frustrations.

**Note:**Consider redirecting to an external service for which SSO can be organised for external users without prior manual configuration. Note that using an external service will lead to a need API based integration (to provide a features list, etc).

#### Reporting

Another area often overlooked at design time or thought of as a nice to have, later unfortunately becoming unavoidable later that in turn extends completion timelines. Consider report generation as a paper-based UI of sorts that is still required by decision-making *stakeholders*, not necessarily system *users*.

Note:  
Consider redirecting to an external service for which SSO can be organised without prior manual configuration. To support this, reporting APIs are required (distinct from UI APIs).

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

|  |  |
| --- | --- |
| [1] | S. Sigal, “ICT Project Guidance - Design - SAD - Baseline Development View”. |
| [2] | S. Sigal, “ICT Project Guidance - Discovery - Obligations, Agreements, Commitments, Standards, Principles (New Zealand Government Agencies)”. |
| [3] | Sigal, ICT Project Guidance - Design - SAD - Baseline Development View (Testing). |
| [4] | Sigal, ICT Project Guidance - Design - SAD - Baseline Development View (Service Client). |
| [5] | Sigal, ICT Project Guidance - Design - SAD - Baseline Development View (Infrastructure). |
| [6] | Sigal, ICT Project Guidance - Design - SAD - Baseline Development View (Deployment). |

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Russell Campbell, Technical Manager |  |
| Rebecca Clark, BA |  |
|  |  |

### 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.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

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

IT

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

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).