SAP Development Standards

- Purpose
- Assumptions
- Related Documents
- AI tools
- ABAP and CDS Programming Rules and Guidelines
  - ABAP Development Environment
  - ABAP Cloud and Clean Core
  - Modern ABAP syntax
  - Package Naming
  - Repository Object Naming
  - Release Contracts
  - Key User Extensibility Apps
  - Business Event Consumption
  - Repository objects that are not allowed
  - Language translation
  - Internal Program Names
  - Modularization
  - Pretty Printer
  - Comments
  - Working with Existing Code and Continuous Improvement
  - Unit and Performance Testing
  - Exceptions
  - Constants
  - Utility Classes
  - ABAP Managed Database Procedures
  - Configuration Table Maintenance
  - Enhancing the SAP Standard
  - ABAP Test Cockpit
  - Tier 2 wrapper
  - SAP RESTful Application programming model
  - Core Data Services
  - Extension CDS View
  - CDS Annotations
  - Enterprise APIs for Integration Scenarios
    - AIF Rules and Naming
  - Event Driven Architecture
    - Metadata Extensions
    - Derived Business Events
    - Business Event Consumption to trigger a custom RAP Business Object
    - Custom RESTful Application Programming (RAP) Business Object
  - Print Forms
  - Workflows
  - ABAP Authorization Check
    - Custom Authorization Objects
  - SAP Screen Personas
  - Developer authorization audit
- Github Repository Guideline
  - Repository Creation Workflow & Responsibilities
  - Version Management in Github
- SAPUI5 and Fiori Elements Programming Rules and Guidelines
  - UI and UX Definition
  - User Experience Principles
  - SAP UX Guidelines for Fiori
  - Development Tools
  - Naming Conventions
  - SAPUI5 ABAP Repository
  - Rules for Fiori applications
  - Unit and Performance Testing
  - Localisation
  - Accessibility
  - Fiori Elements
  - UI Extensibility
- JavaScript Programming Rules and Guidelines
  - Development Tools
  - Naming Conventions
  - Github Repository
  - Version Management in Github
  - JavaScript File Organization
- SAP BTP Development Rules and Guidelines
  - Development Environment for SAP BTP Full Stack Application Development : SAP Build Code & Joule
    - Development Framework: Cloud Application Programming (CAP)
    - Connectivity & Integration: SAP Cloud SDK
    - Development Lifecycle & DevOps Best Practices
    - SAP BTP Space Naming Conventions
  - SAP BTP CAPM Coding Standard and Guidance

Security & Secrets
Code Quality & Standards
CAP-Specific Best Practices
Data & Performance
Localization & UI
SAP Build Process Automation (SBPA) Rules and Guidelines
General Architecture & Deployment
SAP Build Process Automation(SBPA) - Naming Conventions
SAP Build Process Automation : Development Best Practices
SAP Build Process Automation : Integration & Action Projects
Workflow Triggering Mechanisms
Monitoring, Visibility & Error Handling
Mobile Neptune App Programming Rules and Guidelines
Neptune Application Naming
Neptune Application User Experience
Unit and Performance Testing
ABAP Application Class
Backend Calls
Event Handling Success / Error events
Client-side Scripting using JavaScript
Neptune Launchpad Events - OnInit / OnLoad
Mobile Client Build
Publishing custom Plugins to npm
System refresh preparations

Purpose

The guiding principle for programming standards and guidelines at Syensqo is to use what is generally accepted by the industry as “best-practice”, rather than defining a bespoke set of rules. By adopting this approach, it is more likely that developers engaged by Syensqo are already familiar with the “best-practice” approach and can work effectively in the Syensqo environment immediately.

This document’s purpose is to provide developers with the standards and guidance required to develop in Syensqo’s landscape.

Assumptions

All tools required to develop best-practice based are available.

Since SAP development tools and approaches are evolving so does this document.

The SAP Development Approach has been understood.

Related Documents

SAP Development Approach

Integration Architecture

SAP Analytics and Reporting Approach

Security Approach

SAP Integration Development Standards

SAP Analytics and Reporting Development Standards

AI tools

Public web based AI tools can pose significant security risks, and are **not be used** for code generation, test data generation, data formatting or testing in general. Any tools should be Syensqo sanctioned applications that are delivered with the application portfolio or trusted software locally installed.

Any use of LLM based coding agents must be avoided. Syensqo's AI Policy defines guidelines on the use of AI tools and strictly prohibits the use of non-approved AI tools.

Rule	Avoid web based public AI tools for code generation, data manipulation and testing

ABAP and CDS Programming Rules and Guidelines

It is intended that the following rules and guidelines for ABAP and CDS development will align as closely as possible with the SAP Custom Code Lifecycle Management roadmap.

ABAP Development Environment

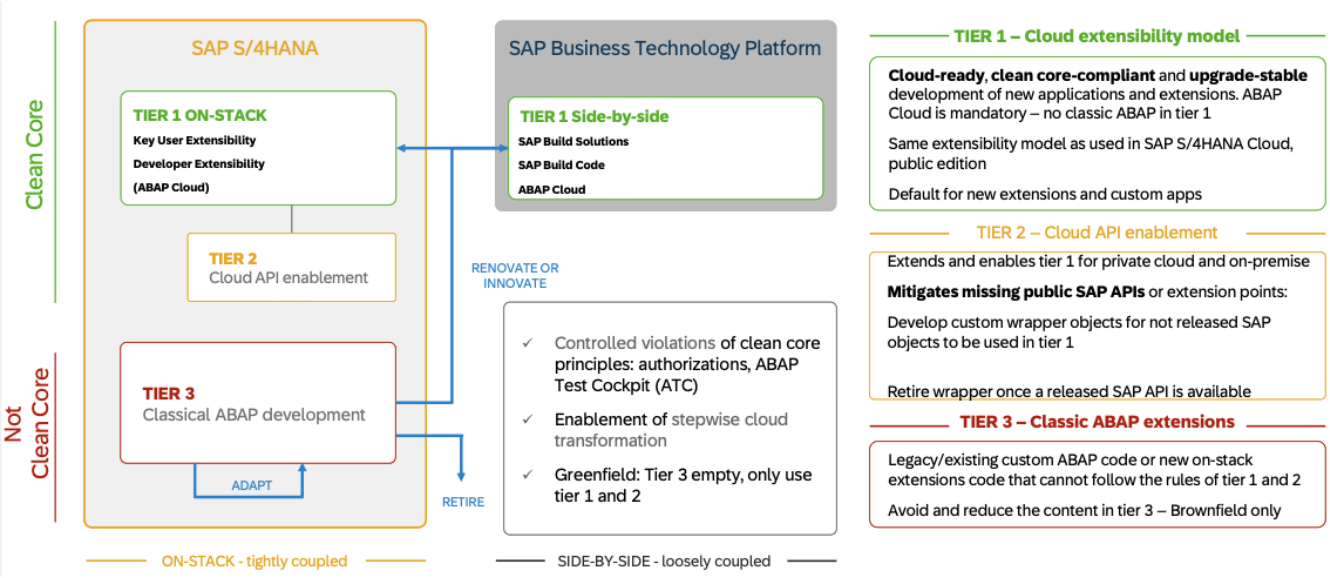
Syensqo has a modern SAP environment. To best meet the development needs the ABAP Development Tools (ADT) in Eclipse will be used.

Rule	The Eclipse-based ABAP Development Tools are to be used as the development environment for all ABAP related development tasks where supported.

ABAP Cloud and Clean Core

ABAP Cloud is the programming model for cloud compliant custom developments. Implementing with ABAP Cloud ensures that the code will be considered Clean Core and upgrade stable. It is the standard for Syensqo.

Please refer to the Basic ABAP Cloud rules and the ABAP Cloud keyword reference documentation.



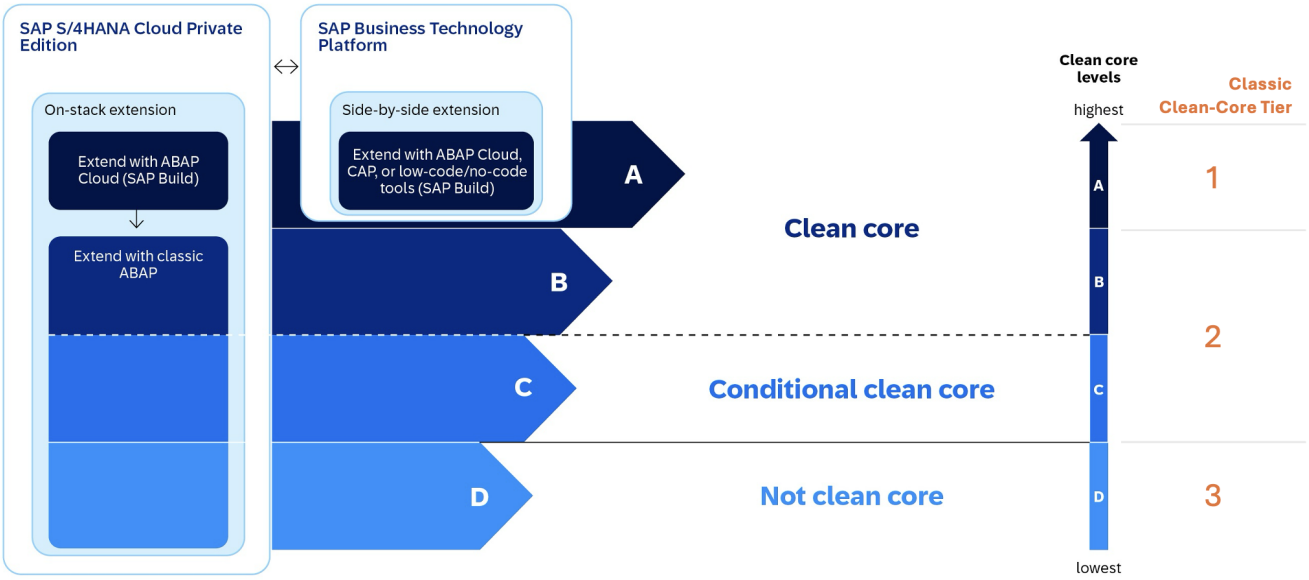
ABAP Cloud is Tier 1 and has to be the default approach for new objects.

Tier 2 is a valid approach only if Tier 1 cannot be done completely.

Tier 3 is deprecated in general, but exceptions can be applied based on each requirement. For example a new classic ABAP print program to be used for an Adobe form, or an implicit enhancement based on business justification and great negative impact if not implemented.

More details of how the different technologies and approaches map to the tiering can be found in the Development Id Definition and Counting Rules document.

Based on the 3 Tier approach SAP refined the tiers into 4 levels with focus on ABAP enhancement techniques and objects. The level-based approach enhances transparency, particularly in the use of classic ABAP code, by offering a more nuanced distinction between upgrade-stable, recommended extensions and those that require timely remediation. The 4 levels fit into Syensqo's 3 Tier approach since it elevates certain legacy objects out of tier 3 into tier 2 due to lack of tier 1 alternatives.



Level A equals to Tier 1 with complete focus on SAP Build tools and key user extensibility. Syensqo's approach has ADT with ABAP Cloud Standard added to Tier 1.

Level B and C compose Tier 2. Level B examples are T2 wrappers and mandated use of legacy extensions, like supported classic Badis, classes or BAPIs. Typical Level C examples include the direct use of unsupported internal function modules, classes, or read access to SAP tables that have not been endorsed for external consumption via CDS or RAP business objects.

Level D equals Tier 3. These are enhancements like modifications or implicit enhancements.

SAP's Cloudification website provides the object classification into "Classic API" (Level B) and "No API" (Level D) for legacy type objects. it is highly recommended to check the object using that website in case of any doubt if a class or function module is Level B, C or D.

We will still classify into 3 tiers, though, since the 4 Levels can be mapped to the 3 tiers. Furthermore, our 3 Tier approach encompasses more than ABAP and includes output forms, workflows, reporting and more.

Rule	ABAP Cloud is to be used for new objects.
Rule	Even if ABAP Cloud cannot be implemented 100%, it is the responsibility of each developer to ensure that the maximum of ABAP Cloud is used.
Rule	For classic ABAP the use of Tier 2 Level C type objects must be limited and only used if no Tier 2 Level B object exists.

Modern ABAP syntax

ABAP syntax has been significantly modernized as of ABAP 7.5, with SAP continually delivering improvements (example). The new syntax supports a cleaner, more concise style resulting in code that is simpler, easier to read and more expressive. Its use results in better code pushdown and more efficient code execution.

Use of ABAP Objects is mandatory. Obsolete syntax has no place in a modern ABAP environment and must not be used.

All new development must adopt *at least* the following ABAP features; these will have the most immediate impact on code quality.

- Inline Declarations and Constructor Expressions. This will remove the need for intermediate variables and verbose initialization routines.
- Table Selection and Comprehension. More expressive use of internal tables will improve readability of the code.

Rule	Modern ABAP syntax (release 7.5x or later) must always be used.

Package Naming

ABAP uses a hierarchical package scheme to organize development artefacts. As discussed in **Code Organization** in the Development Approach, an organizational scheme based on vertical functional decomposition combined with horizontal architectural layering will be used to organize code artefacts in all development environments used at Syensqo.

Good ABAP package naming is critical to ensuring code can be found quickly within the ABAP repository. All custom packages will be created under the **/SYQ/CUSTOM** top parent. Underneath there will be a second layer with a package per business function.

- **/SYQ/RND** - Research and Development
- **/SYQ/MKT** - Marketing, Sales and Service
- **/SYQ/PROC -** Procurement
- **/SYQ/MFG** - Manufacturing
- **/SYQ/QM -** Quality Management
- **/SYQ/SCM -** Supply Chain
- **/SYQ/PM** - Plant Maintenance
- **/SYQ/EHSS** - Safety and Sustainability
- **/SYQ/FIN -** Finance
- **/SYQ/EPPM -** EPPM
- **/SYQ/HR** - HR
- **/SYQ/GTS -** Global Trade Services
- **/SYQ/TECH** - Technical developments independent of a business function
- **/SYQ/UTILITIES** - Re-usable utility classes
- **/SYQ/MIGRATION** - Data migration custom objects

For example:

/SYQ/CUSTOM

/SYQ/FIN (Business Function)

/SYQ/FIN_E_12345 (Business Function plus Type of Functional Specification - Enhancement plus Development Id - 12345)

The parent package depends on the Business Function which can be determined based on the L4 process that needs to be updated in the Functional Specification. The first part of the L4 is based on the L1 that it belongs to. The L1 describes the Business Function.

L1	Package
01	**/SYQ/RND**
02	**/SYQ/MKT**
03	**/SYQ/PROC**
04	**/SYQ/MFG**
05	**/SYQ/QM**
06	**/SYQ/SCM**
07	**/SYQ/PM**
08	**/SYQ/EHSS**
09	**/SYQ/FIN**
10	**/SYQ/EPPM**
11	**/SYQ/HR**
12	**/SYQ/GTS**

Example: **02**.04.03.04. The 02 stands for Marketing, Sales and Service and its package is /SYQ/MKT.

It is mandatory to develop ABAP Cloud Standard compliant code where ever possible the default ABAP language version assigned to the packages should be "ABAP Cloud". If the development is pure classic ABAP, i.e. exits or classic Badis then choose Standard ABAP for the dev id package, Due to a lack of compliant standard objects, if a development requires both types then the main dev id package should be ABAP Cloud and a child package needs to be created of type Standard ABAP with the same name root and "_CL" at the end. That allows us to follow ABAP Cloud standard as a mandatory guideline but legacy objects or classic ABAP as well.

Note the package prefix in **bold** illustrates the requirement for the child package to reflect the parent package name in its prefix.

The above scheme uses only the first level of the Syensqo functional decomposition model. The development package is named per Functional Specification type to reflect the code boundaries and to allow easier navigation between related objects during maintenance/BAU.

To find out the right parent package it is needed to check the L4 process in the Functional Specification.

Classic user exits are implemented in Z includes. The name is mandated by SAP. For these the package **ZCUSTOM** exists. The dev id package, to be created under it, naming stays as is except that /SYQ/ is being replaced with Z. Standard ABAP to be picked for the package.

Rule	The name of the package must follow the prescribed pattern. The package name will use a prefix which clearly identifies its parent.

Repository Object Naming

Each ABAP artefact of a specific repository type must be uniquely named. The name chosen must reflect the use and functional area of the object.

The naming scheme used for ABAP repository objects is comprised of the following components.

1. A namespace prefix that indicates the owner of the object. The SAP standards for this are:
    - Objects starting with a **/SYQ/** are considered Customer Objects. Syensqo has its own custom namespace.
    - Objects starting with **/SYQ/** have been created with key user extensibility tools.
    - Objects starting with a Y are considered local Customer Objects only to be used for temporary POCs.
    - Objects starting with /…/ other than **/SYQ/** have been created using a prefix namespace registered to another specific organization. This may be SAP or a SAP Partner.
    - All other objects belong to SAP
2. In specific circumstances the type of repository object is encoded in the prefix, for example CL for Class or T for Table. Most repository objects are only accessed from the Object Navigator or the Project Explorer where they are organized by object type making encoding redundant. Encoding becomes important if the repository object is referenced in the source code where it can't be seen in the context of the repository.
3. A descriptive name for the object. This **may** contain encoded information about the functional area or business process that the object belongs to. The name needs to ensure that the functional alignment to the process reference model is clear.

The sections below illustrate the naming scheme used a Syensqo: The pattern [NS] refers to the namespace prefix /SYQ/ or Y.

Data Dictionary

Object Type	Pattern	Characters	Example
Table	[NS]**T**_[NAME]	16	/SYQ/T_FAILEDLOG
Data Element	[NS]**E**_[NAME]	30	/SYQ/E_USERNAME
Domain	[NS]**D**_[NAME]	30	/SYQ/D_SYSTEMSTATUS
View	[NS]**V**_[NAME]	16	/SYQ/V_CONNUSER
Lock Object	**E**[NS][NAME]	16	E/SYQ/ORDERDATA
Search Help	[NS]**SH**_[NAME]	30	/SYQ/SH_ACTIVEUSERS
Structure	[NS]**S**_[NAME]	30	/SYQ/S_SKILLPROFILE
Table Type	[NS]**TT**_[NAME]	30	/SYQ/TT_SKILLSPROFILE

Source Code

Object Type	Pattern	Characters	Example
Interface	[NS]**IF**_[NAME]	30	/SYQ/IF_FI_CUSTOMER
Class	[NS]**CL**_[NAME]	30	/SYQ/CL_FI_CUSTOMER
Inbound Proxy Interface	[NS]**II**_[NAME]	30	/SYQ/II_CUSTOMER_MASTER
Outbound Proxy Class	[NS]**CO**_[NAME]	30	/SYQ/CO_WEATHER_SERVICE
Exception Class	[NS]**CX**_[NAME]	30	/SYQ/CX_FI_CUSTOMER
Function Group	[NS][NAME]	26	/SYQ/SUPPLY_CHAIN_PLANNING
Function Module	[NS][NAME]	30	/SYQ/CREATE_PLANNING_DISTRIB

RESTful Application programming model

Object Type	Pattern	Characters	Example
Root node of the data model with transactional processing	[NS]**I_RO**_[NAME]_**TP**	30	/SYQ/I_RO_CUSTOMER_TP
Behavior implementation class	[NS]**CL_BP**_[NAME]_**TP**	30	/SYQ /CL_BP_CUSTOMER_TP
Interface CDS containing Draft Entries only	[NS]**I**_ [NAME] _**DRAFT**	30	/SYQ/I_PurchaseOrder_Draft
Service Binding (oData v2 or v4)	[NS]**[UI or API]**_[NAME]_**[v2 or v4]**	30	/SYQ/UI_Customer_v4
Custom Event	**/syq/**[Operation]	30	/syq/Created
Local Consumption Event Handler Class (optional counter if more than one event handler class)	[NS]**CL**_[REUSE VIEW NAME]_[COUNTER]	30	/SYQ /CL_R_MAINTENANCEORDERTP
Event Binding Type (Type Namespace + Object Type + Operation)	**syqsap.s4.beh.**[REUSE VIEW NAME with no underscore].[Operation]		syqsap.s4.beh.CustomOrder.Updated

CDS

Object Type	Pattern	Characters	Example
Basic/Interface Data Source view	[NS]**I**_[NAME]	30	/SYQ/I_PurchaseOrder
Private Data Source view	[NS]**P**_[NAME]	30	/SYQ/P_PurchaseOrder
Consumption/Projection Data Source view	[NS]**C**_[NAME]	30	/SYQ/C_PurchaseOrder
Value Help view	[NS]**VH**_[NAME]	30	/SYQ/VH_CompCode
CDS extension view	[NS]**E**_[Standard CDS Name]_[2-digit Series Number]	30	/SYQ/E_PURCHASEORDFS_01
DCL	To follow the CDS view entity name	30	/SYQ/C_PurchaseOrder
Basic Reuse View with Transactional processing	[NS]**R**_[NAME]_**TP**	30	/SYQ/R_CustomOrder_TP
Abstract View	[NS]**D**_[NAME]	30	/SYQ/D_CustomOrderCreated
Metadata Extensions of an Abstract View	[NS]**D**_[NAME]	30	/SYQ/D_ProcessOrderCreated

ABAP Managed Database Procedures

Object Type	Pattern	Characters	Example
AMDP Class Name	[NS]**CL**_[NAME]_**AMDP**	30	/SYQ/CL_MM_PURORDER_AMDP
Table Function Methods	[NAME]_**TF**	30	GETPURCHASEDETAILS_TF
CDS Table Functions	[NS]**TF**_[NAME]	30	/SYQ/TF_PURCHASEDETAILS

Classic Enhancements

Object Type	Pattern	Characters	Example
Composite Enhancement Implementation	[NS][NAME]	30	/SYQ/MSC_ARCHIVING
Composite Enhancement Spot	[NS]**ESC**_[NAME]	30	/SYQ/ESC_FI_DATA_MAPPING
Enhancement Spot	[NS]**ES**_[NAME]	30	/SYQ/ES_FI_ADDRESS_FORMAT
Enhancement Implementation	[NS]**EI**_[NAME]	30	/SYQ/EI_ARCHIVING_CONTACTS
BAdI Definition	[NS][NAME]	30	/SYQ/ALERT_NOTIFICATION
BAdI Implementation	[NS][NAME]	30	/SYQ/ALERT_NOTIFICATION_EMAIL

Workflow

Workflow artefacts are assigned a unique identifier by SAP when they are created. Semantic information can only be provided in the Abbreviation field (12 characters) and the Name text. Abbreviations do not need to be globally unique, and the system will warn if a duplicate is found.

The Abbreviation Field should encode the intention of the artefact as best as possible and then use the Name text to clarify the objects' purpose.

BRF+

Object Type	Pattern	Characters	Example
Application	[NS]**APP**_[NAME]	30	/SYQ/APP_MM_GOODS_MOVEMENT /SYQ/APP_MDG_MATERIAL_CHECKS
Rules	[NS]**R**_[NAME]	30	/SYQ/R_ADDRESS_CHECKS
Ruleset	[NS]**RS**_[NAME]	30	/SYQ/RS_TRIGGER_OUTPUT
Function	[NS]**FN**_[NAME]	30	/SYQ/FN_OUTPUT_TYPE
Action	[NS]**A**_[NAME]	40	/SYQ/A_MATGRP_INVALID_ERROR
Decision Table	[NS]**DT**_[NAME]	30	/SYQ/DT_TM_PUR_ORDER_TYPES

Web Dynpro

Object Type	Pattern	Characters	Example
Web Dynpro Application Configuration	[NS]**A**_[NAME]_**CONF**	30	/SYQ/A_FI_CUSTOMER_UPDATE_CONF
Web Dynpro Component Configuration	[NS]**C**_[NAME]_**CONF**	30	/SYQ/C_FI_OVERDUE_INVOICES_CONF

Others

Object Type	Pattern	Characters	Example
Adobe Form (with Fragments)	[NS][NAME]	30	/SYQ/MAIL_REDIRECTION_NOTICE
Adobe Form Interface	[NS][NAME]	30	/SYQ/PRODUCT_FACT_SHEET
Message Class	[NS][NAME]	20	/SYQ/FI_MASTER_DATA
ABAP Transformation	[NS][NAME]	40	/SYQ/COST_OBJECT_TO_JSON
Authorization Object	[NS][NAME]	10	/SYQ/COSTC
Shared Object Area Class	[NS][NAME]	30	/SYQ/PRODUCT_CACHE

Release Contracts

In ABAP Cloud, the "Release Contract" is the formal mechanism that turns a custom object into a stable repository object and determines how it can be used or extended. Without this, the object remains "internal" and restricted.

Contract types are:

- Contract C0: Extend: This contract ensures stability at dedicated extension points in APIs to allow extensibility.
- Contract C1: Use System-Internally: This contract ensures a technically stable public interface for use in custom development objects created in the ABAP development tools for Eclipse, or using key user apps like Custom Business Objects or Custom CDS Views.
- Contract C2: Use as Remote API: This contract ensures a technically stable public interface for use as remote API.
- Contract C3: Manage Configuration Content: This contract ensures a stable persistence for own configuration content that can be exported, imported, displayed, and edited using dedicated APIs and editors.
- Contract C4: Use in ABAP-Managed Database Procedures: This contract ensures a technically stable public interface for use in ABAP-Managed Database Procedures (AMDP).

Contract type C0 is not to be used.

Contract type C1 and C2 are the main contract types.

Contract type C3 is implicitly set by SAP for all configuration tables and does not need to be set.

Contract type C4 is only for AMDP and optional.

Rule	In ABAP Cloud developments assign release contract C1 for locally used CDS views and classes and C2 for APIs.

Key User Extensibility Apps

Key user extensibility apps like the **Custom Fields & Logic applications** are used to create custom fields and Badi implementations in a clean core compliant way. The fields can be added/ published to CDS model, Service model, UI & Reports, Email & Form templates. Clean core compliant BAdIs are also being implemented using the apps.

Key User Extensibility Object Naming		
Syntax	**Convention**	**Example**
Label / Description	<Meaningful Label or Description>	Order No.
Identifier	/SYQ/<Label>_<SAP Suffix>	/SYQ/OrderNo_BDC
Tooltip	<Meaningful Description>	Purchase Order Document Number

The **Custom Fields & Logic** apps support different scope to be activated to use the new custom fields. Activation of all backend related field integrations for available oData, SOAP and BAPI to be done by default. UIs and Reports, Email and Form Templates, Business Scenarios based on specification requirement only. Other Extensibility apps to maintain forms templates, logos and more exist.

Rule	Use the Custom Fields & Logic app whenever possible to follow SAP's roadmap to Clean Core.

Business Event Consumption

Business event consumption provides a mechanism for a loosely coupled and flexible way of triggering follow-on processes after a business event is triggered (ie. Purchase Order Created) where the event producer (originator) does not need to know the consumers of the events. Moreover, it allows a fan-out approach where a single business event can trigger multiple actions that can independently work. In addition, the potential for technical debt is lowered by reducing the tightly coupled integration points (e.g. BADIs and bespoke code within the core application processing). RESTful Application Programming (RAP) business events can be consumed within the system (locally) in a clean core compliant way.

For follow-on processes requirements (ie. Update purchase order after purchase requisition change), consider Business Event Consumption rather than creating a BADI implementation that updates another business object. Moreover, for any requirement of Application Job triggering within a BADI, consider Business Event Consumption first.

Note that Key User Extensibility Custom Logic is still the preferred option for logic and data changes within the same business object.

To be able to use business event consumption, the Behaviour Definition (BDEF) with events must be released for customer use and either extensible or allowed for use in Cloud development (either with Release Contract C0 or C1).

Rule	Use Business Event Consumption whenever possible for follow-on processing rather than using BADI or Key User Extensibility Custom Logic

Repository objects that are not allowed

The following repository objects are **not** to be used at Syensqo. For further information see the SAP help entry on Obsolete Language Elements.

- Business Object Type (BOR). Custom Workflows must use ABAP Objects.
- (Web-)Dynpros and associated artefacts (does not include generated table maintenance screens). Modern SAP web technologies like UI5 are to be used instead.
- Logical Databases – These are now obsolete
- ITS Service – The Internet Transaction Server is an old web technology and should not be used.
- ABAP Type Groups – These are now obsolete
- ABAP Object Services - The object services framework delivers a lifecycle model for ABAP Objects, allowing them to be stored, queried and modified within a defined transaction scope. Although this sounds valuable, the overhead of the framework, the tight coupling to the database and the compromises made to the software architecture when using the framework mean that it's not to be used.

Furthermore, the following repository objects are **not** to be used in ABAP Cloud and Clean Core:

- Custom transaction codes for ABAP report programs. Native Fiori apps are the default.
- Custom ABAP reports and includes except for programs used in background jobs.
- Custom function groups and modules except for generated maintenance views or if a RFC function is required.
- Implicit enhancements (unless the specific approval from the SyWay Design Authority has been secured)
- Classic CDS views. CDS View Entity type to be used.
- SAPScript and Smartforms. Adobe Forms with Fragments is the default, with classic Adobe as fallback
- Custom Search help via exit. Search helps in UI5 apps need to use exposed CDS views or implementation in a RAP BO.
- Gateway service projects for oData services. RAP based services to be used instead.

Language translation

If not specified otherwise in the specification document, all texts and descriptions need to be translated into the 4 core Syway languages, English, French, Italian, Chinese (Mandarin). Please refer to KDD055 - Multi-Language Support for more details.

Internal Program Names

Internal program names are those used within the source code, such as variables, types and methods. As per the naming guidelines in **Code for Readability**, choose names that include detailed semantic information relevant to the current program context. For example, the name should reflect if the object is singular or plural (a collection).

A minimal approach to prefix encoding is encouraged. Technical information about an object can be obtained from the ABAP IDE and so these details **mus t not** be encoded in the name. As ABAP is a strictly typed language the compiler will identify any type safety issues.

Names that conflict with ABAP keywords should be avoided. Although this is not strictly enforced by the compiler, it can affect readability. ABAP keywords are easily identifiable through syntax highlighting, and an effort should be made to change the name when a conflict occurs.

As ABAP is case insensitive, lower case is to be used for internal names. Underscores should be used for clarity between words, for example user_name or address_for_delivery.

Local classes and interfaces are not to be prefixed.

When using inline declarations ensure that the data type of the variable is clear from the right-hand side assignment.

Constants to be prefixed with **c**_.

Global variables must start with the prefix **g**_. This makes identification of global variables during refactoring and debugging clear. Any other Prefix encoding is **not** to be used for variables – this includes local variables and class attributes. In the case when a conflict occurs between class attributes and local variables then the **me->** prefix should be used to explicitly identify the class attribute. This is considered more appropriate than using **l**_ for the local variable.

Parameters should be prefixed with **i**_ (importing), **e**_ (exporting), **c**_ (changing) and **r**_ (returning) allowing them to be distinguished from variables with the same semantic meaning. This applies to both method and function parameters.

Modularization

Modularization in ABAP is a process by which functionality is broken down into smaller manageable components. ABAP supports the following modularization techniques:

- Classes through the use of ABAP Objects (Global or Local)
- Methods
- Subroutines within a Program or Function Module using the FORM statement **(Only allowed in the Syensqo environment for specific use cases)**
- Function Modules within a Function Group **(Only allowed in the Syensqo environment for specific use cases)**

The process of decomposing functionality into smaller modules is a key practice for achieving a **Separation of Concerns** and adherence to the **Single Responsibility Principle** as discussed in **Code for Maintainability** in the Development Approach.

A method or subroutine should **not have more than 200 lines** to fulfil Separation of Concerns and the the Single Responsibility Principle.

Rule	Modularization must be used to achieve a *Separation of Concerns* and adhere to the Single Responsibility Principle. Modularization is considered a key metric for the measurement of code quality.

Pretty Printer

It is generally considered best practice to use the ABAP workbench “Pretty Printer” function to provide a consistent formatting style for ABAP code. The reason consistent formatting is essential for good maintainable code is because the version management of ABAP code does not ignore formatting when comparing versions. Hence by ensuring a consistent formatting style, tools such as version management can provide better clarity on change.

Pretty Printer can be configured several ways, and the most common approach is to set “Indent” to true and set the Convert Uppercase/Lowercase option to “Uppercase Keyword”. The principle behind this is that when code is printed the language keywords standout from the rest of the code. This is not relevant for modern development, and so Syensqo will standardize on the following settings:

Rule	Indent: True Convert Uppercase/Lowercase: Lowercase

Indentation is essential as it is a core style element in code readability. Lowercase for all source is chosen as this aligns with the style guidelines of other languages in use at Syensqo, namely Java and JavaScript. Modern editors, including the ABAP Workbench and the ABAP Development Tools will colour keywords making the use of capitals for distinction irrelevant.

Comments

Although the goal of all developers should be to write code that is easy to understand, comments should be used to highlight areas of code that need further clarification. As described in **Code for Readability**, comments are not to be used as a replacement for well written, intention revealing code.

ABAP supports two types of comments and both are allowed within the Syensqo Development Standards. That said, the following rules should be followed for the use of these comment types.

Line Comments (those starting with a *****) should be used be used in the first instance. These should be placed on the line above the target of the comment, allowing the reader of the code to read the comment before the source code. The comment should be indented to match the line of code it is referencing.

Inline or End of Line Comments (those starting with a **"**) should only be used in the following situations:

- After a type or variable declaration.
- At the end of a code block. Note that although this is allowed it is discouraged as it usually indicates that the section of code has become too complex (as it requires a comment to explain it) and should be broken into smaller parts – see Separation of Concerns.
- If a Pseudo Comment needs further clarification.

The practice of including a standard header in each ABAP artefact is **not** to be followed at Syensqo. The standard header contains information that is already available from the attributes of the relevant artefact and the version control system. Additionally, the standard header will be likely to be out of date almost immediately due to code changes not being reflected in the header.

Rule	Delete dead or unused code that has been commented before migration to test environments.
Rule	Use comments only when further context is helpful. Always maintain code for readability first.

Working with Existing Code and Continuous Improvement

When working with any existing code at Syensqo, the style and structure of the code should be maintained as consistently as possible. Mixing classic and modern ABAP styles significantly impacts the readability of the code.

If the existing code can be easily refactored to a more modern style, then this should be done as part of the change. In this case the scope of the refactoring must cover a single modularization unit, such as a method.

The ability of a developer to do this while implementing an enhancement or fixing a bug will depend on the complexity and size of the particular module, the time available, and the existence of unit tests to confirm no regression bugs have been introduced.

As part of working with existing code, deprecated functions should be removed when identified. Continually making improvements of this type will minimize the remediation work during an upgrade or support package.

Rule	Always leave the code base in a better state than when you found it.

Unit and Performance Testing

Unit Test

Every development, be it a one line enhancement or a complex custom solution, needs to be unit tested after build completion and before handing it over for the Functional Acceptance Test.

Each functional specification has basic test scenarios and cases maintained. These are the baseline for unit test. The unit test results need to be documented showing clearly the passing of all tests and steps involved to achieve this.

The following guidelines should be applied for unit tests.

1. Unit tests should focus on behavior, not individual methods or functions.
2. Unit tests should never directly test private or protected methods; the aim of unit testing is to confirm behavior from the point of view of the consumer.
3. Unit tests must run quickly, be independent (not rely on test execution order or pre-determined state) and be reliable.
4. Only test one behavior per test.
5. Unit testing should confirm that objects are at the right level of abstraction and are easy to work with.

For more complex custom solutions the use of code-driven unit tests is mandatory. Developers should deliver both Unit and Component tests to demonstrate the correctness of the delivered software as part of the quality process. Additional guidelines apply for these.

1. If unit test are hard to write, this usually indicates an issue with the design or implementation of the application.
2. Unit tests are written in code.

ABAP Unit is the ABAP implementation of the standard xUnit pattern for building technical facing unit tests. More details of how to use can be found at SAP Learning.

Rule	Unit Tests are mandatory for ALL developments.

Performance Test

Performance tests are an essential part of delivering good software. Without good performance the solution will not be accepted by the the users. The following development scenarios require a mandatory performance test in a test system with proper data.

- AMDPs
- Complex select queries and logic, eg. loops on big internal tables
- Selects on big tables (tables with more than 100k records and big selection result)

Without a performance test it is not possible to deliver these solutions with the assurance that they will run properly in the production system after go-live. Tcode ST12 is to be used for ABAP developments. For CDS only developments ST01 with DB trace active is to be used. In addition to that the Basis team can also monitor the runtime memory consumption to check if it is within acceptable limits.

Performance test results need to be documented explaining the test case, data volume used and showing the trace results with an evaluation.

Rule	For data or processing intense developments it is mandatory to compete a performance test.

Exceptions

A common approach to exception handling significantly assists with the clarity of the code. Class based exceptions have many advantages over classic exceptions, providing a change in program flow rather than a simple return code. For this reason only class based exceptions are to be used at Syensqo.

When using class-based exceptions write the TRY/CATCH block first before writing any code. The normal program flow should then be written inside the TRY block.

Exceptions should be defined with respect to the caller as this is where the exception will be caught. The exception must provide enough context to allow the caller to handle the failure.

As discussed in **Robust Software**, class-based exceptions should be caught at an appropriate level in the code. If an exception is raised, allow it to propagate up through the call stack until a suitable handler is found.

When the code in the catch block exceeds more than a few lines it should be extracted into a separate error handling method to ensure the intent of the called method is clear.

Rule	Only use class-based exceptions.

Constants

General Approach

The general approach to hardcoding is to avoid hardcoding values directly into your ABAP code. Instead define constants that have the correct data type and a descriptive name.

Constants should be defined as part of the class or interface attribute definition to allow reuse in several methods. In case a constant is only used in a specific method it is allowed to define the constant at the beginning of it.

Rule	Do not hardcode in ABAP code.

Enterprise Structure

Enterprise structure, like company code, sales organization and plant will be maintained in custom table /SYQ/T_CONSTANTS and accessible via a utility class /SYQ/CL_CONSTANT_UTILITY. The table allows for grouping like country or region to allow for quick selects of required constants to be used in selects or application logic.

Rule	Always use Enterprise Structure constants from the ES constants table instead of creating separate constants in different places.

Utility Classes

Several utility classes are available to ease use and standardize certain reoccurring custom logic. These utility classes need to be used when implementing the custom logic requires it. A list of available utility classes is available here.

Rule	Always use use available utility classes to avoid unnecessary code duplication and slightly different implementation approaches.

ABAP Managed Database Procedures

ABAP Managed Database Procedures (AMDP) allow the execution of stored procedures from the ABAP runtime.

AMDP classes are used to perform data intensive logic with complex calculations. The AMDP methods can be exposed as CDS Table functions.

Rule	Only use AMDP classes for complex calculations with high data volume.

Configuration Table Maintenance

Custom transportable configuration tables are not to be maintained via classic generated maintenance views (SM30). Instead they need to be integrated with the **Custom Business Configuration App**.

Via Eclipse it is possible to generate the required RAP object and v4 oData service for a custom table. The generated object is of type "Business Configuration Maintenance Object". A good Tutorial is available. Once created, the new maintenance object for the table is available in the Custom Business Configuration App. **The generated table maintenance objects are to be created in a package named after the table in super package /SYQ /MAINTENANCE_OBJECTS.**

The Custom Business Configuration View needs to be maintained in this tracker, for the security team to add them to the roles.

In the exceptional case that a configuration table's content is deemed as not transportable for business to use it is needed to generate a regular managed RAP object and v4 oData service that is used to generate a Fiori Elements app in BAS. This app will be made available to the business users.

Rule	No custom classic table maintenance views are allowed.
Rule	Authorization checks are mandatory for any configuration table maintenance.

Enhancing the SAP Standard

SAP provides several options to change the standard solution. These generally fall into the categories of **enhancement**, where SAP standard is adjusted through non-invasive extension points, and **modification**, where the SAP supplied source code is modified directly or SAP artefacts are cloned and modified as required.

Syensqo supports the use of the **SAP Enhancement Framework** to make changes to the standard solution assuming all necessary approvals are obtained. This includes the use of BAdIs, User Exits and Enhancement Points. It is mandatory to always check for Tier 1, released Badis, or Tier 2, unreleased Badis, enhancement options first before looking into Tier 3.

When implementing enhancements it is preferable to add just enough code into the original program to collect any required information and then delegate to a global class where the additional functionality has been implemented. Delegation may be as simple as a calling a method on a class, or depending on the requirement, a custom BAdI implementation may be used to take advantage of features such as Multiple Use and Filters, and if required, the BAdI can be added to the switch framework..

If this approach is followed, the enhancement can be quality checked in the same way as all other custom development.

Implicit enhancements are always the last option and are counted as modifications due to their impact to upgrades and overall system stability.

Standard Modification process

1. Exact object(s) that require modification or implicit enhancement are identified and listed.
2. Design Authority must explicitly approve the modification through custom development request process.
3. Critical Developer Role is updated with the requested SAP standard objects (exact name will be mentioned in S_DEVELOP object).
4. Access will be provided to the assigned developer to complete the development.

Rule	If a BADI definition is **not** enabled for multiple implementations, then it is required to implement all Enhancements in the same implementation class that is owned by the initial Enhancement. All included additional Developments need to be encapsulated as a static class method, each assigned to the corresponding package. The static class method needs to be wrapped in a condition to ensure that it is getting executed at the right time. A comment needs to mention the additional Developments for easier maintenance.
Rule	If a VOFM routine needs to be implemented, then it is required to create **one** implicit enhancement. No modification allowed. Encapsulation into a static class method is mandatory
Rule	If a user exit needs to be implemented, then it is required to create **one** implicit enhancement. No modification allowed. Encapsulation into a static class methods is mandatory.
Rule	Enhancement must be chosen over Modification.
Rule	Other Implicit Enhancements are counted as Modifications and to be avoided.

ABAP Test Cockpit

The ABAP Test Cockpit (ATC) is the integration of the SAP Code Inspector into the ABAP development environment. The ATC allows for the continuous checking of developments against an agreed check variant without the overhead of requiring the developer to create Inspections and Object Sets as required by the SAP Code Inspector (SCI). The ATC allows the developer to choose from a list of available global check variants defined by the Code Inspector, or to run against the agreed default check variant.

Syensqo will provide a global check variant which all developments must run successfully agains. This process will be established as part of a quality gate for migration to test and consolidation systems.

SAP has provided a best-practice recommendation for the use of the ABAP Test Cockpit as a quality checking tool and this will be implemented at Syensqo. The approach recommended by SAP is as follows:

- Development System Checks.
    - ATC checks are run in an Ad-Hoc manned by developers on code they are currently working on
    - Scheduled ATC checks are run across the full code base in the development system
- Objects on transports are automatically checked on release (quality gates)
- QA System Checks
    - Checks are periodically scheduled on the code in the QA system

The ABAP test cockpit will be used to automatically inspect 100% of all custom code at Syensqo. The result of each inspection that is run is available to the development team via the ATC results browser in either the ABAP development workbench or the ABAP Development Tools. This provides a worklist for the development team and continuous feedback about the quality of its codebase. All warnings from the ATC are to be treated seriously.

Failing tests would stop the release of the transport. A review of the errors is needed and exceptions to be approved by the development architects.

Rule	Quality and correctness of all ABAP developments must be continuously demonstrated through the use of ABAP specific tools such as the ABAP Test Cockpit.
Rule	Every time an object was changed, it is needed to run the ATC check variant /SYQ/DEFAULT and the result is to be attached in the CD.

Tier 2 wrapper

There are still many unreleased objects without contract in S/4HANA. In order to use them it is necessary to build a tier 2 wrapper with contract C1 that enables the use in ABAP Cloud (tier 1) objects.

Examples are:

- Selects on tables or unreleased standard CDS views in custom CDS view entities: a wrapper CDS to be built
- All Function modules: a custom class to be built that wraps the FM call
- Unreleased standard classes and methods

Rule	Tier 2 wrappers are to be created as classic ABAP but released with a C1 contract. That way the wrappers can be consumed in an ABAP Cloud object.

SAP RESTful Application programming model

SAP Gateway Service Builder is not to be used to create oData services. Instead, the use of RAP generated oData services and Web APIs is mandatory.

Refer also to the RAP keyword documentation on SAP Help.

General Rules:

1. Use Eclipse to build RAP Business Objects
2. RAP business objects to be used when CRUD actions are needed in a Fiori app development
3. SAP delivers many standard RAP business objects already that need to be used instead of standard BAPIs if available.

Rule	The Gateway Service Builder (SEGW) must not be used to create oData services. RAP and ADT Service Definition is to be used instead.

Custom Business Objects App

The Custom Business Object app allows creating managed RAP business objects and should be used for simple and fully custom business logic based on one or two custom tables. The oData service is to be consumed as a Fiori Elements app.

Core Data Services

Core Data Services (CDS) is a modelling tool for defining semantically rich data models using a database independent syntax. The CDS model is then “pushed down” into the target database allowing it to be consumed. ABAP has a specific implementation of the general CDS concept allowing for the definition and consumption of CDS views from ABAP.

ABAP CDS view entities are implemented from the ADT using a Data Definition Language (DDL) artefact. CDS entities utilize a new authorization concept using a data control language (DCL), and this should be used to implement specific entity level authorizations.

If the custom CDS view entity is based on released standard CDS view entities it is required to use the **Custom CDS View app.**

It is mandatory to create a CDS view entity for every custom table to be used in ABAP selects or in custom composite CDS views.

Further rules specific to Analytical CDS are detailed in the SAP Analytics and Reporting Standards.

Rule	CDS view entities need be used instead of direct table selects
Rule	**CDS view entities need be used instead of direct table selects to follow the ABAP Cloud model.**
Rule	**Every custom table requires a base CDS view entity to be created as well to be used in any selects.**

Guide	Use of Custom CDS View app is preferred and needs to be checked first

If the Custom CDS View app cannot be used then ADT has provided various sample CDS templates, which should be used for more efficient development:

CDS VDM types: Rules & Guidelines

CDS can be categorized as Basic, Interface and Consumption CDS.

Rule	Consumption CDS must be exposed for UI Modelling.

Guideline	Basic/Private CDS	Interface/Composite CDS	Consumption/Projection CDS

Behaviour	• Expose all relevant Business Data • Form the Low-redundancy model on the top of database tables • Directly access database table and other basic views	• Can access Basic views • They should be reusable • can have associations to other basic views • They introduce data redundancy	• Can access data from various composite as views • Can Directly expose as OData Service using annotations @OData. Publish = True • can have associations to various consumption or Interfaces views
Important Annotations	@VDM.Viewtype: #BASIC	@VDM.Viewtype: #COMPOSITE	@VDM.Viewtype:#CONSUMPTION
Data Source	Basic view/Tables/views	Basic views/Interface views	Composite views

CDS Selection List: Rules & Guideline

Guide	Use of relevant Arithmetic/Aggregate function at CDS level is recommended as it helps to push down the code to DB layer

Guideline	**Rules & Guidelines**
Column List	Column list are comma separated and alias names are given with prefix "_" underscore
Case Expression	CDS supports the CASE expression. An **Alias name is required for the resulting column except for nested CASE.** The resulting column type is derived from expressions after THEN and ELSE clause.
Built in SQL Functions	String expressions, including string concatenation operator &&, or built in functions should be used at **CDS layer is recommended**
Arithmetic Functions	Aggregate/Arithmetic operations is recommended at CDS level
Cast expressions	Convert the value of the operand to the dictionary type specified by dtype(any data element & predefined data type in ABAP dictionary) •Different operand types supported: Literal, column, path expression, build-in function, arithmetic expression •Various data types in ABAP namespace supported •Result length is determined at activation time
Virtual Elements	Virtual elements allow to implement logic in an ABAP class to retrieve the value of the element. Can be used to reduce the need for AMDPs.

CDS Performance Specific Guidelines

Guideline	**Description**
Replacing for ALL ENTRIES by CDS views and view-on-view concept.	Create CDS and join tables, which could be joined into one or multiple CDS views. If FOR ALL ENTIRES is used then FDA must be used.
Use of UNION/UNION ALL to merge data from 2 or more select statements is recommended instead of the use of JOIN.	Union is performed directly within HANA and SELECT statements can be processed in a parallel way. Keep in mind select list must have the same number of output fields and compatible types.
Use of JOIN only to combine data from multiple tables or views when all related data is used, or when a specific join logic is to be enforced.	Joins are executed immediately when the CDS view is accessed, meaning all join conditions are evaluated regardless of whether all fields are used in the query. Associations define a conceptual relationship between entities, acting as a "join on demand".

Use of expressions and functions: • Concatenate (similar to CONCATENATE on APPL Layer in ABAP) • COALESCE (check on 2 variables: If first variable has null value, then take second variable) • Simple/ Complex Case (similar to CASE on APPL Layer in ABAP) • Substring (get subset of variable/ string)	Expressions and functions should be used to push down data intense logic form the APPL Layer to DB layer.
Use of aggregation functions (SUM, MIN, MAX, COUNT)	Aggregate functions should push down data intense logic from the Application Layer to DB layer
In case HANA DB specific features are used, as e.g. input parameters, a check should build in to check if the requested feature is supported	Use of method CL_ABAP_DBFEATURES=>USE_FEATURES. Also implement proper error handling.
Use of CDS view within ABAP	In general, the CDS view should be used and not the Database table if available.
When using view-on-view concept, use association to simplify the definition and use of relations between entities	Associations simplify the definition of relations between entities compared to an explicit join definition: • Associations are defined within CDS views • Association can be exposed via projection list • Simplification of path navigation by path expressions. While consuming the association in the CDS using path expressions, a join is constituted in HANA.
Improve Security of CDS Views using **ABAP CDS Access Control**	AccessControl.authorizationCheck is mandatory CDS annotation
Use the **Dependency Analyzer** to Evaluate relationships and complexity of CDS Entity with regards to its SQL Definition	Dependency Analyzer helps you to understand SQL dependencies and complexity that might negatively affect the performance of your query.
Extend SAP CDS Views Instead of Creating New CDS views	Extends an existing CDS view cds_entity using a CDS view extension cds_view_extension in CDS source code, example : EXTEND VIEW cds_entity WITH cds_view_extension [association1 association2 ...] { select_list_extension } [;]
Improve Readability in Your CDS Views	Open your CDS entity and choose **Source Code > Format** from the context menu.
Check for CDS performance improvements	• **DISTINCT** should be at the top most level to achieve code push down. • Similarly, **GROUP BY**, which should be avoided in CDS as per above. Use of Lengthy **DISTINCT/GROUP BY** should be certainly avoided – limit it to the key fields of the respective tables. Should be tested for negative effect on Performance through ST05 /Execution Plan.
Secondary index usage in S/4 Hana only if beneficial	SAP Note 1794297 -Secondary Indexes for S/4HANA and the business suite on HANA • For some use case secondary indexes can still be beneficial. This is especially true for highly selective queries on non-primary key fields. These queries can be significantly improved by indexes on single fields which are most selective.

Extension CDS View

A standard CDS view can be extended to add new fields, data source and associations.

General Rules:

1. Extension CDS view name should start with “/SYQ/E”, point to the standard CDS name, should reflect the serial number
2. Association Name should start with an “_” and should reflect type or name of associated data source.
3. Custom Field Name should start with “Z” & and can contain technical field name or reflect the meaning of the field.
4. Field Alias should reflect the meaning of the field from end user’s perspective.

Guide	Extend standard views instead of copying them with reference.

CDS Annotations

An annotation enriches a definition in the ABAP CDS with metadata. It can be specified for specific scopes of a CDS object, namely specific places in a piece of CDS source code.

- Latest Fiori Element based apps are completely CDS annotations driven **with minimum UI Coding.**
- Annotations at oData service can be added at Metadata level and Reference Annotations
- Analytical apps like KPI, OVP can be easily built using CDS Annotations.

Rule	Use CDS annotations to drive UI build.

Enterprise APIs for Integration Scenarios

There are 3 types of APIs possible to be used in CPI integration scenarios.

1. Rest/oData APIs
2. ABAP Proxies (SPROXY)
3. Idocs

Rest/oData APIs are modern RAP based APIs that follow the open API standard. Implementation follows the Restful ABAP Programming section.

ABAP proxies are classic APIs and consist of classes generated to support the consumption and provision of Enterprise Services (via the ESR or Metadata Repository). For standards related to Interface design, please see the SAP Integration Approach.

Once a proxy is generated, the names of the generated repository objects (classes, interfaces, dictionary objects) as well as the field attributes of any structures must be adjusted to support the naming standards defined in this document. The names used for the generated proxy must clearly reflect the intention of the enterprise service it was generated from.

Idocs are only to be used if no modification or enhancement is needed and if SAP supports them long term.

Use of AIF is mandatory for client and server proxy implementations.

Client proxies, those used for the consumption of an enterprise service, only need to follow the defined naming practices.

Server proxies require implementation and must observe the following rules.

1. Business logic must not be included in the proxy implementation. The proxy must delegate to the appropriate business logic. A proxy implementation is an adapter (as described in **Software Architecture**) between the Enterprise Service domain (Infrastructure) and the Application domain.
2. Technical failures should raise detailed exceptions to ensure visibility and transparency of the failure.
3. An explicit COMMIT must not be used in the proxy implementation. This is handled by the framework.
4. Proxy runtime should be minimized. If the business logic requires excessive runtime then appropriate technical level checks should be made, and if successful, the business logic executed via an asynchronous process (bgRFC, application job). This will technically complete the proxy execution and avoid blocked queues and timeouts.

Rule	Classic ABAP proxies are only to be used when the use of RESTful APIs is not a viable option.
Rule	AIF is Syensqo's centralized monitoring framework for APIs, be it RESTful APIs, ABAP Proxies or IDocs.

AIF Rules and Naming

AIF is Syensqo's centralized monitoring framework for APIs, be it RESTful APIs, ABAP Proxies or IDocs. Every standard and custom interface needs to be configured, be it inbound or outbound.

Custom AIF interface Naming		
Object	**Convention**	**Example**
AIF Namespace	**Z**<Meaningful Label or Description>	ZEDOC
AIF Interface	**Z**<Namespace>_<Descriptive abbreviation>	ZEDOC_INV
Other objects	/SYQ/<Prefix>_<Description>	/SYQ/S_RAW

Event Driven Architecture

SAP Enterprise Event Enablement allows integration between S/4HANA and SAP Event Mesh, Integration Suite. It supports an Event Driven Architecture that allows a scalable and flexible approach with a loosely decoupled pattern. This is particularly useful for requirements where the integration scenarios does not require large payloads or a two-way integration. Within S/4HANA system, messages can either be published (Outbound) or subscribed (Inbound) to Event Mesh.

In scenarios where Event Driven Architecture is suitable, below are the different use cases on Event Enablement implementation in S/4HANA in the order of priority:

Metadata Extensions

Metadata Extensions allows new annotation to existing data definition with minimal impact. In the context of event enablement, the common annotation @Event.context.attribute can be used to add a specific property to header which is useful for filter purposes.

Use Metadata Extensions when there is an appropriate Standard Business Object event metadata extension where annotating an existing property in the payload is enough for the requirement.

Requirement: the Behaviour Definition (BDEF) must be released for customer use and extensible in cloud development. In addition, the abstract entity assigned as event parameter must be extensible.

Derived Business Events

Derived Business Events is a mechanism to send a custom event whenever a standard business event is raised (ie. Trigger custom PO create event when standard PO create event is raised) with a redefined payload depending on the business requirement. This is useful in adding new fields to the payload as needed.

Use Derived Business Events when the standard business events requires additional properties in the payload and/or annotation on existing payload and it does not fit the metadata extension scenario.

Requirement: the Behaviour Definition (BDEF) must be released for customer use and available for use in cloud development and the definition is extensible (contains extensible annotation)

Business Event Consumption to trigger a custom RAP Business Object

In line with Business Event Consumption, the same principle can be used to trigger a custom RAP Business Object. This allows a lot of flexibility in the processing logic and payload.

Use Local consumption when the RAP Business Object is not extensible but there is an existing released RAP Business Object event or it requires additional processing logic that is not possible via Derived Events.

Requirement: The Standard Behaviour Definition (BDEF) must be released for customer use and available for use in cloud development.

Custom RESTful Application Programming (RAP) Business Object

For requirements that are fully custom including the processes and underlying tables, a custom RAP Business Object can be created with Business Events.

Print Forms

The development of print forms at Syensqo will be based solely on the Adobe Document Services framework. Since the release of S/4HANA 1511, a new output management technique has been adopted by SAP utilizing BRF+ instead of the previous “NAST” condition-based technique. The target architecture in S/4HANA is now focused only on ADS and Adobe Forms. SAPScript and SmartForms are to be considered deprecated. With Clean Core the forms need to be built as Adobe with Fragments forms as default. Only if there is no output management support for these can a classic Adobe form be built instead or SAPScript/Smartforms as absolute exception.

SAP will continuously deliver Adobe Forms to replace its legacy SAPScript and SmartForms. During this transition period these legacy forms may be configured to be used through Output Management and minor modifications made (such as the addition of an Syensqo logo). Major changes to existing legacy forms are not permitted; instead confirm with SAP when the Adobe Form replacement will be delivered and if this is unsuitable a custom Adobe Form can be considered.

All print forms can be translated via the Translation Editor SE63.

If a form template is used in several countries having the same layout and content, then the translation in SE63 is required.

If a form template has major layout and/or content changes due to country specific versions, then a new form needs to be created to streamline the form configuration and maintenance.

Form Build Rules:

1. Logo:
    a. As default, use the black and white Logo in white background (42mm logo) if not specified otherwise. Logo should not be directly attached. We have the utility class /SYQ/CL_DYNAMIC_COMPANY_LOGO that needs to be used for all forms as default. No logos should be loaded/embedded into a form.
    b. The MIME repository to be used is SAP/PUBLIC/Syensqo (tcode SO2_MIME_REPOSITORY)
2. Font: 8 - 11 points Arial Regular to be used unless specified for different sections of the form.

3. Page Margins: 5 mm at least on all sides so that the whole layout gets printed.
4. Table Header: Should have at least 1 space before the text and after or be center aligned if anything not specified for the build – should never look like it is on the Table border (line).
5. Address: Should have the same format/fonts for all the address in the same form like for Company address, Sold-to address, Customer address, unless specified in the build.
6. Date: Left Aligned as default, unless specified in the build. Use ISO 8601 standard of pattern YYYY-MM-DD if not specified otherwise.
7. Texts: Left Aligned as default, unless specified in the build.
8. Numbers: Should be ideally right aligned like for amount and weight.

Suggestions as below for better and clean output:

1. Form Header: Should stand out from the rest of the texts in the form for e.g. bold, bigger font size.
2. Free Texts: Remove extra space while printing texts.
3. Labels: If one header label has ':', then ideally all header labels should have that.
4. Last page: Should check for blank page as the last page unless required for build. Try to ensure blank page is not getting printed at the end unless required.

Rule	New print forms must prioritize the use of Adobe with Fragments built in the LiveCycle Designer

Workflows

Custom Workflows need to be built using the Classic Business Workflow or Flexible Workflow or BTP BPA. That complies with SAP's clean core approach. The business objects and custom logic have to be done in ABAP OO and RAP.

Transaction code SWDD_SCENARIO instead of SWDD is used for Flexible Workflows. Using Flexible Workflows allows for flow configurations via the app "Manage Workflow", but has many limitations. For the agent determination the new "Responsibility Management" app is used.

Guide	S/4HANA Custom Workflows to be built as Classic Workflows where no standard flexible WF exists.
Rule	Cross-system Custom Workflows to be built in BTP BPA.

ABAP Authorization Group

Custom ABAP code and tables will have a custom authorization group assigned to them to provide the mechanism to restrict access to them by users.

For these custom developments the table below will be used depending on the relevant functional module. The object ***BRGRU*** has a maximum of four characters to create a unique authorization group with. The object(s) will be created using transaction SE54.

To have a consistent naming convention for custom authorization groups, use the entries in the table for tables and programs.

MODULE	CODE	ABAP	TABLE
Financial Accounting	FI	ZFI	ZFIT
Controlling	CO	ZCO	ZCOT
Investment Management	IM	ZIM	ZIMT
Treasury	TR	ZTR	ZTRT
Enterprise Control	EC	ZEC	ZECT
Material Management	MM	ZMM	ZMMT
Sales & Distribution	SD	ZSD	ZSDT
Production, Planning & Control	PP	ZPP	ZPPT
Product Data Management	PDM	ZPDM	ZPXT
Quality Management	QM	ZQM	ZQMT
Plant Maintenance	PM	ZPM	ZPMT
Service Management	SM	ZSM	ZSMT
Project Systems	PS	ZPS	ZPST
Personnel Development	PD	ZPD	ZPDT
Payroll Accounting	PA	ZPA	ZPAT
Organizational Management	OM	ZOM	ZOMT

Time Management	TM	ZTM	ZTMT

Rule	SAP supplied authorization group &NC& will **not** be used at Syensqo

ABAP Authorization Check

Authorization checks are needed to limit access to data or to restrict who can create/update/delete data.

The ABAP keyword AUTHORITY-CHECK is deprecated in ABAP Cloud. Instead, the authorization checks need to be added to CDS view entities using DCLs (Data Control Language) to restrict access or inside the RAP Behavior handler class.

DCL authorization

RAP authorization

Rule	Custom code must always be executed in the context of an authorized user, i.e. after an authorization check has been performed by the system. For code directly invoked by a user, an explicit authorization check must be performed before business or data access logic is executed. Relying purely on Start authorization is not enough to meet these criteria. Custom code executed within an already-authorized context (e.g. most BAdIs, user exits, enhancement spot implementations, etc), is exempt from this requirement because the check would have been performed by the standard SAP code.

SAP Screen Personas

SAP Screen Personas can be used to turn existing ERP Webgui screens into a simplified application for desktop and mobile. Native Ui5 apps are always preferred, but some transaction codes (T-codes) are still required to be used and can benefit from SAP Screen Personas especially to enable use on mobile devices.

Possible use cases are:

- Simplification of important and still used Tcodes (https://developers.sap.com/mission.screen-personas.html)
- Mobile device enablement of important and still used Tcodes (https://developers.sap.com/tutorials/personas-mobile-workbook-optimize-mobile.html)

Rule	Screen Personas are not to be used, except for mobile device adjustments of T-codes or for simple GUI adjustments like field removal.

Developer authorization audit

Developer authorization is subject to a bi-yearly review as per external auditors and internal controls. **Report needed.**

Github Repository Guideline

Syensqo uses Github (www.github.com/SCo-SyWay) to host its source code git repositories.

To access the Syensqo Github account users will first need to be granted access by an administrator of the Github account. Two-factor authentication on the Github account is mandatory to be enabled to provide an additional layer of security for the organization.

Once a user has been granted access, they will then need to create a Github Personal Access Token. This can be found under User > Settings > Developer Settings > Personal Access Tokens (https://github.com/settings/tokens). This Personal Access token needs to be saved and used in lieu of a password when accessing a git repository from SAP Build Code.

The source code for all SAPUI5 applications needs to be managed in Github and a new repository should be created for each application. For other source code this may not be needed.

Git commits should follow best practice. This means commits should be small and done often; committed code should be complete and tested; and commit messages should be meaningful and explain what was changed and why.

Repository Creation Workflow & Responsibilities

- **Repository Creation :** A Tech Lead/Architect or designated project member identifies the need for a new repository. The respective Tech Lead /Architect is responsible for creating the repository.
- **Naming:** The Tech Lead/Architect ensures the repository adheres strictly to the naming convention.
    - For SAPUI5 apps the naming is [DEV_ID][MEANINGFUL_DESCRIPTION]
- **Visibility:** ALL repositories MUST be created as **private by default**. Public repositories are not permitted without explicit, high-level approval.
- **Initial Setup:** Upon creation, the Tech Lead must:
    - Add a comprehensive README.md file.
    - Configure a .gitignore file suitable for the project's technology stack.
- **Team Assignment**: Access control will be managed SOLELY through GitHub Teams. No direct user assignments to repositories.

Team Structure and Access Control

Rule : Repository access is granted exclusively by assigning users to designated GitHub Teams. This ensures consistent permissions and simplifies management.

Defined Teams & Responsibilities:

- SAP BTP Dev Team: Responsible for repositories related to SAP Business Technology Platform development.
- SAP Integration Team: Responsible for repositories focused on SAP integration scenarios (middleware, APIs, etc.).
- SAP UI/ABAP Team: Responsible for SAP UI-specific development (Fiori Freestyle).
- SAP ABAP Team: Responsible for SAP UI-specific development (Fiori Elements).
- ICertis Team: Responsible for repositories related to the ICertis platform.

Version Management in Github

Every change to an existing app needs to be done via branch created in Github. A new repository will be initialized and contain the main branch. But the actual development will be done in a feature branch. The feature branch will be merged with the main branch after SIT sign off which is used to import into the production systems.

UAT defect fixes will need a new branch (maintenance branch) and follow the same process as the feature branch.

The app will still be assigned to a transport and moved to QA and Prod via it. Github is only for version management, collaboration and governance review.

SAPUI5 and Fiori Elements Programming Rules and Guidelines

UI and UX Definition

User Interfaces (UI)

This term relates primarily to the screen technologies. The intended SAP footprint spans a number of SAP user interface solutions, including cloud solutions acquired by SAP (eg. Ariba, SuccessFactors etc.), solutions developed by SAP (eg. Fiori applications), traditional screens (eg. classic SAPGUI transactions) yet to be updated by SAP, as well as non-SAP frameworks (eg. React). This section seeks to provide a coherent approach to accessing the various technologies on offer in as simple a manner as possible. This includes access across various channels (eg. desktop / tablets / mobiles).

User Experience (UX)

The concept of user experience (UX) is a broad domain and concerns itself more with how users feel about a system after using it. As such it is more difficult to measure and less technically focused than the area of user interfaces. One definition of UX can be succinctly summarised as …

A person's perceptions and responses that result from the use or anticipated use of a product, system or service - Ergonomics of human system interaction, ISO 9241-210

Concepts such as visual design, interaction design, theming, accessibility and performance can all contribute to the end UX. When we describe the 'look and feel' of an application, whilst the user interface may deliver the 'look', the user experience can denote the 'feel' of that application. From an SAP standpoint given the charter for 'Standardization' we place heavy reliance on SAP's ability to delivery UX in its technology suite. Therefore from a strategy perspective we align with SAP's UX direction which is based on the 'Fiori' concept.

User Experience Principles

The User Experience principles represent the guiding principles when making all decisions related to the User Experience.

Principles	Description

Consistent	• Feels like one solution with a single and holistic experience • Syensqo branding and corporate identity is enforced • Keeping SAP standard theme across all systems whenever possible
Simple	• Intuitive to use, minimizing training • Users are not bombarded with more functionality than is useful to them
Desirable	• An exciting prospect, not something to be dreaded • The system is desirable to use (i.e. easy to use, self-explanatory etc.)
Productive	• Supports users achieving as much as possible in the minimum time, especially those “experts” users • Responsive and fast to use
Futureproof	• Uses only strategic technology that provides the best user experience and is supported for the foreseeable future (i.e. Fiori) • Ongoing cost and maintainability are considered (i.e. use Standard vs Custom)

SAP UX Guidelines for Fiori

Custom UI5 Fiori apps need to follow the SAP UX guidelines. Its principles were described in section 2.2.

The SAP UX guidelines for Fiori can be found here:

https://experience.sap.com/fiori-design-web/

https://sapui5.hana.ondemand.com/

The standard app library for Fiori apps is also a good source of information.

Development Tools

SAP offers **SAP Build Code** as the new entry point to app development and the **SAP Business Application Studio (BAS)** for the development and extension of SAPUI5 applications. Although there are other tool choices available, like VS Code, the SAP Build Code and BAS are to be used for all SAPUI5 developments at Syensqo.

Rule	SAP Build Code and the SAP Business Application Studio is to be used for all SAPUI5 development.

Naming Conventions

This guide is to be considered the Syensqo standard for SAPUI5 naming conventions and is to be followed for all SAPUI5 development. By adopting these conventions, code developed at Syensqo will be of a standard accepted by SAP.

The JavaScript standards defined in the **Java and JavaScript Programming Rules and Guidelines** section are to be considered in the context of working with the SAPUI5 library, SAPUI5 templates are regularly updated and we recommend developers to familiarize themselves with above document as per SAPUI5 release version.

SAPUI5 application namespaces are used to guarantee code written by Syensqo does not clash with SAP delivered code or 3rd party frameworks. The namespace scheme used at Syensqo will start with the prefix com.syensqo as per SAP best practice. Following this prefix, a combination of the Process Reference Model hierarchy will be used to uniquely assign the SAPUI5 component to a relevant functional process.

Once the root namespace of the SAPUI5 component is determined, it should be used as a prefix for all other namespaces used in the repository.

For example, the following namespace is used for a new SAPUI5 component in the learning and development process type in HR.

com.syensqo.hr.learninganddevelopment .Component – Component Configuration

com.syensqo.hr.learninganddevelopment .controller.createcoursematerial – Course Material Controller

The namespace postfix will vary based on the application design. The application structure should always follow the current guidelines from SAP.

SAPUI5 ABAP Repository

Syensqo will host a majority of its SAPUI5 applications in the embedded Fiori server in S/4HANA, with the exception being extensions to the SAP Cloud Portfolio. When SAPUI5 applications are hosted on an ABAP system the rules below need to be followed to ensure the integrity of the application.

- Do not modify any SAPUI5 applications directly in the ABAP editor. This can lead to source code inconsistencies
- The SAPUI5 ABAP Upload/Download tool should not be used in the development process. Only the SAP Build Code and BAS IDE can be used for working with SAPUI5 applications using the ABAP Repository.

Note that although SAPUI5 applications are stored as BSP applications they do not support the BSP runtime hence BSP concepts such as Flow Logic are not supported in SAPUI5 development.

Rules for Fiori applications

The ABAP application server offers CDS, transactional application frameworks and traditional ABAP coding that expose user interface consumption services through the OData protocol, which significantly simplifies template-based SAP Fiori applications.

Backend Modelling (CDS Development)	**OData Service Modelling**	**UI Development**
CDS with annotations (oData v2)	CDS Exposed directly as OData	• KPI Modeler • OVP • APF Modeler
CDS with Annotations (oData v2/v4)	CDS consumed as Reference data source	List Report / Object Page (Read only)
CDS with Annotations with RAP for Draft persistence (oData v4)	CDS Consumed as reference data source	List Report / Object Page (includes CUD Operation)
CDS, RAP, BAPI, Classes	• CDS Map to Data source • RAP / BAPI / RFC for CUD operation	Free style development

Guide	Use of CDS annotation-based application is to be used over conventional Free style UI Application

Guide	Fiori UI5 App is preferred for any enhancements of UI.

Guide	UI Adaptation and clean core key user extensibility are preferred.

Guide	Neptune’s DX platform to be used for mobile app clients.

Unit and Performance Testing

UI unit testing follows the ABAP guideline for all S/4HANA backend developments that are used in the app, i.e. oData services.

The app itself needs to be tested to ensure it is working as per the requirement detailed in the FS. Use of the UI5 Test Recorder to simulate interactions and help automate UI validation is recommended.

Thorough test of the UI in several browsers and devices the app is intended for (e.g., Chrome, Edge, Safari, mobile/tablet) is mandatory.

In addition to that it is needed to test and measure the following if relevant:

- BTP backend logic and performance, i.e. CAP services.
- UI5 and JavaScript compliance using the UI5 Inspector browser plugin and ESLint.
- For non UI5 based apps, i.e. ReactJS, compliance using ESLint.
- Round trip calls to the backend to be optimized and at a minimum.
- No backend call or app internal processing should take more than 5 seconds.

Rule	Thoroughly test any custom app for all required screen sizes.
Rule	Any deviation is to be documented and justified.

Localisation

To support translatable texts, all viewable labels and literals must use the UI5 localization concept and provide a resource bundle. Resource bundles are stored in the **i18n** folder of the web app. The default bundle is called i18n.properties and must be provided for all SAPUI5 applications. Additional language bundles can then be created on an as needed basis.

Accessibility

All SAPUI5 applications must at a minimum not interfere with any accessibility features delivered by the framework. For more information about accessibility see the Accessibility section of the SAPUI5 documentation.

Fiori Elements

Fiori Elements allow SAPUI5 applications built directly from annotations provided by the consumed OData services. As each template is provided by SAP, applications built using Fiori Elements Templates will benefit from enhancements delivered by SAP in future releases of the SAPUI5 library. Additionally, applications built from templates will meet Fiori design guidelines and follow established floorplan patterns.

Application built with Fiori Elements require no or minimal code to be written as the application is based on views and controllers provided by SAP. For this reason, Fiori Elements should be considered before creating a custom SAPUI5 application.

Rule	Custom SAPUI5 apps to be built with Fiori Elements whenever possible.

UI Extensibility

SAP introduced several new UI extensibility options in S/4HANA that are to be used as of 1809 (See appendix for full extensibility presentation slides).

Customer enhancement of SAP delivered applications is managed by an extension process that does not modify the standards application directly. This means that any changes made by the customer will not be lost when the original application is upgraded.

Rule	Only the adaptation project concept can be used to change SAP delivered SAPUI5 applications. Applications must not be modified directly.
Rule	Fiori Key user extensibility is preferred and to be used if supported by the app.

JavaScript Programming Rules and Guidelines

The JavaScript standards defined in this document are to be considered in the context of working with the SAPUI5 library or Cloud Application Programming (CAP).

Development Tools

SAP offers the SAP Build Code and SAP Business Application Studio (BAS) for the development and extension of JavaScript. Although there are other tool choices available, eg. Visual Studio Code, the SAP tools are to be used for all JavaScript developments at Syensqo.

Rule	The SAP tools are to be used for all JavaScript development.

Naming Conventions

SAP has provided the Development Conventions and Guidelines document for working with JavaScript. This guide is to be considered the Syensqo standard for naming conventions and is to be followed for all SAPUI5 development. By adopting these conventions, code developed at Syensqo will be of a standard accepted by SAP for contribution to the SAPUI5 library.

The following high level rules should be noted

- Use clear and descriptive names for files, directories, functions and variables that reflect their content or purpose.
- Classes and Controls are to be started with a capital letter and then follow camel case.
- Functions and Variables are to start with a lower case letter then follow camel case.
- Pre-fixing of variables is encouraged due to the weak typing in JavaScript. A list of acceptable prefixes is provided in the SAP Development Conventions and Guidelines document.
- Constants are to be all UPPER CASE
- Use “strict mode” during development to identify any potential coding issues.

Github Repository

Syensqo uses Github (www.github.com/SCo-SyWay) to host its SAPUI5 and CAP git repositories.

To access the Syensqo Github account users will first need to be granted access by an administrator of the Github account. Two-factor authentication on the Github account is mandatory to be enabled to provide an additional layer of security for the organization.

Once a user has been granted access, they will then need to create a Github Personal Access Token. This can be found under User > Settings > Developer Settings > Personal Access Tokens (https://github.com/settings/tokens). This Personal Access token needs to be saved and used in lieu of a password when accessing a git repository from SAP BAS.

The source code for all SAPUI5 applications needs to be managed in Github and a new repository should be created for each application. When creating new git repositories, the repository should be named with the same name as the BSP application name and Development Id plus more information added as description. This convention makes it easier to identify the git repository for any SAPUI5 application.

Git commits should follow best practice. This means commits should be small and done often; committed code should be complete and tested; and commit messages should be meaningful and explain what was changed and why.

Version Management in Github

Every change to an existing app needs to be done via branch created in Github. A new repository will be initialized and contain the main branch. But the actual development will be done in a feature branch. The feature branch will be merged with the main branch after SIT sign off which is used to import into the production systems.

UAT defect fixes will need a new branch (maintenance branch) and follow the same process as the feature branch.

The app will still be assigned to a transport and moved to QA and Prod via it. Github is only for version management, collaboration and governance review.

JavaScript File Organization

JavaScript file organization involves structuring your project's '.js' files and directories to promote maintainability, scalability, and collaboration. It is mandatory to adhere to the following approaches and considerations:

- Group related files within directories based on the features or functionalities they implement (e.g., components/, utilities/, services/, pages/).
- Distinguish between UI-related logic, data handling, and business logic, placing them in different files or directories.

SAP BTP Development Rules and Guidelines

This section outlines the architectural standards and tooling strategy for building side-by-side extensions on the SAP Business Technology Platform (BTP).

Primary objective is to achieve **"Clean Core"** extensibility while maximizing developer productivity through the use of SAP **Build Code, Generative AI (Joule), and the Cloud Application Programming Model (CAP)**.

Development Environment for SAP BTP Full Stack Application Development : SAP Build Code & Joule

We will utilize SAP Build Code for cloud-native development that combines the Business Application Studio (BAS) with the power of Generative AI.

Integrated Development Environment (IDE)

- Tool: SAP Business Application Studio (Build Code Edition).
- Guideline: All development occurs in this cloud-based environment to ensure consistent runtime versions and tool extensions across the team.
- Standard: Use "Dev Spaces" specifically configured for Full Stack Cloud Application.

AI-Assisted Development (Joule Copilot)

We will leverage Joule to accelerate the coding lifecycle. Joule is not just a chatbot; it is embedded into the coding workflow.

- Use Case A: Data Modeling: Developers will use natural language to describe entities (e.g., *"Create a SalesOrder entity with fields for amount, customer, and date"*). Joule generates the CDS syntax automatically.
- Use Case B: Business Logic: Instead of writing boilerplate JavaScript/Java, developers will prompt Joule (e.g., *"Write a handler to validate that the SalesOrder amount does not exceed 10,000"*).
- Use Case C: Unit Testing: Joule will be generating initial unit test blocks for CAP services.

Rule	Use SAP Build Code (Business Application Studio - Build Code Edition) with configured Dev Spaces and embedded Joule AI to ensure consistent cloud-native tooling and accelerate code, model, and test generation.

Development Framework: Cloud Application Programming (CAP)

The SAP Cloud Application Programming Model (CAP) is a mandatory framework for building backend services. It is "opinionated," meaning it enforces best practices by default.

Core Data Services (CDS)

Standard: All data models and service definitions must be defined in .cds files.Best Practice: Keep the data model (Database layer) separate from the service definition (API layer) to allow for future reuse.

Service Layer (Node.js, Python)

Selection: (Choose one for the client, usually Node.js for speed.Productivity: CAP automatically handles OData V4 provisioning, multi-tenancy implementation, and localized error messages. Developers focus only on domain logic.

Event-Driven Architecture

Standard: Use CAP's built-in eventing to subscribe to S/4HANA events (e.g., BusinessPartner Created).Tool: SAP Event Mesh. CAP handles the connection details; developers simply write event handlers: srv.on('BusinessPartner/Created', ...).

Rule	Use SAP Build Code (Business Application Studio - Build Code Edition) with configured Dev Spaces and embedded Joule AI to ensure consistent cloud-native tooling and accelerate code, model, and test generation.

Connectivity & Integration: SAP Cloud SDK

To interact with the S/4HANA core or other external systems, we will strictly use the SAP Cloud SDK.

Virtualized Connectivity

- Standard: Never hardcode URLs or credentials in the code.

Implementation: All external calls must go through the Destination Service via the SAP Cloud SDK execute() method.

- This ensures security and allows switching between Test and Production systems without changing code.

Resilience

- Best Practice: The Cloud SDK includes built-in resilience patterns (Circuit Breakers, Retry Logic, Bulkhead). These must be enabled for all S/4HANA calls to prevent cascading failures.

Rule	Route all external calls through the SAP Cloud SDK and Destination Service, and enable built-in resilience (retries, circuit breakers, bulkheads) for secure, environment-independent connectivity.

Development Lifecycle & DevOps Best Practices

Version Control (Git)

- Standard: A "Feature Branch" workflow.
- Structure:

- main: Production-ready code.
- develop: Integration branch.
- feature/BusinessRequirement: Individual developer workspace.

Application Security (XSUAA)

- Standard: Security is defined in **xs-security.json**.
- Use **Role Collections to group scopes**. Do not assign individual scopes to users.
- CAP Integration: Protect services using the @requires annotation (e.g., @requires: 'Admin') directly in the CDS file.

Continuous Integration/Delivery (CI/CD)

- Tool: SAP Cloud Transport Management Service (cTMS) or CI/CD Service.Pipeline: Code committed to main must automatically trigger a build, run unit tests (generated by Joule), and prepare the MTA (Multi-Target Application) archive for deployment.
- Pipeline: Code committed to main must automatically trigger a build, run unit tests (generated by Joule), and prepare the MTA (Multi-Target Application) archive for deployment.

Rule	Adopt a feature-branch Git workflow, define security in xs-security.json and CAP @requires, and enforce CI/CD that runs Joule-generated tests and packages MTA artifacts on main commits.

SAP BTP Space Naming Conventions

A BTP Space is the logical boundary for resources, access control, and runtime isolation in SAP BTP — choose and document the correct Space before creating Dev Spaces to ensure proper entitlements, quota management, and team-level isolation.

Dev Space Naming Follow the same organizational hierarchy used for ABAP packages but append a "/" suffix to indicate environment-specific Spaces. Place all custom Spaces under the /syq parent to keep them discoverable and consistent.

/syq/BusinessFunction

Suggested Dev Space names (one per business function)

- /syq/rnd/ — Research and Development Space
- /syq/mkt/ — Marketing, Sales and Service Space
- /syq/proc — Procurement Space
- /syq/mfg — Manufacturing Space
- /syq/qm — Quality ManagementSpace
- /syq/scm — Supply Chain Space
- /syq/pm — Plant Maintenance Space
- /syq/ehss — Safety and Sustainability Space
- /syq/fin — Finance Space
- /syq/eppm — EPPM Space
- /syq/hr — HR Space
- /syq/gts — Global Trade Services Space
- /syq/tech — Technical developments (cross-functional) Space
- /syq/utilities — Re-usable utilities Space
- /syq/migration — Data Migration Dev Space

Naming notes / best practices

- Use lowercase, short functional codes (as above) for readability and consistency.
- Ensure Space-level naming and permissions map to teams responsible for each function to simplify quota, entitlement, and security management.

CAP Naming Convention (BTP) — /syq/ Standard

Namespace: Use /syq/ as the root namespace for all CAP models/services to mirror ABAP discoverability and ownership.

Object Type	Pattern	Example
CDS model file (domain/data)	db/<area>/<name>.cds	db/fin/finance-core.cds
CDS service file	srv/<area>/<name>-service.cds	srv/fin/finance-service.cds
CSV data load file	db/data/<namespace>-<entity>.csv	db/data/FinancialDocument.csv
Entity (persistent table)	/<Area><BusinessObject>	/FinancialDocument
Entity (master data)	/<Area><Object>	/BusinessPartner
Entity (transactional)	/<Area><Object> + child compositions	/SalesOrder, /SalesOrderItem
Association/Composition name	lowerCamelCase descriptive	items, customer, toPlant
Type (CDS type)	/types/<Name> (or type <Name>)	/types/CurrencyCode
Aspect (reusable fields)	/aspects/<Name>	/aspects/AuditFields
Service definition	/<Area><Domain>Service	/FinanceService
Service entity (projection)	<BusinessObject> or <BusinessObject>View	SalesOrders, SalesOrderItems
Action	action <Verb><Object>()	action ApproveSalesOrder()
Function	function Get<Functionalty>() returns	function GetSalesOrderStatus()
Bound action/function (on entity)	action <Verb>() on entity	entity SalesOrder { action Approve(); }
Event (CAP)	event <PastTenseVerb><Object>	event SalesOrderApproved
Annotation file (optional)	app/<appname>/annotations.cds	app/sales/annotations.cds
Node.js service implementation	srv/<service-name>.js	srv/finance-service.js
UI app name (Fiori/UI5)	<area>-<app> (kebab-case)	fin-sales-approval
MTAR/MTA id	syq.<area>.<app>	syq.fin.salesapproval
Destination name (BTP)	SYQ_<APPID>_<PURPOSE__	SYQ_<<APPID>>_SupplierApp

Rule	Naming of CAP artifacts should be environment-agnostic (dev/qa/prod handled by BTP Spaces/Destinations), not embedded in CDS names.

SAP BTP CAPM Coding Standard and Guidance

Security & Secrets

1. **No secrets in code or repository** - No credentials, API keys, or sensitive data in source code, comments, or .**env files**
2. **XSUAA authentication implemented** - Mandatory security service configured for all environments
3. **Input validation and secure connections** - All user inputs validated, secure API calls without hardcoded credentials
4. **Latest stable dependencies** - Up-to-date CDS/Node.js versions with secure, fixed dependency versions in package-lock.json

Rule	Never store secrets in code or repos — keep credentials and API keys out of source files.

Code Quality & Standards

1. **Naming conventions and clean code** - Consistent naming, no dead code, proper comments for non-standard functions
2. **ESLint configuration** - .eslintrc.json file included with no unused variables or imports
3. **Externalized configuration** - All config parameters externalized, proper .gitignore with excluded credentials
4. **API versioning** - All provided APIs are properly versioned

Rule	Enforce XSUAA authentication in all environments — require platform security service.Validate inputs and use secure connections — avoid hardcoded credentials in API call. Follow consistent naming, remove dead code, and enforce linting with an .eslintrc.json.

CAP-Specific Best Practices

1. **Managed aspects for entities** - Use framework-managed fields for DB logging (avoid manual modifiedBy assignments)
2. **Efficient CQL queries** - No "SELECT *" unless fetching 80%+ fields, no queries inside loops
3. **Multiple service architecture** - Services created based on roles/requirements, not single service for all functionality
4. **Proper associations** - No unmanaged associations unless justified with documented reasons
5. **Remote services handling** - External services handled via extend entity in separate CDS files

Rule	Let the CAP framework manage entity fields and use efficient, well-structured services and queries

Data & Performance

1. **No binary data in key fields** - Binary data not used in primary key fields
2. **Optimized data handling** - Large binary data via Object Store / DMS (not HANA Cloud), with malware scanning enabled
3. **Auto-scaling configured** - Load balancing setup where required for performance

Rule	Naming of CAP artifacts should be environment-agnostic (dev/qa/prod handled by BTP Spaces/Destinations), not embedded in CDS names.

Localization & UI

1. **Annotations for reusability** - Entity annotations created at bottom level for reusability
2. **Proper localization** - Text tables handled by locale framework, i18n used for frontend labels
3. **No assert.unique on primary keys** - Primary key constraints handled properly without assert.unique

Rule	Naming of CAP artifacts should be environment-agnostic (dev/qa/prod handled by BTP Spaces/Destinations), not embedded in CDS names.

SAP Build Process Automation (SBPA) Rules and Guidelines

The purpose of this document is to establish a standardized approach for designing, developing, and deploying workflows using **SAP Build Process Automation** on the SAP Business Technology Platform (BTP). Adherence to these guidelines ensures:

- **Maintainability:** Consistent naming and structure allow for easier handover and debugging.
- **Scalability:** Architectural decisions that support high-volume processing.
- **Governance:** Clear separation of concerns between Pro-Code and No-Code artifacts.

General Architecture & Deployment

Project Structure

All development should occur within the **SAP Build Process Automation Lobby** for standard implementation, or **SAP Business Application Studio (BAS)** for complex, pro-code extensions.

- **No-Code (Lobby):** Use for standard approval flows, simple form-based triggers, and drag-and-drop automation.
- **Pro-Code (BAS):** Use for complex data transformations, custom UI integration, or scenarios requiring deep git-based version control integration.

Deployment

- **Dev/Test/Prod:** All artifacts must be developed in the Development sub account and transported via **SAP Cloud Transport Management (cTMS)** Direct modification in Production is strictly prohibited.

SAP Build Process Automation(SBPA) - Naming Conventions

To ensure clarity across the project lifecycle, the following naming standards must be applied to all artifacts. Names should be in **PascalCase** (Capitalize Each Word) with no special characters or underscores unless specified.

Artifact Naming Standards

Artifact Type	Convention Pattern	Example
Project	[LOB] [Process Name]	Finance InvoiceApproval
Process	[ProcessName]Process	OrderFulfillmentProcess
Form	[Function]Form	VendorOnboardingForm
Decision	[Logic]Rule	DiscountEligibilityRule
Data Type	[Entity]DT	EmployeeDetailsDT
Action Project	[System]Actions	S4HANAOrderActions

Variable Naming

- Use **camelCase** for internal variables and input/output parameters (e.g., invoiceAmount, approverEmail).
- Ensure context names in the workflow context match the API payload keys where possible to reduce mapping effort.

SAP Build Process Automation : Development Best Practices

Process Design

- **Modularity:** Do not build monolithic processes. Break complex workflows into sub-processes.
- **State Management:** Always define a "Happy Path" and an "Error Path." Use **Parallel Gateways** carefully; ensure all branches converge correctly to avoid orphaned tokens.
- **User Assignment:** avoid assigning tasks to specific email addresses. Always assign tasks to **BTP Role Collection** or dynamic variables determined by logic

Interactive Forms

- **Simplicity:** Keep forms user-friendly. Break long forms into sections or multiple steps.
- **Validation:** Use built-in constraints (Mandatory, RegEx patterns) within the form designer to validate data *before* the process moves to the next step.

Decision Management (Business Rules)

- **Externalize Logic:** Do not hardcode logic inside Process Condition branches. Use **Decision Tables**.
- **Reusability:** Design Decision Tables so they can be reused across different processes (e.g., a central "Approval Limits" decision table).

SAP Build Process Automation : Integration & Action Projects

Integration with external systems (S/4HANA, Ariba, Third-party) is handled via **Actions**.

Action Project Configuration

When creating Action Projects to consume APIs:

- **File Support:** Ensure API specifications are uploaded in valid **JSON** or **XML** format (OpenAPI specification)
- **Uniqueness:** Action names must be unique within the specific Action Project.

API Selection Criteria

Select the integration method based on the requirement:

1. **SAP Graph:** Preferred for navigating complex SAP landscapes with a unified data model.
2. **SAP CAP (Cloud Application Programming Model):** Use when custom business logic or data persistence is needed before the workflow step.
3. **Standard OData APIs:** For direct CRUD operations on S/4HANA or SuccessFactors.

Destination Configuration

Destinations in the BTP Sub-account must be configured correctly for SBPA to discover them.

Property	Value / Standard
Name	[SystemID]_HTTP (e.g., S4H_HTTP)
Type	HTTP
Proxy Type	Internet or OnPremise (via Cloud Connector)
Authentication	PrincipalPropagation (Preferred) or OAuth2ClientCredentials
Additional Properties	sap.processautomation.enabled = true

Guide	Principal Propagation is critical for audit trails, ensuring the backend system knows *who* triggered the action.

Workflow Triggering Mechanisms

Select the appropriate trigger mechanism based on the business use case.

API Triggering (Technical Start)

- **Use Case:** High-volume scenarios or when the process is initiated by an external system (e.g., a CAP application or a legacy system).
- **Implementation:** The workflow is exposed as a REST API instance.
- **Requirement:** Requires a dedicated service key and proper OAuth token handling.

Event-Based Triggering (Asynchronous)

- **Use Case:** Loosely coupled architecture. Example: A purchase order is created in S/4HANA, generating an event that automatically starts the workflow.
- **Implementation:** Requires **SAP Event Mesh**. The SBPA project subscribes to the specific topic (e.g., sap/s4/beh/purchaseorder/created).

Rule	Select API-triggered workflows for high-volume/external starts and event-triggered workflows (via Event Mesh topics) for loosely coupled, asynchronous business events.

Monitoring, Visibility & Error Handling

Process Visibility

- **Dashboards:** Every critical business process must have a corresponding **Process Visibility Scenario** enabled.
- **KPIs:** Define at least three KPIs per project (e.g., "Cycle Time," "Open Tasks," "Error Rate").
- **Attributes:** Mark relevant context data (e.g., Order ID, Customer Name) as "Visible in Monitor" to allow support teams to search for specific instances.

Error Handling & Troubleshooting

- **Technical Errors:** If an API call fails, the workflow should route to a "Technical Support" user task or an error-logging automation, rather than silently failing.
- **Support Portal:** For platform-level issues (e.g., API gateway failures), incidents should be raised via the SAP Support Portal with logs from the "Monitor > Automations" view.

Rule	On failures, route workflows to a Technical Support task or error-logging..

Mobile Neptune App Programming Rules and Guidelines

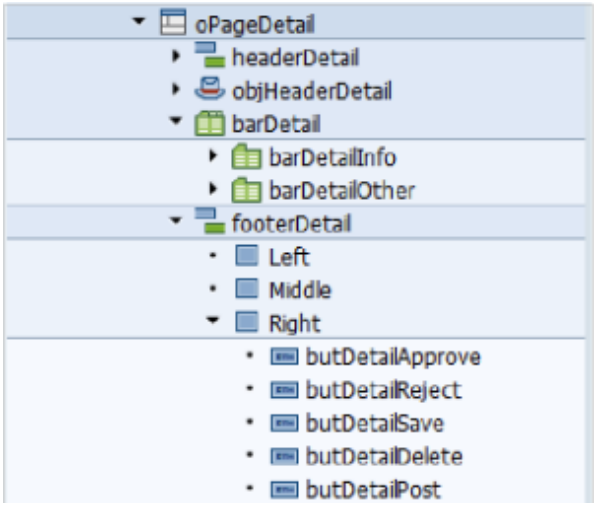
For Fiori the SAPUi5 Standard must be followed and for ABAP and JavaScript development, the Syensqo Standards must be followed. However, this section revisits the SAP Fiori Guidelines from a Neptune access perspective.

- Neptune Launchpad which is the home page must be the single point of entry and central point of all navigation paths. The tiles on the home page of the Neptune Launchpad represent navigation anchors to the individual apps. By selecting a tile, the user navigates to the corresponding app. It is also possible to integrate legacy UI technology through such tiles.
- Neptune Launchpad is configured in the Neptune Cockpit hosted on one S/4HANA server, it can host multiple Launchpads and mobile clients.
- It is Neptune and SAP Fiori best practice to design all UI5 apps stateless.
- Neptune is utilizing SAP UI5 as the default HTML5 GUI framework for Neptune apps. Even though it is technically possible to use any HTML GUI framework, you should use Neptune applications on the UI5 framework.
- Minor design adjustments can easily be added by applying custom CSS classes directly in Neptune Application Designer. Custom icons can be included using the sap.m.Image control.
- It is recommended to define the Title/Group attribute in the app settings as this is the wording that will be displayed on the browser tab.
- Every app should start with the sap.m.Shell control, typically named 'oShell'.

- Beneath sap.m.Shell, always place the sap.m.App or sap.m.SplitApp control as a subnode. The App/SplitApp control is the de-facto root control of the app that will be used to control all navigation and busy-state of the app. For example, you can name it 'oApp'. You can have multiple App /SplitApp controls under one Shell.
- As subnodes to the App/SplitApp control, it is recommended to define a set of sap.m.Page controls that define the needed screens of the app. Hierarchically all pages should be placed at the same level.
- The page control that is placed as the first subnode of the App/SplitApp control will be the starting page which is rendered/displayed at app start.

Neptune Application Naming

- Every object of an app must have a unique name (exception: neptune.BarContent).
- Just as SAP Fiori, the 'camelCase' notation applies - starting lowercase and using uppercase letters for all leading characters of the composite name.
- It is UI5 best practice to start every object name with a lowercase 'o' (for object). Neptune only uses the 'o' notation on Pages, App/SplitApp and Shell controls.
- The maximum name length for an UI5 control in Neptune is 23 characters.
- Name the sap.m.Shell control 'oShell'
- Name the sap.m.App/SplitApp control 'oApp'
- Following the above guidelines for the object names should be similar to this example:



Neptune Application User Experience

- SAP Fiori Design Guidelines must be followed when designing UI5 apps with Neptune.
- The app should be based on sap.m.Page as the basic design container.
- Every page should include a header section.
- Every page should include a footer section.
- All process specific design, e.g. forms, tables, tabs, etc., should be placed between header and footer of a page.
- Basic process triggers like buttons to save, cancel, submit should be placed in the right section of the footer.
- Back buttons should be placed in the left section of the page header or footer.

SplitApp environment:

- The Master Page should be the leading function, providing e.g.
- a list of (searchable) items, or an input form to define/search for datasets, or a list of sub-functions, etc.
- The Detail Page should display process/data details and functional triggers, e.g.
- Infos to a specific dataset selected on the Master Page, or Sub-items of a dataset, or Input fields to create a new dataset, etc.
- Both sections should have a footer displaying buttons to trigger further actions.

Buttons:

- Buttons should follow a basic color concept, which can be set via property 'type':
- blue (type: Emphasised): Core function trigger of a page, e.g. Search, Submit. In general used to emphasise a certain button object in contrast to un-styled buttons on the same footer/toolbar
- green (type: Accept): Positive function trigger used mostly in combination with a negative trigger, e.g. Save/(Delete).
- red (type: Reject): Negative function trigger used mostly in combination with a positive trigger, e.g. Decline/(Approve).
- none (type: Default/none): Used for standard functions. Button blends in with footer/toolbar, e.g. Settings.
- The button function should either be displayed applying text OR SAP-UI5 icon. All available UI5 icons are listed in the Neptune Icon Explorer.

Unit and Performance Testing

Unit Testing follows the guidelines of the ABAP and UI sections.

In addition, it is mandatory to test the Neptune apps in a desktop browser and a mobile device to ensure compatibility.

ABAP Application Class

All Neptune Applications have an associated backend ABAP Class. The application class is where the logic of the application is implemented. In terms of the Model View Controller design pattern the application class hence plays the role of ‘model’.

- Follow Syensqo's ABAP Development guidelines.
- The app-corresponding backend class must implement the /NEPTUNE/IF_NAD_SERVER interface.
- Using UI5, the only applicable handling method is HANDLE_ON_AJAX.
- In HANDLE_ON_AJAX, implement a case statement based on input parameter AJAX_ID to redirect to the corresponding private methods of the class.
- All attributes that are used for binding in Neptune Application Designer need to be declared public.
- Attributes for binding can be flat structures or internal tables.
- Attributes can be based on local structures or data dictionary.
- Define small/lean private methods.

Backend Calls

In order to successfully design a backend call, you need to define the following parts:

- Bind the backend attributes to the corresponding UI5 controls.
- Define the Ajax ID for the backend call at one of the bound controls.
- In online scenarios, the main receiving UI5 control should define the Ajax ID.
- In offline scenarios, use sap.m.Ajax control to define the Ajax ID for backend calls.
- If you want to send/receive more than one data model, use the 'Additional Model Send/Receive' option next to the Ajax ID.
- Implement corresponding backend code in the ABAP class based on the incoming Ajax ID.
- Based on the definition of the Ajax ID (e.g. at UI5 table control 'tabMyTable'), Neptune will automatically create an anonymous function that can be used to trigger the backend call via Javascript function call.
- This function always uses the following naming convention:

    'getOnline' + nameOfUI5ObjectWhereAjaxIDisDefined().

Event Handling Success / Error events

- **AjaxSuccess:**

  When the server answers with an expected result, the AjaxSuccess event at the UI5 control with the Ajax ID is triggered automatically. We recommend making use of this event in order to handle further process control after fetching data from the backend. This will also prevent wrongful process control on the client-side if the server does not answer or the connection is broken during a call. Typical examples of needed client-side code in the AjaxSuccess event are disabling the busy state, navigation, and further data processing/updates. Below you find a common AjaxSuccess coding example:

- **AjaxError:**

  Implementing the AjaxError event (always implement both!) will prevent wrongful process control on the client-side if the server does not answer or the connection is broken during a call. For example, if you trigger a server-side search call, you do not want to show an empty result list if the server is down, the ABAP logic dumps, or connection is lost. Correct process-handling would be to let the user know that the server did not respond.

Client-side Scripting using JavaScript

- All custom JS functions create a ScriptCode control named 'Functions'.
- More ScriptCode can be added to the controls based on the need for specific Javascript functions grows.
- For all code that should run initially when starting the app, create a ScriptCode control named 'Init'.

Neptune Launchpad Events - OnInit / OnLoad

- There are two events provided by sap.n (which is the Launchpad-specific namespace) which are executed when launching an App in the Neptune Launchpad:
- sap.n.Shell.attachInit triggered the first time an app is launched. It is the equivalent to sap.ui.getCore().attachInit.
- sap.n.Shell.attachBeforeDisplay triggered every time an app is displayed (launch and switching views).

Mobile Client Build

Mobile Clients can be built in 2 ways:

- Neptune Mobile Build Service (MBS) (Preferred option)
- Manual build using the Cordova CLI

Rule	Use the Mobile Build Service. Only in case of issues should the Cordova CLI be used.

Using MBS

1. Logon to the Neptune Portal

1. Go to the Mobile Build Service (MBS)
2. Choose account
3. Choose Neptune app project file
4. Set the parameters for Android or iOS as needed
5. Generate and download the app

Using the Cordova CLI

- Download the cordova project from Neptune dev system and unzip.
- Make sure that the the correct SDK tool version is installed (for Androis build to be set in Android Studio)
- Run in terminal in project root folder:
    - cordova platform add android
- Copy build.json to root folder.
- Copy keystore file to root folder (if applicable)
- Change App name in config.xml to [sysid] -
- Run in project root folder:
    - cordova build [android/iOS] --release --buildConfig=build.json OR
    - cordova build [android/iOS] --debug --scan (for debug version)
- Change file name to ([sysid] - [Application Name] [version].apk)

Publishing custom Plugins to npm

To publish custom plugin to npm, follow below steps:

1. Plugin can be published via VS Code or Node JS command prompt.
2. Open Command prompt and login to npm
    - Command - > npm login
3. After login use below command to publish the changes to npm
    - Command - > npm init --scope=@syensqo
4. After initializing publish to npm with below command
    - Command - > npm publish --access public
5. Plugin is now plublished to npm

6. It usually takes 24hrs for the plugin to be available in MBS tool. If it doesn’t appear then contact volt build team.

System refresh preparations

Any S/4HANA system refresh needs to be prepared for in case of Neptune.

Since the configuration of the Neptune environment will be overwritten it is necessary to export it BEFORE the system refresh and imported again after the refresh.