Coding Standard

# API Standard

|  |  |
| --- | --- |
| API Standard | Detail |
| Provide an Open API specification |  |
| Follow RESTful Best Practices | * Delete actions should be idempotent. * Avoid implementing chatty POST and GET operations (minimize API call round trips) * POST actions that create new resources should not have unrelated side-effects (Single responsibility) * Follow the HTTP specification when returning a response (returning the correct and meaningful HTTP Status Code to the client) * Capture exceptions and return a meaningful response to clients (do not have uncaught exception propagate to the framework even you should catch and * Handle exceptions consistently and log information about errors * Provide an Unauthenticated Route for Monitoring * Following Security Standards, Policies and Requirements (Azure API Management) * Use Managed Service Identities |

# Application Development Standard

|  |  |
| --- | --- |
| Application Development Standard | Detail |
| Design Principles | * SOLID * IOC (Dependency Injection) * DRY (Don’t Repeat Yourself) * Use RESTful API * Document the API properly * Anticipate your API will be consumed by others so make it “friendly”. |
| Security Concerns | * SOS (Statement of Sensitivity) * PIA (Privacy Impact Assessment) |
| Code comments | * Comment on methods that are less obvious. * Indicate JIRA ticket number and what is being fix when fixing bugs. * Do not leave old code behind (that is the reason why we use GIT) * Archive code 100% coverage by testing   Module Comments   * A module is any source code file, a standard module, a class, a component or user control, an interface definition, or a .NET Structure. The comment should include: * Purpose. A few lines about what the module does. * Author. Who wrote the module, and when. Also list major updates and who made them. * Requirements. What other modules or libraries the module requires to compile and run. * Usage. How you use the module. Is there a special way to instantiate objects of this class? Is there a required calling sequence? How should you dispose of the object? If there are certain key procedures in the module, mention them too. * Limitations. Are there any special limitations to the use of the module?   Procedure Comments   * A procedure includes Sub, Function, Property block, or Event declarations. The comment should include: * Purpose. One or two lines describing what the procedure does. * Parameters. Describe each parameter individually. What values do they accept? For each parameter, indicate if it's for input or output (in/out). If a parameter returns a value, describe it as you would describe a function return value. * Function return value. Describe what values will be returned, and in which range. For a boolean or enum return value, describe what the individual return values mean. Mention any special error values. * Side effects. Does the procedure update any variables? * Error conditions. If an error condition may arise, describe if the procedure will handle the error, or let the caller handle it. Does the function return an error code? Should the caller be prepared with an error handler? Should the caller read the Err object? * Pre-conditions. Are there any conditions that must be met before calling this procedure? Should you call the procedures in a certain order? Should you set some variables? * Algorithm. If the procedure implements a complex algorithm, it may be worth describing it too.   In-line Comments   * The intention of any code development is to make the code as self-documenting as possible. This is achieved through good naming conventions. In cases where the code become a little complex, in-line comments are used to assist in describing the context of the code being executed.   Comments When Fixing a Bug   * When addressing a bug, indicate in a comment in the code the JIRA ticket number and what is being fixed (in general terms). |
| Git branching and branch policies | * The following base branch names shall be used in Git (case sensitive): * main – the main production branch * develop – the main development branch * New projects should start with the develop branch only and create the main when ready for the initial release. * If using release branches, they must be in a folder called “release” (case sensitive) * Releases must be locked once released * When working on a new feature/bugfix create your working branch in either “feature/branch\_name” or “bugfix/branch\_name”.  “feature” and “bugfix” must be lowercase and singular * For emergency production fixes use a hot fix branch “hotfix/hranch\_name” (again lowercase and singular) from the main branch. * Once released the hotfix will need to be reverse integrated into any development branches that are being worked on. * Feature branch names should include the JIRA ticket number that describes the bug or feature. * If the JIRA story includes multiple features, use the ticket number and a description of the feature being worked on. * All merges to release/develop/main branches shall be done using Pull Requests (PRs). * Typically, a PR cannot be approved by the requestor when more than one developer is working on the same project. * When requesting a PR ensure to set your branch to delete after the PR is successful so that extra branches are not maintained on the server for nothing – your local will be retained and will get recreated when/if you do another check-in to that branch. * Sync frequently from the source branch (ideally daily) |
| Project Wikis | All projects should have a Wiki that contains pertinent project information.  At a minimum this includes:   * A project overview * Roles * Business rules * Dependencies * Application URLs for all environments * Database name(s) * KeyVault details * Known issues/troubleshooting tips * Special instructions on how to run the project locally (web.config changes, installed libraries, etc.) |
| KeyVault and Secrets | * All sensitive credentials or values shall be stored in KeyVault and only accessed at runtime, not from configuration.  This always keeps the credentials out of source control.  Developers only have access to the DEVOPS and Dev KeyVaults, the DBA’s have access to KeyVaults in the QA environment and above. * The only exception to this shall be in local configuration scenarios when connecting to a database on the developer’s VM via a trusted connection – no user credentials in code. |
| Things to avoid | The following shall not be acceptable in any code checked in for deployment (applies to new development – may not necessarily apply to legacy apps):   * Text literals in UI – everything must be in resource files and translated. * Hard coded values – use configuration, KeyVault, constants, enums, etc. * Security and Accounts in code or configuration anywhere. * Unparameterized SQL * SQL that produces “select \* from xyz.”, reasons for that are: * Unnecessary IO * Increased network traffic * More application memory * Dependency on Order of Columns on DataSet * Breaking views while adding new columns to a table when adding or removing columns. * Conflict in JOIN Query because of same column name * Copying data from one table to other using INSERT … INTO … SELECT … will copy incorrect data if the order of column is not the same between two tables. * Inline or dynamic SQL anywhere. * Dead / Unused Code * Back door Test, Debug and Training code should use Preprocessor Directives to remove for Production deployment |
| Things to do | * Global error handling * Server-side input validation * WET 4 – uses jQuery validator to validate client side. You need to validate everything on the server side as well. * Run accessibility checks on your application. There are many tools available to do this, some that you can use are: * W3C markup validation - https://validator.w3.org/ * CSE HTML Validator (installed on all developer VM’s) * NVDA free version is available for our developer VM’s (https://nvda.en.lo4d.com/windows) * Consult with the UI/UX resource when in doubt. |
| Develop Maintainable Code | This is broad statement but is derived from the notion of Maintainability Index that can be calculated within Visual Studio. There are three main areas that the developer can focus on to keep the code more maintainable. They are.   * Cyclomatic Complexity * Depth of Inheritance * Class Coupling |
| New Web Development | * .Net 6.0 wherever possible. * Use web application or API depending on the need. * Where possible, favor ajax Http requests to backend API’s over server-side post-backs for accessing data to keep the UX fluid and improve perceived performance * Use client-side templating where appropriate – favor Vue for new development where feasible. * Implement the WET look and feel along with it’s WCAG enhancements by one of 2 means: * CDTS Nuget package * TBS.Foundation |
| Database Development | * Use Entity Framework Core (database first) * Use LINQ to Entities for simple data access (CRUD) * Use stored procedures for complex data access/reports. * Stored procedures must return explicit columns (no select \* from …) * Map stored procedure result sets to Entities |
| TBS NuGet packages | Several Nuget packages exist to implement common functionality in all TBS custom apps:   * TBS KeyVault – common library for accessing KeyVault * TBS ContentServer – used by TAP based apps to connect to GCDocs for document repository * POM Framework – automated test framework (work in progress) * TBS.PowerBI.Embed – used for applications that have embedded Power BI reports * TBS.Common.Javascript * TBS.Foundation – MVC helper library incorporating WET |
| Visual Studio templates | Two NuGet packages are available:   * At least have 3 tiers: * UI * Business * Data access layers * Web API * Web Application |
| C# Naming Convention | * Use PascalCasing for all public member, type and namespace names. * Use camelCasing for parameter names. * Favor readability over brevity * Overall be consistent in your use of naming conventions. * Microsoft provides guidance in general for OO naming conventions, but not specifically for naming UI controls. Since UI controls are ultimately variables used in code, they should follow the same convention as any other variable - no hungarian notation prefix. |
| JavaScript Naming Convention | * Use camelCasing for all functions, methods, parameters and properties. * Use UPPERCASE for constants/global variables. * Use PascalCasing for classes. * Put all Javascript in external JS files. This will allow us to apply the proper Content Security Policies headers as required by IT Sec. * Overall be consistent in your use of naming conventions |
| Data Naming Convention | * The business and data layers will need to implement DTO (data transformation objects) to convert back and forth. * Stored procedures shall be named as follows: * Must start with “usp\_” * Should end with an action for what the procedure is doing – Select, Insert, Update, Delete, GetList etc * The middle must include a reference to the object being acted upon * Separate words with “\_” * User Defined Functions shall use the following prefixes: * User defined scalar function – “udf\_” * Table valued function – “tvf\_” |
| Global Error Handling | * The global error handler is used catch all errors and remove the need for duplicated error handling code throughout the .NET api. It's configured as middleware in the configure HTTP request pipeline section of the Program.cs file. |
| Exception Handling | Design your solution to handle exceptions and errors and prevent the application from crashing.   * Use try/catch/finally blocks to recover from errors or release resources. * Handle common conditions without throwing exceptions. * Use specific exceptions in a catch block. |
| Validation | Both client-side and server-side validation have their advantages and disadvantages. In general, it is recommended to use both client-side and server-side validation for maximum security and usability. Client-side validation can provide instant feedback to the user, while server-side validation can ensure that all data is validated correctly. By combining both approaches, you can create a robust validation system that ensures the integrity and security of your data.   * Client-Side Validations * Use Client-side validation to provide instant feedback to the user or to inform them of an error so they can immediately verify and correct it before they take the next step.      * Server-Side Validations * Use server-side validations for more advanced validations against business logic. * Use server-side validations when there is a increased security need. Since server-side validation occurs on the server, it is less vulnerable to tampering. |
| HTML checkers | During the authoring of this document, there are several validators and accessibility checkers available. These tools, when used, are designed to aid the development team to ensure the solution meets the standards that are being adhered to by TBS. These tools are currently identified as   * W3C markup validation - https://validator.w3.org/ * CSE HTML Validator (installed on all developer VM’s) * NVDA free version is available for our developer VM’s (https://nvda.en.lo4d.com/windows) |