Documentation

Coding rules

Software architecture

11 août 2023

Toenn Vaot
SIREN 833908502
EURL enregistrée RCS Evry
Capital social 1000€

1. Table des matières

2. Confidentiality purpose

This document is CONFIDENTIAL. No diffusion of this document is authorized without authorization as inside as outside of TOENN VAOT company. Only authorized people can access this document and should be aware of the confidentiality level of this document and should respect the usage and diffusion conditions.

3. Project definition

Denmark project is a ticketing solution for Denmark transportation supervisor company, Rejsekort. This solution is deployed on the quasi-totality of the country and lets customers to travel all around the country thanks to one card.

4. Introduction

This document describes the development rules to follow in the aim to work in team and let all developers understand the structure of solution, project and code.

5. Repository structure rules

IFS_CS_NG
- IFS_DEV
  - IFS_APP
    - **Definition:** Mobile application projects
  - IFS_DB
    - **Definition:** Database projects
  - IFS_SRC
    - **Definition:** Component projects
  - IFS_SRC_NDP
    - **Definition:** Non-deployed component projects (i.e Tooling)
  - IFS_WEB
    - **Definition:** Website projects (excluded Web API projects)
- IFS_GEN
  - **Definition:** Output folders of any projects
- IFS_INTEG
  - **Definition:** Integration projects

6. Solution naming rules

1. Solution name should be a trigram described the role and content inside
2. Solution name should be written in CamelCase (**i.e:** `Acm` and not `ACM`)

7. Project naming rules

1. Project name should begin with solution name at minimum
2. Project describe a sub-component of solution should be in this format `ToennVaot.<SolutionName>.<SubComponentName>`
3. Project name should be written in CameCase (**i.e:** `ToennVaot.Acm.Api`)
4. Project name can reflect ONLY the solution name (**i.e:** `ToennVaot.Acm`)

8. Project structure rules

a- MVC Website / API

1- Partials should be on folder `_Partials` (need code to put in place) or `Shared` (standard implementation)

2- Partials should be named starting with an underscore (**i.e:** `_Footer.cshtml`)

3- Editor templates should be on folder `Shared/EditorTemplates`.
4- All output models should be defined in the API project not in `<project>.Models` project. No system models should be exposed. Mapping between them should be assumed by a mapper class which could be manage by an auto-mapper (**i.e:** AutoMapper) Coding rules

a- General

1- All names should be written in CamelCase
2- All class should be described thanks to a comment
3- All public methods should be described thanks to a comment
NOTE: An EMPTY comment IS NOT a comment, it should FILLED with conscious and professionalism
4- Parameter name should be the most descriptive of its content but in case of generic treatment, it is better to adopt generic naming like *item, component, …* which let the method/function to be agnostic of functional name against its generic treatment!
5- Parameter type should be interfaced if it isn't a native type (string, int, long, short, …) in the aim to be mock and testable however we want.
6- Don't use specialize name when the output is generic. (**i.e:** `int HttpStatusCode` should `int StatusCode` or should be `HttpStatusCode Code`)

b- Magic but not too much

A special chapter to talk about "magic" numbers, string, date and so on. Developer has always the trend to minimum of code. But minimum of code shouldn't be done without minimum of understanding. And "magic" is the opposite of understanding – the principal goal of magic – so developer should replace "magic" like this:

- **Magic numbers:** enumeration if multiple values are possible for the same field, constants instead
- **Magic strings:** constants
- **Magic dates:** constants

And don't hesitate, abuse of comments to explain the business case of each values to be sure the knowledge around this "magic" be shared between current developers but also for next ones.

c- Classes

1- Avoid to declare class having the same name as .Net native class

d- Enumerations

1- Enumeration name should end with *Enum* (**i.e:** `OrderStatusEnum`)

e- Exception classes

1- Exception class name should end with *Exception* (**i.e:** `ApplicationNotExistsException`)

2- Extension classes

1- Extension class name should start with the type name extended (**i.e:** `ApplicationExtensions` extends `Application` object)
2- Extension class name should end with *Extensions* (**i.e:** `ApplicationExtensions`)
3- Extension of type should be done on Object classes not on native type (string, int, long)
Example:
    - `<string>.GetId()` is forbidden because it is not an extension but a business case that could be resolve in static helper class

- `<int>.GetDate()` is forbidden because it is not an extension but a personal choice to transform a datetime representation as integer to its datetime version
- `<DateTime>.To1545Date()` is authorized in this case because it is a special date representation applied in your own business case.

3- Configuration

1- Configuration should be extend per environment (**i.e:** `appsettings.Development.json`)

4- Styling

1- Styling in website should be written in **Error! Reference source not found.** format (.scss)
INFO: Learn **Error! Reference source not found.** on the website: https://sass-lang.com/guide
2- The color should be written in HEXADECIMAL format and in FULL version, with 6 characters (#FFFFFF and not #FFF)
3- The reference files should be named starting with an underscore (**i.e:** `_forms.scss`) and should be integrated in the main **Error! Reference source not found.** file (**i.e:** `site.scss`)
4- Style for partials in MVC website should be written in a distinct **Error! Reference source not found.** stylesheet to be used ONLY on pages integrating the partial.
5- Style for pages in MVC website should be written in a distinct **Error! Reference source not found.** stylesheet to be used ONLY on the page
NOTE: This **Error! Reference source not found.** stylesheet can include the partial **Error! Reference source not found.** stylesheets if it needs them (page using some partials), to limit the number of minimize
6- INLINE Style are forbidden BUT in case of animation/integration (**i.e:** `display:none` changed by JS)

9. Framework usage rules

a- Entity Framework

The usage of Entity Framework is authorized but with some logic constraints:

- Use in simple cases (SELECT, UPDATE, DELETE just on one table)
- In complex cases, the study of performance AND maintainability for **Error! Reference source not found.** should be done
- In complex cases, a comment should be written above to describe the role and the goal of the request

b- Error! Reference source not found.

1- The usage of **Error! Reference source not found.** should be done thanks to SCSS version.
2- The extension or override of any **Error! Reference source not found.** part should be done by inheritance of SCSS
3- The file `variables.scss` should be kept as provided in the original package

NOTE: If **Error! Reference source not found.** is used, all the styles outside **Error! Reference source not found.** should be based on. Colors, Sizing, and so one basically managed by the framework should be used as calculation base in style. The goal is to only have one reference to manage all the styles easily.

c- Error! Reference source not found.

1- The usage of **Error! Reference source not found.** should be thanks to SVG+HTML version in website

2- The usage of **Error! Reference source not found.** in styles should be done thanks to SCSS version
3- The extension or override of any Bootstrap part should be done by inheritance of SCSS and/or inheritance of JS part (for SVG part)

10. Manager or Store?

Manager is the orchestrator to manage data but it is not the responsible. It always delegates.
- To STORE if the data can be access from database
- To EXTERNAL INTERFACE (API) if the data can be provided by external interface (AX, FARM, …)

Store is the responsible of data access and data formatting from database.

a- Can I call multiple stores?

Yes, you can BUT only if stores are in the same solution. If not, you have to call manager of external store from manager of your solution or study if the store should be shared in transverse solution (TVS)

b- Where calling external interface

External interface SHOULD be called from manager and NOT FROM store. Manager is the chief of decision and should keep its role. External interface is considered as external store of the solution so according to the previous point, manager is responsible of the call.

c- Store NG vs/and Store Legacy

If you have to use store for new generation and store for legacy OR only store for legacy, you have to create a new project for legacy called `ToennVaot.<SolutionName>.Storage.Legacy`.
WHY? This is to eventually prepare the disappear of legacy part in the future.

11. API project rules

a- Global exception handler

A global exception handler should exist to avoid technical error be displayed to the final user and reveal some secrets on your API. So during development you can disable this handler to see all the details but in Production you should enable it to hide them.

b- Exceptions catch or not catch

API should implement a global exception handler. You shouldn't catch exceptions but the business exceptions. If your workflow uses some specific business exceptions and you need to catch in the aim to return a specific HTTP Status Code or a specific message, you should catch but leave other exceptions to the global handler.

c- No Business in non-business place

API is not the right place to implement business rules. As you can do, always delegate the business rules to components to be sure that components reused for other applications (internal purposes) continue to apply or continue to be able to apply the business rules.
The goal behind this point is to avoid the duplicate code in the API and in the component if your API plays multiple times with the business rules and also to reuse a maximum of business code.