THE STANDARD

TEAM PRACTICES

HASSAN HABIB

Culture, Purpose and Practices
for Engineering Teams

The Standard Team Practices

Hassan Habib

Free to read online at hassanhabib.com/books/the-standard-team

Thank you, Lord for all you taught me. Forgive me if I forget anything

Contents

0 **Introduction**

1 **Purpose**

1.1 Scenarios

2 **Contributing Code**

2.1 Branching and Forking

2.2 Naming Your Work

2.3 Commits

2.4 Pull Requests

2.5 Solution and Project Structure

3 **Configuration**

3.1 App Settings and Environment Variables

3.2 Secrets in GitHub Actions

4 **Measurements**

4.1 Ranks

4.2 Estimating Effort

0

Introduction

The Team Standard is a set of overall principles that shall guide any engineering team to create a prosperous productive engineering culture. A culture that prefers people over processes, and considers any end-result or success as a side-effect to a successful engineering community.

Products, projects and deliverables come and go. Your best product today shall become legacy tomorrow. And what really lasts is the relationships you've developed with your team during the process of building such products.

An engineering team with strong culture can become an unstoppable force, powered by passion and compassion. Aiming to fulfill its goals on both the individual and community levels.

This Standard is written by an imperfect man. Therefore it shall also be imperfect. Incomplete. Subject to be changed or modified and completely scrapped off. It's an accumulation of personal experiences in the engineering industry, and proposed solutions to protect future generations from toxic and counter-productive practices.

This Standard is also forever changing. It's a reflection of ongoing learnings, experiences and interactions with many engineers within several communities on daily basis. It's always open for input, no matter where you are, what you are or how long you've been in the industry. This Standard is built to make sure your voice is heard, recognized and appreciated.

The main intent of this Standard is not to create obedient cogs. The type of cogs you may see today following orders without questioning. Heartless, cold and brainless cogs, willing to let go of their principles (if they had any) to appease their masters. This Standard would rather inspire the re-birth of prosperous, ambitious and innovative humans - embodying every single aspect of their humanity into what they create.

The principles of The Team Standard are aiming towards protecting any engineering team from organizational diseases and toxic patterns such as silos, saviors, favoritisms, levelisms and many other patterns that play destructive roles into slowing down if not blocking the progress of any team on both the individual and community levels.

The Team Standard mainly outlines three main factors that directly impacts the engineers in any team on daily basis in terms of productivity. These factors are: The Individual, The Community and The Purpose.

Let me briefly explain why these factors are critical to any engineering culture existing or just starting from scratch.

Engineers are the building blocks of any team. They are the foundation of any culture. Without engineers there's no community. Without a community there's no culture, purpose or product. If engineers are not taken care of, the entire community falls.

But Individual engineers without strong relationship between them are no different than a bunch of random individuals who happen to be in the same elevator at the same time. Nothing in common between them except for the environmental situation. Therefore, a community has to be established, to outline the communications, interactions and relationships between the individuals within that community.

A community without a purpose is like a ship with no compass. You may have the strongest, smartest most ambitious individuals with the greatest relationships with one another. But without a real purpose you have a ship with no compass. Being lost shall burn through the strong ties of any team till there's nothing left but empty shells.

An engineering team that makes their culture a higher priority than their deliverables are destined to succeed into whatever they do. This Standard favors people over numbers. People over profits. People over processes. It doesn't just aim to help you deliver a good product, but rather empowers you to look beyond the material and embrace a more fulfilling life pushing towards the survival of our humankind, its evolution and fulfillment.

Sincerely,

Hassan Habib

1

Purpose

Every engineering team is responsible for defining and clarifying their purpose. A purpose for every team is the compass that guides the team and ensures they stay on track. Engineering teams should not identify their purposes as means. For instance, making profits is not a purpose in and of itself. But rather means to achieve higher aspects and bigger goals. The ultimate purpose for every engineering team is to contribute to the survival of the humankind, its evolution and fulfillment. A team of engineers that keeps this purpose as their north star is destined to create the highest impact in their lifetimes and after. But the purposing process requires intentional, systematic and deliberate efforts to define the material purpose. For instance, a team of engineers working on a restaurant software have to incorporate the fact that building efficient software falls directly into warding off someone's hunger. The human impact is what connects us at the end of the day. Purposing requires chasing the human impact regardless of whether it's direct or indirect. For instance, building software for the turbo-machinery industry requires thinking about the people the software impacts. Building a simple human-machine-interface (HMI) will result in saving the workers who are going to be using the software's time. Making their work environment safer. Expediting the production process therefore these workers become more productive. Any software out there built to cause humans harm in terms of destruction, spreading misinformation, scamming people or replacing their daily jobs with automated processes without offering an alternative is a software that aims to the destruction of the humankind, its devolution and depression. As software engineers we must always consider the human aspect of what we build. How does it largely impact beyond our immediate family but looking beyond that horizon and ensuring we are building healthy, productive software.

The purposing process requires these main aspects to be considered.

1. Overall Purpose

2. Scenarios
3. Features
4. User Stories
5. Tasks
6. Defects

The Standard Team is to clearly understand and define each and every aspect of the aforementioned points.

The Overall Purpose

The overall purpose of every project is what defines the boundaries of the project. For instance, if we are building a Bakery system, we need to define the following aspects:

- Nature of the business
- Goals of the business
- Future considerations

Business Nature

The nature of the business is what defines what the business is. Our Bakery system would define its nature as follows: A bakery is an establishment that produces and sells flour-based food baked in an oven such as bread, cookies, cakes, pastries, and pies.

Business Goals

Deliver the highest quality of bakeries to as many customers as possible.

Future Considerations

Expanding to become a nation-wide bakery business selling across the country.

These above three aspects should help the engineering team understand the nature, goals and potential growth of the business to ensure they build software that can support that expansion. For instance, knowing that one of the goals of the business is that it will be nation-wide will help engineers determine whether to build a simple POS system running as a desktop application or a web application with mobile app capability to help customers order on-the-go as fast and as simple as possible.

1.1

Scenarios

Now that we have clarified the overall purpose of the business, we start going into scenarios where we talk about higher-level goals of achieving these purposes. Let's talk about that.

Scenarios of any business are not meant to be technical. But they can dive into that level if needed. You shouldn't expect any technical language in any given scenario unless the business itself is technical. Every scenario needs to fulfil the following aspects:

- Actors (Given)
- Interactions (When)
- Outcomes (Then)

Here's an example for a scenario:

> **GIVEN** a customer of ABC Bakery
> **WHEN** I want to order a pie
> **THEN** I should be able to place the order from home

This is an overall scenario that highlights the actor – which is the customer of the bakery. And the interactions which in case is the urge/intention to order a pie. Then the outcome which is the ability to order a pie online. This scenario outlines the actions, the trigger (the interaction) and then the outcome. It's a very high-level requirement as it doesn't define API endpoints, certain user interface or anything of that nature. Scenarios need to always stay with the end-users, the actual customers of the product. And what they do and the outcomes they expect out of the software. Scenarios sometimes can hand-over to each other. The above scenario is a high-level scenario that can be followed by this scenario here:

> **GIVEN** a customer of ABC Bakery
> **WHEN** I order a pie online
> **THEN** I should receive it within 5 minutes timeframe

This scenario here added up on top of the previous one. It's already established that customers can order online. Now we have a new scenario that is marketable to everyone where we can say: "We deliver our pies in less than 5 minutes guaranteed!" Think about your outcomes as something that can be advertised on TV for everyone including non-technical people. What do they care about in terms of outcomes? That's where these scenarios live.

Here's a bunch of other examples for reference:

> **GIVEN** a cooker of ABC Bakery
> **WHEN** I need to see pending orders
> **THEN** I should be able to view all my pending orders

> **GIVEN** a manager of ABC Bakery
> **WHEN** I need to monitor the progress of my business
> **THEN** I should be able to see all activity of my business

The overall theme here is generic overall results. The cooker wants to be able to view all the incoming pending orders. The manager wants to see all activities on the business. Nothing technical or specific to how this is going to be implemented. More details are welcomed at this level. Scenarios are written in face-to-face meetings with customers. They are raw requirements not yet broken down into features, stories or tasks.

2

Contributing Code

Having defined practices for code contributions, branch, commit strategies and project structures is essential for any software development project. These practices provide a clear and consistent approach to managing the codebase, ensuring that contributions are made in a controlled and organized manner. This helps to avoid confusion and conflicts, and ensures that the codebase remains stable and maintainable. Additionally, well-defined practices for code contributions and project structure can help to improve collaboration and communication within the development team, making it easier to track and review changes, and to identify and resolve issues. Overall, defined practices for code contributions, branching and commit strategies as well as project structures are a key part of any successful software development project.

Defining a contribution guideline is crucial for any open-source or collaborative software development project. It establishes clear rules and expectations for how contributors should submit and review changes, which helps to ensure that all contributions are made in a consistent and organized manner. Additionally, a contribution guideline can provide guidance on best practices for coding, testing and documentation, which can help to improve the overall quality of the codebase. It also serves as an educational tool to new contributors, providing them with the necessary information and instructions on how to contribute to the project effectively. (Furthermore, having a well-defined GitHub contribution guideline can help to attract new contributors and maintain a healthy open-source community.)

2.1

Branching and Forking

A branch strategy for code is essential for maintaining a stable and maintainable codebase. It allows for multiple developers to work on different features or bug fixes simultaneously, without interfering with each other's work. Each developer can create their own branch to work on, which can then be merged into the main branch once it has been reviewed and approved. This helps to avoid conflicts, and ensures that the codebase remains stable and consistent. Furthermore, a branch strategy enables version control and allows different versions of the codebase to be created and tracked. This can be useful for rolling back changes if necessary, and for maintaining multiple versions of the software for different environments. Overall, a branch strategy is a critical aspect of any software development project and is essential for ensuring that the codebase remains stable and maintainable.

There are several different types of branching strategies that can be used in software development, including:

1. Gitflow: A popular branching strategy that follows a strict branching model, where development happens in feature branches and is merged into a main development branch. This strategy is good for large projects with many contributors.
2. Trunk-Based Development: This strategy involves working on the main branch, or trunk, and committing changes directly to it. This is a good strategy for smaller projects with a small number of contributors.
3. Feature Branching: This strategy involves creating a separate branch for each feature or bug fix, and merging it into the main branch when it is complete. This is a good strategy for larger projects with multiple contributors.
4. Forking: This strategy involves creating a copy of the repository and working on it separately. This is good for open-source projects where multiple contributors are working on the same codebase. The main advantage of the Forking

workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

5. Release Branching: This strategy involves creating a separate branch for each release, and merging it into the main branch when it is ready to be released. This is a good strategy for projects that have a regular release schedule.
6. Continuous Integration: This strategy involves merging code changes as soon as they are made, and running automated tests to ensure that the codebase remains stable. This is a good strategy for projects that have a high frequency of changes.

Ultimately, it's important to choose a branching strategy that fits the needs and constraints of the project, and that can be easily understood and followed by all the contributors.

Forking for Open Source

As The Standard promotes open-source work, let's have a quick look at the Forking strategy.

How it works

The Forking workflow begins with an official public repository stored on a server. When a new developer wants to start working on the project, they do not directly clone the official repository.

Instead, they fork the official repository to create a copy of it. This new copy serves as their personal public repository. After they have created their server-side copy, the developer performs a git clone to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository (not the official one). They then file a pull request with the main repository, which lets the project maintainer know that an update is ready to be

integrated. The pull request can then be reviewed and it also serves as a convenient discussion thread if there are issues with the contributed code.

The following is a step-by-step example of this workflow:

1. A developer ‘forks’ an ‘official’ server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A new local feature branch is created.
4. The developer makes changes on the new branch.
5. New commits are created for the changes.
6. The branch gets pushed to the developer’s own server-side copy.
7. The developer opens a pull request from the new branch to the ‘official’ repository.
8. The pull request gets reviewed and build success and tests are verified. At this point the maintainer can either request changes or approve for merge upon which the changes are then merged into the original server-side repository.

2.2

Naming Your Work

Every branch, commit and pull request carries a category. The same list decides how a piece of work is named and, later, how it is counted.

Branch Names

For branch names we can apply the following naming convention:
`users/[username]/[category]-[entity]-[action]`

The variables can be substituted

- `[username]` => your github username
- `[category]` => the category that you are working on
- `[entity]` => the entity that your service / broker uses
- `[action]` => the action

Categories

The categories below are grouped by the kind of work they describe.

Infrastructure and delivery

Category	Description
`INFRA`	Initial project setup, creating the build project / build scripts.
`MAJOR INFRA`	Major updates, removing or adding simple configurations or files.
`MEDIUM INFRA`	Medium updates, removing or adding simple configurations or files.
`MINOR INFRA`	Minor updates, removing or adding simple configurations or files.

Category	Description
`PROVISIONS`	Creating provision project / scripts.
`RELEASES`	Infrastructure work to release software.
`CONFIG`	Any configuration changes i.e. setting up appsettings.json for your various environments.

Data

Category	Description
`DATA`	Creation of a data model (and its migration when EF is used)
`MAJOR DATA`	Changing existing data model with 5+ property additions/modifications.
`MEDIUM DATA`	Changing existing data model with 3-4 property additions/modifications.
`MINOR DATA`	Changing existing data model with 1-2 property additions/modifications.
`MIGRATION`	Moving or transforming data from one system to another, updating existing data, or generating reports.
`MAJOR MIGRATION`	Major data migration or transformation tasks involving significant data volume or complexity.
`MEDIUM MIGRATION`	Medium complexity data migration or updates involving moderate data volume or effort.
`MINOR MIGRATION`	Minor data migration or simple data updates with low complexity or small data volume.

Brokers and services

Category	Description
`BROKERS`	When creating a broker to wrap external libraries, resources, services, or APIs
`MAJOR BROKERS`	Major changes to existing broker (significant functionality changes).
`MEDIUM BROKERS`	Medium changes to existing broker (multiple significant modifications).

Category	Description
`MINOR BROKERS`	Minor changes to existing broker (simple modifications).
`FOUNDATIONS`	When creating a Foundation Service
`MAJOR FOUNDATIONS`	Changing existing foundation service with writing or updating all or to 5+ tests.
`MEDIUM FOUNDATIONS`	Changing existing foundation service with writing or updating to 3-4 tests.
`MINOR FOUNDATIONS`	Changing existing foundation service with writing or updating to 1-2 tests.
`PROCESSINGS`	When creating a Processing Service
`MAJOR PROCESSINGS`	Changing existing processing service with writing or updating all or to 5+ tests.
`MEDIUM PROCESSINGS`	Changing existing processing service with writing or updating to 3-4 tests.
`MINOR PROCESSINGS`	Changing existing processing service with writing or updating to 1-2 tests.
`ORCHESTRATIONS`	When creating an Orchestration Service
`MAJOR ORCHESTRATIONS`	Changing existing orchestration service with writing or updating all or to 5+ tests.
`MEDIUM ORCHESTRATIONS`	Changing existing orchestration service with writing or updating to 3-4 tests.
`MINOR ORCHESTRATIONS`	Changing existing orchestration service with writing or updating to 1-2 tests.
`COORDINATIONS`	When creating a Coordination Service
`MAJOR COORDINATIONS`	Changing existing coordination service with writing or updating all or to 5+ tests.
`MEDIUM COORDINATIONS`	Changing existing coordination service with writing or updating to 3-4 tests.
`MINOR COORDINATIONS`	Changing existing coordination service with writing or updating to 1-2 tests.

Category	Description
`MANAGEMENTS`	When creating a Management Service
`MAJOR MANAGEMENTS`	Changing existing management service with writing or updating all or to 5+ tests.
`MEDIUM MANAGEMENTS`	Changing existing management service with writing or updating to 3-4 tests.
`MINOR MANAGEMENTS`	Changing existing management service with writing or updating to 1-2 tests.
`AGGREGATIONS`	When creating an Aggregation Service
`MAJOR AGGREGATIONS`	Changing existing aggregation service with writing or updating all or to 5+ tests.
`MEDIUM AGGREGATIONS`	Changing existing aggregation service with writing or updating to 3-4 tests.
`MINOR AGGREGATIONS`	Changing existing aggregation service with writing or updating to 1-2 tests.

Exposers

Category	Description
`CONTROLLERS`	When creating a Controller
`MAJOR CONTROLLERS`	Changing existing controller with writing or updating all or to 5+ tests.
`MEDIUM CONTROLLERS`	Changing existing controller with writing or updating to 3-4 tests.
`MINOR CONTROLLERS`	Changing existing controller with writing or updating to 1-2 tests.
`CLIENTS`	When creating a client on a library that others can use
`MAJOR CLIENTS`	Changing existing client with writing or updating to 5+ tests.
`MEDIUM CLIENTS`	Changing existing client with writing or updating to 3-4 tests.
`MINOR CLIENTS`	Changing existing client with writing or updating to 1-2 tests.

Category	Description
`PROVIDERS`	When creating a provider on a SPAL provider library that others can use
`EXPOSERS`	When creating any other kind of exposer i.e. Program.cs
`MAJOR EXPOSERS`	—
`MEDIUM EXPOSERS`	—
`MINOR EXPOSERS`	—

Front end

Category	Description
`VIEWS`	When creating a View Service
`MAJOR VIEWS`	Changing existing view with writing or updating all or to 5+ tests.
`MEDIUM VIEWS`	Changing existing view with writing or updating to 3-4 tests.
`MINOR VIEWS`	Changing existing view with writing or updating to 1-2 tests.
`BASES`	When creating a Frontend Base Component
`MAJOR BASES`	—
`MEDIUM BASES`	—
`MINOR BASES`	—
`COMPONENTS`	When creating a Component
`MAJOR COMPONENTS`	Changing existing component with writing or updating all or to 5+ tests.
`MEDIUM COMPONENTS`	Changing existing component with writing or updating to 3-4 tests.
`MINOR COMPONENTS`	Changing existing component with writing or updating to 1-2 tests.
`PAGES`	When creating a Blazor Page
`MAJOR PAGES`	—
`MEDIUM PAGES`	—

Category	Description
`MINOR PAGES`	—

Tests

Category	Description
`ACCEPTANCE`	When writing an Acceptance Test
`MAJOR ACCEPTANCE`	—
`MEDIUM ACCEPTANCE`	—
`MINOR ACCEPTANCE`	—
`INTEGRATION`	When writing an Integration Test
`MAJOR INTEGRATION`	—
`MEDIUM INTEGRATION`	—
`MINOR INTEGRATION`	—

Fixes and clean-up

Category	Description
`CODE RUB`	Small change to fix things styling, code formatting, spelling (Not for bug fixes)
`MAJOR CODE RUB`	Significant code cleanup affecting multiple files.
`MEDIUM CODE RUB`	Moderate code cleanup affecting several files.
`MINOR CODE RUB`	Minor code cleanup affecting few files.
`MINOR FIX`	A minor bug fix / change that do not significantly alter the overall functionality of the program.

Category	Description
`MEDIUM FIX`	A fix that has a moderate level of impact on the functionality or performance of a program or system.
`MAJOR FIX`	A major bug fix is a significant repair or correction made to a software program or system that addresses a critical or major issue.
`IMPORT`	When you are copying code and tests over from another system with no or minor changes like namespaces. Should include the component being imported, format: IMPORT: [COMPONENT]: [Description]

Design and documentation

Category	Description
`DOCUMENTATION`	General documentation
`STANDARD`	When you update or introduce a new thing in The Standard
`DESIGN`	Creating documentation has design details for your architecture (High Level Design / Low Level Design)
`MAJOR DESIGN`	Designing 15+ service methods.
`MEDIUM DESIGN`	Designing 10-14 service methods.
`MINOR DESIGN`	Designing <10 service methods.
`BUSINESS`	Creating documentation that outlines your business processes / Standard Operating Procedures.
`STATUS`	When updating STATUS information in your design documentation.

People and process

Category	Description
`REVIEW`	Reviewing submitted work in the context of project standards, and Standard compliance.
`PLANNING`	Planning should occur once per feature and involve collaboration across services. Individual work and planning for themselves doesn't quality as planning. Planning involves multiple engineers with a leader who assigns and distributes the tasks across the team.
`MAJOR PLANNING`	Planning 10+ tasks.
`MEDIUM PLANNING`	Planning 5+ tasks.
`MINOR PLANNING`	Planning <5 tasks.
`MENTORSHIP`	60+ minute session.
`MAJOR MENTORSHIP`	60+ minute session.
`MEDIUM MENTORSHIP`	45 minute session.
`MINOR MENTORSHIP`	30 minute session.
`DISCUSSION`	When conducting a discussion meeting under the DISCUSSION title, surrounding technical issues.

Category	Description
`MAJOR DISCUSSION`	60+ minute discussion.
`MEDIUM DISCUSSION`	45 minute discussion.
`MINOR DISCUSSION`	30 minute discussion.

2.3

Commits

The Standard follows a TDD (Test-Driven Development) software development methodology that emphasizes writing tests before writing the actual code. It is a process of writing a test case, running it and observing it failing, then writing the code to pass the test. It's an iterative process that helps developers to focus on the requirements, design, and implementation of the code. When using TDD, developers write test cases using a testing framework such as xUnit or NUnit, and use the framework to run and assert the tests. The goal is to have a high percentage of test coverage, ensuring that the code is thoroughly tested and any bugs or issues are caught early on in the development process. This approach helps to ensure that the code is maintainable, easy to understand, and that it meets the requirements of the project.

When following this TDD process of writing a test case we can use the test name and its outcome to commit the work. The work done to write the test and observing it failing can be committed as `[Test Name] -> FAIL` e.g. `ShouldAddPostAsync -> FAIL` and subsequently the work done to make it pass can be committed as `[Test Name] -> PASS` e.g. `ShouldAddPostAsync -> PASS`

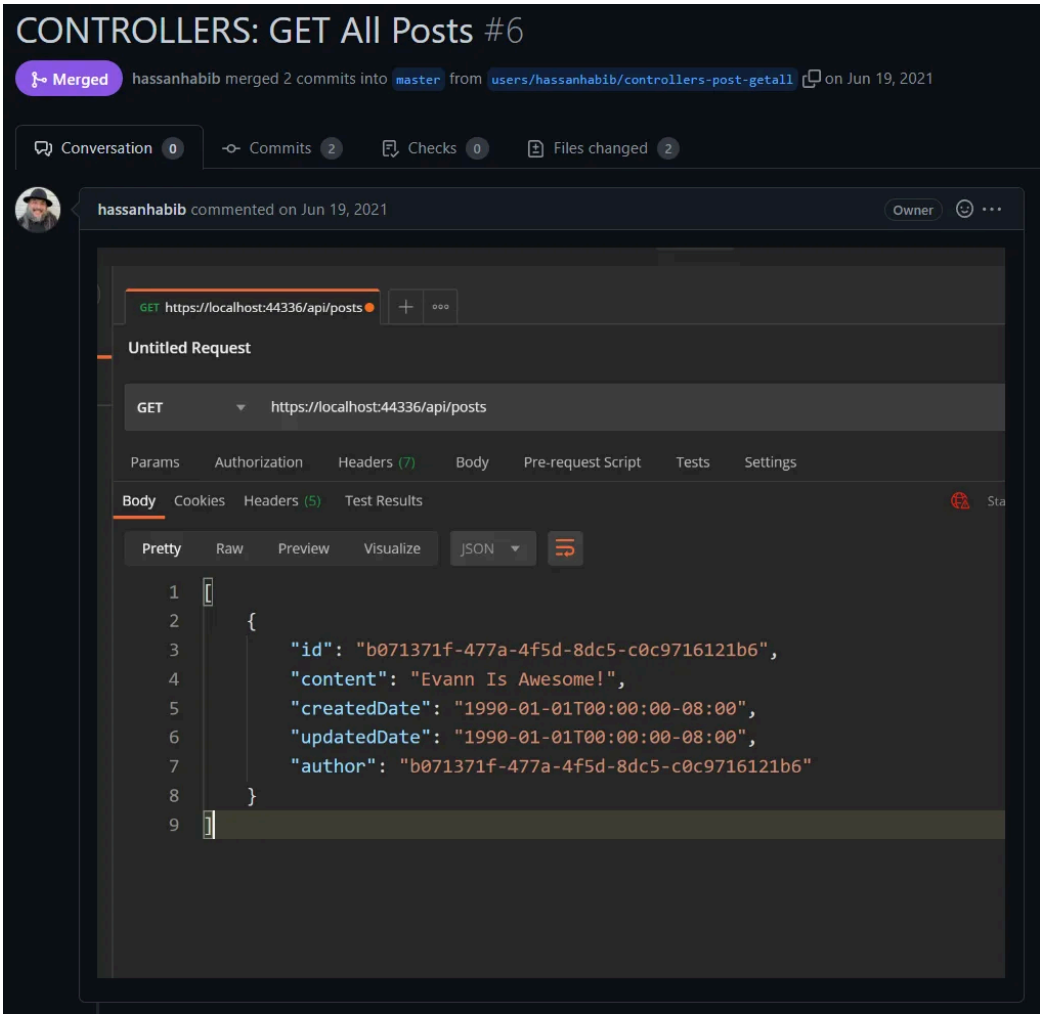
If you are working on any code that does not require testing i.e. DATA, BROKERS, CONTROLLERS, then we would adopt the same naming convention as used for Pull Requests which uses the category and a description of the work done using this syntax `[CATEGORY]: [Description Of Work Completed]`, where the category is always in CAPS and the description in Pascal Case e.g. `DATA: Add Student Model` OR `BROKERS: Insert Student` OR `CONTROLLERS: POST Student` (See Naming Your Work for the complete list of categories.)

2.4

Pull Requests

When developers have completed their work, they can create a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The Pull Request (PR) can be created with the name using this syntax `[CATEGORY]: [Description Of Work Completed]`, where the category is always in CAPS and the description in Pascal Case e.g. `FOUNDATIONS: Add Student`

The comment section of the PR can also be completed with any additional information that describe the work done and that can be helpful / relevant to the approver. e.g. a screenshots of the CONTROLLER actions working as expected showing the various response outcomes like `201 - Created` or `400 - Bad Request` or `424 - Failed Dependency` etc.



You can also link a pull request to an issue to show that a fix is in progress and to automatically close the issue when the pull request or branch is merged by adding a KEYWORD #ISSUE-NUMBER anywhere in the comment section. e.g. `Closes #10` or if the PR cover multiple things, then you could do `Closes #10, closes #123`

If you simply wish to link an issue without a closing action on merge, you can omit the keyword and just add #ISSUE-NUMBER e.g. `#10`

2.5

Solution and Project Structure

Visual Studio utilizes a hierarchical project structure that allows developers to organize their code and resources in a logical and easy-to-navigate manner. At the top level, you have a solution that is a container for one or more projects. A project is the container for all of the files and resources that make up an application.

Projects in the Solution

A typical solution structure would look like this...

```
  |-- Taarafo.Core                              (API)
  |-- Taarafo.Core.Infrastructure.Build         (Console App)
  |-- Taarafo.Core.Infrastructure.Provision     (Console App)
  |-- Taarafo.Core.Tests.Acceptance             (xUnit Test Project)
  |-- Taarafo.Core.Tests.Build                  (xUnit Test Project)
```

Project Structure

A typical project folder structure would look like this...

```
Taarafo.Core
  |-- Brokers
        |-- DateTimes
        |-- Loggings
        |-- Storages
  |-- Models
        |-- Foundations
            |-- Posts
               |-- Exceptions
            |-- Comments
               |-- Exceptions
  |-- Migrations
  |-- Services
        |-- Foundations
            |-- Students
        |-- Processings
            |-- Students
```

```
        |-- Orchestrations
            |-- Students
  |-- Controllers
```

The above folder structure can also be carried over into the test projects to organise models and to group the various tests by Services\ `[Service Type]` \ `[ServiceName]`

3

Configuration

Configuration settings are values that determine the behavior of a software application. They can include anything from database connection strings and API keys to feature flags and logging levels. By using configuration settings, developers can customize the behavior of their application without modifying the code, making it easier to deploy and manage different environments.

One popular way of storing configuration settings in .NET applications is by using the `appsettings.json` file. This file is a JSON-formatted file that contains a set of key-value pairs representing configuration settings. Developers can easily modify these settings to change the behavior of their application.

However, not all configuration settings can be safely stored in `appsettings.json` files. Sensitive data, such as API keys, database passwords, or any other data that could grant unauthorized access to the application or its resources, should be kept secret. Storing sensitive data in configuration files, especially in plain text format, can put the application's security at risk.

It is very important for developers to follow secure coding practices and never store passwords or other sensitive data in configuration provider code or in plain text configuration files. Such sensitive data should be stored in a secure location like Azure Key Vault or in environment variables that are set in a deployment environment. Storing passwords in local settings files is also not recommended as this can easily lead to scenarios where passwords are accidentally checked into code repositories, either because a developer has forgotten to exclude the file from version control or as a result of someone changing it and then inadvertently including the sensitive files, putting the application's security at risk.

One way of protecting sensitive data is by using **user secrets**. User secrets is a feature in .NET that provides a convenient way for developers to store and retrieve sensitive data during development. This data is stored locally on the developer's

machine and is not intended to be used in production. While user secrets can be useful for keeping sensitive data out of source control and easily accessible during development, it is important to note that they are not compatible with GitHub workflow actions. Instead, developers should consider using more secure methods for storing secrets, such as environment variables or storing them in a secure location like `GitHub Action Secrets` or `Azure Key Vault`. As user secrets is not directly compatible with GitHub build pipelines we will not explore this option further.

Another way of protecting sensitive data is the use of **environment variables**. Environment variables are variables that are set in the operating system, which can be accessed by applications at runtime.

By using both **appsettings.json** and **environment variables**, developers can separate sensitive data from other configuration settings, making it easier to manage and secure. Additionally, this approach enables developers to store their configuration settings securely in a build pipeline, ensuring that sensitive data is protected throughout the development lifecycle.

3.1

App Settings and Environment Variables

A .NET app will automatically load and register `appsettings.json` and `appsettings.{Environment}.json` with the IConfiguration interface during application startup, if the files are located in the application's root directory. This means that the key-value pairs defined in the appsettings.json file will be accessible through the IConfiguration object, allowing developers to easily access and use the configuration settings throughout their application.

It's worth noting that in addition to the appsettings.json file, there are other configuration providers available in .NET that allow developers to load configuration settings from various sources, such as environment variables or command-line arguments. (Developers can also create their own custom configuration providers to load configuration settings from other sources if needed.)

Configuration sources are read in the order that their configuration providers are specified. Order configuration providers in code to suit the priorities for the underlying configuration sources that the app requires.

A typical sequence of configuration providers is:

1. appsettings.json
2. appsettings.{Environment}.json
3. User secrets
4. Environment variables using the Environment Variables configuration provider.
5. Command-line arguments using the Command-line configuration provider.

Let's look at some code samples to see how all this works.

An Example

appsettings.json

```
{
  "MySettings": {
    "ApiUrl": "https://api.somesite.com/",
    "ApiKey": "1ae4e397-ec3c-4ed7-8280-a17d0e2cbe78",
    "OrganisationId": "1bac6df0-cd68-4ce5-9c29-b2beb58201cd"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

With the above `appsettings.json` file we can then load our variables like this:

```
    string apiUrl = configuration["MySettings:ApiUrl"];
    string apiKey = configuration["MySettings:ApiKey"];
    string organisationId = configuration["MySettings:OrganisationId"];
```

Environment Variables

We can add environment variables to a web application by doing this:

```
var builder = WebApplication.CreateBuilder(args);
...
builder.Configuration.AddEnvironmentVariables();
...
var app = builder.Build();
```

or if you want to play with this in a Console App or Unit Test, you can do this:

```
var configurationBuilder = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables("MYAPP_ACCEPTANCE_");

this.configuration = configurationBuilder.Build();
```

NOTE that we have defined a prefix that environment variables must start with. This prefix will automatically be removed from the environment variable names by the configuration builder. This is a very useful feature as you can now store environment variables with the same name for multiple apps and/or environments.

Up to this point we will still get the same values if we get our config items as we have not set the values for any environment variables yet.

In C# .NET, working with environment variables is straightforward. The Environment class provides several methods for reading and setting environment variables. For example, to read an environment variable, you can use the `Environment.GetEnvironmentVariable` method, passing in the name of the variable you want to retrieve. To set an environment variable, you can use the `Environment.SetEnvironmentVariable` method, providing the name of the variable and its value. You can also use the command line to set the environment variables by running the SETX command.

For demonstration we will leave the value of `MySettings:ApiUrl` in appsettings.json

Next we will set the value of `MySettings:ApiKey` through this line of code

```
Environment.SetEnvironmentVariable("MYSETTINGS:APIKEY", "3d4a1c55-fcd7-4b34-8536-99c8ae6ae33c");
```

and for the `MySettings:OrganisationId` we will use the command line argument through a console window with administrative privileges

```
setx MYSETTINGS:ORGANISATIONID "b2440ae9-cad2-4d70-b138-4a807abe1bb7"
```

NOTE Visual Studio preloads the environment variables when it starts, and it caches them until the application is closed. Unlike environment variables set through code, those set using the command line will not be immediately available due to the preload behaviour. **You will need to close Visual Studio and re-open it.**

Once you have reloaded Visual Studio we can get the values from the configuration again by doing this:

```
    string apiUrl = configuration["MySettings:ApiUrl"];
    string apiKey = configuration["MySettings:ApiKey"];
    string organisationId = configuration["MySettings:OrganisationId"];
```

You will notice that the ApiUrl is the same as before, but the values have now changed for `MySettings:ApiKey` and `MySettings:OrganisationId` and they are now equal to the values we specified for the environment variables via code and through SETX on the command line.

You can also remove environment variables through code by setting the value to `null`.

```
Environment.SetEnvironmentVariable("MYSETTINGS:APIKEY", null);
```

OR via the command line you can do

```
setx MYSETTINGS:APIKEY /delete
```

Or if you prefer a GUI, you can also go to SYSTEM PROPERTIES > ENVIRONMENT VARIABLES where you can ADD / REMOVE environment variables.

3.2

Secrets in GitHub Actions

To setup your environment variables for use within GitHub we will need to setup repository secrets and add a section at the beginning of your workflow file.

1. Go to the GitHub repository where you want to use these settings and click on “Settings” in the top navigation bar.
2. In the left sidebar, click on “Secrets” and then click on “New repository secret” button.
3. Create two new secrets named “MYAPP_ACCEPTANCE_MYSETTINGS__APIKEY” and “MYAPP_ACCEPTANCE_MYSETTINGS__ORGANISATIONID”, with the corresponding values for “ApiKey” and “OrganisationId” from the given settings. (“ApiUrl” stays in appsettings.json.)

   NOTE

   - We have prefixed our environment variables with `MYAPP_ACCEPTANCE_` to match the earlier configuration of `.AddEnvironmentVariables("MYAPP_ACCEPTANCE_");`.
   - The other difference is that a colon (:) is not allowed on GitHub Secrets to represent a hierarchy and you have to use a double underscore (__) instead.
4. To expose these secrets as environment variables in the GitHub Action Workflow, add the following code snippet at the beginning of your workflow file:

```
env:
  ApiKey: ${{ secrets.MYAPP_ACCEPTANCE_MYSETTINGS__APIKEY }}
  OrgId: ${{ secrets.MYAPP_ACCEPTANCE_MYSETTINGS__ORGANISATIONID }}
```

A full example of the `dotnet.yml` file will look like this:

```
name: MyApplication Build
on:
```

```yaml
  push:
    branches:
    - main
  pull_request:
    branches:
    - main
jobs:
  build:
    runs-on: windows-latest
    env:
      ApiKey: ${{ secrets.MYAPP_ACCEPTANCE_MYSETTINGS__APIKEY }}
      OrgId: ${{ secrets.MYAPP_ACCEPTANCE_MYSETTINGS__ORGANISATIONID }}
    steps:
    - name: Pulling Code
      uses: actions/checkout@v2
    - name: Installing .NET
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 7.0.201
        include-prerelease: false
    - name: Restoring Packages
      run: dotnet restore
    - name: Building Solution
      run: dotnet build --no-restore
    - name: Running Tests
      run: dotnet test --no-build --verbosity normal
```

4

Measurements

The Standard promotes ideals of fairness to all. The Standard recognizes that innovative ideas, great contributions and high performance are not exclusive to individuals with certain titles. In order to truly and fairly ensure everyone's contributions are measured fairly, The Standard introduces values and points to contributions at different levels and categories. The accumulations of these contributions increases the rank of the contributor from one level to the next.

Contribution Points

The following is a list of all types of contributions and the value for each. A contribution is scored by the category its title starts with; each category is defined in Naming Your Work.

Infrastructure and delivery

Title starts with	Points
`INFRA`	10
`MAJOR INFRA`	10
`MEDIUM INFRA`	3
`MINOR INFRA`	1
`PROVISION`	10
`RELEASE`	10
`CONFIG`	5

Data

Title starts with	Points
`DATA`	5
`MAJOR DATA`	5
`MEDIUM DATA`	3
`MINOR DATA`	1
`MIGRATION`	20
`MAJOR MIGRATION`	20
`MEDIUM MIGRATION`	10
`MINOR MIGRATION`	5

Brokers and services

Title starts with	Points
`BROKER`	5
`MAJOR BROKER`	5
`MEDIUM BROKER`	3
`MINOR BROKER`	1
`FOUNDATION`	10
`MAJOR FOUNDATION`	10
`MEDIUM FOUNDATION`	5
`MINOR FOUNDATION`	1
`PROCESSING`	15
`MAJOR PROCESSING`	15
`MEDIUM PROCESSING`	10
`MINOR PROCESSING`	5
`ORCHESTRATION`	20

Title starts with	Points
`MAJOR ORCHESTRATION`	20
`MEDIUM ORCHESTRATION`	15
`MINOR ORCHESTRATION`	10
`COORDINATION`	20
`MAJOR COORDINATION`	20
`MEDIUM COORDINATION`	15
`MINOR COORDINATION`	5
`MANAGEMENT`	20
`MAJOR MANAGEMENT`	20
`MEDIUM MANAGEMENT`	15
`MINOR MANAGEMENT`	5
`AGGREGATION`	10
`MAJOR AGGREGATION`	10
`MEDIUM AGGREGATION`	5
`MINOR AGGREGATION`	1

Exposers

Title starts with	Points
`CONTROLLER`	5
`MAJOR CONTROLLER`	5
`MEDIUM CONTROLLER`	3
`MINOR CONTROLLER`	1
`CLIENT`	5
`MAJOR CLIENT`	5
`MEDIUM CLIENT`	3
`MINOR CLIENT`	1

Title starts with	Points
`EXPOSER`	5
`MAJOR EXPOSER`	5
`MEDIUM EXPOSER`	3
`MINOR EXPOSER`	1

Front end

Title starts with	Points
`VIEW`	10
`MAJOR VIEW`	10
`MEDIUM VIEW`	5
`MINOR VIEW`	1
`BASE`	5
`MAJOR BASE`	5
`MEDIUM BASE`	3
`MINOR BASE`	1
`COMPONENT`	15
`MAJOR COMPONENT`	15
`MEDIUM COMPONENT`	10
`MINOR COMPONENT`	5
`PAGE`	5
`MAJOR PAGE`	5
`MEDIUM PAGE`	3
`MINOR PAGE`	1

Tests

Title starts with	Points
`ACCEPTANCE`	10
`MAJOR ACCEPTANCE`	10
`MEDIUM ACCEPTANCE`	5
`MINOR ACCEPTANCE`	3
`INTEGRATION`	15
`MAJOR INTEGRATION`	15
`MEDIUM INTEGRATION`	10
`MINOR INTEGRATION`	10

Fixes and clean-up

Title starts with	Points
`MAJOR FIX`	20
`MEDIUM FIX`	10
`MINOR FIX`	5
`MAJOR CODE RUB`	10
`MEDIUM CODE RUB`	5
`MINOR CODE RUB`	1
`CODE RUB`	1
`IMPORT`	5

Design and documentation

Title starts with	Points
`DESIGN`	20
`MAJOR DESIGN`	20
`MEDIUM DESIGN`	10

Title starts with	Points
`MINOR DESIGN`	5
`DOCUMENTATION`	1
`STANDARD`	100
`BUSINES`	50
`STATUS`	1

People and process

Title starts with	Points
`PLANNING`	15
`MAJOR PLANNING`	15
`MINOR PLANNING`	10
`MINIOR PLANNING`	5
`MENTORSHIP`	20
`MAJOR MENTORSHIP`	20
`MINOR MENTORSHIP`	15
`MINIOR MENTORSHIP`	10
`DISCUSSION`	15
`MAJOR DISCUSSION`	15
`MINOR DISCUSSION`	10
`MINIOR DISCUSSION`	5
`REVIEW`	1

Unrecognized

Title starts with	Points
`UNKNOWN`	0

4.1

Ranks

The following are the ranks based on the life-time accumulation and contribution to The Standard Community and Standard-Compliant projects:

POSITION	SCORE
SWE I	10,000
SWE II	25,000
SR. SWE	50,000
PRINCIPAL	100,000
PARTNER/DIRECTOR	300,000
VP	500,000
DISTINGUISHED	1,000,000
THE STANDARD BEARER	1M + DELEGATION

4.2

Estimating Effort

The following table provides the **average estimated time** required to implement different layers or components in The Standard architecture. These times are based on practical experience and are used as a general reference for planning and effort estimation.

Layer / Component	Average Time
Brokers, Base	~1 hour
Foundations, Components	~3 hours
Processing, Orchestration, Coordination, Orchestration Components	~5 hours
Management, Uber Manager	~3 hours
Controllers, Pages, Clients	~1 hour

These estimates represent typical implementation durations for a single service or component, assuming standard complexity and no external dependencies.