Entain pre-Nx analysis

This document outlines initial findings in the organization and state of Entain's codebases from the context of frontend applications. Based on these findings, high-level goals for migrating to the Nx toolchain are outlined in the *Entain Nx migration goals* document.

Workspaces

Entain currently uses a full-stack multi-repository setup with dozens of Git repositories served by self-hosted GitLab servers. .NET solutions and projects dominate the workspace structure of each Git repository while Angular frontend source code is located in a small corner of the workspace, more specifically in one or more *Client* folders despite the fact that product Git repositories like *Mobile Sports* have more frontend source code than backend source code.

Additionally, frontend applications are server-side rendered by ASP.NET Core. This means that backends and frontends are not independently deployable. One of the reasons for server-side rendering is passing a configuration JSON object from the backend to the frontend.

Code sharing

Entain currently uses a full-stack multi-repository setup where both backend and frontend code is shared through a private package registry, Artifactory. Shared code is located in the *Vanilla* GitLab project. Versioned packages are published from the Vanilla GitLab project to Artifactory and consumed from product GitLab projects like *Mobile Sports* where each Git repository declares its version of the shared code packages.

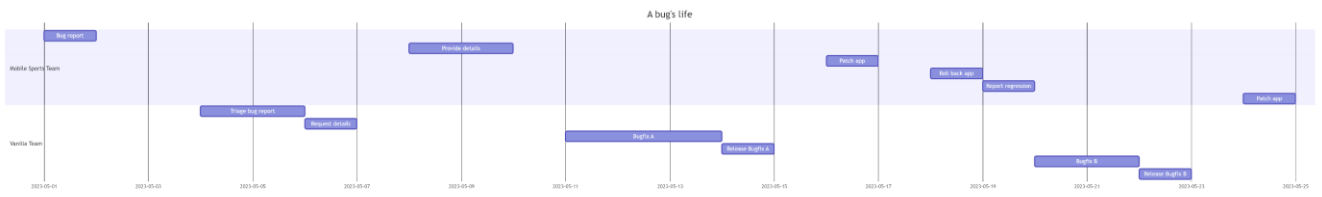
The release cycle for breaking changes to the Vanilla packages is 6 weeks. For backward-compatible changes, bug fixes, and other patches to Vanilla packages, the release cycle is weekly.

Changes to shared code are primarily made by the Vanilla team. An example of the workflow involving two teams is described below.

1. The Mobile Sports application uses Vanilla version A.
2. Mobile Sports Team requests a change in Vanilla.
3. Vanilla Team release Vanilla version B.
4. Mobile Sports Team updates the Mobile Sports application to Vanilla version B.
5. Mobile Sports Team becomes aware of a regression caused by Vanilla version B.
6. Mobile Sports Team rolls back the Mobile Sports application to Vanilla version A.
7. Mobile Sports Team reports the regression caused by Vanilla version B.

8. Vanilla Team release Vanilla version C.
9. Mobile Sports Team updates the Mobile Sports application to Vanilla version C.

Each effort requires careful coordination, yet any amount of time can pass between each effort. Context switching happens, team members change, other issues are worked on, events get in the way, the release cycles must be taken into account, and information is lost. This all increases the chance of errors and increases the lead and cycle times as seen in the following chart.



The risk of errors is reduced if a team has access, knowledge, expertise, and other prerequisites of working in both the Vanilla Git repository and for example the Mobile Sports Git repository. Even then, to make changes to shared code, the following outlines the workflow.

1. Apply the desired change to Vanilla.
2. Make a local release of Vanilla.
3. Make the local release available to the Mobile Sports application, locally.
4. Update the Mobile Sports application to the local Vanilla release, locally.
5. Run and test the Mobile Sports application locally to verify the Vanilla change and guard against regressions, locally.
6. Repeat steps 1-5 until the change is satisfactory.
7. Publish a Vanilla release.
8. Update the Mobile Sports application to the new Vanilla release.

This improved workflow can be performed faster and by a single team with the mandatory prerequisites but is tedious, error-prone, unclear, undocumented, ineffective, and performed in multiple ways by different teams and team members.

Maintenance

Although many third-party dependencies are used in multiple Git repositories, there are differences in third-party dependencies and versions used between multiple Git repositories. For example, some unit tests use Karma while others use Jest. Custom scripts are maintained across many repositories and outdated third-party dependencies are used, for example for bulk task running and packaging Angular libraries.

For the third-party dependencies that are shared, little knowledge or code is shared related to maintaining and migrating dependencies to a major version with breaking changes as well as how to avoid known bugs and performance degradations when using these dependencies.

Each team performs the following tasks that are error-prone and require expert knowledge and care.

- Maintain configurations for third-party toolchains.
- Maintain integrations of third-party dependencies.
- Resolve breaking changes when updating third-party toolchains and dependencies.
- Ensure compatibility between multiple third-party toolchains and dependencies.

The Vanilla packages do not ensure or validate compatibility with older or more recent major and minor versions of third-party dependencies than what the Vanilla Git repository is using at the time a package version is released.

Static analysis

The TypeScript and Angular compiler settings are lax, exposing the systems to a high risk of bugs that could have been avoided at build time and deteriorating the developer experience, in particular not taking advantage of the *Ivy*-specific Angular Language Service features.

For example, the Vanilla Git repository only uses the deprecated `fullTemplateTypeCheck` Angular compiler setting and no strict TypeScript compiler settings. Critically, the `strictNullChecks` TypeScript compiler setting and the `strictNullInputTypes` Angular compiler settings are disabled. With this setup, Angular component templates are barely checked for type errors, and all type annotations implicitly allow `null` and `undefined` types.

These are missed opportunities and result in a large amount of technical debt making refactoring code, merging Git repositories, and restructuring a workspace risky efforts.

The current lint rules allow implicit and explicit use of the infamous `any` type. Fortunately, few explicit uses of the `any` type have been identified and mostly in testing-related files. However, both implicit and explicit `any` types should be avoided to prevent type errors.

Circular dependencies

TypeScript path mappings are used to make import aliases in the current workspaces. Unfortunately, many wildcard TypeScript path mappings are used and they are very broad like `@sports/*`, `@betslip/*`, and `@frontend/vanilla/features/*`. Wildcard TypeScript path mappings circumvent a single entry point (`index.ts/public_api.ts`), resulting in a patchwork of dependencies between individual files rather than between projects through their entry points. Additionally, relative paths are likely not used to refer to the same part of the codebase.

The virtual project boundaries are pierced. This presents a huge issue when migrating to a workspace with many projects as should be the case for an Nx monorepository.

Build time

The Vanilla and Mobile Sports workspaces each have hundreds of thousands of lines of frontend source code between only a few projects. The current build toolchain primarily consists of Gulp, Angular CLI, and Webpack, resulting in slow build and serve times for individual projects.

The following samples were measured on May 15th, 2023 and give an impression of the current state of workspaces.

Build task	Note	Local/CI	Time
`build client:sports`	Mobile Sports	CI	20 minutes 9 seconds 149 milliseconds
`build client:betstation`	Mobile Sports	CI	12 minutes 53 seconds 696 milliseconds
`client-build`	Vanilla	CI	12 minutes 14 seconds 558 milliseconds
`serve mobilesports`	Approximation with hot Angular CLI cache	Local	5 minutes 20 seconds