Using the NGST Monrepo

This monorepo makes it easy to develop and publish node libraries that can be used across different NGST projects.

Changesets

Changesets is tool that makes it easier to do versioning and publishing of packages:

https://github.com/changesets/changesets

Just need to install changesets cli as a dev dependency and initialize for the project:

npm install --dev @changesets/cli

npx changset init

Also need to setup workspaces in package.json to show which packages changesets will include.

To get the versioning set up you need to run:

npx changeset

This will generate changeset versioning file. This doesn't actually change the versions yet, but allows you to review version changes using:

npx changeset status –verbose

To actually change the versions in the package run:

npx changeset version

Now you could actual publish to npm using:

npx changset publish

This would require the creation of .npmrc file with:

//registry.npmjs.org/:_authToken=<<npm token>>

where npm is created using npm account. Make sure token doesn't require MFA.

Also you can change the npm registry for different scope (but probably won't need):

#@aevan04:registry=https://registry.different.org/

When using pnpm I also set some stuff up to make things easier:

```
#We need to have packages in “workspace”, but that automatically links stuff we might not want
link-workspace-packages=false
#Set this to true so that when running scripts root is included
#This not always best though, turns out better to just run root separately
#include-workspace-root=true
```

It turns out the best way to publish packages is to use a github action. changesets has some samples to work from:

https://github.com/changesets/action

This involves creating .github folder in root of monorepo with workflows folder full of .yml file for each workflow. The nice thing is that changesets has an action changesets/action@v1 that not only publishes packages but also creates releases in github.

Import vs Require

Node has traditionally used commonjs modules that worked well and used require statement to import them. ES6 introduced ES modules to JS which allowed front end JS to now be able to import modules natively. ES modules are imported using either static import statement that must be at the top of file, or dynamic import statements that can be used anywhere similar to require but use async/await. It seems that the best way to use ES module exports is to export each function, variable, etc instead of exporting a “default”. I think default was kept around to make it be able to more resemble what require returns but seems unnecessary in most cases.

In order to use ES modules in a file you must make it a ES module by setting type=”module” in package.json or else setting extension equal to .mjs. I think .mjs script allows a mix of commonjs with ESM because setting type=”module” in package will make everything ESM. It is possible to read ESM into commonjs and vice versa. To read ESM into commonjs use use the dynamic import statement. To read commonjs into ESM you can use static or dynamic import.

It is important to note you need Node 12 to experiment with modules and Node 16 to officially used them. I originally wanted to be able to upgrade project to Node 16 but that wasn't so easy for older projects like GP dashboard. It seemed like the older mongo libraries weren't playing so well with Node 16 but newer mongo libraries weren't backward compatible so lots of dashboard code would have to change. I also wanted to use pnpm in production but it is not compatible with older node versions so might have to use npm still in production and only use pnpm locally for development and linking local packages.

Package Management

While doing local development it turns out that pnpm package manager is a bit better than npm when doing monorepo dev and publishing:

https://pnpm.io/

pnpm uses linking so that modules are not duplicated when they are used in multiple dependencies. Newer versions of npm do this by hoisting all the dependency modules to top level but that can lead to "ghost" modules being available. Eg. project depends on dependency A depends on dependency B. Project then has access to dependency B even though it doesn't explicitly have it in it's package.json. If dependency A is removed then project will not have access to dependency B anymore causing possible breaks.

When installing packages in monorepo you can run this so that packages folder package.json gets installed:

```
pnpm -r install
```

Note: pnpm installs the lowest version in range eg. if ^1.0.0 installs 1.0.0 but, npm installs highest version.

In order to link local packages to node_modules use pnpm link. First add package to global link store by going to root folder of package and running:

```
pnpm --global link
```

Note: When running pnpm --global link it adds the linked package to:

<global package root>/package.json

where < global package root> is like:

C:\Users\aevans04\AppData\Local\pnpm\global\5\

global package root node modules can found using:

pnpm root -g

This is of note because I couldn't figure out how to delete link from global modules. It can be manually deleted from global package root node_modules but then when pnpm link –global is run again it installs everything in <global package root>/package.json dependencies.

Now the link to this packages local code can be linked to by name by going to project or package where you want to link to the dependency and run:

pnpm --global link <package name>

This linking feature is used in @usepea-ngst/dev-link tool to simplify linking to local packages.

Another tool that is helpful for removing node_modules in case want to start over with install is call npkill. Either install globally or just run npx npkill and choose which node_modules to kill. Much easier to do than removing them by hand when you have multiple packages in the monorepo each with package.json and node_modules folder.

Requiring local packages when in development

When developing a package, it would be very inconvenient to have to publish the package every time you made a change in order to test it. It is much better to directly consume the local version of the file in the repo before it is published. Then the package can be debugged and finetuned before it gets published to npm. To do this a cli was created to link local versions to node_modules:

@usepa-ngst/dev-link

The cli tool can either be globally installed or else run using:

npx @usepa-ngst/dev-link <cmd>

where cmd = link or register. It will process package in current working directory using config called .dev-link in that directory.

Note: Do not check in .dev-link to git to ensure we don't accidentally link in production.

Here is a sample .dev-link file:

```
{
	register: true,
	devLinks: {
		packageA: {},
		packageB: {disabled:false},
		packageC: {disabled:true},
	}
}
```

cmd = link will run pnpm --global <package name> for all packages names in devLinks object from .dev-link file where disabled not equal to true (disabled defaults to false).

cmd = register will run pnpm --global in current working directory if register is equal to true (register defaults to false).

Another sample .dev-link file:

```
{
	exclude: true,
	devLinks: {
		packageD: {},
		packageE: {disabled:false},
		packageF: {disabled:true},
	}
}
```

cmd = link will run pnpm --global <package name> for ALL dependencies in package.json (that are installed in global link store) except for those names in devLinks object where disabled not equal to true (disabled defaults to false). This format allows every dependency in package.json to be easily linked by setting devLinks equal to {} and exclude = true. Note: If package isn't installed in global link store then we won't attempt to link it and it will be ignored. In that case

might be better to set exclude = false and explicitly list all links so that error is thrown if link is not in store.

Note: after linking to global store the package.json will also be update to the version linking to. This is usually what we want of course because we are testing the use of the new version.

First it is necessary to set up npm script to run dev-link register and link commands in all the monorepo packages like:

```
"link": "npx @usepa-ngst/dev-link link",
"register": "npx @usepa-ngst/dev-link register"
```

In the root of the monorepo we set up npm run to run dev-link to register and link all packages in monorepo:

```
"link:all": "pnpm unlink * && pnpm run register:all && pnpm run link
&& pnpm -r run link",
"register:all": "pnpm -r run register"
```

Note: when linking all we have to unlink all of the current links first. I couldn't figure out how to unlink individual links or individual packages.

Other options explore for requiring local packages

npm link: This would have been a very simple solution that could have been used in the dev-link tool to link local packages to node_modules. It turns out that some funky things occurred some times when linking. For example, we have package A at version 2 and package B which depends on package A at version 1. When linking node_modules folder in package B with version 1 package A is removed and version 2 package gets consumed by package B which isn't necessarily desired. in dependents that are still needed because dependent itself depends on version of module different than main . Also when using npm 7+ if workspaces are being set (like needed to use changsets tool for releasing/publishing), then linking is done by default which limits some flexibility. For dev-link tool we ended up using pnpm and pnpm link instead.

relative-deps: npm module which worked ok but you have to run npm install every time there is a change or do a watch command. In addition it is necessary to debug the version of the file in node_modules somewhere instead of actual location of local version in development. Also I couldn't figure out how to have a different version of module that is consumed by another module to work. For example we have package A and package B where package B depends on package A. The project X package.json uses version 1 of package B which used version 1 of package A. Locally package b has changes that will be version 2 which uses version 2 of package a. If we set project X relative-deps to use package B locally, then package B will use version1 of package A instead of version 2 like as expected (because local version 2 package B package.json calls for version 2 package A). I don't see how this really makes sense.

@usepa-ngst/dev-require: This is custom tool that would allow local version of modules to be required based on a global config file. The disadvantage of dev-require is that it embeds development tooling into production code eg. function from the module must be used to require that decides whether to do local require or remote require. It's not a very complex process in production ie. Just checks for env var of path to config and if not exists then returns null and wrapper then just does normal require. But still asking a bit much to force every module developed by us to be required using this. Also, there is no way to use the tool if using a static import such as used in front end dev. We could do dynamic import but that isn't used all the time in front end (tree shaking, etc).