# The big Tiny monorepo

Welcome to the TinyMCE monorepo. For TinyMCE itself look to the [modules/tinymce](modules/tinymce) folder.

## Some background

As TinyMCE transitioned to a modern codebase through 2017 and 2018 many external dependencies were added from previously closed-source projects. This became unwieldy to develop, so in June 2019 the decision was made to bring those projects together in a monorepo.

This repo is built with Yarn workspaces and uses publish tooling support from Lerna. NPM is not supported and attempts to use it will fail.

An important feature of this monorepo is the use of TypeScript 3.0 features "project references" and "build mode":

https://www.typescriptlang.org/docs/handbook/project-references.html

These are still quite new and some of the tooling is still catching up (e.g. webpack needs `tsc -b -w` running in the background).

### A quick note about `modules`

Most monorepos use a `packages` folder to hold the included projects, but we have chosen `modules` instead. There are few reasons for this:

\* These projects are not extra packages of TinyMCE, they are self contained libraries used as module dependencies for the editor.

\* Enough examples exist of projects not using `packages` that we don't think it will be difficult to understand

\* It tab completes better (`packages` overlaps with `package.json`)

## Getting started

Install [Node.js](https://nodejs.org/en/) on your system.

Clone this repository on your system

```

$ git clone https://github.com/tinymce/tinymce.git

```

### Install dependencies

\* `yarn`

### Build TinyMCE

\* `yarn build`

This will produce an editor build in `modules/tinymce/js`, with distribution zips in `modules/tinymce/dist/tinymce\_[number].zip`.

## Developing TinyMCE

\* `yarn dev`

This performs compilation steps which webpack requires but are usually once-off. It also runs `tsc` to make later commands faster (`tsc -b` enforces incremental compilation).

### Builds

To build the editor in development, use `yarn tinymce-grunt`. This will output to the `modules/tinymce/js` folder (`build` is effectively `dev` followed by `tinymce-grunt`).

Task names can be included, for example `yarn tinymce-grunt bundle` will execute the bundle task. More information on TinyMCE grunt tasks is available in the [TinyMCE readme](modules/tinymce/readme.md).

## Development scripts

There are many top-level helper scripts for TinyMCE and Oxide (the default skin) defined in `package.json`.

### TinyMCE

`yarn start`

This boots the TinyMCE webpack dev server at http://localhost:3000 and also starts a TypeScript watch process. With this running changes to \_any\_ `.ts` source file in the monorepo (excluding tests) should be reflected in WebPack within a few seconds.

If WebPack later adds supports for TypeScript build mode the background TypeScript watch process won't be necessary which will speed up round trip time.

`yarn watch`

runs `tsc -b -w` for those times when you don't need to iterate in the browser.

`yarn tsc`

an alias to `tsc -b` just in case you forget

`yarn eslint`

runs `eslint` in \_all\_ projects, with a rule set that is far more strict than most projects were previously subject to. This is a good source of things to improve when bored.

`yarn tinymce-grunt`

easy access to the TinyMCE grunt commands from the root folder.

### Oxide

```

yarn oxide-build

yarn oxide-icons-build

```

These commands build the skin and icons but should not normally be required outside of other development scripts.

`yarn oxide-start` will set up a watch and rebuild process for creating custom skins.

### Focussed development

If you are working in a single module and don't want to deal with the overheads of whole-monorepo compilation, you can run `yarn --focus` from that module's folder to install the latest published versions of monorepo projects in a local `node\_modules`. For more information see this yarn blog post:

https://yarnpkg.com/blog/2018/05/18/focused-workspaces/

## Testing

Testing relies on `yarn lerna changed` to determine which modules need testing, and a grunt script then separates them into two groups depending on whether they need GUI browser testing or can be tested with phantomjs.

### CI test scripts

```

yarn browser-test

yarn phantomjs-test

```

### Dev test scripts

```

yarn browser-test-manual

yarn phantomjs-test-manual

```

Development testing will be adjusted in future so that there's only one manual entry point for ease of development. They are still separate for now because there are two projects that use bedrock route configurations; a route config combination process is required to run them at the same time.

### Running a subset of tests

To run a single test: `yarn bedrock -f file`

To run a whole folder of tests: `yarn bedrock -d folder`

## CI builds

CI builds rely on the `ci` and `ci-all` package.json scripts, in addition to the above testing scripts, to run type checking and linting before executing the full test suite. A `ci-local` package.json script has also been added for convenience to simulate this process in development and then run tests.

## Version management

It is important that you never hand-edit a `package.json` file, particularly the `dependency` and `version` fields. And we do mean \_never\_. Doing so may break the automated scripts. See the `publishing` section below for more information.

### dev dependencies

All dev dependencies are in the project root, so to add or upgrade a specific dependency:

`yarn add -D -W <package>`

### normal dependencies

To add a dependency inside a monorepo package:

`yarn workspace <fullname> add <othername>`

This works whether adding an external dependency or a dependency on another monorepo package.

Note that both names must be the entire scoped `name` of the package, not the folder, for example

`yarn workspace @tinymce/oxide add @tinymce/oxide-icons-default`

## Publishing process

We have a CI process set up to publish all changed libraries as patch releases twice a day. This simply describes that process, is is not intended that it be performed manually.

### Side note: major and minor version bumps

> In the future these will likely be automated via the lerna-supported [conventional commit](https://conventionalcommits.org) specification, for now this is done manually.

In theory minor bumps can be done in the package.json by hand but for consistency we recommend using the lerna tooling for both. `yarn lerna version` is the only way to do this without breaking links between packages.

For each changed package choose major, minor or patch as appropriate depending on the flow-on effects of this version change. Afterwards, you \_must\_ run the git commands below to push the version and related tags correctly.

Changes to minor and major versions are such a rare occurence that this manual process will suffice until we switch to conventional commits. Unfortunately manual version changes mean the next automated build will run all repository tests, since nothing has changed, but that's probably a good idea for serious version changes anyway.

### CI publish process

`yarn lerna publish patch`

This is configured via `lerna.json` to exclude TinyMCE. We will not be using lerna to publish TinyMCE itself as it places far greater importance on the version number than library projects.

`yarn lerna publish from-package`

This is run after `publish patch` to catch cases where `lerna version` was run manually for a non-patch bump. It compares the source version to the NPM registry and publishes anything that doesn't match.

Lerna's publish process is configured to not `git push` in case of failure, so after a successful publish this must be done manually:

```

git push

git push --tags

```

## Adding globals

TinyMCE puts a `tinymce` object in the global namespace, and has a tree of objects down from there.

If you wish to add to this, you need to do the following:

1. Ensure your module is located under `modules/tinymce/src/core/main/ts/api`

2. Expose any types under `interface TinyMCE` in `modules/tinymce/src/core/main/ts/api/Tinymce.ts`

3. Expose any objects/functions as part of `const publicApi` in `modules/tinymce/src/core/main/ts/api/Tinymce.ts`

4. Ensure your values are exposed in `modules/tinymce/src/core/main/json/globals.json`

Ensure the paths all match. e.g. if you're exposing `modules/tinymce/src/core/main/ts/api/dom/StyleSheetLoader.ts`

1. The module is under `.../api/dom/StyleSheetLoader.ts`

2. The type is `TinyMCE.dom.StyleSheetLoader`

3. The constructor function is `publicApi.dom.StyleSheetLoader`

4. globals.json contains `"tinymce.core.api.dom.StyleSheetLoader"`