[WIP] [RFC] Lexical Builder API

See also:

- https://lexical-builder.pages.dev/
- https://lexical-builder.pages.dev/docs/intro
- https://lexical-builder.pages.dev/docs/migration
- https://lexical-builder.pages.dev/docs/faq
- https://github.com/etrepum/lexical-builder

Use Case

Plug-ins are not the solution that users want. Lexical Builder attempts to solve several problems with code reuse and scaffolding in Lexical.

- Plug-ins can only support code that can be asynchronously added to a LexicalEditor after construction, but most use cases also require configuration that must be specified synchronously during construction (such as any new nodes or replacements that are being registered)
- Plug-ins are all React dependent. There is no standard to package code that can support both React and non-React use cases from the same package, whether or not the package has React dependencies
- Nodes and Plug-ins can not declare their dependencies or conflicts, for example @lexical/yjs conflicts with @lexical/history. Markdown pretty much requires @lexical/rich-text. There is the workaround where Plug-ins can throw runtime exceptions if their Node requirements are not met, but there is no reasonable way for a node to detect a missing plug-in!
- Configuration is verbose, several properties of the editor or LexicalComposer have "obvious" defaults that must be specified
- Configuration is not composable, it is not easy to copy and paste an editor together because at minimum you must edit two places (config and plug-ins) and the config section requires manual merging of the nodes and theme properties.
- Using a context and hooks to set up the editor with bespoke plug-ins is cumbersome, most people are not used to authoring top-level components for each piece of code that has to interact with the editor.

Requirements

- Be the entrypoint to creating and configuring the editor
    - Synchronous configuration of the editor

        - Synchronous registration with the editor after creation
            - These synchronous calls can dispatch asynchronous work
            - Synchronous cleanup
- Composable
    - Plug-ins depend on and extend other plug-ins
    - Configuration is specified in pieces, with sensible merge policies
    - Sensible and extensible defaults
- Framework Independent
    - Plug-ins that don't use React shouldn't need React!
    - Should still provide a best-in-class React experience, with no reason (other than legacy) to continue with the existing plug-in convention
- Not dependent on bundlers and is not a package manager
    - This mostly means that a Plan must be imported by the user, not referenced by name (as is common in node environment configurations, like eslint or webpack, to allow them to be written in JSON)
    - Shouldn't know anything about versions, parsing version requirements, etc. That should be done by npm/pnpm/yarn/etc. from metadata in package.json.

Terminology

I'm not married to any of these terms. They seemed like reasonable choices given the real-world metaphor (architects/engineers make plans, builders execute those plans) and the Builder pattern in software parlance. I think plug-in would've been the obvious choice, but that was already used for the existing convention.

Plan - What plug-ins should've been

Plan is what all plug-in developers would author. It's metadata, partial configuration for the editor, plus a register method to add behavior to the editor after creation.

Builder - Creates an editor from Plans

This is the (mostly opaque/internal) API that combines Plans to create and manage the Editor.

Core Changes

This is designed to be an entirely additive solution, but the following changes would be very useful for adoption

Add Plan types & helpers to `lexical` module

This has almost no runtime or bundle size effects (<1kb, see @etrepum/lexical-builder-core index.js). The helper functions are identity functions only for type inference and will only add a few bytes to the runtime. They are not strictly necessary but would be very helpful for users.

Types

- LexicalPlan - This is the type that matters, that all plug-in use cases should implement and export. The associated utility/helper types below would also make sense to export and have no runtime representation.
    - AnyLexicalPlan
    - AnyLexicalPlanArgument
    - PlanConfigBase
    - NormalizedLexicalPlanArgument
    - RegisterState
    - LexicalPlanArgument
    - LexicalPeerConfig
    - LexicalPlanConfig
    - LexicalPlanName
    - LexicalPlanOutput
    - LexicalPlanDependency
    - RootPlan
    - RootPlanArgument
    - RegisterCleanup
    - NormalizedPeerDependency
    - EditorHandle
    - InitialEditorStateType

Helper functions (for inference only)

- definePlan - provides inference assistance in creating a LexicalPlan
- configPlan - provides inference assistance in creating a NormalizedLexicalPlanArgument for specifying configuration of a Plan dependency
- defineRootPlan - convenience for creating an app's root plan which has a default name and no configuration
- provideOutput - convenience for attaching the output property to the return value of a Plan's register (RegisterCleanup) which aids in inference of the Output of a Plan
- safeCast - an identity function that's like the satisfies operator in TypeScript, but does not narrow. Useful for defining configuration.
- declarePeerDependency - allows a peer dependency tuple to be constructed with a type argument

New `@lexical/builder` package

This implements the LexicalBuilder class which can take a set of Plans and create/manage an Editor. We may choose to only expose this only as a factory function (e.g. createEditorFromPlan) and keep the class as an implementation detail.

```javascript
/**
 * @param {AnyLexicalPlan} plan
 * @param {AnyLexicalPlan[]} plans
 * @returns {EditorHandle}
 */
export function buildEditorFromPlans(plan, ...plans) {
  // implementation details using the LexicalBuilder class
  return { editor, dispose };
}
```

Export Plans from each package

Packages that implement plug-ins and nodes should export a Plan to configure them accordingly, e.g. @lexical/rich-text, @lexical/plain-text, @lexical/code, and so on. The runtime overhead of exposing this object should be negligible as it's only a little bit of metadata and generally does not require adding any new code at all (the register property uses the same calling convention as existing functionality. For example:

```javascript
export const RichTextPlan = definePlan({
  config: {},
  conflictsWith: ["@lexical/plain-text"],
  name: "@lexical/rich-text",
  nodes: [HeadingNode, QuoteNode],
  register: registerRichText,
});
```

New @lexical/react/LexicalPlanComposer module & component

LexicalPlanComposer is a replacement for LexicalComposer that uses a plan instead of an initialConfig. Alternatively, LexicalComposer could be modified to take an initialConfig or a plan, but I think it would be cleaner to have a separate component and not have to maintain that sort of branching for legacy code.

Add Plan/Builder awareness to @lexical/devtools

Since everything LexicalBuilder does during construction is synchronous, it should be straightforward to capture metadata when the editor is constructed. We could know exactly which Plan registered a transform, command, or listener by using module-level state (or equivalent) on enter/exit of a Plan's register method.

WIP Prototype

Monorepo with prototype
https://github.com/etrepum/lexical-builder
Docs page [WIP]
https://lexical-builder.pages.dev/docs/intro

Implementation notes:

- All of the plans in @lexical/builder should be colocated with the modules that implement their functionality (e.g. RichTextPlan should be exported by @lexical/rich-text). I centralized them all to make it very clear that this is an additive change and to prevent rebase issues at this early stage. The overhead of exporting these Plans should be negligible.
- If this were accepted it would also make sense to move the Plan related types into the lexical package so that any package can export a Plan without any sort of dependency on @lexical/builder (maybe also add the definePlan and configPlan helper functions to help with type inference too)

Usage:

Vanilla JS

```javascript
const { editor, dispose } = buildEditorFromPlans({
  dependencies: [
    RichTextPlan,
    configPlan(EmojiPlan, { emojiBaseUrl: "/assets/emoji" }),
  ],
});
editor.setRootElement(document.getElementById("lexical-editor"));
```

React:

```JavaScript
const plan = defineRootPlan({
  dependencies: [
    RichTextPlan,
    // By default a ContentEditable renders first in the composer
    configPlan(ReactPlan, { contentEditable: null }),
    configPlan(EmojiPlan, { emojiBaseUrl: "/assets/emoji" }),
  ],
});
function Editor() {
  return (
    <LexicalPlanComposer plan={plan}>
      <h1>The Editor</h1>
      <ContentEditable className="editor-input" />
    </LexicalPlanComposer>
  );
}
```

What is a Plan?

- What users expected plug-ins to be. Adding functionality to the editor is as simple as importing it, and then specifying the plan in the configuration
- A convention and schema that plug-in developers want to support to reduce documentation and support burdens
- A data type including the arguments to createEditor, metadata (name, dependencies, conflicts), Plan-specific configuration with defaults that can be augmented, and optional functions to add behavior after the editor is created
- May depend on some specific framework using a Plan dependency, e.g. ReactPlan

Plan can:

- Be used at most once when building a LexicalEditor
- Have configuration
- Have dependencies on other Plans by concrete reference (like commands), optionally augment their Config, and access their finalized config (using the concrete reference as a key) during register

- Can have (always optional) peerDependencies on other plans by name, although you could throw a runtime exception during register if they are not present (used by ReactPlan to detect whether LexicalPlanComposer or ReactPluginHost are present)
- Register Nodes, replacements, transforms, etc. and any other initial configuration in general by manipulating the initial configuration
- Update the initial state before reconciliation
- Configure the editor
- Optionally do registration/cleanup with the editor
- Use an AbortSignal to help manage finalization of asynchronous tasks
- Allow a Vanilla JS user to host React plug-ins (it's all just portals anyway)
- (TODO - only React or no framework right now) Optionally have framework-specific integrations
- (TODO) Allow dev tools to capture metadata about the Plan (e.g. commands that are registered while the Plan is executing could be tagged with the plan's name)

Plan can not:

- Be loaded asynchronously, but their registration function can do async work, so an async import could be done at that time

What is a Builder?

- A framework agnostic tool to manage creating the LexicalEditor from PlanArguments, which may either be a Plan or a [Plan, Config] pair that is used to augment the configuration for that plan

Builder does:

- Synchronously build the LexicalEditor's initial configuration from an Array of plans
- Merge Plan configurations after flattening dependencies w/ a topological sort (e.g. merge([...dependencyConfigs, explicitConfig]))
- Provide callbacks for plans to do registration (and clean-up) on the LexicalEditor after it is constructed
- Provide a decorate protocol so that any Plan can render Vanilla JS (compatible with any framework) or Framework-Specific UX (e.g. an alternative to the current React Plug-in Convention)
- Demonstrate maximal automatic and manual lazy-loading strategies for a Plan (e.g. dynamically import UX code only if the Editor becomes editable)

- Exports TypeScript types that plug-in developers can use to author LexicalBuilder Plans with only a devDependencies entry and no runtime import

Builder does not:

- Offer any sort of module resolution, e.g. you can not specify a plan by the name of a module, it must be a module that was already imported
- Allow for plans to be asynchronously loaded. A Plan can have behavior that does asynchronous work (possibly asynchronously loading its supporting code) after the LexicalEditor is constructed, but the Plan itself is already completely reified before the Builder constructs the LexicalEditor.
- Allow modifications of any sort to the Builder's inputs after the LexicalEditor is constructed. To make such changes, the LexicalEditor must be re-created, because Nodes can not be unregistered or replaced after construction.
- Add any cross-framework compatibility. Vanilla JS Plan Decorators can work in any framework, but React decorators only work in React projects.