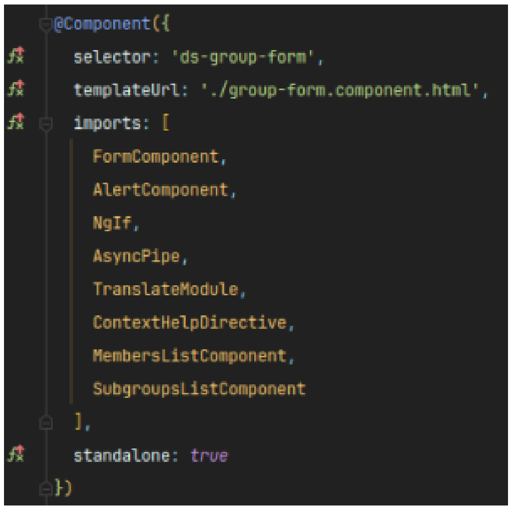
Migration to Standalone Components

The goal of this document is to describe the migration of DSpace angular application to standalone components approach which is the default behavior by Angular 17 release.

It describes steps done for the migration and all the problems encountered with the solution we have adopted.

Migrating Components, Directives and Pipes to Standalone via the official Angular Migration Script

We used the official migration script to start migrating components, directives and pipes to a standalone architecture. For each object, this script changes its decorator in order to let the object itself import its requirements directly.



```
@Component({
  selector: 'ds-group-form',
  templateUrl: './group-form.component.html',
  imports: [
    FormComponent,
    AlertComponent,
    NgIf,
    AsyncPipe,
    TranslateModule,
    ContextHelpDirective,
    MembersListComponent,
    SubgroupsListComponent
  ],
  standalone: true
})
```

It also changes the Module that declared such object, moving it from the `declarations` array into the `imports` array.

The script managed to correctly migrate a lot of components, but some of them required more attention, such as components in theme folders and components with custom decorators, that were not immediately compatible with a standalone approach.

Already existing themes (*dspace* and *custom*) have already been migrated to standalone.

Additional themes, however, can be migrated with the following console command even at a latter time:

Migration script

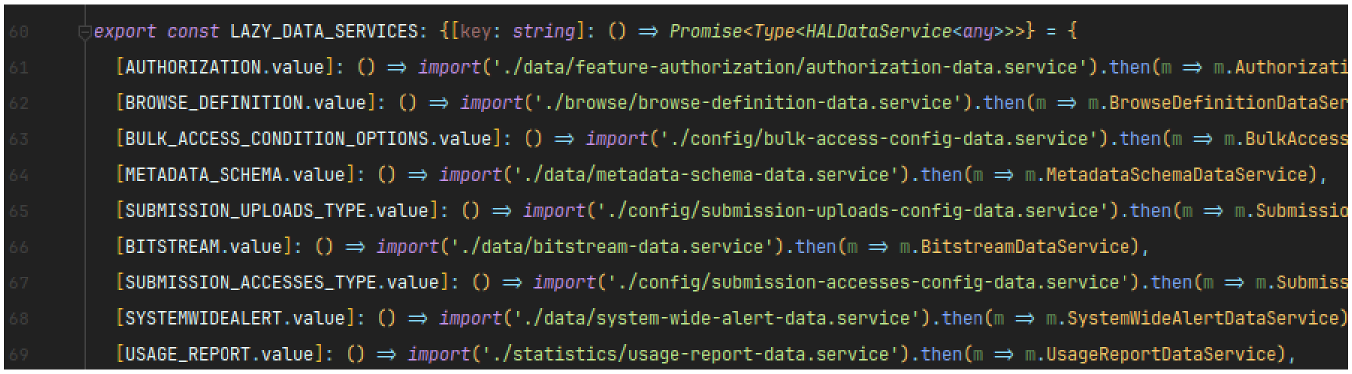
ng generate @angular/core:standalone --path src/themes/<theme-name>

Unfortunately, this script is not going to resolve any import of standalone component in the `src/app` folder, so those imports should be resolved manually.

Handling CoreModule providers

`CoreModule` providers were eagerly loaded in bootstrap time, becoming root providers and breaking tree-shaking, becoming an issue for bundle size.

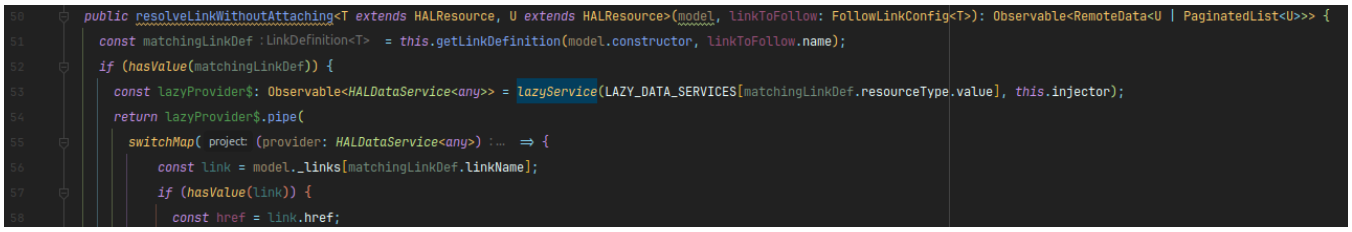
In order to solve this issue, we introduced a map (`LAZY_DATA_SERVICES`) that uses the `ResourceType` as key and dynamically imports the service.



```
export const LAZY_DATA_SERVICES: {[key: string]: () => Promise<Type<HALDataService<any>>>} = {
  [AUTHORIZATION.value]: () => import('./data/feature-authorization/authorization-data.service').then(m => m.Authorizati
  [BROWSE_DEFINITION.value]: () => import('./browse/browse-definition-data.service').then(m => m.BrowseDefinitionDataSer
  [BULK_ACCESS_CONDITION_OPTIONS.value]: () => import('./config/bulk-access-config-data.service').then(m => m.BulkAccess
  [METADATA_SCHEMA.value]: () => import('./data/metadata-schema-data.service').then(m => m.MetadataSchemaDataService),
  [SUBMISSION_UPLOADS_TYPE.value]: () => import('./config/submission-uploads-config-data.service').then(m => m.Submissio
  [BITSTREAM.value]: () => import('./data/bitstream-data.service').then(m => m.BitstreamDataService),
  [SUBMISSION_ACCESSES_TYPE.value]: () => import('./config/submission-accesses-config-data.service').then(m => m.Submiss
  [SYSTEMWIDEALERT.value]: () => import('./data/system-wide-alert-data.service').then(m => m.SystemWideAlertDataService)
  [USAGE_REPORT.value]: () => import('./statistics/usage-report-data.service').then(m => m.UsageReportDataService),
```

Since the services in the core module were eagerly loaded, they became part of root providers, so we also refactored all the services to become explicitly `{ providedIn: 'root'}`.

These services are used in the `LinkService` which before used the service class directly, because it was eagerly loaded. Now we use a helper function called `lazyService` which uses the dynamic import to get the service. `LazyService` will load the service and return an observable with its value, which we use to get the value in the `resolveLinkWithoutAttaching()` method.



```
  public resolveLinkWithoutAttaching<T extends HALResource, U extends HALResource>(model, linkToFollow: FollowLinkConfig<T>): Observable<RemoteData<U | PaginatedList<U>>> {
    const matchingLinkDef : LinkDefinition<T>  = this.getLinkDefinition(model.constructor, linkToFollow.name);
    if (hasValue(matchingLinkDef)) {
      const lazyProvider$: Observable<HALDataService<any>> = lazyService(LAZY_DATA_SERVICES[matchingLinkDef.resourceType.value], this.injector);
      return lazyProvider$.pipe(
        switchMap( project: (provider: HALDataService<any>) :...  => {
          const link = model._links[matchingLinkDef.linkName];
          if (hasValue(link)) {
            const href = link.href;
```

This way we have lazy loaded services that are loaded when they are needed. The `@dataService` decorator is not needed anymore because we don’t have all the services eagerly loaded anymore.

This will also help later on, when we will move to a plugin architecture: since services are provided in root, it would be easy to move them outside the app code to library code.

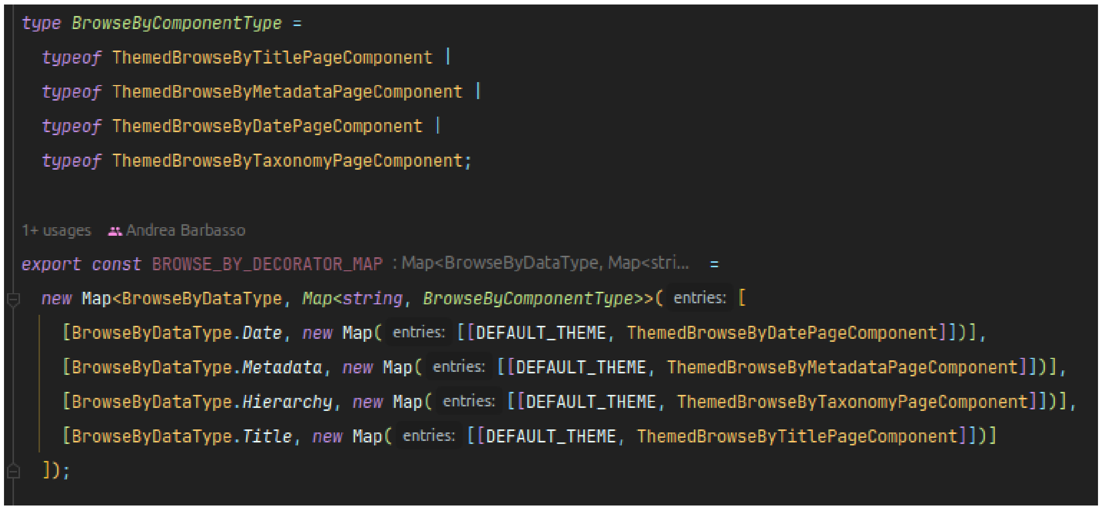
Handling Custom Decorators for Maps of Injectable Components

In DSpace we used the decorators pattern to create maps of components, which are retrieved and injected by a given key (e.g. *`@RendersSectionForMenu`* or *`@listableObjectComponent`*).

Such custom decorators were functioning thanks to a workaround: a module method (usually called withEntryComponents()) was serving decorated components the same way that entryComponents worked before Ivy.

This approach is not compatible anymore since we are removing modules thanks to the Standalone architecture.

To fix this problem we replaced the maps - that were previously built **dynamically** - with **static** maps.



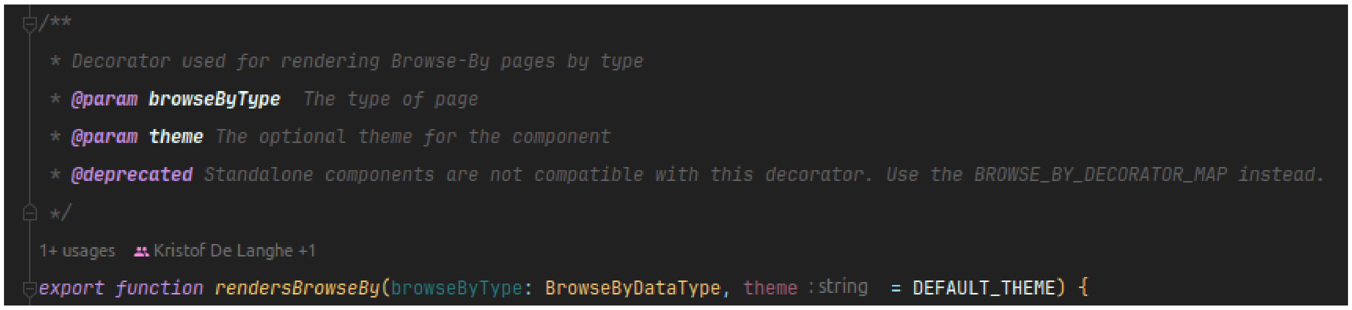
```
type BrowseByComponentType =
  typeof ThemedBrowseByTitlePageComponent |
  typeof ThemedBrowseByMetadataPageComponent |
  typeof ThemedBrowseByDatePageComponent |
  typeof ThemedBrowseByTaxonomyPageComponent;


1+ usages   Andrea Barbasso
export const BROWSE_BY_DECORATOR_MAP : Map<BrowseByDataType, Map<stri...  =
  new Map<BrowseByDataType, Map<string, BrowseByComponentType>>( entries: [
    [BrowseByDataType.Date, new Map( entries: [[DEFAULT_THEME, ThemedBrowseByDatePageComponent]])],
    [BrowseByDataType.Metadata, new Map( entries: [[DEFAULT_THEME, ThemedBrowseByMetadataPageComponent]])],
    [BrowseByDataType.Hierarchy, new Map( entries: [[DEFAULT_THEME, ThemedBrowseByTaxonomyPageComponent]])],
    [BrowseByDataType.Title, new Map( entries: [[DEFAULT_THEME, ThemedBrowseByTitlePageComponent]])]
  ]);
```

This has been done for *almost* every decorator. The exception lies in the @ListableObjectComponent decorator. This decorator is creating a very large map of Components, both in depth (the decorator is called 119 times) and width (the map uses 4 keys to retrieve the correct component), and these components are scattered throughout the application.

We temporarily moved all these components into a single module (called ListableModule) that uses the old approach (with the withEntryComponents() method), this means that all such components **are not standalone** for now. The ListableModule is imported in the AppModule, even if this is not strictly necessary. We are aiming for a better and definitive solution.

In order to maintain some backwards compatibility with custom decorated components in custom-made themes, we did not remove the decorator function, instead we marked it as deprecated.



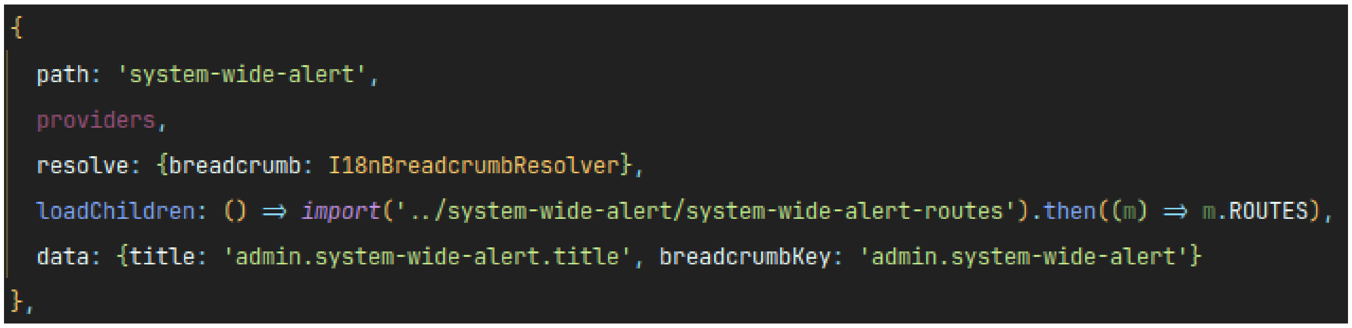
```
/**
 * Decorator used for rendering Browse-By pages by type
 * @param browseByType  The type of page
 * @param theme The optional theme for the component
 * @deprecated Standalone components are not compatible with this decorator. Use the BROWSE_BY_DECORATOR_MAP instead.
 */
1+ usages  Kristof De Langhe +1
export function rendersBrowseBy(browseByType: BrowseByDataType, theme :string  = DEFAULT_THEME) {
```

Migrating Routing and Lazy Loading

Since modules are now (mostly) gone, Routing and Lazy Loading had to shift to a non-module-based approach.

Following the official docs, routes are now stored in plain typescript files. These files are lazy loaded (thanks to the `import()` clause) and keep the same tree hierarchy as before.

Providers are now determined in those routing files, and not in the module that imported the routing module.



```
{
  path: 'system-wide-alert',
  providers,
  resolve: {breadcrumb: I18nBreadcrumbResolver},
  loadChildren: () => import('../system-wide-alert/system-wide-alert-routes').then((m) => m.ROUTES),
  data: {title: 'admin.system-wide-alert.title', breadcrumbKey: 'admin.system-wide-alert'}
},
```

Deleting unnecessary modules

According to the Angular Standalone Migration Guide, a module can be safely deleted if:

- Has no `declarations`.
- Has no `providers`.
- Has no `bootstrap` components.
- Has no `imports` that reference a `ModuleWithProviders` symbol or a module that can't be removed.
- Has no class members. Empty constructors are ignored.

After using the migration script to delete such modules, we checked for other ones that could be safely deleted (e.g. any module that had all the properties in the bullet list, but with an `withEntryComponents()` method that was not called anymore).

Migrating Unit Tests

Before, in unit tests, we had the component in the declarations array and the NO_ERRORS_SCHEMA applied, meaning that every other tag inside the component that was not part of the declaration would not be initialized and there would be no error. But, with a standalone component having an imports array itself, this means that every component and directive that is inside of it would be initialized in the tests.

This breaks our testing because of missing providers needed for the other imported components. So, the solution for this cases would be to override the component and remove those imports from the component via the `TestBed.overrideComponent()` method. This way, we explicit what we are testing, and what we are removing from the tests.

This would also catch cases where we introduce a new component inside the component and its providers would be missing, since we now have to explicitly tell the test that we want to remove it and not test it.