Plone Angular SDK Documentation

Release 1.3.1

ebrehault sunew tdesvenain fulv

Dec 29, 2019

Contents:

CHAPTER 1

Basic usage

In `src/app.module.ts`, load the module and set the backend URL:

```
import { RESTAPIModule } from '@plone/restapi-angular';

...

@NgModule({
  ...
  imports: [
    ...
    RESTAPIModule,
  ],
  providers: [
    {
      provide: 'CONFIGURATION', useValue: {
        BACKEND_URL: 'http://localhost:8080/Plone',
        CLIENT_TIMEOUT: 5000,
      }
    },
  ],
  ...
```

And you have to set up the Plone views for traversal in `src/app.component.ts`:

```
import { Component } from '@angular/core';
import { Traverser } from 'angular-traversal';
import { PloneViews } from '@plone/restapi-angular';

@Component({
  ...
})
export class AppComponent {

  constructor(
```

(continues on next page)

(continued from previous page)

```
    private views:PloneViews,
    private traverser: Traverser,
  ) {
    this.views.initialize();
  }
}
```

Now you can use the Plone components in your templates, for example in `src/app.component.html`:

```
<plone-navigation></plone-navigation>
<traverser-outlet></traverser-outlet>
```

1.1 Customize components

WORK IN PROGRESS (we will propose a better customization story)

If you want to change the component's rendering, you can provide your own template by extending the original Plone component.

In this example we will override the template used by the `Navigation` component in order to use Material Design styling. The navigation menu is actually provided by two separate components, `Navigation` and `NavigationLevel`. The actual customization will happen in the latter, but we also need a custom `Navigation` in order to refer to our custom `NavigationLevel`.

Let's use Angular CLI to create our custom components:

```
ng generate component custom-navigation
ng generate component custom-navigation-level
```

This will create two new folders: `./src/app/custom-navigation` and `./src/app/custom-navigation-level`.

We will start with `./src/app/custom-navigation/custom-navigation.component.ts`:

```
import { Component } from '@angular/core';
import { Navigation } from '@plone/restapi-angular';

@Component({
  selector: 'custom-navigation',
  template: `<custom-navigation-level [links]="links"></custom-navigation-level>`
})
export class CustomNavigationComponent extends Navigation {}
```

- We add an `import` for the default `Navigation`.
- Rename the `selector`.
- Put the `template` inline (using backticks) instead of using an external `templateUrl`, since the template is very short.
- Replace `implements` with `extends` and extend from `Navigation`.
- Delete the `constructor` and `ngOnInit`.

Let us now turn to `./src/app/custom-navigation-level/custom-navigation-level.component.ts`:

```
import { Component } from '@angular/core';
import { NavigationLevel } from '@plone/restapi-angular';

@Component({
  selector: 'custom-navigation-level',
  templateUrl: './custom-navigation-level.component.html',
})
export class CustomNavigationLevelComponent extends NavigationLevel {
}
```

This is very similar to the custom navigation component, except that we point to a `templateUrl`, because in this case the template (`./src/app/custom-navigation-level/custom-navigation-level.component.html`) is a little more involved.

```
<md-nav-list>
  <md-list-item *ngFor="let link of links">
    <a md-line [traverseTo]="link.properties['@id']">
      {{ link.properties.title }}
    </a>
    <custom-navigation-level
      [links]="link.children"
      *ngIf="link.children"></custom-navigation-level>
  </md-list-item>
</md-nav-list>
```

Note that we are using the same structure as in the default `NavigationLevel` template, only using markup from Angular Material. Before we can call this done, we also need to install the dependencies (see the setup here):

```
npm install --save @angular/material
npm install --save @angular/animations
```

Finally, edit your app module (`./src/app/app.module.ts`):

```
...
import { CustomNavigation } from './src/custom-navigation/custom-navigation.component
↪';
...
@NgModule({
  declarations: [
    ...
    CustomNavigation,
  ],
...
```

And load the CSS for Angular Material in the "main template" `./src/index.html`:

```
<link href="../node_modules/@angular/material/prebuilt-themes/indigo-pink.css" rel=
↪"stylesheet">
```

Now you can use your `<custom-navigation>` component in templates, for example by using it instead of `<plone-navigation>`.

1.2 Customize views

Customizing a view is quite similar to component customization, the only extra step is to declare it for traversal. In this example we will modify the default view so that it will display the context's summary under its title.

Let's use Angular CLI to create our custom view:

```
ng generate component custom-view
```

This will create a new folder: `./src/app/custom-view`.

Edit `./src/app/custom-view/custom-view.component.ts`:

```
import { Component } from '@angular/core';
import { ViewView } from '@plone/restapi-angular';

@Component({
  selector: 'custom-view',
  template: `<h2>{{ context.title }}</h2><h4>{{ context.description }}</h4>`,
})
export class CustomViewView extends ViewView {}
```

You can see in the inline template that we added the `context.description`.

In `app.module.ts`, you will need to put our custom view in `declarations` and in `entryComponents`:

```
import { CustomViewView } from './custom-view/custom-view.component';
@NgModule({
  declarations: [
    AppComponent,
    CustomViewView,
  ],
  entryComponents: [
    CustomViewView,
  ],
...
```

And in `app.component.ts`, you will need to register it for traversal this way:

```
...
import { CustomViewView } from './custom-view/custom-view.component';

...
export class AppComponent {

  constructor(
    private views:PloneViews,
    private traverser: Traverser,
  ) {
    this.views.initialize();
    this.traverser.addView('view', '*', CustomViewView);
  }
}
```

Now your custom view will replace the original one.

CHAPTER 2

Installation

2.1 NodeJs

We will need NodeJS 6.10+.

We recommend using NVM to install NodeJS.

Install nvm on our system using the instructions and provided script at:

https://github.com/creationix/nvm#install-script

Using `nvm` we will look up the latest LTS version of node.js and install it:

```
$ nvm ls-remote --lts
$ nvm install 6.10
```

Then, each time we want to use this version of NodeJS, we just type:

```
$ nvm use 6.10
```

2.2 Angular CLI

Angular CLI is the commande line interface provided by Angular.

Note: We need CLI 1.0.0+

We install it with NPM:

```
$ npm install -g @angular/cli
```

The `-g` option install the CLI globally, meaning it is available wherever we activate our NVM.

`ng` will be available from the command line and we are ready to bootstrap an application.

2.3 Backend

We need a running instance providing the Plone REST API.

TODO: provide deployment options here.

2.4 Setup a new Angular project

Enter the command:

```
$ ng new myapp
```

It will setup a standard Angular project structure and install all the default dependencies.

The app can be served locally with:

```
$ ng serve
```

The result can be seen on http://localhost:4200, and any change in the project code triggers an automatic reload of the page.

2.5 Add the @plone/restapi-angular dependency

Stop the local server and type:

```
$ npm install @plone/restapi-angular --save
```

Note: the `--save` option ensures the dependency is added in our `package.json`.

We are now ready to use Plone Angular SDK.

CHAPTER 3

Principles

CHAPTER 4

Deployment

4.1 Basic Deployment

Deployment can be achieved in two very basic steps:

- build the app: *ng build –prod*,
- push the resulting *./dist* folder to any HTTP server.

But we need to tell the HTTP server to not worry about traversed URL. Basically any requested URL must be redirected to *index.html*, so Angular Traversal takes care about the requested path.

If you use Nginx, it can be achieved with this very simple configuration:

```
location / {
    try_files   $uri $uri/ /index.html;
}
```

Basically any existing file (like index.html, JS or CSS bundles, etc.) will be served directly, and anything else is redirected to index.html.

4.2 Server-side rendering

For a single page app, it might be interesting to be able to render pages on the server-side:

- it improves the first-page display time,
- it improves SEO,
- it makes social network sharing more accurate.

Angular provides a server-side rendering solution named Universal. Universal uses NodeJS to render the requested page as plain HTML which is delivered to the client directly. But once the first page is delivered, the page is rehydrated, meaning the JavaScript application is loaded on the background and takes the control back smoothly, so when the user clicks on any link or performs any action offered by the UI, it is processed on the client-side.

@plone/restapi-angular is Universal compliant.

A little extra configuration is needed to allow it in a regular Angular CLI project, and an example will be provided soon.

CHAPTER 5

Development

To make development and debugging of this library easy, you can run on a linked git clone when using it from an angular-cli based app.

Goals:

- Run on a git clone, not just on a released version of the library in node_modules. Making it possible to run on a branch, (master, feature branch for a later pull request...)
- Sourcemaps of the library Typescript code in the browsers developer tools.
- `debugger;`-statements can be placed in the typescript sourcecode of the library, as well as of the app.
- instant recompile and reload of both app and library code changes when using `ng serve`.
- keep imports the same: `import { RESTAPIModule } from '@plone/restapi-angular';` should work both when we run on a release in node_modules or on a git clone.

Prerequisites:

You have created an app with angular-cli.

5.1 Setting up development

The method is:

1. clone the library (or libraries).
2. symlink the src-folder of the library into a packages-folder in your apps src-folder.
3. configure the module resolution
4. configure angular-cli build to follow symlinks

This method will build the library with the methods and configuration of your app. Production releases can behave differently.

1 and 2: The following script clones two libraries - plone.restapi-angular and angular-traversal, and symlinks them into src/packages

Run it from inside your app.

```
#!/bin/sh
# Run me from project root
mkdir develop
cd develop
git clone git@github.com:plone/plone.restapi-angular.git
git clone https://github.com/makinacorpus/angular-traversal.git
cd ..

mkdir src/packages
mkdir src/packages/@plone
ln -sT ../../../develop/plone.restapi-angular/src ./src/packages/@plone/restapi-
↪angular
ln -sT ../../develop/angular-traversal/src ./src/packages/angular-traversal
```

For `@plone/restapi-angular`, we need to create the full namespace folder hierarchy (`@plone`).

3: Module resolution: We want to keep being able to import from `@plone/restapi-angular`, just as when running on a released version of the library:

```
import { RESTAPIModule } from '@plone/restapi-angular';
```

In `tsconfig.json` it is possible to configure a `paths`-mapping of module names to locations, relative to the baseUrl (the location of your apps main entry point).

See https://www.typescriptlang.org/docs/handbook/module-resolution.html#path-mapping

Add the paths mapping to the `compilerOptions` in the `tsconfig.app.json` of your app, (I assume you have the layout of an angular-cli standard project), and make sure the location matches with your `baseUrl`-setting.

```
"baseUrl": "./",
"paths": {
  "@plone/restapi-angular": ["packages/@plone/restapi-angular"],
  "angular-traversal": ["packages/angular-traversal"]
}
```

With some IDEs, like IntelliJ, you will have to put those settings into root `tsconfig.json`. Note that the baseUrl will be your source directory (probably `./src`) there.

```
"baseUrl": "./src",
"paths": {
  "@plone/restapi-angular": ["packages/@plone/restapi-angular"],
  "angular-traversal": ["packages/angular-traversal"]
},
```

4: Add the following to the `defaults` section of your `.angular-cli.json`:

```
"defaults": {
  "build": {
    "preserveSymlinks": true
  }
}
```

CHAPTER 6

Advanced

6.1 Configuration options

The CONFIGURATION provider gets some values:

- *BACKEND_URL*: the url of the backed
- *CLIENT_TIMEOUT* the time (in ms) client waits for a backend response before it raises a timeout error. Defaults to 15000.

6.2 Registering a custom marker for view registration

TBD

CHAPTER 7

References

7.1 Components

7.1.1 Breadcrumbs

```
<plone-breadcrumbs></plone-breadcrumbs>
```

Displays the breadcrumbs links for the current context.

7.1.2 Forms

Based on Angular2 Schema Form.

7.1.3 Global navigation

```
<plone-global-navigation></plone-global-navigation>
```

Displays the first level links.

7.1.4 Navigation

```
<plone-navigation root="/news" [depth]="2"></plone-navigation>
```

Display navigation links.

`root` can be either a string (to specify a static path like `/news`) or a null or negative number to specify an ancestor of the current page (0 means current folder).

`depth` defines the tree depth.

Note: be careful, in Angular templates, inputs are considered as string unless they are interpolated, so `root="/events"` returns the string `"/events"` and it works. It is equivalent to `[root]="'/events'"`. But `root="-1"` is wrong, as it would return the string `"-1"` which is not a number. To get an actual number, interpolation is mandatory: `[root]="-1"`.

7.1.5 Comments

```
<plone-comments></plone-comments>
```

Display the existing comments and allow to add new ones.

7.1.6 Workflow

```
<plone-workflow [showHistory]="true" [haveCommentInput]="true"></plone-workflow>
```

Display workflow history and actionable list of available transitions.

7.1.7 Toolbar

```
<plone-toolbar></plone-toolbar>
```

TO BE IMPLEMENTED

7.2 Directives

7.2.1 Download directive

Download directive makes the component to start a file download at click.

You have to provide a NamedFile object to the directive:

```
<a href="#" [download]="context.thefile">Click here to download {{ context.thefile.
↪filename }}</a>
```

This works with any html element:

```
<button [download]="context.thefile">Click here to download {{ context.thefile.
↪filename }}</button>
```

The directive has three outputs,

- *onBeforeDownloadStarted*,
- *onDownloadSucceeded*,
- *onDownloadFailed*

7.3 Services

7.3.1 Services injection

To make injection easier, all the following services are available in a unique service named *Services*. Example:

```
import { Services } from '@plone/restapi-angular';
...

  constructor(public services: Services) { }

  ...

    this.services.resource.find(...);
```

7.3.2 Configuration

It manages the following configuration values:

- *AUTH_TOKEN_EXPIRES*: the expiration delay of the authentication token stored in local storage, in milliseconds (1 day by default).
- *BACKEND_URL*: the URL of the backend searver exposing a valid Plone REST API
- *PATCH_RETURNS_REPRESENTATION*: if true (by default), successful patch requests return a 200 with full modified content representation as body. If false, it returns a 204 response with no content.
- *RETRY_REQUEST_ATTEMPTS*: the number of times client will try a request when server is unavailable. (3 by default).
- *RETRY_REQUEST_DELAY*: the retry delay in milliseconds (2000 by default).

Methods:

get(key: string): returns the configuration value for the given key.

urlToPath(url: string): string: converts a full backend URL into a locally traversable path.

7.3.3 Authentication

Properties:

isAuthenticated: observable indicating the current authentication status. The *state* property is a boolean indicating if the user is logged or not, and the *error* property indicates the error if any. The *username* property is the name of the logged in user, if any.

Methods:

getUserInfo(): returns an object containing the current user information.

login(login: string, password: string): authenticate to the backend using the provided credentials, the resulting authentication token and user information will be stored in localstorage. It returns an observable.

logout(): delete the current authentication token.

7.3.4 Comments

Methods:

add(path: string, data: any): add a new comment in the content corresponding to the path.

delete(path: string): delete the comment corresponding to the path.

get(path: string): get all the comments of the content corresponding to the path.

update(path: string, data: any): update the comment corresponding to the path.

7.3.5 Resources

This service gives access to all the Plone RESTAPI endpoints to manage resourcezs (i.e contents).

Properties:

defaultExpand: array of string indicating the default expansions that will be asked to the backend when we call *get*.

Methods:

breadcrumbs(path: string): return the breadcrumbs links for the specified content.

copy(sourcePath: string, targetPath: string): copy the resource to another location. Returns an observable.

create(path: string, model: any): create a new resource in the container indicated by the path. Returns an observable.

delete(path: string): remove the requested resource as an observable. Returns an observable.

find(query: any, path: string='/', options: SearchOptions={}): returns the search results as an observable.

> See http://plonerestapi.readthedocs.io/en/latest/searching.html#search. The *options* parameter can contain the following attributes:
>
> - sort_on: string, name of the index used to sort the result.
> - metadata_fields: string[], list of extra metadata fields to retrieve
> - start: number, rank of the first item (used for batching, default is 0),
> - size: number, length of the batching (default is 20)
> - sort_order: string, *'reverse'* to get a reversed order,
> - fullobjects: boolean, if *True*, the result will be fully serialized objects, not just metadata.

getSearchQueryString: (static) get a query string from a criterion/value(s) mapping and options object. Used by *find* method.

get(path: string, expand?: string[]): returns the requested resource as an observable. *expand* allow to specify extra expansion (they will be added to *defaultExpand*).

lightFileRead(file: File): Observable<NamedFileUpload>: (static) get a plone file field from a javascript File object. Not suitable for big files.

move(sourcePath: string, targetPath: string): move the resource to another location. Returns an observable.

navigation(): get the global navigation links. Returns an observable.

transition(path: string, transition: string, options: WorkflowTransitionOptions): perform the transition on the resource. You can set a workflow comment. Returns an observable of the last action information.

workflow(path: string): get the workflow history and the available transitions on the content. Returns an observable.

update(path: string, model: any): update the resource by storing the provided model content (existing attibutes are not overidden). Returns an observable.

save(path: string, model: any): update the resource by replacing its model with the provided model content. Returns an observable.

type(typeId): return the JSON schema of the specified resource type.

vocabulary(vocabularyId): return the specified vocabulary object. Returns an observable.

7.3.6 API service

This service allows to call regular HTTP verbs (for instance to call non-standard endpoints implemented on our backend):

- *get(path)*
- *post(path, data)*
- *patch(path, data)*
- *delete(path)*

They all takes care to add the appropriate headers (like authentication token), and return an observable.

In addition, it provides a specific method to download a file as a blob:

download(path) returns an observable containing a Blob object.

A Blob object can be turned into an URL like this:

```
import { DomSanitizer } from '@angular/platform-browser';

constructor(
  ...
  public sanitizer: DomSanitizer,
) { }

...
  this.services.api.download(path).subscribe(blob => {
    this.downloadURL = this.sanitizer.bypassSecurityTrustUrl(
      window.URL.createObjectURL(blob));
  });
```

It also exposes a *status* observable which returns an object containing:

- *loading*, boolean, true if call is pending, false if finished
- *error*, the HTTP error if any.

It exposes a *backendAvailable* observable that emits *false* when backend server can't be reached or consistently responds 502, 503 or 504.

7.3.7 Cache service

The CacheService service provides a *get* method which wraps *get* method from Api service with caching features.

The http request observable is piped into a Subject that repeats the same response during a delay. This delay can be set while providing *CACHE_REFRESH_DELAY* property of *CONFIGURATION* provider.

You can clear the cache emitting the *revoke* event of the service. It revokes all the cache if you give no argument to the emission. It revokes cache for a single path if you give it a string.

```
this.cache.revoke.emit('http://example.com/home')
```

The cache can't store more than as many entries as set on *CACHE_MAX_SIZE* property.

A *hits* property contains the hits statistics (number of hits by path).

Cache service is massively used by *resource* and *comments* service. All get requests are cached and all create/update/delete requests revokes cache.

7.3.8 Loading service

Loading service stores ids for what is currently loading. You declare here which loadings have begun and finished.

The service provides observables that emits when loading status changes. This is useful when you want to display a reactive loader.

You give an id to each 'thing' you mark as loaded using the *begin* method. You mark loading as finished using the *finish* method.

status behavior subject changes when there is nothing left to load or if there is at least one thing loading.

isLoading method provides an observable that emits the loading status for a specific id.

```
loading.status.subscribe((isLoading) => {
    this.somethingIsLoading = isLoading;
});

loading.isLoading('the-data').subscribe((isLoading: boolean) => {
    this.dataIsLoading = isLoading;
});

loading.begin('the-data')  // mark 'the-data' as loading
dataService.getData().subscribe((data: string[]) => {
    loading.finish('the-data');
    this.data = data;
}, (error) => {
    loading.finish('the-data');
    this.data = [];
    this.error = error;
});
```

This service is used by LoadingInterceptor http interceptor that marks a loading status when any http request is done.

7.4 Traversal

Based on Angular traversal.

The Traversal service replaces the default Angular routing. It uses the current location to determine the backend resource (the **context**) and the desired rendering (the **view**).

The view is the last part of the current location and is prefiexd by @@. If no view is specified, it defaults to *view*.

The rest of the location is the resource URL.

Example: */news/what-about-traversal/@@edit*

When traversing to the location, the resource will be requested to the backend, and the result will become the current context, accessible from any component in the app.

According the values in the *@type* property of the context, the appropriate component will be used to render the view.

Note: We can also use another criteria than *@type* by registring a custom marker (the package comes with an *InterfaceMarker* which marks context according the *interfaces* attribute, which is supposed to be a list. At the moment, the Plone REST API does not expose this attribute).

Outlet:

```
<traverser-outlet></traverser-outlet>
```

It allows to position the view rendeirng in the main layout.

Directive:

traverseTo allows to create a link to a given location.

Example: .. code-block:: html

    <a traverseTo="/events/sprint-in-bonn">See the sprint event

Methods:

addView(name: string, marker: string, component: any): register a component as a view for a given marker value. By default, we use the context's *@type* value as marker.

traverse(path: string, navigate: boolean = true): traverse to the given path. If *navigate* is false, the location will not be changed (useful if the browser location was already set before we traverse).

7.5 Views

7.5.1 @@add

Example: *http://localhost:4200/site/folder1/@@add?type=Document*

Display the form to add a new content in the current context folder. The content-type is specified in the query string.

7.5.2 @@edit

Example: *http://localhost:4200/site/folder1/@@edit*

Display the current context in an edit form.

7.5.3 @@layout

Example: *http://localhost:4200/site/folder1/@@layout*

Display the layout editor for current context.

TO BE IMPLEMENTED

7.5.4 @@login

Example: *http://localhost:4200/site/@@login*

Display the login form.

7.5.5 @@search

Example: *http://localhost:4200/site/@@search?SearchableText=RESTAPI*

Display the search results for the specified criteria.

7.5.6 @@sharing

Example: *http://localhost:4200/site/folder1/@@sharing*

Display the sharing form for the current context.

TO BE IMPLEMENTED

7.5.7 @@view

Example: *http://localhost:4200/site/folder1* or *http://localhost:4200/site/folder1/@@view*

Display the current context.