Build the content list

Two ways to generate pages in PlatformUI

Tutorial path

As written in the PlatformUI technical introduction, pages in PlatformUI can be generated either by the browser based on the REST API (or any other API) responses or by doing part of the rendering on the server side for instance with some Twig templates called by a Symfony controller. Both options are perfectly valid and choosing one or the other is mainly a matter of taste. This step will examine both strategies even if the later steps will be based on the server-side rendering.

Browser side rendering

In this case, the browser uses the REST API to fetch the necessary objects/structures and then the logic of transforming that to an HTML page is written in JavaScript and executed by the browser.

View service to fetch the Location list

Minimal view service

The first thing to do is to create a view service. A view service is a component extending `Y.eZ.ViewService`. So we first need to declare and create a module and then this module will create the minimal view service class:

yui.yml

```
ezconf-listviewservice:
     requires: ['ez-viewservice']
     path:
%extending_platformui.public_dir%/js/views/services/ezconf-listviewservice.js
```

Then in `ezconf-listviewservice.js` we can write the minimal view service:

Minimal ListViewService

```
YUI.add('ezconf-listviewservice', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListViewService =
Y.Base.create('ezconfListViewService', Y.eZ.ViewService,
[], {
        initializer: function () {
            console.log("Hey, I'm the ListViewService");
        },
    });
});
```

This is the minimal view service, it only writes a "hello world" message in the console when instantiated but for now it's not used anywhere in the application.

Configure the route to use the view service

To really use our view service in the application, we have to change the route so that the PlatformUI application instantiates and uses the view service when building the page. To do that, we have to add the route `service` property to hold the constructor function of the view service so the application plugin that adds the route will also have to require the `ezconf-listviewservice` module:

yui.yml

```
 ezconf-listapplugin:
     requires: ['ez-pluginregistry', 'plugin', 'base',
'ezconf-listview', 'ezconf-listviewservice'] # the view
module has been added
     dependencyOf: ['ez-platformuiapp']
```

After doing that, `Y.eZConf.ListViewService` becomes available in the application plugin code and we can change the `eZConfList` route to:

Creating a route with a view service

```
app.route({
    name: "eZConfList",
    path: "/ezconf/list",
    view: "ezconfListView",
    service: Y.eZConf.ListViewService, // constructor
function to use to instantiate the view service
    sideViews: {'navigationHub': true, 'discoveryBar':
false},
    callbacks: ['open', 'checkUser', 'handleSideViews',
'handleMainView'],
});
```

After this change, the `Y.eZConf.ListViewService` is used when a user reaches the `eZConfList` route.

Fetching Locations from the view service

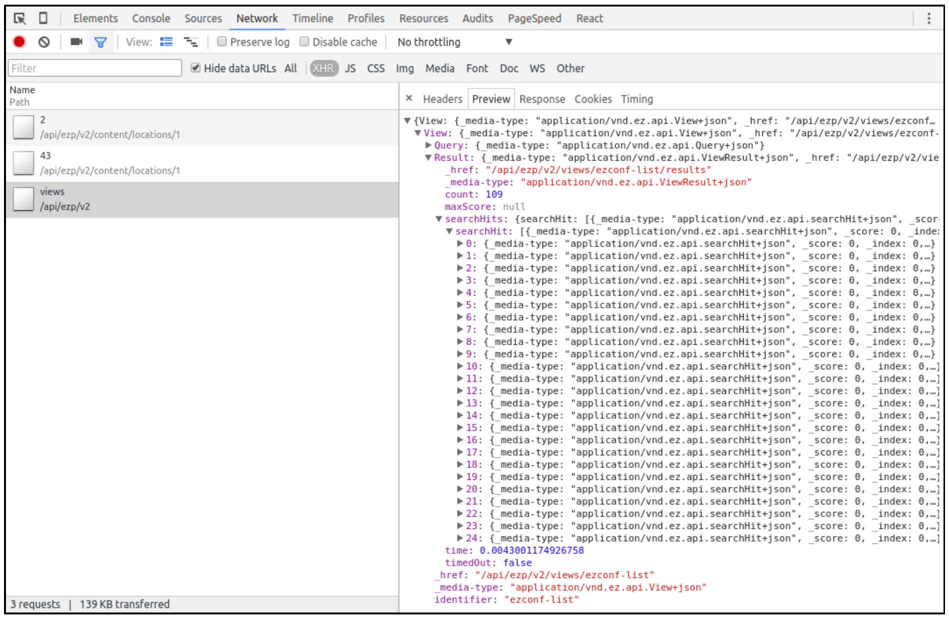
A view service is responsible for fetching data so it can be rendered. For a given route, the view service is instantiated the first time the route is accessed and then the same instance is reused. On that instance, the `_load` method is automatically called. This is where the loading logic should be in most cases. This method also receives a callback as its only parameter. This callback function should be called once the loading is finished. Typically, a view service will use the JavaScript REST Client to request eZ Platform REST API. To do that, a JavaScript REST Client instance is available in the `capi` attribute of the view service.

In this tutorial, we want to display the Content in a flat list and filter this list by Content Types. For now, let's fetch everything; to do that, the view service will create a REST view to search for every Location in the repository:

ezconf-listviewservice.js

```
YUI.add('ezconf-listviewservice', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListViewService =
Y.Base.create('ezconfListViewService', Y.eZ.ViewService,
[], {
        initializer: function () {
            console.log("Hey, I'm the ListViewService");
        },

        // _load is automatically called when the view
service is configured for
        // a route. callback should be executed when
everything is finished
        _load: function (callback) {
            var capi = this.get('capi'), // REST API
JavaScript client
                contentService =
capi.getContentService(),
                query =
contentService.newViewCreateStruct('ezconf-list',
'LocationQuery');

            // searching for "everything"
            query.body.ViewInput.LocationQuery.Criteria
= {SubtreeCriterion: "/1/"};
            contentService.createView(query,
Y.bind(function (err, response) {
                // parsing the response and storing the
location list in the "location" attribute
                var locations;

                locations =
Y.Array.map(response.document.View.Result.searchHits.sea
rchHit, function (hit) {
                    var loc = new Y.eZ.Location({id:
hit.value.Location._href});


loc.loadFromHash(hit.value.Location);
                    return loc;
                });
                this.set('locations', locations);
                callback();
            }, this));
        },
    }, {
        ATTRS: {
            locations: {
                value: [],
            }
        }
    });
});
```

At this point, if you refresh the PlatformUI application and follow the link added in the previous step, you should see a new REST API request to `/api/ezp/v2/views` in the network panel of the browser:



```
2
/api/ezp/v2/content/locations/1
43
/api/ezp/v2/content/locations/1
views
/api/ezp/v2

3 requests | 139 KB transferred
```

```
▼{View: {_media-type: "application/vnd.ez.api.View+json", _href: "/api/ezp/v2/views/ezconf…
  ▼View: {_media-type: "application/vnd.ez.api.View+json", _href: "/api/ezp/v2/views/ezconf-
    ▶Query: {_media-type: "application/vnd.ez.api.Query+json"}
    ▼Result: {_media-type: "application/vnd.ez.api.ViewResult+json", _href: "/api/ezp/v2/vie
       _href: "/api/ezp/v2/views/ezconf-list/results"
       _media-type: "application/vnd.ez.api.ViewResult+json"
       count: 109
       maxScore: null
      ▼searchHits: {searchHit: [{_media-type: "application/vnd.ez.api.searchHit+json", _scor
        ▼searchHit: [{_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _inde
          ▶0: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶1: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶2: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶3: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶4: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶5: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶6: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶7: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶8: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶9: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…}
          ▶10: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶11: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶12: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶13: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶14: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶15: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶16: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶17: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶18: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶19: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶20: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶21: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶22: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶23: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
          ▶24: {_media-type: "application/vnd.ez.api.searchHit+json", _score: 0, _index: 0,…]
       time: 0.0043001174926758
       timedOut: false
     _href: "/api/ezp/v2/views/ezconf-list"
     _media-type: "application/vnd.ez.api.View+json"
     identifier: "ezconf-list"
```

The Locations are built in the application but not yet used anywhere.

Passing the Location to the view

Now that we have the Locations, we have to give them to the view. For that, we have to implement the `_getViewParameters` method in the view service. This method is automatically called when view service loading is finished, it should return an object that will be used as a configuration object for the main view.

In our case, we just want to give the Location list to the view, so the `_getViewParameters` method is quite simple:

_getViewParameters of the view service

```
_getViewParameters: function () {
    return {
        locations: this.get('locations'),
    };
},
```

With that code, the view will receive the Location list as an attribute under the name `locations`.

Why implement _load and _getViewParameters and not load and getViewParameters?
When implementing a custom view service, you should always implement the protected `_load` and `_getViewParameters` methods, not their public counterparts `load` and `getViewParameters`. By implementing the protected versions, you keep the opportunity for a developer to write a plugin to enhance your view service.

View to display a list of Location

The view now receives the Location list in the `locations` attribute so we have to change the view to take that into account. For now, let's change it to just display the Location it receives as an unordered HTML list of links to those Locations. To do that, we have to:

1. Declare the `locations` attribute in the view
2. Give that list to the template in a form it can understand
3. Update the template to generate the list

Point 2 is required because Handlebars is not able to understand the complex model objects generated by YUI. So we have to transform those complex object into plain JavaScript objects. After doing the changes in steps 1 and 2, the view looks like this:

ezconf-listview.js

```
YUI.add('ezconf-listview', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListView = Y.Base.create('ezconfListView',
Y.eZ.TemplateBasedView, [], {
        initializer: function () {
            console.log("Hey, I'm the list view");
        },

        render: function () {
            this.get('container').setHTML(
                this.template({
                    locations:
this._getJsonifiedLocations(),
                })
            );
            return this;
        },

        _getJsonifiedLocations: function () {
            // to get usable objects in the template
            return Y.Array.map(this.get('locations'),
function (loc) {
                return loc.toJSON();
            });
        },
    }, {
        ATTRS: {
            locations: {
                value: [],
            }
        }
    });
});
```

Then the template has to be changed to something like:

ezconflistview.hbt

```
<h1 class="ezconf-list-title">List view</h1>

<ul class="ezconf-list">
{{#each locations}}
    <li><a href="{{path 'viewLocation' id=id
languageCode=contentInfo.mainLanguageCode}}">{{
contentInfo.name }}</a></li>
{{/each}}
</ul>
```

PlatformUI provides a `path` template helper that allows you to generate a route URL in PlatformUI. It expects a route name and the route parameters.

Results
The resulting code can be been in the `6_1_list_client` tag on GitHub, this step result can also be viewed as a diff between tags `5_navigation` and `6_1_list_client`.

The rest of this tutorial is focused on the server side rendering strategy. Completing the browser side rendering strategy to get the expected features is left as an exercise.

Server side rendering

In this case a part of the rendering is delegated to the server. When building a PlatformUI page this way, the application will just do one or more AJAX request(s) and inject the result in the UI. The PlatformUI *Admin part* is built this way. To be easily usable in the JavaScript application, the server response has to be structured so that the application can retrieve and set the page title, the potential notifications to issue and the actual page content. This is done by generating an HTML fragment, in the following way:

Example of server side response

```
<div data-name="title">Title to set in the
application</div>
<div data-name="html">
    <p>Page content</p>
</div>
<ul data-name="notification">
    <li data-state="done">I'm a "done" notification to
display in the application</li>
</ul>
```

View service to fetch an HTML fragment

Minimal view service

In the case, the minimal view service is exactly the same as the one produced in the previous Minimal view service paragraph.

Configure the route to use the view service

The eZConfList route also has to be configured exactly in the same way as in the previous Configure the route to use the view service paragraph.

Generate the HTML fragment server side

This will be done by a Symfony Controller that will use the Search Service and a Twig template to generate the HTML code. As you can see in this Github commit, it's a very basic Symfony Controller that extends `EzSystems\PlatformUIBundle\Controller\Controller`.

> Extending `EzSystems\PlatformUIBundle\Controller\Controller` is not strictly required. By doing that, the actions provided by the controller are automatically restricted to authenticated users. This base controller also provides the base API to handle notifications if needed.
>
> To learn how to write Symfony controllers, please read the Symfony Controller documentation.

The `list` action in `ListController` uses the following Twig template:

list.html.twig

```
{% extends "eZPlatformUIBundle::pjax_admin.html.twig" %}

{% block header_title %}
    <h1 class="ezconf-list-title">List view</h1>
{% endblock %}

{% block content %}
<ul class="ezconf-list">
{% for value in results.searchHits %}
    <li><a href="">{{ value.valueObject.contentInfo.name
}}</a></li>
{% endfor %}
</ul>
{% endblock %}

{% block title %}List{% endblock %}
```

Again, it's a quite a regular template but to ease the generation of the expected structured HTML fragment, this template extends `eZPlatformUIBundle::pjax_admin.html.twig` and redefines a few blocks, the main one being `content` where the actual page content is supposed to be generated.

Update the view service to fetch the generated HTML fragment

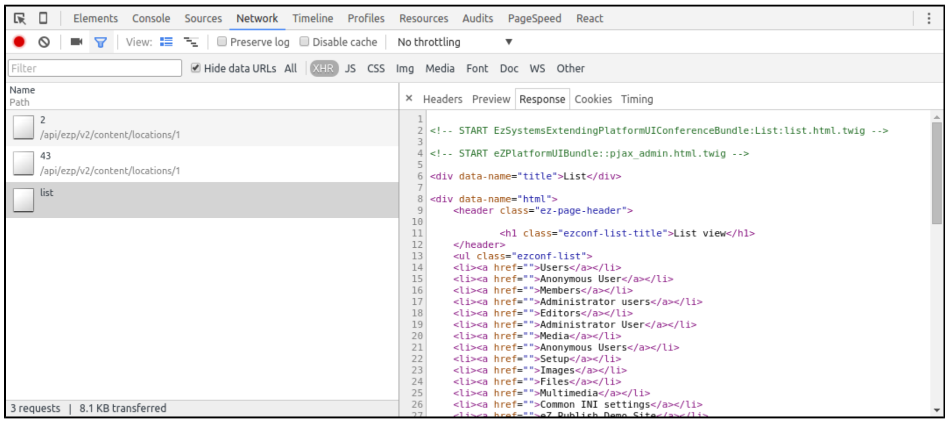
We now have a Symfony Controller able to generate our Location list but this list is not yet available in the application. As in Fetching Locations from the view service and Passing the Location to the view, we have to add the code in the view service. But in the case of a server side rendering, we can reuse `Y.eZ.ServerSideViewService` which provides the base API to parse an HTML fragment which also provides a ready to use `_getViewParameters` method. All we have to do then is to implement the loading logic in `_load`:

ezconf-listviewservice.js

```
YUI.add('ezconf-listviewservice', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListViewService =
Y.Base.create('ezconfListViewService',
Y.eZ.ServerSideViewService, [], {
        initializer: function () {
            console.log("Hey, I'm the ListViewService");
        },

        _load: function (callback) {
            var uri = this.get('app').get('apiRoot') +
'list';

            Y.io(uri, { // YUI helper to do AJAX
request, see http://yuilibrary.com/yui/docs/io/
                method: 'GET',
                on: {
                    success: function (tId, response) {
                        this._parseResponse(response);
// provided by Y.eZ.ServerSideViewService
                        callback(this);
                    },
                    failure: this._handleLoadFailure, //
provided by Y.eZ.ServerSideViewService
                },
                context: this,
            });
        },
    });
});
```

The resulting `_load` method will just do an AJAX request to the action provided by our Symfony controller.

At this point, if you refresh your browser, nothing should have changed but you should see the AJAX request in the network panel of your browser.



```
2
/api/ezp/v2/content/locations/1
43
/api/ezp/v2/content/locations/1
list
```

```

<!-- START EzSystemsExtendingPlatformUIConferenceBundle:List:list.html.twig -->

<!-- START eZPlatformUIBundle::pjax_admin.html.twig -->

<div data-name="title">List</div>

<div data-name="html">
    <header class="ez-page-header">

            <h1 class="ezconf-list-title">List view</h1>
    </header>
    <ul class="ezconf-list">
    <li><a href="">Users</a></li>
    <li><a href="">Anonymous User</a></li>
    <li><a href="">Members</a></li>
    <li><a href="">Administrator users</a></li>
    <li><a href="">Editors</a></li>
    <li><a href="">Administrator User</a></li>
    <li><a href="">Media</a></li>
    <li><a href="">Anonymous Users</a></li>
    <li><a href="">Setup</a></li>
    <li><a href="">Images</a></li>
    <li><a href="">Files</a></li>
    <li><a href="">Multimedia</a></li>
    <li><a href="">Common INI settings</a></li>
    <li><a href="">eZ Publish Demo Site</a></li>
```

View to handle the server side generated code

Change the view to be server side view

To have a visual change, we now have to change the `ListView` to be a server side view. This operations involves removing some code added in the Define a View step. Basically, the View does not need any template but it will inherit from `Y.eZ.ServerSideView`. As a result, the view module definition becomes:

View module definition

```
ezconf-listview:
    requires: ['ez-serversideview']
    path:
%extending_platformui.public_dir%/js/views/ezconf-listvi
ew.js
```

And the View component also has to be simplified to:

ezconf-listview.js

```
YUI.add('ezconf-listview', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListView = Y.Base.create('ezconfListView',
Y.eZ.ServerSideView, [], {
        initializer: function () {
            console.log("Hey, I'm the list view");
            this.containerTemplate = '<div
class="ez-view-ezconflistview"/>'; // make sure we keep
the same class on the container
        },
    });
});
```

At this point, you should see the same list as the one that was generated in Browser side rendering section. The only difference lies in the non-working links being generated in the server side solution.

Navigating in the app from the view

There are several ways to fix the link issue. In this step we are going to add some metadata to the generated HTML links, then we'll change the view to recognize the enhanced links and finally we'll change the server side view to achieve the navigation.

The server side code has no knowledge of the JavaScript application routing mechanism as a result, it can not directly generate any PlatformUI Application URI, but we know while generating the HTML fragment that we want each link to allow the navigation to the `viewLocation` route for the Location being displayed. We can then change the Twig template to add the necessary metadata on each link so that the application has a way of guessing where the user is supposed to go when clicking on the link:

"content" block in the Twig template

```
{% block content %}
<ul class="ezconf-list">
{% for value in results.searchHits %}
    <li><a class="ezconf-list-location" href=""
        data-route-name="viewLocation"
        data-route-id="{{
path('ezpublish_rest_loadLocation', {locationPath:
value.valueObject.pathString|trim('/')}) }}"
        data-route-languageCode="{{
value.valueObject.contentInfo.mainLanguageCode }}"
        >{{ value.valueObject.contentInfo.name
}}</a></li>
{% endfor %}
</ul>
{% endblock %}
```

For each link we are basically saying to the application that the user should directed to the `viewLocation` route for the given Location id and the given language code.

> In PlatformUI code, the Locations, Content items and Content Types are identified by their REST id, that is the REST resource URL which allows you to fetch the object in the REST API. That's why we are using the `path` Twig template function to build the Location id.

Then we have to change the view to add a special behavior when the user clicks on them. The view can not directly trigger the navigation to the expected route. So in this case, we are firing an application level event with all the data we have on the link and we'll let the view service handle this application level event to take the user to the expected page. So, we have to configure our view to recognize the click on the links and to fire the `navigateTo` custom event:

ezconf-listview.js

```
YUI.add('ezconf-listview', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListView = Y.Base.create('ezconfListView',
Y.eZ.ServerSideView, [], {
        // this is YUI View mechanic to subscribe to DOM
events (click, submit,
        // ...) and synthetic event (some custom event
provided by YUI) like
        // 'tap' here.
        events: {
            '.ezconf-list-location': {
                // tap is 'fast click' (touch friendly)
                'tap': '_navigateToLocation'
            }
        },

        initializer: function () {
            console.log("Hey, I'm the list view");
            this.containerTemplate = '<div
class="ez-view-ezconflistview"/>';
        },

        _navigateToLocation: function (e) {
            var link = e.target;

            e.preventDefault(); // don't want the normal
link behavior

            // tell the view service we want to navigate
somewhere
            // it's a custom event that will be bubble
up to the view service
            // (and the app)
            // the second parameter is the data to add
in the event facade, this
            // can be used by any event handler function
bound to this event.
            this.fire('navigateTo', {
                routeName: link.getData('route-name'),
                routeParams: {
                    id: link.getData('route-id'),
                    languageCode:
link.getData('route-languagecode'),
                }
            });
        },
    });
});
```

The DOM event handling is one of the main YUI View features. It is documented in the YUI View guide.

Now the click on the Location link is transformed in a `navigateTo` event, we have to subscribe to that event in the view service to trigger the expected navigation:

ezconf-listviewservice.js

```
YUI.add('ezconf-listviewservice', function (Y) {
    Y.namespace('eZConf');

    Y.eZConf.ListViewService =
Y.Base.create('ezconfListViewService',
Y.eZ.ServerSideViewService, [], {
        initializer: function () {
            console.log("Hey, I'm the ListViewService");

            // we catch the `navigateTo` event no matter
from where it comes
            // when bubbling, the event is prefixed with
the name of the
            // component which fired the event first.
            // so in this case we could also write
            // this.on('ezconflistview:navigateTo',
function (e) {});
            // `e` is the event facade. It contains
various informations about
            // the event and if any the custom data
passed to fire().
            this.on('*:navigateTo', function (e) {
                this.get('app').navigateTo(
                    e.routeName,
                    e.routeParams
                );
            });
        },

        _load: function (callback) {
            // [...] skip to keep the example short
        },
    });
});
```

With that in place and after a refresh, the Location list should allow you to the navigate to the expected Location in PlatformUI.

Application level events
PlatformUI uses a lot of custom application level events thanks to the EventTarget YUI component. Those events are very similar to DOM events but they are attached to the application components instead of the DOM elements. Like for DOM events, there is a bubbling mechanism. For instance, here the view is firing an event and unless the propagation of the event is stopped, it will bubble to the view service and then to the application. The event basically follows the components tree until the application. **An application level event is way for a deeply nested component to communicate with a higher level component.**

Results and next step
The resulting code can be seen in the `6_2_list_server` tag on GitHub, this step result can also be viewed as a diff between tags `5_navigation` and `6_2_list_server`.

The next step is then to add the pagination.