Alter the JavaScript Application routing

PlatformUI routing mechanism

The essential task of the PlatformUI Application component is to handle the routing. It is based on the routing capabilities provided by the YUI App component and it uses hash-based URIs. By default, the PlatformUI Application will recognize and handle several routes which are declared in the app component itself.

A route is described by an object with the following properties:

- `path`: the path to match
- `view`: the identifier of the main view to render when the route is matched
- `callbacks`: a list of *middlewares* to execute
- `name`: an optional name to generate links
- `sideViews`: an optional side view configuration
- `service`: an optional reference to a view service constructor

Modifying the routing from the bundle with a plugin

To tweak any behavior in the application, the way to go is to write a plugin and in this case a plugin for the Application.

Declaring the module providing plugin

The module has to be declared in the extension bundle's `yui.yml` file. It can be done in the following way:

```
system:
    default:
        yui:
            modules:
                # use your own prefix, not "ez-"
                ezconf-listapplugin: # module identifier
                    dependencyOf: ['ez-platformuiapp']
                    requires: ['ez-pluginregistry', 'plugin', 'base'] # depends on the
plugin code
                    path:
%extending_platformui.public_dir%/js/apps/plugins/ezconf-listappplugin.js
```

This configuration means we are declaring a module whose identifier is `ezconf-listapplugin`. It will be added to the dependency list of the module `ez-platformuiapp` (the one providing the application component). The plugin module requires `ez-pluginregistry`, `plugin` and ba `se.` It is stored on the disk in `%extending_platformui.public_dir%/js/apps/plugins/ezconf-listappplugin.js`.

> `%extending_platformui.public_dir%` is a container parameter which was added in the previous step. It is here to avoid repeating again and again the base path to the public directory. Of course, it is also perfectly possible to write the full path to the module.

Module creation

Before creating the actual plugin code, we have to first create the module in the configured file. The minimal module code is:

%extending_platformui.public_dir%/js/apps/plugins/ezconf-listappplugin.js

```
YUI.add('ezconf-listapplugin', function (Y) {
    // module code goes here!
 // this function will executed when the module loaded in the app,
    // not when the file is loaded by the browser
    // the Y parameter gives access to the YUI env, for instance the components
    // defined by others modules.
});
```

The first parameter of `YUI.add` should be exactly the module identifier used in `yui.yml` otherwise the module won't be correctly loaded in the application. If the module code does not seem to be taken into account, it is the very first thing to check.

Base plugin code

After the module creation, it is time to create the minimal Application plugin:

%extending_platformui.public_dir%/js/apps/plugins/ezconf-listappplugin.js

```
YUI.add('ezconf-listapplugin', function (Y) {
    // Good practices:
    // * use a custom namespace. 'eZConf' is used as an example here.
    // * put the plugins in a 'Plugin' sub namespace
    Y.namespace('eZConf.Plugin');

    Y.eZConf.Plugin.ListAppPlugin = Y.Base.create('ezconfListAppPlugin',
Y.Plugin.Base, [], {
        initializer: function () {
            var app = this.get('host'); // the plugged object is called host

            console.log("Hey, I'm a plugin for PlatformUI App!");
            console.log("And I'm plugged in ", app);
        },
    }, {
        NS: 'ezconfTypeApp' // don't forget that
    });

    // registering the plugin for the app
    // with that, the plugin is automatically instantiated and plugged in
    // 'platformuiApp' component.
    Y.eZ.PluginRegistry.registerPlugin(
        Y.eZConf.Plugin.ListAppPlugin, ['platformuiApp']
    );
});
```

The added code creates a plugin class and registers it under `Y.eZConf.Plugin.ListAppPlugin`, then the PlatformUI plugin registry is configured so that this plugin is automatically instantiated and plugged in the PlatformUI App component.

The PlatformUI's plugin system comes almost entirely from the YUI plugin. While that's not a strict requirement, you should use the *Advanced Plugins* strategy mentioned in the YUI documentation. That's why in this example and in most cases, the plugin will have the `plugin` and `base` YUI plugin as dependencies. `base` also provides the low level foundations for most PlatformUI component, so reading the Base YUI documentation will also help understanding several concepts used all over the application.

At this point, if you open PlatformUI in your favorite browser with the console open, you should see the result of the `console.log` calls in the above code.

Adding a route to the application

Finally, the plugin is ready to add a new route to the application. As written in the previous code sample, the plugged object, the application here, is available through `this.get('host')` in the plugin. The App object provides a `route` method allowing to add route.

%extending_platformui.public_dir%/js/apps/plugins/ezconf-listappplugin.js

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

    Y.eZConf.Plugin.ListAppPlugin = Y.Base.create('ezconfListAppPlugin',
Y.Plugin.Base, [], {
        initializer: function () {
            var app = this.get('host'); // the plugged object is called host

            app.route({
                name: "eZConfList",
                path: "/ezconf/list",
                view: "dashboardView", // let's display the dashboard since we don't
have a custom view... yet :)
                // we want the navigationHub (top menu) but not the discoveryBar
                // (left bar), we can try different options
                sideViews: {'navigationHub': true, 'discoveryBar': false},
                callbacks: ['open', 'checkUser', 'handleSideViews', 'handleMainView'],
            });
        },
    }, {
        NS: 'ezconfTypeApp' // don't forget that
    });

    Y.eZ.PluginRegistry.registerPlugin(
        Y.eZConf.Plugin.ListAppPlugin, ['platformuiApp']
    );
});
```

Now, if you refresh your browser, you still need not see any visible change but the application should recognize the `/ezconf/list` hash URI. Going to `/ez#/ezconf/list` should display the same thing as `/ez#/dashboard`.

> The PlatformUI Application component extends the YUI App component, as a result the complete API of this component can be used.

> **Results and next step:**
> The resulting code can be seen in the 3_routing tag on GitHub, this step result can also be viewed as a diff between tags `2_configuration` and `3_routing`.
>
> The next step is then to define a new view and to use it when the newly added route is matched.