# Technical Exercise

In this phase, the candidate will need to deliver a small working application that attends to the requested requirements. The candidate will have 3 days to deliver what was asked.

### General Questions

* What best practices have you used to ensure that the front-end of your websites is as fast as possible?
* Describe your preferred new JS feature (ES2015/2016 or under active specification)

### Single Page Application Exercise

Implement a simple Single Page Application engine that attend to the following requirements:

* You cannot use any third party libraries
* It should be pluggable to existing web sites, requiring minimal changes. Take the following application as an app that could benefit from SPA: [Link](https://github.com/brunobasto/liferay_fi_spa_exercise).
* It should have a **routes** attribute**.** The purpose of this attribute is to define which links the SPA Engine recognize so that it knows that, when clicking on those links, the page shouldn't refresh, but the navigation should go through an Ajax request instead.
* It should have an **enabled** attribute. When enabled is **false** the application links go through traditional navigation (No SPA).
* After the content is loaded, It should replace the current content of **<body>** with the new one that came from the SPA request.
* If there are JavaScript scripts (either inline or referenced externally with the src attribute) inside the new content loaded, the engine should parse them so that the application works as expected.

An example of how to start the engine would be like follows:

<script type="text/javascript">

var engine = new Engine(

{

routes: ['\*.html', '/site/\*'],

enabled: true,

}

);

</script>

Bonus features (Not required, but desirable)

* Implement unit tests.
* Show a loading bar while the content is being loaded.
* Implement history management using the History API (history.pushState, history.replaceState, etc)
* Use ES6 syntax
* Avoid SPA for links that have the "data-no-spa" data attribute. For example:

<a href="no\_spa.html" data-no-spa="true">No SPA</a>

### Metal TreeView Component

Implement a **TreeView** component using [Metal.js](http://www.metaljs.com/) that attend to the following requirements:

* There should be a **TreeView** class that renders nodes according to a ***nodes*** attribute. An example of usage of that component would be like this:

<script type="text/javascript">

var treeView = new TreeView(

{

nodes: [

{

label: 'nodeA'

},

{

label: 'nodeB'

nodes: [

{

label: 'nodeB\_child1'

},

.

.

.

{

Label: 'nodeB\_childn'

}

]

}

]

}

);

</script>

* You cannot use any other JavaScript third party libraries other than **Metal.js.** You can use other CSS libraries like Bootstrap, for example.
* You can use Bootstrap’s CSS to help you with the basic styling of the components.
* You can use either [JSX](http://www.metaljs.com/docs/jsx-components.html) or [Soy](http://www.metaljs.com/docs/soy-components.html) as template languages.
* The user should be able to **expand** and **collapse** nodes that have children inside it. Expanding a node should show all its children. Collapsing a node should hide all its children. There should be icons showing if the node is collapsed or expanded. Clicking on the icon should **toggle** between **collapsed** and **expanded**.
* The TreeView should be **accessible**. Acceptance criterias for accessibility are:
  + User should be able to focus nodes using the keyboard.
  + Using **up and down arrow keys** user can navigate vertically to the previous or next node at the root of the tree or inside a parent node.
  + Using **left and right arrow keys** user should be able to navigate into a subtree of a node or out of a subtree of a node.
  + Using the **enter key,** user should be able to collapse or expand a node.

Bonus features (Not required, but desirable)

* Implement unit tests.
* Implement an option that enables the TreeView nodes to fetch its children using ajax. When clicking to expand a node, an Ajax call would fetch its children. An example of usage would be:

<script type="text/javascript">

var treeView = new TreeView(

{

nodes: [

{

label: 'Click to fetch children',

**childrenURL: '/data/children.json'**

}

}

);

</script>

* Implement an exposed ***TreeViewNode*** class so that users of your API are able to customize their nodes. An example of a custom node would be:

<script type="text/javascript">

class **ColoredTreeViewNode** extends TreeViewNode {

… custom logic here …

}

var treeView = new TreeView(

{

nodes: [

new **ColoredTreeViewNode**({

label: 'my custom node',

**color: 'blue'**

}),

{

label: 'nodeB'

nodes: [

{

label: 'nodeB\_child1'

},

.

.

.

{

Label: 'nodeB\_childn'

}

]

}

]

}

);

</script>

* Implement the **RadioTreeViewNode** and **CheckboxTreeViewNode**. The names are self explanatory: The first should render a radio button inside of it and the second should render a checkbox inside of it.
* Add an attribute called ***draggable*** that adds the ability to drag nodes from one place of the TreeView to the other. Think of it as a file explorer: You can move folders to different directories just by dragging them.

**Note: You MUST NOT make any information about the exercise publicly available. You MUST NOT publish the code you create on repositories which are publicly available.**