React JS

Integration with Liferay DXP Using Headless API

Things To Note

The purpose of this documentation is to show how users can use a popular Javascript library called React.js with Liferay and Headless API to create stand-alone React applications.

1. Primary requirements include knowing a bit of JavaScript, how to set up/download software onto Terminal for MacOSX. Also, knowing how to initialize a Liferay instance locally will help with the set up portion of this walk through.

2. For this project, I developed on a MacOSX (Linux) system, using VS Code as a barebones code editor and GitHub to push my files onto the internet for public access.


3. This documentation is set up into separate parts, divided by column. These include Getting Started, Headless API's Into React, and Component Explanations.

Getting Started

https://github.com/sasajallana/react-for-liferay-fe/tree/master
For reference if you're ever stuck

Diving In

https://github.com/sasajallana/react-for-liferay-fe/tree/master
For reference if you're ever stuck

NPM

NPM is the orchestration tool used to create and manage React Apps. Make sure you have these latest versions (as of 2/27/2020), which work with Liferay DXP 7.2.

npm/npx	6.13.6
java sdk	openjdk version "1.8.0_232"
node	v13.7.0

Liferay and Blade

Liferay is being used as a platform management system in this project. It provides the Headless API that our stand-alone React application(s) call in order to retrieve data.

Blade is the CLI tool used to initialize a Liferay project with the correct files.

This tutorial uses 7.2 DXP with Tomcat.

Setting Up

One thing that you should have already is **Homebrew**, a MacOSX package installer. Using Homebrew makes it easier to download software using Terminal:
https://docs.brew.sh/Installation

Before beginning to setup ReactJS for Liferay Workspace, you will need a few things. As mentioned above, you should first download things like Npm/npx, node, Java. On MacOSX, the easiest way to download is by typing these commands in $USER home on a Terminal window...

node	brew install node
npm/npx	(will be installed w node, check via) npm --version

And then, you're going to need to download Liferay Workspace:
https://sourceforge.net/projects/lportal/files/Liferay%20IDE/3.8.0/

LiferayWorkspace-202002250521-osx-installer.dmg	2020-02-26	22.6 MB

Click on this line from the above link and wait for the download. Once done, go through the process of setting up from this site:
https://portal.liferay.dev/docs/7-1/tutorials/-/knowledge_base/t/liferay-workspace

Note: If you try to open the Installer directly, it might issue a 'Untrusted Developer' warning. The easiest work around is to right-click the Installer icon and from the Menu, click Open.

Why We Use SDK from DMG

We are using the SDK installer to initialize Liferay and Blade CLI. From now on, we will use Blade and npm to create Liferay Workspaces to host our React App for development.

Setting Up Workspace

Now with Node, Java, NPM installed and the Liferay installer done, let's start from anew.

Go ahead and open your Terminal to your $USER home directory.
Let's create a new folder where your Liferay Workspace will live and cd into it.

cmd

```
mkdir react-for-liferay-fe
```

cmd

```
cd react-for-liferay-fe
```

Now, in your new empty folder "**react-for-liferay-fe**", initialize a Liferay workspace through:

cmd

```
blade init -v 7.2
```

The **-v** tag indicates what version of Liferay we're using. Now if you list the files in the directory (**ls** command), you will see this:

Terminal

```
Setups-MacBook-Pro:react-for-liferay-fe setup$ blade init -v 7.2
Setups-MacBook-Pro:react-for-liferay-fe setup$ ls
build.gradle            gradle                  gradle.properties       gradlew.bat             settings.gradle         wars
configs                 gradle-local.properties gradlew                 modules                 themes
```

These are all Liferay Workspace files. Next, we need Liferay Server files, in order to run and setup the rest of Liferay. Still in the **react-for-liferay-fe** directory, run command

cmd

```
blade server init
```

Terminal

```
Setups-MacBook-Pro:react-for-liferay-fe setup$ blade server init
Executing gradle task initBundle...


server init completed successfully.
Setups-MacBook-Pro:react-for-liferay-fe setup$ ls
build.gradle            configs                 gradle-local.properties gradlew                 modules                 themes
bundles                 gradle                  gradle.properties       gradlew.bat             settings.gradle         wars
```

This will create a **bundles** folder. In the new bundles folder, there will be a bunch of server files. To access the command to run Liferay Workspace, you will need to execute this command to move into the correct folder:

cmd

```
cd bundles/tomcat-9.0.17/bin
```

Terminal

```
/Users/setup/react-for-liferay-fe/bundles/tomcat-9.0.17/bin
Setups-MacBook-Pro:bin setup$ ls
bootstrap.jar                   configtest.bat                  setclasspath.sh                 tomcat-native.tar.gz
catalina-tasks.xml              configtest.sh                   setenv.bat                      tool-wrapper.bat
catalina.bat                    daemon.sh                       setenv.sh                       tool-wrapper.sh
catalina.sh                     digest.bat                      shutdown.bat                    version.bat
ciphers.bat                     digest.sh                       shutdown.sh                     version.sh
ciphers.sh                      makebase.bat                    startup.bat
commons-daemon-native.tar.gz    makebase.sh                     startup.sh
commons-daemon.jar              setclasspath.bat                tomcat-juli.jar
```

You will notice a bunch of .sh and .bat files. These are executionary files that can be launched or stopped by using commands like "**run**" and "**stop**". In this case, we will be setting up Liferay Workspace for the first time, so we need to *run* the server. You do this by running the **catalina.sh** file with the command:

cmd

```
./catalina.sh run
```

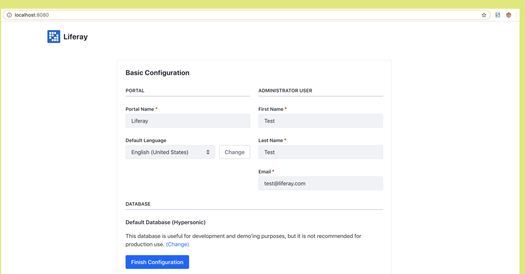
Use this command anytime you would like to run the Liferay Workspace. Since we are going to be consistently building off data from the Liferay Workspace, the server will need to be up. If nothing goings wrong, in the Terminal the below Liferay logo and extra lines of code should appear, and then a browser window will open connected to web address **localhost:8080**

Terminal

```
   __   _____________________ ____  ___  __
  / /  / _/ ____/ ____/ __ \/ __ \/   |/ /
 / /  / // /_  / __/ / /_/ / /_/ / /| |/ /
/ /___/ // __/ / /___/ _, _/ ____/ ___ / /
/_____/___/_/   /_____/_/ |_/_/   /_/  |_/

Starting Liferay Community Edition Portal 7.2.0 CE GA1 (Mueller / Build 7200 / June 4, 2019)
```

localhost:8080



Continue set-up as usual with **Default Configuration** and **Default Database** as Hypersonic database, and make sure to create the **Admin user as the initial test@liferay**, just incase you ever get locked out somehow. **Agree to the Terms of Service** and set your password. You should have a fresh, working Liferay Instance working!

Setting Up React

To start integrating React with Liferay, open up Terminal and redirect into the workspace directory created before that hosts all of the server and Liferay files (if following the tutorial, **react-for-liferay-fe**).

Assuming that you have npm/node installed, we will now be able to use **npm/npx** commands in order to build our ReactJS application.

One warning/issue you might bump into is the **mismatch of compatible Node.js version to work with Liferay.** As of **3/3/2020,** the latest version of Node.js that works with Liferay DXP 7.2 is **Dubnium.** You can change your Node version via **nvm.**

From the **react-for-liferay-fe** directory in your Terminal, type the following command:

```
npx create-react-app reactapp
```

npx: is like npm, but runs the following command temporarily, and is best in this case to just instantiate a new React application in the local **react-for-liferay-fe** directory

create-react-app: is the command to build React apps/generate fresh React application files

reactapp: is the name of our new React application

After the command runs and says it's creating and finishes installing a new React project called **reactapp,** you should get a few "Success" messages, and recommended commands to run in order to start the project. Go ahead and **cd into reactapp and ls.**

cd reactapp ; ls

```
Setups-MacBook-Pro:reactapp setup$ pwd
/Users/setup/react-for-liferay-fe/reactapp
Setups-MacBook-Pro:reactapp setup$ ls
README.md               package-lock.json       public
node_modules            package.json            src
```

node_modules: where external dependencies/files needed for the project as you develop will be stored and referenced

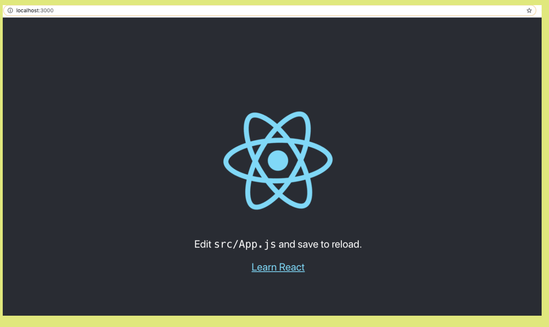
package.json: metadata for the project, including dependency version numbers, script commands

public: on initialization, contains barebones files that renders the homepage (**index.html**) and other files

src: contains JavaScript and style files that helps to render the homepage programatically

You don't have to exactly know everything about React from the get-go; you will learn as you work with it continuously. Anyways, let's go ahead and start up the application with command:

```
npm start
```



If successful, the app will launch in browser on **localhost:3000.**
You have successfully created a work React App in a Liferay Workspace directory!

The Project

Now that we have created both a working Liferay Workspace and a new React App within it, we can begin to look into how to make them work together.

One thing to be pointed out is that **it will help to have a functional understanding of ReactJS in order to better understand this tutorial.** Things like knowing about states, props, components, etc.

For this project, we will be creating a simple **single page blog application** that displays blog posts hosted on tag's set in Liferay.

The data will be fetched using **Liferay headless API commands** (more on that later).

The front end design will be in **our React app.**

Simple Interface

For this project, we will follow a simple interface as pictured below.

This display is **separated into two main components:**

A place for the **listing of blog posts,** as indicated on the left
And a place for the **blog post contents, information and title,** as indicated on the right.

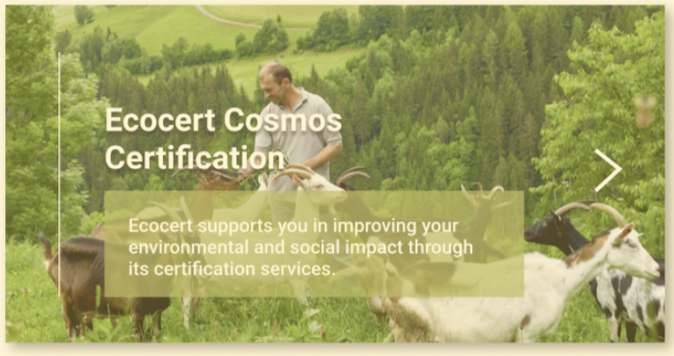
To create components and orchestrate how our website will better look, it’s important to conceputalize layout before we code.

This whole outlined area we will call the **BlogWindow**

The highlighted yellow area is the area for displaying **lists of all blogs, BlogList**

This highlighted pink area is the area to display **the corresponding blog selected from the left blog list, BlogContent**

Environmental Organizations



1 ›

When You Need A Realiable, International Body of Certification

By: Alexa Javellana
Published 02/25/2020

What is JOCA (Japan Organic Cotton Association)?
What is JOCA (Japan Organic Cotton Association)JOCA (Japan Organic Cotton Association) started as an association to think what we should protect and what we should do through organic cotton in this 21st century, and take action. It is an association contributing to the preservation of the global environment with the spread of cultivation of organic cotton and the method of minimizing the environmental stress on its production process.

About Organic Family Cotton Tag
About Organic Family Cotton TagAvanti is a member of Japan Organic Cotton Association. Its original brand PRISTINE was made by taking advantage of Japanese technology and sensibility in a way that is friendly to people and the environment, with the aim of spreading organic cotton based on JOCA Charter. The tags proving its firm safety and security are attached to all products before delivering to you.

In 1993, our representative, Chieko Watanabe established “Japan Texas Organic Cotton Association”, with the cooperation of nine domestic textile enterprises, and came in the board chairman. In 2000 it was promoted to “NPO Japan Organic Cotton Association (JOCA)”, and she came in the vice chairman. She uses all her strength for the enlightenment and spreading of organic cotton products that are relied by consumers. JOCA is aiming to increase the production of organic cotton by 10 percent, and spreading production methods and recycling system friendly to the earth. It is our action assignment to carry out activities that contribute to conservation of the global environment in

For more information about Japan Organic Cotton Association, refer to the official site.

Let's Begin

With a plan in mind, let's begin with a mini-lesson on how to correctly organize files in a simple ReactJS project with Liferay Workspace in order to better under the structure of our application.

Assuming that we are still in the Terminal running the npm command, **open a new Terminal tab/window and go ahead and cd back into our react-for-liferay-fe/reactapp directory.**

There, we will see the **src** folder. Go ahead and **cd into src**. (If you **pwd** or **dir**, your path should be something like **/Users/$username/react-for-liferay-fe/reactapp/src**)

In this **src** folder, if you ls, you will see a bunch of JS files, beginning with App or index. These files determine what you see when you go to the website at localhost:3030 in your browser. For this project, we will only need to see how to use Headless API's with ReactJS, so our setup will be simple.

```
Setups-MacBook-Pro:src setup$ pwd
/Users/setup/react-for-liferay-fe/reactapp/src
Setups-MacBook-Pro:src setup$ ls
App.css                 App.test.js             index.js                serviceWorker.js
App.js                  index.css               logo.svg                setupTests.js
```

Here, we will create a **components** folder.

The **components** folder in a ReactJS project contains the types of components you create to integrate into your application/website. This can range from things like **forms, cards, buttons, and more.** For this project, we will start by creating **a Blogs component.**

src

```
mkdir components
```

src/components

```
mkdir Blogs
```

Now, we have a folder called **Blogs** to hold the code for our custom Blog component in the **src/components** folder.

In **Blogs,** let's create a simple React JS file to act as the root code for our component. Create a file in **Blogs** called **BlogWindow.js, open the project and BlogWindow.js file in your favorite code editor,** and in the new file copy and paste the code from the **skeleton.txt** under the **BlogWindow** portion into it.

(**Skeleton.txt is located in the reactapp/src/ folder**)

It should look like this:

reactapp > src > components > Blogs > JS BlogWindow.js > ...

```
import React from 'react';

export default class BlogWindow extends React.Component {
  constructor(props) {
    super(props);
    this.state = {

    }
  }

  render() {
    return (
      <div className="blogContainer">
       <div>Hello!</div>
      </div>
    )
  }
}

```

How do we view this component in our React app?

For this case scenario, we will simply replace the contents of our **src/index.js** file to render **BlogWindow.js.** This is done by **replacing the code in src/index.js with the skeleton code in skeleton.txt for index.js.**

Now, **index.js** should look like this:

reactapp > src > JS index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

import BlogWindow from './components/Blogs/BlogWindow';

ReactDOM.render(<BlogWindow />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

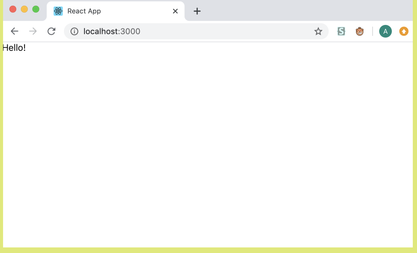
What did we do exactly?

In this bare bones application, **index.js** is responsible for generating the initial HTML you see on your screen at localhost:3030 (**at index.html**). In the above code, we are calling the ReactDOM to render the component **BlogWindow.js** from the **BlogWindow** component, in this case simply being the returned in **BlogWindow.js.** That's why we also have to import BlogWindow in index.js as indicated on the top of the index.js file.

Now, refresh your **localhost:3000** or start your React project if not running (make sure you're somewhere in the react-for-liferay-fe directory!) by the command:

react-for-liferay-fe/..

```
npm start
```



Generating Data

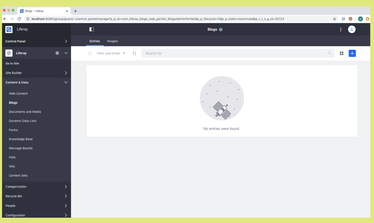
So now that we have our basic structure, let's see how we integrate Liferay Headless API's into the mix.

First, we have to create data that we will fetch.

In this project, we will be dealing with **Blog data.**

In your browser, **after making sure your local Liferay Workspace is running,** go ahead and open up **localhost:8080** and log into your administrative account (hopefully, test@liferay.com).

Once logged in, navigate to open up the **Control Panel**, and redirect to **Content & Data -> Blogs**



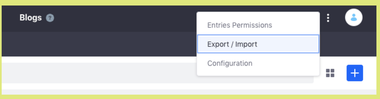
You should have a blank screen with no Blogs if you are in your initial Liferay Workspace.

In this **Blogs** page, we will have the blog posts that we will need to fetch and display in our React app. I've created a few blogs (6) to test out for this tutorial.

In Liferay, you can import data files called **Lars into Liferay and get viable entries of data to test with Headless.**

So, with that being said, let's upload out Blog posts. Go ahead and download the **ReactBlogs.portlet.lar** file from the **react-for-liferay-fe/reactapp/ directory.**

Now, in your **localhost:8080 Blogs page, click on the three dots on the upper right hand corner and select Export/Import:**

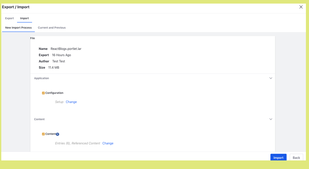


On selecting Export/Import, a new window will appear. In that window, select **Import** and in the Import screen, in the **Drop a Lar File or Select File section, go ahead and upload the ReactBlogs.portlet.lar you have in the react-for-liferay-fe/reactapp/lars folder (or wherever you downloaded it):**

Export / Import : Upload

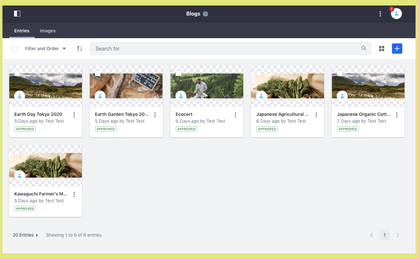
Once uploaded, click on the **Continue** button for the window to process the lar file:

Export / Import : Continue to Import



Then, we finally click on **Import** and wait a few seconds for everything to process correctly:

If successful, if you go back to the main **Blogs** page, you will see the following:

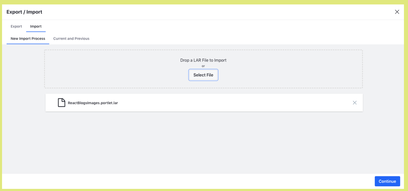


Now we have the main Blog data up. If you click on any of the Blogs, you should see their contents: all have a basic Title, subtitle, description and one word in their Keywords value (either 'environmental organization' or 'volunteer').

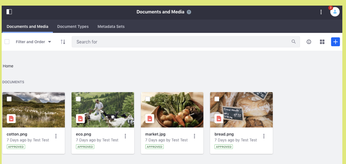
One thing you might notice as well is the these Blog posts have **images.** These images will be pulled from the Documents and Media section, so we will have to upload these images.

Go ahead to the **Documents and Media section,** which should be right underneath **Blogs.** Again, in the Documents and Media section this time, click on the three dots in the upper right hand corner, then select Export/Import.

In the **Import screen, upload and import ReactBlogImages.portlet.lar from your lar/ folder or wherever you downloaded it:**



Just like before, go ahead and **Continue -> Import.** Once successful, your **Documents and Media** section should look like this:



Now, we have our fully updated data.

The next question that follows: how do we access it in React?

Where Are The API's?

With our data set and our component skeleton's in place, we can finally start to think about **Headless API's.**

What exactly is a Headless API is rather complex to explain, but for the general purpose of this exercise, I will limit it to this explanation:

Headless API:
A link that acts as a service to get and return JSON data hosted on our Liferay Workspace.

With **Liferay running** on localhost:8080, let's try to see if we can find an example.

In your browser, go ahead and go to link:
http://localhost:8080/api/jsonws

At this link, you will be able to peruse through the many **Liferay provided Headless API's that you can use in any project to interact with the Workspace to return data to your project.**

In this tutorial, we will be dealing with **blogs** API's.

If you go to **Context Name** and select **blogs** from the dropdown that appears, you will see a bunch of new methods underneath the **Search bar that are methods of interacting with data in Blogs Liferay Workspace.**

Additionally, at

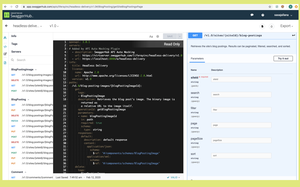
https://app.swaggerhub.com/apis/liferayinc/headless-delivery/v1.0#/

You can also see and interact (a more limited) amount of API's for Liferay.

At localhost:8080/api/jsonws, there are a bunch more API's than in the swaggerhub.com link as provided directly above, but in this case, it's **much easier to use swaggerhub.com for visual follow-through.**

So, with that being said, let's choose an API to focus on.

Go to the link:
https://app.swaggerhub.com/apis/liferayinc/headless-delivery/v1.0#/BlogPosting/getSiteBlogPostingsPage



This should lead you to the this API:

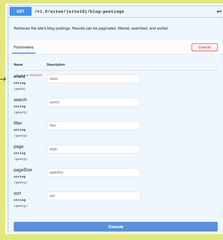
that we will be using in order to get
data of the blogs we created in our local Liferay Workspace from uploading the Lar files in the last part of this tutorial.

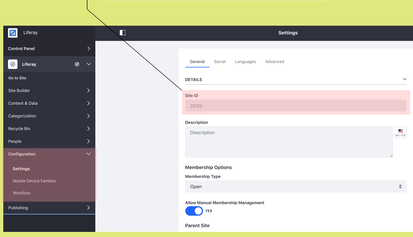
Before we start using this, one of the first things you should do in swaggerhub hub is **change the server.** Do this by **in the blue area/rightmost area, scroll completely upwards. Under the "Servers" drop down, choose and change the address to**

https://localhost:8080/o/headlessdelivery

By changing the address, it will allow us to access data stored at the localhost Liferay instance.

Now, go ahead and scroll back down to the **GET API for blog-postings** and click on the **Try it out** button.





Once you click on **Try it out,** your going to notice the only required field is the **siteId.** You will need the **Site ID of the site you uploaded your blog data to, in this case the default Liferay site that was created on configuration.**

To find the Site ID, it will be under **Configuration -> Settings on the right hand menu in Liferay Workspace.** Use the Site ID (as indicated in the red box above) in the **siteId requirement input on swaggerhub.**

Once you put in the siteId, go ahead and **Execute the API.**

On **Execute,** you will get a **Curl** response. Copy, paste and run it in the terminal to see the JSON of all the blog posts we uploaded under Blogs in Liferay Workspace!

More than likely, an error will occur if you simply copy paste into Terminal as is.

Fix 1:
Change link from "https://localhost..." to **"http://localhost..."**, as our basic workspace localhost:8080 is more likely HTTP not HTTPS.

Fix 2:
Use **-u** parameters to **log admin credentials if there is an access error.**

Fixed CURL Link

curl -X GET
"**http**://localhost:8080/o/headless-delivery/v1.0/sites/20123/blog-postings" -H "accept: aplication/json" **-u test@liferay.com:test**

Utilizing API's

Now that we have an API we can work with and know what type of data is returned on a successful GET, we can integrate it into our **Blog** code back inside of our **create-react-app reactapp.**

If not already, **go ahead and reopen your react-for-liferay-fe project in a code editor of your choice (VS in this tutorial).**

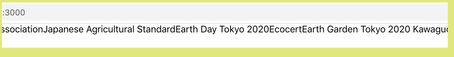
Navigate back to the **BlogWindow.js** component. It should still have the code previously entered from the earlier part of this tutorial, derived from skeleton.txt.

Now, we will add a function that will use the **headless API.**

From the skeleton.txt, copy the portion of code between the **BlogWindow.js API** portion and replace whats currently in the **BlogWindow.js with it.**

Now, your code should look like this:

If you refresh the page in the React localhost:3030/3000, you should see a list of all the blog titles appear in one uniform line (if left unstyled):



Problems Encountered

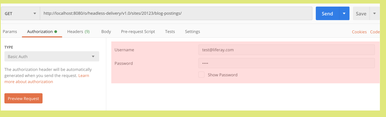
More than likely, it won't be that straightforward to get the results we want; we are going to have to make a few changes that will differ from each individual going through this tutorial.

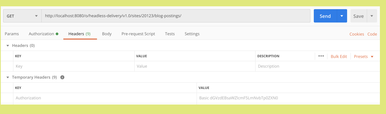
Scenario 1: Incorrect Authentication

In the **getAllBlogs()** function that we add into **BlogWindow,** there is a portion of the **fetch** that is labeled **"headers"** and under that, we have **Authorization.**

In this code, our string for Authorization is:
"**Basic dGVzdEBsaWZlcmF5LmNvbTp0ZXN0"**

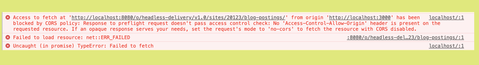
But, for your Liferay instance, this will be different. Use and test the fetch URL in something like **Postman**, and get the generated header from setting up a simple Basic Authrization to use instead.





Scenario 2: CORS

If you view in the **Console from localhost:3000 an error like this:**



It mentions something about **CORS. CORS is** Web Contexts Cross Resource Origin Sharing (CORS), and this needs to be enabled in order for the Liferay instance and React application to talk to each other/pass data. To enable this, go back on the Liferay Workspace under

Configuration -> System Settings -> Security Tools

There, you will see this page:

On this page, to enable **CORS,** go ahead and press the **Add button. Once clicked,** it will bring you to another form pre-filled. Go ahead and **Save** the default that comes up.

Now, if you refresh the page, you should see the titles show up if everything else is okay!

So What Did We Do?

We managed to integrate a Liferay Headless API into our React application in order to display blog data we retrieved locally!

The **getAllBlogs()** function uses a **fetch paired with Authentication to talk to the Liferay Workspace.**

The returned data JSON is then **set into a state in BlogWindow.js, which is then rendered to the browser screen.**

Further Styling and Testing

Up to now, we are basically done with the main part of the tutorial.

This integration is still a work in progress, but will built on further.

This project is continued further in Github repo:

https://github.com/sasajallana/introduction

Where the **Blog component** is built on further with more components using the headless API.

To view and test that code, replace the
react-for-liferay-fe/reactapp/src/components/Blogs folder with the introduction/reactapp/src/components/Blogs folder.

You will probably have to change a few things, such as the
Authentication in header anytime we use a fetch call.

今のところそれは以上です！