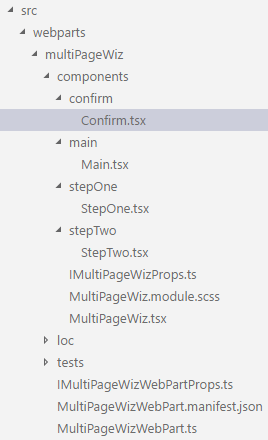
# Tutorial 1: SharePoint Multi-Step Wizard Using React and TypeScript

Figure 1: Project structure

1. Create a SharePoint project with React using Yeoman generator: <https://dev.office.com/sharepoint/docs/spfx/web-parts/get-started/build-a-hello-world-web-part>.
2. Start the sharepoint framework in thest environment using gulp serve. It will automatically start the app in the browser at <https://localhost:4321/temp/workbench.html>.
3. Structure your project.

Create a folder under components for every step of your wizard. In each folder create a .tsx file like seen on **Figure 1**. Note that the order is alphabetic, the real order of the wizard is **Main -> StepOne -> StepTwo -> Confirm**.

1. Yeoman automatically created an app that renders from multiPageWiz.tsx under multiPageWiz folder. As we want to make a multiple view app, we put all the views In separate tsx files and treat them as separate components, including the main.

You can replace all the code in multiPageWiz.tsx with the following:

import \* as React from 'react';

import { IMultiPageWizProps } from './IMultiPageWizProps';

export default class SpGroup extends React.Component<IMultiPageWizProps, void> {

public render(): JSX.Element {

return (

<div>

</div>

);

}

}

Note that right now the app is a blank sheet. Don’t worry, we will fill it up.

1. Go to Main and replace the code with the following:

import \* as React from 'react';

import styles from '../MultiPageWiz.module.scss';

import MultiPageWiz from '../MultiPageWiz';

export default class Main extends React.Component<any, any> {

public render(): JSX.Element {

return (

<div>

<div className={styles.helloWorld}>

<div className={styles.container}>

<div className={`ms-Grid-row ms-bgColor-themeSecondary ms-fontColor-white ${styles.row}`}>

<span className="ms-font-xl ms-fontColor-white">

<b>Create a group for your client!</b>

</span>

</div>

</div>

</div>

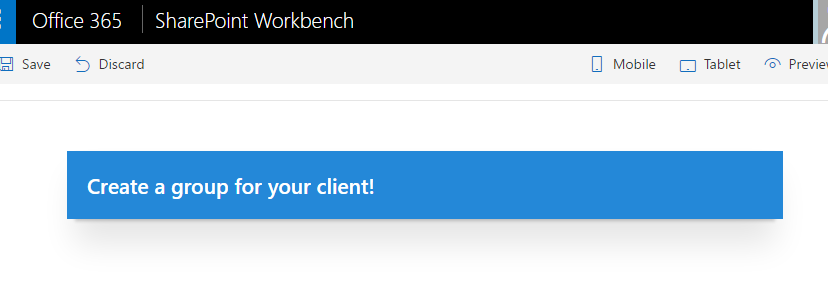
</div>

);

}

}

There is still nothing rendered, because you need to render it to MultiPageWiz.tsx. Open the file and import Main (import Main from './main/Main';). Then add <Main /> in between the div’s. If you save the file you can see that an image appears on the app page. This is the html that was added to Main!



1. You have successfully rendered one component. Now, how to move on to the next? Quite obviously, you need a “Next” or a “Start” button that will take you to the next view. Create the button to Main under span with the following html:

<div className='ms-Grid-col ms-u-lg10 ms-u-xl8 ms-u-xlPush2 ms-u-lgPush1'>

<a href="#" className={`${styles.button}`}>

<span className={styles.label}>START</span>

</a>

</div>

Note that this button currently takes you nowhere, because there is no function yet that will do that. But first, let’s create the second view.

* 1. Open StepOne.tsx and add the following code:

import \* as React from 'react';

import styles from '../MultiPageWiz.module.scss';

import MultiPageWiz from '../MultiPageWiz';

import Main from '../main/Main';

export default class StepOne extends React.Component<any, any> {

public render(): JSX.Element {

return (

<div>

<span className="ms-font-xl ms-fontColor-white">

<b>Good yob! Your 'start' button worked! </b>

</span>

</div>

);

}

}

Note that this is practically a copy-paste of the Main component, only the class names, html and imports differ a bit. **This is basically a blueprint of any component, so keep that in mind when creating next ones**.

* 1. Go to Main and create a router. This is a function that checks where you are at the moment and chooses what to display. First, create a constructor for state that shows where you are at and has a default value of 0. Add the following code just above the render method:

constructor(props) {

super(props);

this.state = {step: 0};

}

Then create a function called nextStep() that increments the state value with each click:

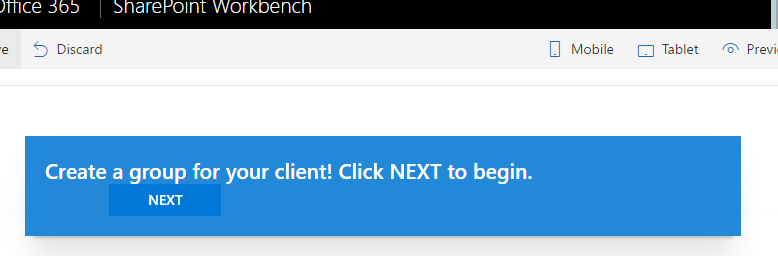
private nextStep(): void {

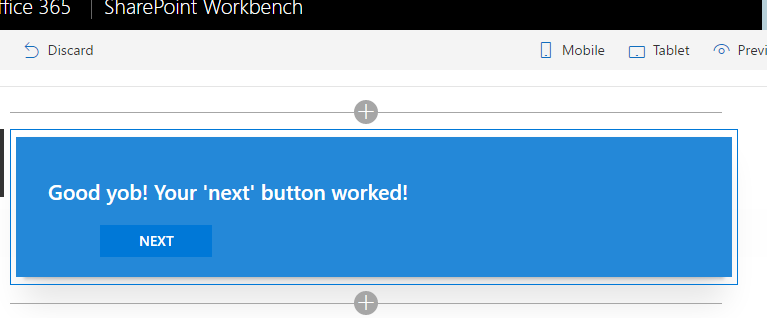
this.setState({step: this.state.step + 1});

console.log(this.state.step);

}

Then add the function to the ‘next’ button (<a href="#"className={`${styles.button}`} onClick={() => this.nextStep()}>…</a>). Now when you click the button, nothing visible happens, but you can see in the console that each time you press the button it prints out a number incremented by 1. To bind this to the view, you must tell the main what to render. First, add the condition to the first view that is in the <span>. Do it like this: { this.state.step < 1 && <span>…</span> }. To add the second view, render StepOne below the <span> like this: { this.state.step == 1 && <StepOne /> } and don’t forget to import it. Press the next button and you should see the view change!





PS! You can add a ’back’ or ‘previous’ button by using a similar function that decrements the state value by 1. If you don’t want it to render in the first view, simply restrict it with a rule that allows rendering only if state value is larger than 0. To prohibit rendering of the ‘next’ button on the last view, just add a rule that prohibits it if the state value is larger than 2 (or allow it until state value is smaller than 3!).

# Tutorial 2: Adding input fields to wizard

Now you should have created all the components you want to use in your wizard and integrated them to the main. The next step is to add input fields and bind them to the data in SharePoint. **Make sure you are now running your application in the SharePoint online test environment**, because the REST service won’t work in local. If you are not sure how to do so, please refer to the section “Preview the web part in SharePoint” [here](https://dev.office.com/sharepoint/docs/spfx/web-parts/get-started/build-a-hello-world-web-part). Also, please create a list (my list’s name will be ‘Clients’) in your SharePoint tenant to test the app.

My wizard is going to retrieve a specific list from SharePoint through the SharePoint REST service. Follow this link for more information: <https://dev.office.com/sharepoint/docs/sp-add-ins/working-with-lists-and-list-items-with-rest>.

1. First, let’s create a dropdown menu to StepOne with the <select> tag. As we don’t have any options yet, the dropdown is empty.
2. Go to main and create an interface for the list items like this:

export interface **IListItem** {

Title?: string;

}

PS! As the IListItem will store values that are needed in multiple components, it must be kept in one place. In this example, we store that kind of code in Main.

1. Import it to StepOne (with curly braces, like this: import { IListItem } from '../main/Main';) and construct a generic component named *items* under the StepOne’s render method:

const items: **JSX**.**Element**[] = this.state.listItems.**map**(

(item: **IListItem**, i: number): **JSX**.**Element** => {

return (

<option *value*={item.Title}>{item.Title}</option>

);

}

);

This creates the <option> tag that will later be rendered between the <select> tags.

1. To put this aside for a minute, we first need to get the clients from the list. For this, we need to make a method under StepOne that would retrieve that list from your SharePoint account and put the list items in the dropdown. This is a bit complicated so I will break it into parts.
   1. First, you need to make sure you have all the props you need. Go to MultiPageWizWebPart.tsx under the multiPageWiz folder. Currently it holds the code for the property pane. You can use it later to create configuration settings for your app, but right now you can delete or ignore everything that is below getPropertyPaneConfiguration(). Important place here is the render method. This doesn’t exactly render anything, but passes on properties to main.

In order to pass on the properties, they must be defined in the IMultiPageWizProps.ts in the components folder. Paste this code:

import { HttpClient, SPHttpClient } from '@microsoft/sp-http';

export interface **IMultiPageWizWebPartProps** {

spHttpClient: **SPHttpClient**;

httpClient: **HttpClient**;

siteUrl: string;

}

Then amend the MultiPageWizWebPart.tsx render method like this:

…

MultiPageWiz,

{

spHttpClient: this.context.spHttpClient,

httpClient: this.context.httpClient,

siteUrl: this.context.pageContext.web.absoluteUrl

}

…

* 1. Then, import SPHttpClient and SPHttpClientResponse from @microsoft/sp-http.
  2. Then create a method that uses the REST service for reading the clients in the list. I called my method readItems():

private **readItems**(): void {

this.props.spHttpClient.**get**(

`${this.props.siteUrl}/\_api/web/lists/getbytitle('Clients')/items?$select=Title’,

SPHttpClient.configurations.v1, {

headers: {

'Accept': 'application/json;odata=nometadata',

'odata-version': ''

}

})

}

Note that *this.props.siteUrl* is the url of the site you are currently at.

*‘Clients’* in the API is the name of my list that I am fetching. */items?$select=Title* fetches the title column in the list. That’s sufficient , as we only need client names.

HttpClientConfigurations.v1 provides standard predefined *HttpClientConfiguration* objects for use with the *HttpClient* class. You can read about the sp-http classes [here](https://dev.office.com/sharepoint/reference/spfx/sp-http-module).

* 1. Then add a response. Simply put it after *spHttpClient.get*:

.**then**((response: **SPHttpClientResponse**): **Promise**<{ value: **IListItem**[] }> => {

return response.**json**();

})

In my example I use JSON as the response format, you can also fetch it in xml if it suits you better.

* 1. In order to be able to store the fetched list items in the array, you need to define it’s state. For this create a constructor in the StepOne class:

constructor(props) {

super(props);

this.state = {

listItems: []

}

this.**readItems**();

}

* 1. Then you store the fetched list items to the created listItems array by amending the following to your readItems method:

.**then**((response: { value: **IListItem**[] }): void => {

this.**setState**({

listItems: response.value

});

}, (error: any): void => {

**console**.**log**(error);

});