Global Office Development BootCamp

Extensions

Índice

[Preparación del entorno 4](#_Toc497497696)

[Creación de la primera Extensión 5](#_Toc497497697)

[Como depurar una extensión fácil 8](#_Toc497497698)

[Crear una línea de migas 10](#_Toc497497699)

[Creación Custom Action 15](#_Toc497497700)

# Preparación del entorno

El Spfx está basado en un modelo de desarrollo web moderno haciendo uso de las últimas herramientas de Front-End para ello antes de empezar con el desarrollo tendremos que tener listo el entorno.

Para ello en primer lugar instalaremos NodeJS <https://nodejs.org/en/>

A continuación, abrimos un terminal (MS-DOS,PowerShell, etc) e instalaremos Yeoman y Gulp con la siguiente instrucción:

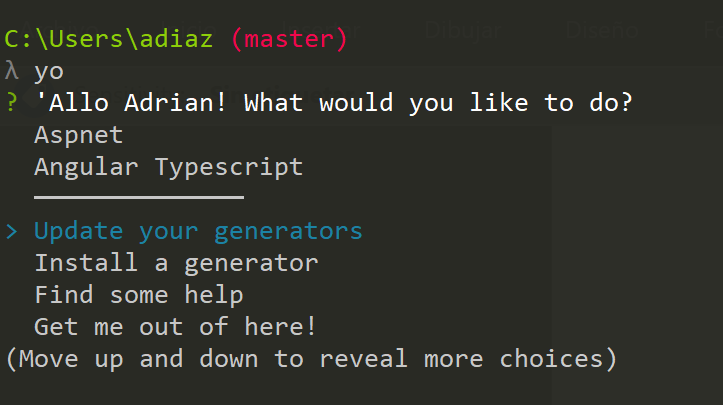
npm install -g yo gulp

El siguiente paso sería instalar la plantilla del SharePoint Framework que nos ha proporcionado Microsoft. Para ello ejecutaríamos la siguiente línea:

npm install -g @microsoft/generator-sharepoint

En caso de que ya tengamos instalado el generador es conveniente comprobar que tenemos la última versión, para ello tendríamos que ejecutar:

Yo y seleccionar la opción de update your generators



# Creación de la primera Extensión

1. Creamos un nuevo proyecto en nuestra ubicación favorita

md global

1. Vamos al directorio que acabamos de crear

cd global

1. Creamos la Extensión utilizando la plantilla de Yeoman

yo @microsoft/sharepoint

1. Cuando se muestre

Aceptamos el nombre de la solución

Seleccionamos SharePoint Online only (latest)

Seleccionamos Create a subfolder with solution name

Seleccionamos N

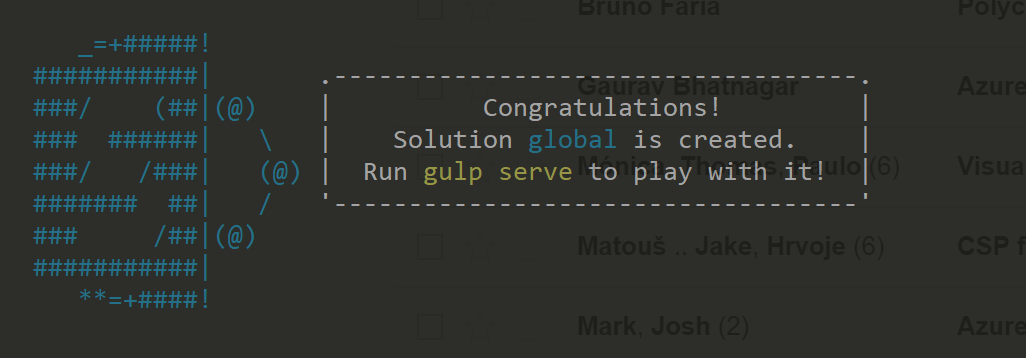
Seleccionamos Extension como el tipo de componente que se ha creado

Seleccionamos “Applicattion Customizer” como tipo de extensión para crear

1. Las siguientes características son relativas sobre la Extension.

Aceptamos el nombre por defecto de la Extension y apretamos Enter

Aceptamos el nombre de la descripción de la extensión y apretamos Enter

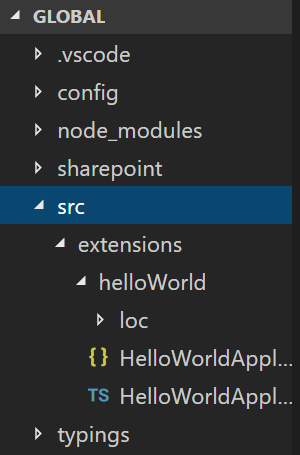


1. A continuación, ejecutamos la siguiente instrucción para verificar las dependencias que tiene el proyecto.

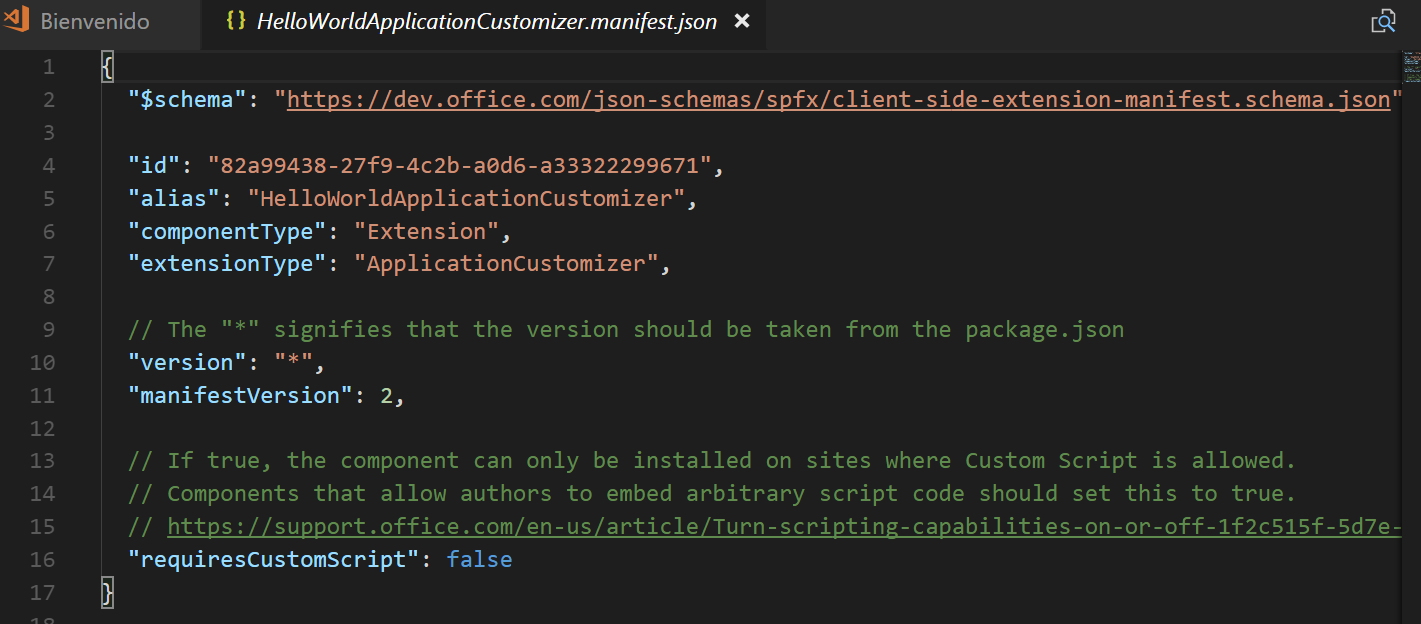
npm shrinkwrap

1. Arrancamos Visual Studio Code

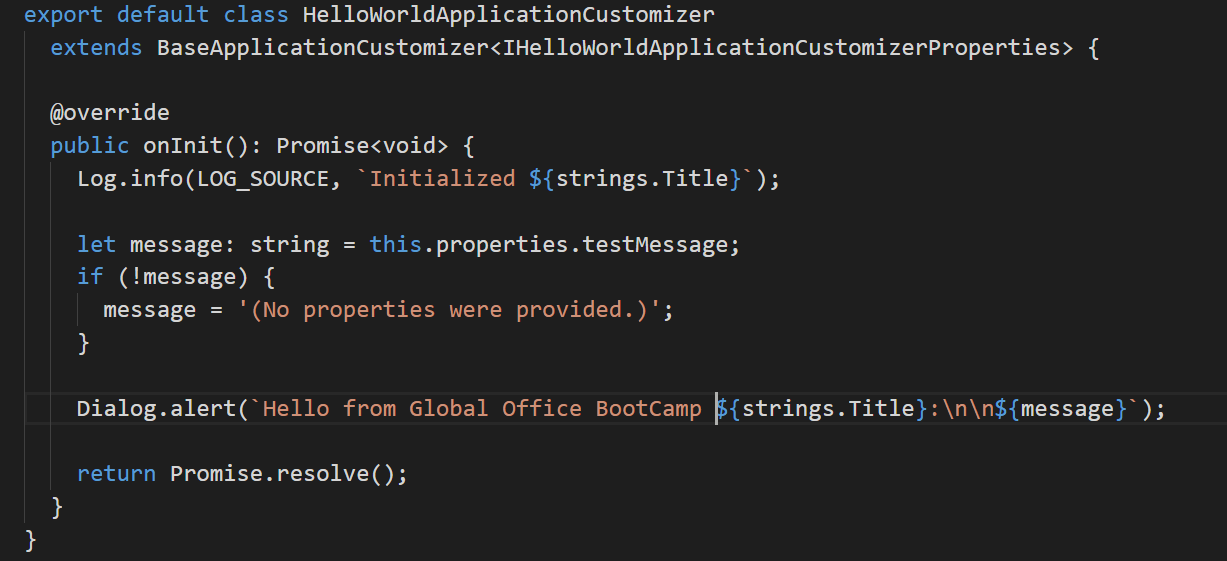
code .



1. Abrimos el fichero Open **HelloWorldApplicationCustomizer.manifest.json** que se encuentra dentro de src\extensions\helloWorld folder. Dentro de este fichero buscamos el Identificador que lo vamos a utilizar posteriormente.



1. Agregaremos lógica a nuestro desarrollo para ello abriremos el fichero **HelloWorldApplicationCustomizer.ts que está dentro de la carpeta src\extensions\helloWorld y dentro del método OnInit es donde pondremos la lógica de nuestra aplicación.**



1. Depuramos nuestra aplicación

Gulp serve --nobrowser

1. Para depurar nuestra aplicación tendremos que ir a nuestro entorno de SharePoint y agregar los siguientes parámetros a la Query String.

?loadSPFX=true&debugManifestsFile=https://localhost:4321/temp/manifests.js&customActions={"e5625e23-5c5a-4007-a335-e6c2c3afa485":{"location":"ClientSideExtension.ApplicationCustomizer","properties":{"testMessage":"Hello as property!"}}}

Donde

loadPSFX=true -> Asegura que el SPFX esta cargado en la página

debugManifestsFile- > Especifica que componentes quieres cargar de tu servidor local

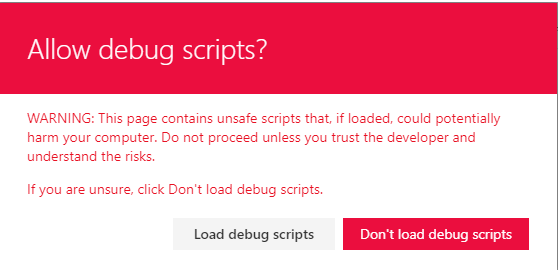
CustomAction ->

Key= usa el guid de la extensión asociada

Location: Tipo de custom action que utilizamos

Propertis

1. Cuando se cargue la página mostrará un mensaje como el siguiente:



1. Seleccionamos Load Debug Script y se mostrará un alert por pantalla

# Como depurar una extensión fácil

1. Crearemos una carpeta llamada gulp-tasks, dentro de dicha carpeta nos creamos un fichero llamado gulpfile-serve-info.js que tendrá el siguiente contenido:

'use strict';

const build = require('@microsoft/sp-build-web');

const common = require("@microsoft/sp-build-common");

const originalBundleTask = build.rig.getBundleTask;

build.rig.getBundleTask = function () {

const originalTask = originalBundleTask.apply(build.rig);

return common.serial(originalTask, serveInfo);

}

const serveInfo = build.subTask('serve-info', (gulp, config, cb) => {

var serveTask = config.uniqueTasks.find((task) => {

return task.name === 'serve' /\* SPFx < 1.2.0 \*/ ||

task.name === 'spfx-serve' /\* SPFx >= 1.2.0 \*/ ;

});

var url = `http${serveTask.taskConfig.https ? 's' : ''}://${serveTask.taskConfig.hostname}:${serveTask.taskConfig.port}/temp/manifests.js`;

for (var key in config.properties.manifests) {

var manifest = config.properties.manifests[key];

if (manifest.componentType !== 'Extension') {

continue;

}

console.log(`${manifest.alias}:`);

switch (manifest.extensionType) {

case "ApplicationCustomizer":

console.log(`?loadSPFX=true&debugManifestsFile=${url}&customActions={"${manifest.id}":{"location":"ClientSideExtension.ApplicationCustomizer","properties":{"prop1":"val1"}}}`);

break;

case "FieldCustomizer":

console.log(`?loadSPFX=true&debugManifestsFile=${url}&fieldCustomizers={"FieldName":{"id":"${manifest.id}","properties":{"prop1":"val1"}}}`);

break;

case "ListViewCommandSet":

console.log(`?loadSPFX=true&debugManifestsFile=${url}&customActions={"${manifest.id}":{"location":"ClientSideExtension.ListViewCommandSet.CommandBar"}}`);

break;

}

console.log();

}

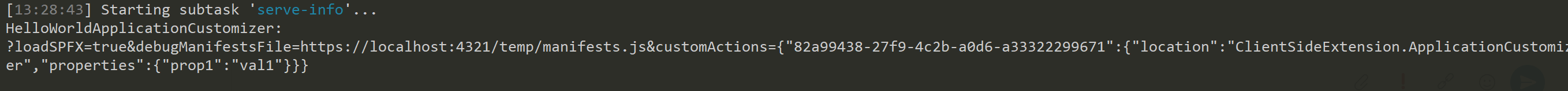
cb();

});

1. A continuación, tendremos que indicar dentro del fichero gulpfile que va a cargar una Tarea más a la hora de lanzar la compilación. Para ello vamos al fichero gulpfile.js y agregamos la tarea que acabamos de crear:

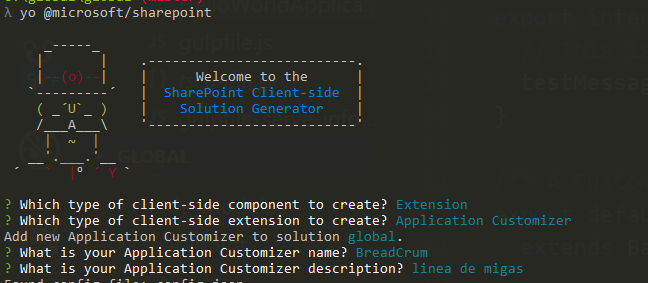
require('./gulp-tasks/gulpfile-serve-info');

1. Arrancamos el servidor Gulp local y si todo ha ido bien se muestra las url que necesitamos para depurar las Extensions que tengamos dentro de la solución:



# Crear una línea de migas

Sobre la solución anterior crearemos una nueva Extension, para ello nos pondremos en el directorio de la solución y ejecutamos el siguiente comando:



Al abrir Visual Studio Code, tendremos una nueva carpeta dentro de las Extensions, dentro de dicha Carpeta vamos a implementar un componente ReactJS. Por lo que dentro de BreadCrum nos creamos una carpeta llamada components con tres ficheros: ISiteBreadCrumb.ts, SiteBreadCrumb.tsx y SiteBreadCrumb.module.scss.Estos tendrán el siguiente contenido.

ISiteBreadCrumb.ts,

import ApplicationCustomizerContext from "@microsoft/sp-application-base/lib/extensibility/ApplicationCustomizerContext";

import { IBreadcrumbItem } from "office-ui-fabric-react/lib";

export interface ISiteBreadcrumbProps {

context: ApplicationCustomizerContext;

}

export interface ISiteBreadcrumbState {

breadcrumbItems: IBreadcrumbItem[];

}

export interface IWebInfo {

Id: string;

Title: string;

ServerRelativeUrl: string;

error?: any;

}

SiteBreadCrumb.tsx

import \* as React from "react";

import { ISiteBreadcrumbProps, ISiteBreadcrumbState, IWebInfo } from "./ISiteBreadcrumb";

import { Breadcrumb, IBreadcrumbItem } from 'office-ui-fabric-react/lib/Breadcrumb';

import { SPHttpClient, HttpClientResponse } from "@microsoft/sp-http";

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

export default class SiteBreadcrumb extends React.Component<ISiteBreadcrumbProps, ISiteBreadcrumbState> {

private \_linkItems: IBreadcrumbItem[];

constructor(props: ISiteBreadcrumbProps) {

super(props);

// Initiate the private link items variable

this.\_linkItems = [];

// Initiate the component state

this.state = {

breadcrumbItems: []

};

}

/\*\*

\* React component lifecycle hook, runs after render

\*/

public componentDidMount() {

// Start generating the links for the breadcrumb

this.\_generateLinks();

}

/\*\*

\* Start the link generation for the breadcrumb

\*/

private \_generateLinks() {

// Add the current site to the links list

this.\_linkItems.push({

text: this.props.context.pageContext.web.title,

key: this.props.context.pageContext.web.id.toString(),

href: this.props.context.pageContext.web.absoluteUrl,

isCurrentItem: !this.props.context.pageContext.list.serverRelativeUrl

});

// Check if the current list URL is available

if (!!this.props.context.pageContext.list.serverRelativeUrl) {

// Add the current list to the links list

this.\_linkItems.push({

text: this.props.context.pageContext.list.title,

key: this.props.context.pageContext.list.id.toString(),

href: this.props.context.pageContext.list.serverRelativeUrl,

isCurrentItem: true

});

}

// Check if you are already on the root site

if (this.props.context.pageContext.site.serverRelativeUrl === this.props.context.pageContext.web.serverRelativeUrl) {

this.\_setBreadcrumbData();

} else {

// Retrieve the parent webs information

this.\_getParentWeb(this.props.context.pageContext.web.absoluteUrl);

}

}

/\*\*

\* Retrieve the parent web URLs

\* @param webUrl Current URL of the web to process

\*/

private \_getParentWeb(webUrl: string) {

// Retrieve the parent web info

const apiUrl = `${webUrl}/\_api/web/parentweb?$select=Id,Title,ServerRelativeUrl`;

this.props.context.spHttpClient.get(apiUrl, SPHttpClient.configurations.v1)

.then((response: HttpClientResponse) => {

return response.json();

})

.then((webInfo: IWebInfo) => {

if (!webInfo.error) {

// Check if the correct data is retrieved

if (!webInfo.ServerRelativeUrl && !webInfo.Title) {

this.\_setBreadcrumbData();

return;

}

// Store the current site

this.\_linkItems.unshift({

text: webInfo.Title,

key: webInfo.Id,

href: webInfo.ServerRelativeUrl

});

// Check if you retrieved all the information up until the root site

if (webInfo.ServerRelativeUrl === this.props.context.pageContext.site.serverRelativeUrl) {

this.\_setBreadcrumbData();

} else {

// retrieve the information from the parent site

webUrl = webUrl.substring(0, (webUrl.indexOf(`${webInfo.ServerRelativeUrl}/`) + webInfo.ServerRelativeUrl.length));

this.\_getParentWeb(webUrl);

}

} else {

// Set the current breadcrumb data which is already retrieved

this.\_setBreadcrumbData();

}

});

}

/\*\*

\* Set the current breadcrumb data

\*/

private \_setBreadcrumbData() {

this.setState({

breadcrumbItems: this.\_linkItems

});

}

/\*\*

\* Default React component render method

\*/

public render(): React.ReactElement<ISiteBreadcrumbProps> {

return (

<div className={styles.breadcrumb} >

<div className="ms-bgColor-themePrimary">

<Breadcrumb

items={this.state.breadcrumbItems}

ariaLabel={'Website breadcrumb'}

className={styles.breadcrumbLinks} />

</div>

</div >

);

}

}

SiteBreadCrumb.module.scss

.breadcrumb {

>div {

padding-left: 7px;

}

div[role="navigation"] {

margin-top: 0;

}

.breadcrumbLinks {

ol {

li {

a {

color: #fff;

font-size: 14px;

padding: 4px 8px 8px 8px;

&:hover {

background-color: #fff;

}

}

i {

color: #fff;

font-size: 10px;

margin: 10px 0;

}

}

}

}

}

A continuación hay que invocar dicho Componente desde la Extensión para eso en primer lugar hay que añadir las referencias de dichos modulos para ello añadiremos el siguiente código:

import SiteBreadcrumb from './components/SiteBreadCrumb';

import { ISiteBreadcrumbProps } from './components/ISiteBreadCrumb';

import \* as React from "react";

import \* as ReactDom from 'react-dom';

También vamos ha utilizar los placeholder habilitados para poder añadir nuestro componente para ello hay que reemplazar la referencia de sp-application-base por la siguiente:

import {

BaseApplicationCustomizer,

PlaceholderContent,

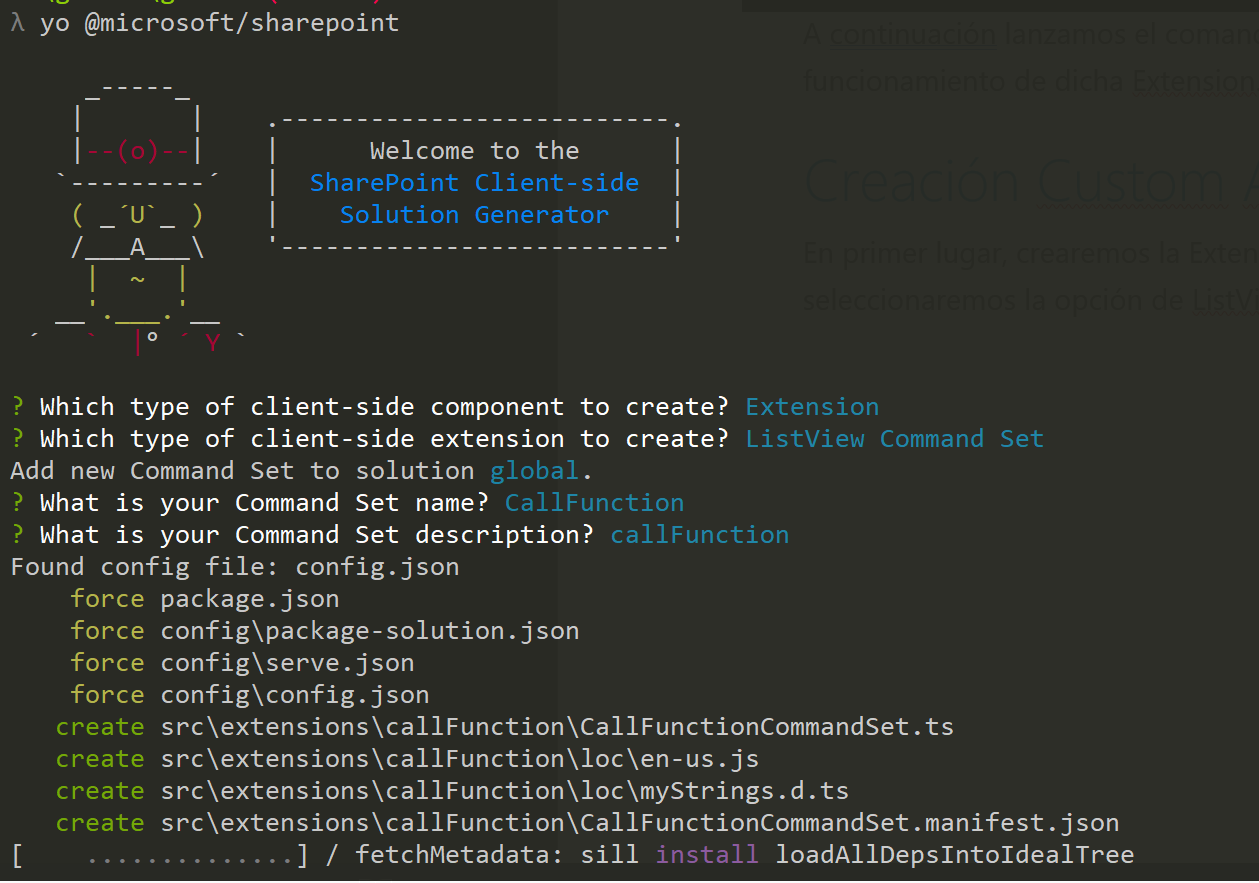
PlaceholderName

} from '@microsoft/sp-application-base';

A continuación lanzamos el comando gulp server –nobrowser y verificamos el correcto funcionamiento de dicha Extension.

# Creación Custom Action

En primer lugar, crearemos la Extensión utilizando yo @microsoft/sharepoint y seleccionaremos la opción de ListView Custom Action.



Dentro de dicha clase vamos a añadir el siguiente contenido:

protected functionUrl: string = "https://spointelligence.azurewebsites.net/api/News?code=1OFts7dd2OaMT2z/bKr7KDHKL/q3Yf0OiN9iRSfemeWPaJIM14e8nQ==";

protected runFunction(id:string): void {

const requestHeaders: Headers = new Headers();

requestHeaders.append("Content-type", "application/json");

requestHeaders.append("Cache-Control", "no-cache");

let noticia:string="Images";

const postOptions: IHttpClientOptions = {

headers: requestHeaders,

body: `{ Id: '${id}', Lista:'${noticia}',site:'${this.context.pageContext.site.absoluteUrl}' }`

};

debugger;

let responseText: string = "";

let resultMsg: HTMLElement = document.getElementById("responseContainer");

this.context.httpClient.post(this.functionUrl, HttpClient.configurations.v1, postOptions).then((response: HttpClientResponse) => {

response.json().then((responseJSON: JSON) => {

responseText = JSON.stringify(responseJSON);

Dialog.alert("Contenido Analizado"+ responseText);

})

.catch ((response: any) => {

let errMsg: string = `WARNING - error when calling URL ${this.functionUrl}. Error = ${response.message}`;

console.log(errMsg);

});

});

}

Y ahora lo que nos queda es invocar este método desde el propio Custom Action. Para ello dentro del custom Action que estamos personalizando añadiremos la siguiente información:

if (event.selectedRows.length >= 0) {

const id: string = event.selectedRows[0].getValueByName("ID");

this.runFunction(id);

}

Si lo ejecutamos hay un error… Sabéis cual?