Progress

PREPARING YOUR TOOLBOX FOR THE SHAREPOINT FRAMEWORK

WITH ANGULAR, WEBPACK AND KENDO UI

Contents

Introduction 4

Future of SharePoint Development 4

Selecting our Web Stack and UI Library 7

Building the App: Contract Register 10

- The App and the App Component 13
- The Contract List 14
- The Modal Service 17
- The Contract Form 21
  - Validation 24
  - PDF Generation 26

The Contract Service 27
PnP JavaScript Core Library 28
Chaining Promises—Always be Making Promises 30
Batching Requests, the Easy Way 31
Building the App: Behind the Scenes 34
Typings 34
Gulp 35
Webpack Bundling 39
Getting Started 46
Summary 46
About the Authors 47

Introduction

This whitepaper discusses the future of SharePoint development and where Kendo UI® by Progress can provide value in custom SharePoint application development.

We want to follow the trails of a SharePoint developer, where we've been (with Full Trust, Sandbox, App/Add-In Web) and where we are heading next with the open web stack. This was an inevitable change that was happening with or without Microsoft, based on all client projects that we see and on talking with the community at large.

With the upcoming SharePoint Framework ("First release this summer"), Microsoft is no longer dictating our toolset, but embracing open web technologies and wanting to support industry trends as first class supported customizations within SharePoint.

There is a lot to catch up on with the SharePoint Framework so we better get started!

Our main goals for this whitepaper are to:

- Get you excited about the new SharePoint Framework and related web stack technologies.

- Use a great set of web stack tools that work well together to build a practical SharePoint business application that can get you started with these technologies today.

Future of SharePoint Development

The future of SharePoint development and customization is the SharePoint Framework.

It is a client-side based framework that allows JavaScript customizations to work on top of SharePoint Online. It will also work with SharePoint Server on-premises in a future update.

If you think SharePoint and Microsoft have been standing still in the last few years—you might be shocked. They have silently come a very long way!

The Framework and Why it's Awesome

Firstly, it is a recognition that Microsoft should adopt and build on top of progressive web technologies. Instead of learning Microsoft's version of a technology that's rapidly outdated, we can use the latest and greatest in jQuery, KnockoutJS, AngularJS or ReactJS—even mix and choose technology based on our needs. These open web technologies are all supported on SharePoint Online.

We are no longer asked to learn strange object models or peculiar syntaxes like CAML or DisplayTemplates. Instead, we just need the common, modern web technology stack to be effective and productive.

For businesses, this means we can train our developers in skills that are useful even outside of SharePoint. It is also far easier to hire web developers and have them contribute to a SharePoint project quickly.

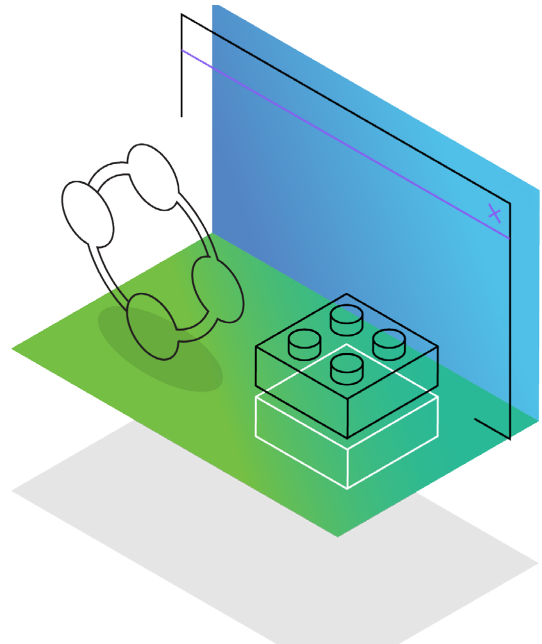
For traditional SharePoint developers, this means we are required to familiarize ourselves with (yet again) a new set of tools. This may be a big negative for some, but we believe it to be a great opportunity. It will open us up to a much broader world of web development and at the same time open up SharePoint to a whole world of web developers. Yes, we will have to step up our game, but it is a small price to pay, compared to the past when we had to hack around ancient, unloved, SharePoint-specific technology.

Secondly, Microsoft promised that they will stop "cheating" by creating backdoors for themselves. This time, they are committed to only build on top of this framework. Starting with the Office 365 Video Portal, the improved Delve Blogs and the new Document Library experience are all the beginnings of Microsoft's own work building on top of the SharePoint Framework. The upcoming new Team Sites are the next step, along with several sneak peeks of the publishing framework.

The Tools

The tools that Microsoft adopted for the SharePoint Framework are best of breed tools from the web world: Template and project generators from Yeoman, build pipeline and tasks with Gulp and project bundling with Webpack—distributed open source on GitHub.

In addition, Microsoft is releasing a tool called the SharePoint Workbench, which will allow us to run and test our controls and applications within an offline environment so we can work on our projects and run automated tests when we are not connected to SharePoint Online.



Open Source

All these framework, libraries and new tools are released Open Source. We can now take a copy of the code, improve it and create a pull request to submit back to the community.

There are also several community projects spinning up around the official releases, adding capabilities, libraries and components available to our toolkit. Most recently, the release of the PnP-JS-Core library provides a new wrapper around the modern REST API that should help developers transition into client side development for the SharePoint Framework.

Not too long ago, SharePoint developers using open source libraries like SPServices to build their web parts and application were somewhat considered cowboys. Today, developers are being encouraged to use them as best practice.

Jeff Teper and the Focus on Modern and Mobile

One big exciting development on the roadmap was the return of Jeff Teper, the father of SharePoint, back to lead SharePoint efforts within Microsoft.

The team had zeroed in on several key updates to SharePoint.

While the backend of SharePoint Online and On-Premises continues to update forward in an evergreen way, new experiences modernize the SharePoint experience.

All modern browsers are supported as first class. And all the new experiences are designed to be responsive and rescale accordingly to the UI—whether it is desktop or mobile, across all platforms.

This matches very well with developments in the web world, with technologies like Cordova allowing web applications to run within native applications, the SharePoint Mobile App was born.

It promises a smooth interface, caching and seamless connection to Office Client Apps, OneDrive and the new Team Site experience.

The Mobile App also had the promise that customizations on top of the SharePoint Framework will run within the Mobile App. This icing on the cake may yet prove to be the biggest selling factor to businesses. If we come to Microsoft for this ride and develop the "new way," we get the experience across all mobile devices—free.

Timeframe

The SharePoint Framework (SPFx) was in private preview, and through the writing of this whitepaper has gone into developer preview and released to the public.

Update: Developer Preview (August 2016)

The Developer Preview, our first taste and our feedback to Microsoft, is in two areas. Firstly, Microsoft's own guidance doesn't cover Angular yet. We believe this to be crucial before public release as a majority of full client-side applications developed in the last few years are using the Angular framework.

Secondly, SPFx build Gulp tasks and Webpack configurations are hidden from the user. This hides some of the 'magic' but also makes the build process difficult to customize for different teams. For example, if we want to add more tasks to the build pipeline, we want to integrate a different testing framework or even deploy the solution to existing SharePoint 2013 On-Premises—the current Gulp tasks hides that.

But we are expecting public release to happen very soon this year. And we hope to work with Microsoft to address some of these issues.

Selecting Our Web Stack and UI Library

Ready, Set, Go! Onward to the SharePoint Framework

So, you are all set and excited about the SharePoint Framework. You are looking to customize Office 365. You want to bring your customizations down to SharePoint Mobile. You want to build apps that will run within SharePoint, or even in Office Add-Ins.

There are a multitude of scenarios that have taken us to this point. The next step is to evaluate the components and technologies that are available to us and pick the ones that we will focus on.

Next, we will discuss web technologies that we believe align well with the direction in the open web stack. We present this technology set as a good solution that works really well, but keep in mind many more technologies, e.g. KnockoutJS or ReactJS will also work fine. You should evaluate this suggested list and make adjustments accordingly, relating to your team's situation.

Technologies

Node, NPM—In the JavaScript world, Node is the runtime that lets us run code without a browser. The modern web stack runs on JavaScript—script running in Node on the server and script running in browser on the client.

NPM is like nuget for getting libraries and tools that we need to run tools as well as libraries for the web application. Historically, Bower was used for client-side JavaScript libraries—technologies such as Webpack module bundling means that there is no longer a real distinction between what is a client-side library or what is just a JavaScript library. If it's needed on the client side, the bundler will pack it for us.

AngularJS—Framework that helps us manage databinding, components and composition. This is one of the most popular web frameworks that likely doesn't need any introduction. Many companies have already invested skills and code to this framework.

For this project, we pick Angular over React because of our own experiences, having used Angular in client projects in the past and the readily available UI frameworks like Kendo UI that are optimized for Angular.

Yeoman—'The web's scaffolding tool for modern webapps' is one of the most exciting pieces to the SharePoint Framework puzzle. Before, we were dependant on Visual Studio Wizards to create the new SharePoint project. Now, we can generate the web project using the Yeoman generator running on node.js from the command prompt.

Microsoft currently has an ongoing very successful Yeoman template for creating Office Add-ins called generator-office (yo office). The SharePoint Framework generator will be called generator-sharepoint (yo sharepoint).

Although we are highly anticipating the SharePoint template, you won't be restricted to it. You can make your own adaptations and templates and share them within your development team or with the entire community.

For our project, we will use the PnP JSCore Yeoman Template also released by Microsoft to get started.

Gulp—Task runner, build, test, deploy to SharePoint. Traditionally, Gulp takes a more active role in minifying and concatenating files before deployment. That task has largely been taken over by Webpack. Gulp remains useful in our stack because it is still more flexible as a task runner to connect different tasks, and ultimately, manage the deploy tasks to SharePoint.

Webpack—It's one of the most interesting pieces of the modern web stack that appeared in the last year and is rapidly gaining momentum. At its core, its unique philosophy is to utilize parallel code splitting to load JavaScript quickly into the browser. And it is designed to not only minify and uglify JavaScript, but also able to generate source-maps and run a development Webpack server that could recompile on the fly. Webpack is fairly opinionated about how things should work, but has taken over much of the functionality that used to depend on several different tools—Browserify, Gulp and require/systemjs. In the SharePoint Framework, Webpack will be one of the pieces used to package our resources both to run in a local development server, or for deployment to a CDN to be used in SharePoint itself.

Visual Studio Code—We choose to use Visual Studio Code, as it's a much simpler code editor. VS Code doesn't try to be a fully Integrated Development Environment (IDE) with full SDKs and wizards. Instead, it tries to be a good code editor that understands files grouped in folders really well.

Where does Kendo UI fit in and what are the Alternatives?

If we are building business applications, we are probably building forms and we need great controls with out-of-the-box validation capabilities. Basic controls like textbox, text area and date pickers are a given. As we consider additional controls like grid view, rich text editing and multi-lingual support for global customers, using a UI framework is a no brainer.

A Grasp of What's Out There and Our Experience with it:

- UI Bootstrap—is well tested and used with Angular, but not particularly optimized for SharePoint or Office 365. You may have some growing pains as you adjust CSS and loading orders to get UI Bootstrap to work nicely. Bootstrap's Responsive CSS doesn't work with SharePoint MasterPage out of box, so you will need to tinker with that. When single page applications (SPAs) are supported properly with SharePoint Framework, this may change.

- Office UI Fabric Core, Components and ngOfficeUIFabric—Office UI Fabric is a set of styles and components provided and used by Microsoft to build their own customizations. The focus is Office 365 friendly CSS, and components designed for plain JavaScript or React. As Microsoft doesn't provide an Angular implementation, ngOfficeUIFabric is a community attempt to create native wrappers for Office UI Fabric components, but along with the components themselves they are not yet widely adopted and you may find support difficult to come by. UI Fabric Core is mainly supported for SharePoint Online, Office Add-Ins and latest versions of SharePoint 2013/2016.

- Kendo UI comes in three plans—Core, Professional and Complete, where Complete goes beyond the scope of this project as it includes additional libraries such as wrappers for MVC, JSP and PHP. Kendo UI Core is free, open source and community-supported. Kendo UI Professional is next step with 70+ advanced controls (grid, spreadsheet, Gantt and scheduler being our recent favorites). While Kendo UI Pro isn't open source, we do get access to dedicated support and full source code to review so we can still take a crack at a particular problem by reading the source code ourselves. It's the best of both worlds.

Kendo UI controls support jQuery and Angular 1.x. Angular 2.x and React support is coming soon.

All these libraries will work with any SharePoint and Angular projects and you don't have to use Kendo UI. Our example does, but it can be replaced with ngOfficeUIFabric.

There are plenty of things to learn and get going in the new framework—spending time on making a control work is probably less important in the grand scheme of things.

> "I have some first-hand experiences here [at a client] customizing both UI-Bootstrap and ngOfficeUIFabric on several client projects. Halfway through each project I'm thinking, maybe I should have just gone for a commercial supported framework and not spend all this time making controls work."

Building the App: Contract Register

We want to demonstrate how the technologies discussed in the previous section work together and build a simple yet useful application: The Contract Register.

Use Case

When a government department shares information with a third party, which is not covered by any other legal agreement, they require the information receiving party to sign a contract.

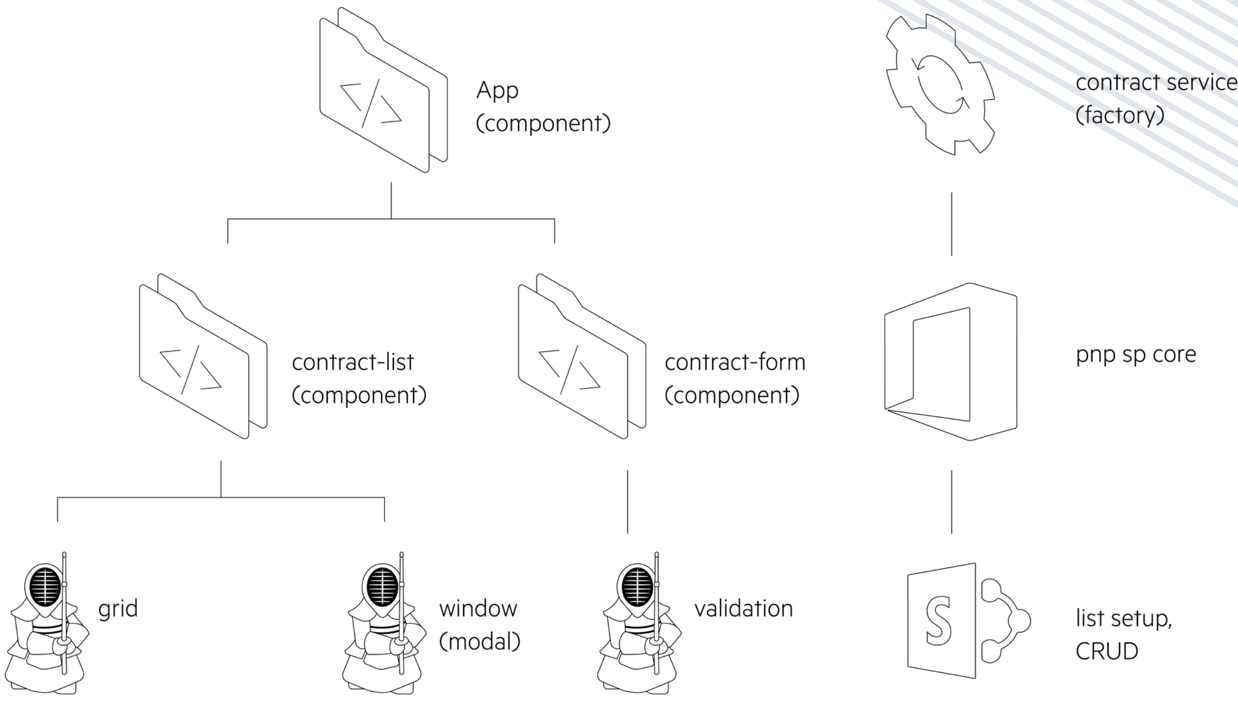
A 'System Access Agreement' describes a code of conduct for external parties to access any of the department's information systems and the general terms of usage.

On a functional level, the application should:

- Show a list of all contracts and status (the register)
- Allow for adding, editing and viewing contracts
- Signed contracts should be uploaded and stored within the register

High Level

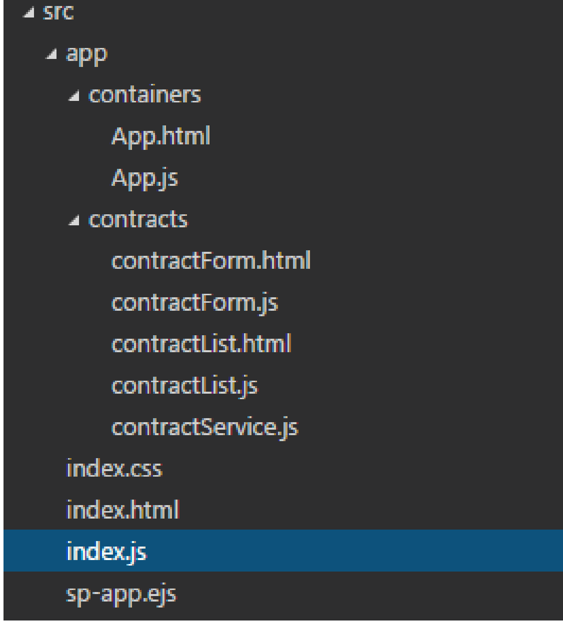
What are we building? Below is a high-level diagram of the components. Each component isn't a lot of code as we aim to show a simple, but working system and how each component interacts.



Group Artefacts by Function—Not by Type

In many starter examples of Angular projects, the SPA is grouped by type of the object. Whether it is a View, Model or Controller—they are often lumped and grouped in separate directories. You see a component like DataService fairly regularly as the only component to talk to the backend.

What many people later find out is that while that sort of grouping makes a good simple starter project or learning demo, it is actually not a great way to keep related functions together. When you are working on the controller for a contract-list, you probably want the contract-list view template right next to it.



Files organized by function, with views next to their controllers.

See Todd Motto's Angular Style guide for in-depth examples: https://github.com/toddmotto/angular-styleguide

Entry – index.js

Every application has a starting point. Ours is index.js. Here we defined our Angular single page application, 'app'. It has dependency on Kendo UI Angular Directives. It has four parts—a contract service where we will talk to the backend system (in this case, SharePoint), a contract list, a contract form and the top level 'app component' which is the root component that contains everything inside.

```
var app = angular
  .module('app', ['kendo.directives', 'kendo.window']);

app.service('contractService', ContractService);

app.component('app', App);

app.component('contractList', ContractList);

app.component('contractForm', ContractForm);
```

The base HTML page that we bind to is SPApp.html which is generated from sp-app.ejs file via the Webpack HtmlWebpackPlugin. We discuss this in a later section. The HTML is similar to index.html.

```
<div ng-app="app">

  <app>Loading...</app>

</div>
```

This is where we bootstrap our Angular application and load the root App component.

The App and the App Component

The App component is loaded in the default page replacing the <app> element.

Here we see a simple example of Angular 1.5's new components syntax, combining best practices of directives with isolated scopes and controllerAs $ctrl by default, forward looking towards a web-components future in Angular 2.0.

```
<contract-list></contract-list>

module.exports = {
  template: require('./App.html'),
  controller: App
};

function App() {
  // App is just a top level component
}
```

A component in Angular can reference either templateUrl. We then need to pass the template as a URL or preload it into the Angular Template Cache. With the template option, we then need to specify inline HTML string.

Webpack excels at loading and packing HTML as string into a single JavaScript file. So one of the early choices we end up with is to rely on Webpack to do templating, and essentially do away with Angular Template Cache.

Note: Should you still want to use Angular Template Cache, there is a Webpack ng-cache loader that will take template files and preload them so they are still available via the templateUrl property.

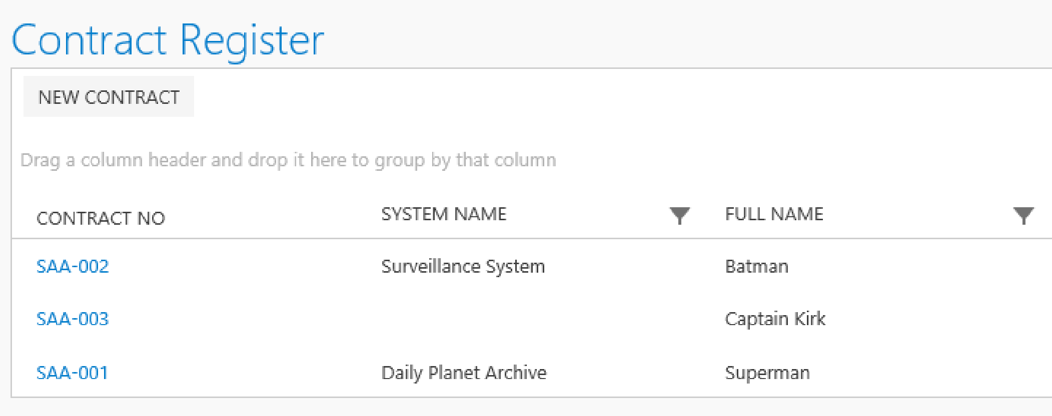
We use template string consistently through the project.

The Contract List

Our first interesting component is a list of the contracts we want to show in the application. The template will look extremely simple.

```
<h1>Contract Register</h1>

<kendo-grid options="$ctrl.gridOptions"></kendo-grid>
```

Note that we wanted to use the Kendo UI Grid control for some fairly sophisticated functions. The Kendo UI Grid control is part of the Kendo UI Professional pack. A simpler example we started with uses the Kendo ListView control, which is part of the Kendo UI Core pack, and works very well for list rendering.



Kendo UI controls follow a more traditional way where we bind the data to a Kendo UI DataSource object first. The data source object tracks paging, filters and grouping, and feeds that information to one or more controls on the page (e.g. a Grid and a Pager).

Angular 1.5 Controller

```
module.exports = {
    template: require('./contractList.html'),
    controller: ContractListController
};

 ContractListController.$inject = ['$scope', '$element', '$attrs', 'contractService',
'$kWindow'];
function ContractListController($scope, $element, $attrs, contractService, $kWindow) {
    var $ctrl = this;

    $ctrl.source = null;
    $ctrl.openFormWindow = openFormWindow;
    $ctrl.refresh = refresh;
    $ctrl.newContract = newContract;

    activate();

    ...
}
```

The ControllerList is written with readability in mind. We follow several best practices from Todd Motto's excellent Angular Style Guide. We list controller members near the top, and initialize them with activate() method.

In Angular 1.5, the controllerAs is on by default, and the default name is "$ctrl" so we reflect that in the code to keep the code simple. (Previously before 1.5, we used vm from our MVVM background).

Because Webpack will take care of bundling and uglification of the parameters, use $inject header to tell Angular which arguments need to be provided by dependency injection.

The Controller activate()

```
function activate() {
     $ctrl.source = new kendo.data.DataSource({
         data: [],
         schema: {
                 model: {
                     fields: {
                         ...
                     }
                 }
             }
     });

     $ctrl.gridOptions = {
         dataSource: $ctrl.source,
         sortable: true,
         filterable: true,
         groupable: true,
         columns: [
             ...
         ],
         toolbar: [
         {
           name: "add",
           text: "New Contract",
           template: '<a ng-click="$ctrl.newContract()" class="k-button k-button-icontext
k-grid-add" href="\\#">New Contract</a>'
         }]
     };

     $ctrl.refresh();
 }
```

We set up the Kendo UI data source.

We set up grid options for the Kendo Grid component.

And finally, we call refresh on the list. Refresh is a method we will come back to later. It is exposed as a public method on the controller.

The Controller refresh()

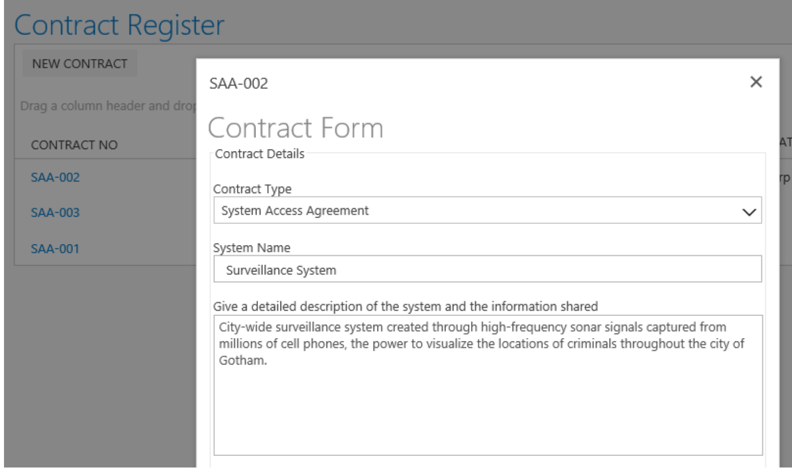
```
function refresh() {
     contractService.getItems().then(function (items) {
         $ctrl.source.data(items);
     });
 }
```

The refresh method asks the contract-service to get a list of items then bind that list to Kendo UI DataSource's data member.

Kendo UI's datasource is an observable array, and it will work with Angular to correctly refresh the grid list.

The Modal Service

When we click on a form we want to bring up, the form in a modal dialog, the Angular way.



Control Approach: Bootstrap's $modal, Office UI Fabric UIF-Dialog, Kendo UI Kendo-Window

All component libraries for a pop up window (dialog) have simple syntax designed for opening an individual dialog. This pattern is useful for scenarios to create dialogs to ask for confirmation, or to bring up a terms and conditions when a user visits an intranet site.

The control-based approach is not a suitable pattern when you want to bind different views and controllers to different items in a set of items.

The control approach also usually does not support multiple stacks of window dialogs. So we couldn't pop up a window dialog and have that window dialog pop up yet another confirmation dialog in a well-supported way.

Factory Approach: SharePoint SP.UI.ModalDialog, UI Bootstrap $modal, $kWindow

Both the UI Bootstrap library and Kendo UI have implementation and examples for a factory-based approach. Incidentally, even SharePoint's own SP.UI.ModalDialog works like a service and lets us create dialog windows and stack them.

Here is a method to open a contract form component within a popped up dialog window.

```
function openFormWindow(dataItem) {
    // http://plnkr.co/edit/PjQdBUq0akXP2fn5sYZs?p=preview
    var windowInstance = $kWindow.open({
        options: {
            modal: true,
            title: dataItem.Title,
            resizable: true,
            visible: false
        },
        template: '<contract-form contractid="$ctrl.contractid"
$close="$close(result)" $dismiss="$dismiss(reason)"></contract-form>',
        controller: ['contractid', function (contractid) {
            var $ctrl = this;
            $ctrl.contractid = contractid;
        }],
        controllerAs: '$ctrl',
        resolve: {
            contractid: function () {
                return dataItem.Id;
            }
        }
    });

    windowInstance.result.then(function (result) {
        if (result) {
            $scope.result = 'confirmed!';
            $ctrl.refresh();
        }
        else {
            $scope.result = 'canceled!';
        }
    });
}
```

The $kWindow service lets us open a window, and at the same time bind an Angular template and a controller to this template. The service also attaches $close and $dismiss methods to the controller for the form. So the child component can control when it is ready to be closed.

Angular's template binding kicks in and allows the contract-form component to be rendered within the popped up modal dialog.

Finally, the service returns an instance that has a result promise that can be used by the parent controller to watch for the result of the opened dialog window.

More information is available on https://angular-ui.github.io/bootstrap/#/modal.

Kendo UI window-service is available here and the code itself is on GitHub.

One specific implementation note—because our project utilizes Webpack bundling, we do not have the template 'window.html' which is needed by angular-kendo-window service. Instead, we bundle the content of window.html via a template string in our implementation.

```
kwin.directive('kWindowFrame', ['$kModalStack', '$q', '$animate', '$injector',
    function ($modalStack, $q, $animate, $injector) {
        ... snipped
        return {
            scope: {
                index: '@'
            },
            replace: true,
            transclude: true,
            template: '<div kendo-window="myKendoWindow" k-options="options" modal-
render="{{$isRendered}}" tabindex="-1" role="dialog" >    <div><div k-window-
transclude></div></div>  </div>',
            /*
            templateUrl: function (tElement, tAttrs) {
                var windowInstance = $modalStack.getTop().value;
                return windowInstance.windowTemplateUrl || 'window.html';
            },
            */
            ... snipped
        };
    }])
```

The Contract Form

The Contract Form is our third component in the application. It is in many ways similar to the contract list component. Because it is loaded in a dialog window, it also has unique bindings to the dialog window's $close and $dismiss methods.

```
module.exports = {
  template: require('./contractForm.html'),
  controller: ContractFormController,
  bindings: {
    '$close': '&',
    '$dismiss': '&',
    'contractid': '<'
  }
};
```

The contractid is a property provided by the contract list when the contract form is opened. The contract form's activate method requests the contract from the contract-service.

```
 activate();

  function activate() {
    //ID is known get item refresh
    if($ctrl.contractid){
        contractService.getItem($ctrl.contractid).then(function (item) {
            $ctrl.item = item;
        });
    }
  }
```

The contract is set to the controller's item property. This is then bound to the template for contract form.

```
<fieldset>
    <legend>Contract Details</legend>
    <p class="forms">
        <label>Contract Type</label>
        <select kendo-drop-down-list style="width: 100%;" ng-model="$ctrl.item.Title">
            <option value="SAA" selected>System Access Agreement</option>
            <option value="NDA">Non Disclosure Agreement</option>
        </select>
    </p>
    <p>
        <label>System Name</label>
        <input type="text" class="k-textbox" ng-model="$ctrl.item.SystemName" required />
    </p>
    <p>
        <label>Give a detailed description of the system </label>
        <textarea ng-model="$ctrl.item.InformationDescription" required></textarea>
    </p>
    <p>
        <label>End Date</label>
        <input kendo-date-picker ng-model="$ctrl.item.ContractEndDate" required />
    </p>

</fieldset>
```

Kendo UI controls here can be bound simply to the model on the controller.

```
<p>
    <button kendo-button type="submit" ng-click="$ctrl.save()">Save</button>
    <button kendo-button type="button" ng-click="$ctrl.dismiss()">Cancel</button>
</p>

  function save() {
    ... snip
    //ID is known update item
    if ($ctrl.contractid) {
      contractService.updateItem($ctrl.item).then(function (item) {
        $ctrl.close();
      });
    }
    else {
      //no ID add as new item
      contractService.newItem($ctrl.item).then(function (item) {
        $ctrl.close();
      });
    }
  }

  function close() {
    $ctrl.$close({
      result: 'save'
    });
  }

 function dismiss() {
  $ctrl.$dismiss({
    reason: 'cancel'
  });
 }
```


Progress

The buttons at the bottom of the form can also be bound to methods on the controller. When invoked, they call the contract-service to update or create new contract item. When that's successful, they call the $close or $dismiss methods on the dialog window to close down the dialog, disposing the form component.

If we don't want to have save() close the modal window, then we need to take the saved item object from the resolved promise and assign it back to $ctrl.item. This lets the UX bind to the latest version of the object, and if we are using calculated fields in SharePoint, this is necessary to bring back the server updates during the save.

Validation

We bind any validation messages to the bottom of the form. Kendo UI validation is very simple to set up.

Third Party Details

Full Name

Organisation

Email

e.g. myname@example.net

Email is required

Fields that are required, add a required attribute. Specify a kendo-validator in the form, and attach validate method to the submit event of the form. (This is triggered by the save button, which is a type="submit").

```
<form kendo-validator="$ctrl.validator" ng-submit="$ctrl.validate($event)">

    <fieldset>
        <legend>Contract Details</legend>
        <p>
            <label>System Name</label>
            <input type="text" class="k-textbox" ng-model="$ctrl.item.SystemName"
required />
        </p>
        <p>
            <label>Start Date</label>
            <input kendo-date-picker ng-model="$ctrl.item.ContractStartDate" required
/>
        </p>
    </fieldset>
    <p>
        {{ $ctrl.validationMessage }}
    </p>

</form>

  function validate(event) {
    //block submit from making a postback
    event.preventDefault();

    if ($ctrl.validator.validate()) {
      $ctrl.validationMessage = "";
      $ctrl.validationClass = "valid";
    } else {
      $ctrl.validationMessage = "There is invalid data in the form.";
      $ctrl.validationClass = "invalid";
    }
  }
```

To be fair, UI-Bootstrap also has similar validation capabilities. Office UI Fabric components does not currently have validation, but it is on the roadmap in the future.

```
function save() {
    if (!$ctrl.validator.validate()) {
      $ctrl.validationMessage = "There is invalid data in the form.";
      $ctrl.validationClass = "invalid";
      return;
    }
    //ID is known update item
    if ($ctrl.contractid) {
      contractService.updateItem($ctrl.item).then(function (item) {
        $ctrl.close();
      });
    }
    else {
      //no ID add as new item
      contractService.newItem($ctrl.item).then(function (item) {
        $ctrl.close();
      });
    }
  }
```

Update contract-form's save() method—it should check and make sure the form is valid.

PDF Generation

The application uses Kendo UI to save DOM to PDF.

Internal Details

Internal Representative

Save | Cancel | Refresh | Upload Signed Contract | Generate Contract

```
function generatePdf(selector) {
  kendo.drawing.drawDOM($(selector)).then(function (group) {
    kendo.drawing.pdf.saveAs(group, "contract.pdf");
  });
}
```

When the Generate Contract is clicked, the click event calls a Kendo function to generate the current DOM into PDF for download.

An alternative scenario may be to write the PDF binary directly to SharePoint's document library.

The Contract Service

The contract service is injected by most of the application components and exposes the methods for talking to the SharePoint list backend.

```
ContractListController.$inject = ['$scope', '$element', '$attrs', 'contractService',
'$kWindow'];
```

In our example project, we also have the contract service provisioning the Contract Register List and fields. You may consider moving this into a separate application component (handling configuration) altogether.

The contract services use the methods provided by the PnP SharePoint JavaScript library to perform actions in SharePoint using simple and clean looking code.

contractService.js

```
module.exports = ContractService;

var pnp = require('sp-pnp-js');

//added here and as webpack external to prevent compile warnings
var _spPageContextInfo = require('_spPageContextInfo');

//Use angular promises
ContractService.$inject = ['$q'];

function ContractService($q) {
    var self = this;

    self.getItems = getItems;
    self.newItem = newItem;
    self.getItem = getItem;
    self.updateItem = updateItem;

    function getItems() {
         return $q(function (resolve, reject) {
    ...
}
```

PnP JavaScript Core Library

What is this library? Currently, it contains a fluent API for working with the full SharePoint REST API as well as utility and helper functions. This takes the guess work out of creating REST requests, letting developers focus on the what and less on the how. (Source: PnP JS Core Wiki)

Even though using the library simplifies working SharePoint REST API, developers still need to know how to complete basic operations with the REST endpoints to take advantage of all the options available.

Together with the PnP JS Core API documentation, performing list operation in SharePoint has never been this easy.

Let's compare creating a SharePoint list through a 'normal' jQuery (ajax) SharePoint REST API call vs the PnP JS Core Library:

Using jQuery, the web request would look something like this:

```
jQuery.ajax({
    url: "http://<site url>/_api/web/lists",
    type: "POST",
    data: JSON.stringify({
        '__metadata': { 'type': 'SP.List' },
        'Title': 'My List Title',
        'Description': 'My list description',
        'BaseTemplate': 100,
        'AllowContentTypes': false,
        'EnableVersioning': true,
    }),
    headers: {
        "accept": "application/json;odata=verbose",
        "content-type": "application/json;odata=verbose",
    ...
}).then(..
```

Making this same list configuration call through the PnPJSCore add method would look like:

```
var additionalSettings = { EnableVersioning: true }
pnp.sp.web.lists.add('My List Title', 'My list description', 100, false, additionalSettings).
then(..
```

As you can see this PnP example is much leaner and readable. The add method parameters allow us to specify the required properties to create the list with one line of JavaScript.

Additional settings like 'EnableVersioning' can be passed in as an object of key value pairs and will be added to the list creation request body. For finding all additional settings, we would still rely on documentation available around the SharePoint REST API like the available list properties.

Chaining Promises—Always Be Making Promises

Continuing to look at our contract service code, it will probably not come as a surprise that all methods return promises and often chaining promises within.

Where possible, we use Angular promises ($q), we use $q() as a promise constructor and return it directly without creating a deferred object first.

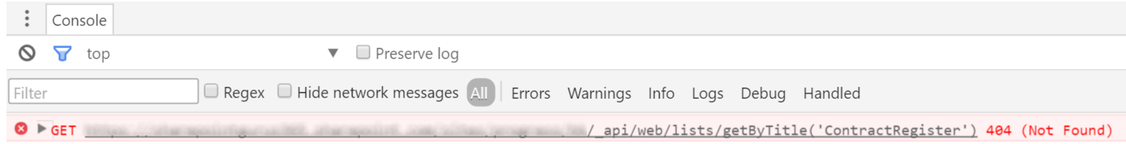
The PnP JS Core library uses ES6 Promises and requires a polyfill for Internet Explorer to work.

```
function getItems() {
    return $q(function(resolve,reject){
        getList().then(function (list) {
            list.items.get().then(function (items) {
                resolve(items);
            });
        }).catch(reject);
    });
}

function getList() {
    return $q(function (resolve, reject) {
        pnp.sp.web.lists.ensure('ContractRegister', '').then(function (result) {
            if (result.created) {
                setupContractRegister(result.list).then(function (list) {
                    resolve(list);
                });
            } else {
                resolve(result.list);
            }
        });
    }).catch(reject);;
}
```

All of the contract service methods: getItems(), newItem(), getItem(), updateItem() use the getList() method to get the contract register list instance first.

The getList() method uses the ensure() method to detect whether the register list already exists, and if not creates it right away. Note: the ensure methods do not update the list settings if the list already exists.

The Lists.ensure() method is a wrapper around the Lists.getByTitle() method, and when fails, catches the error and calls Lists.add(). This event will be logged to the console like:



The ensure method returns a promise that once completed resolves an object containing the list instance and the list creation data.

In the contract service when the list is created we call the setupContractRegister() method only once.

Any consecutive call of the ensure method will effectively be the same as calling the getByTitle () method directly.

Batching Requests, the Easy Way

Although batching has been available through the SharePoint Online REST API for some time, making batching request work was a fairly complicated task. Batching was added in the PnPJSCore 1.0.2 release, the announcement has some great examples for batching (as well as caching). It is important to note that batching is not supported on SharePoint 2013 On-premise as of yet.

Even though the PnPJSCore library makes batching very easy, it is recommended to have a read of Andrew Connell's excellent series, SharePoint REST API Batching—Understanding Batching Requests, to understand the inner workings.

In our contract service, we use batching to provision the list fields for the contract register.

```
function setupContractRegister(list) {
    return $q(function (resolve, reject) {
        // create batch
        var batch1 = pnp.sp.createBatch();
        $q.all([
            //Information System Details fields
            list.fields.inBatch(batch1).addText('SystemName'),
            list.fields.inBatch(batch1).addMultilineText('InformationDescription', 8, false),
            list.fields.inBatch(batch1).addNumber('InformationSensitivity'),
            ...
        ]).then(function () {
            // add some demo entries
            $q.all([
                list.items.add({'Title': 'SAA-001', 'ThirdPartyContactFullName': ... }),
                list.items.add({'Title': 'SAA-002', 'ThirdPartyContactFullName': ... }),
                list.items.add({'Title': 'SAA-003', 'ThirdPartyContactFullName': ... })
            ]).then(function () {
                resolve(list);
            });
        });

        batch1.execute();

        });
    }
```

As explained in the version 1.0.2 announcement, the same applies here. The key thing is we can keep chaining our promises or add them to an array of $q.all([…]) promises and they won't be resolved until the batch execute command is called.

We can almost write the same code as before and just add in the inBatch() method to the chain and get the benefits of REST batching. When we want to add the list fields without batching, (for example to make this work on-premises) we would need to chain them one after the other. Calling them in parallel (array of promises) will return conflict errors as list configuration can't be changed by multiple requests at the same time.

This is different for list items as they can be added simultaneously as the above method demonstrates. This does however result in three separate http requests, where if we would have used batching there would have only been one.

Building the App: Behind the Scenes

There's always two sides to a great meal—there is the recipe and there are the tools that do the cooking. AngularJS, Kendo UI and PnP JS Core work well to create the application. And then we look to the build side—which tools do we use to really improve the build experience.

> "You might not be cooking in the SharePoint Dev Kitchen, but that really doesn't stop you from watching and learning how to use the same tools to cook at home."

Typings

Intellisense is for Everyone, not just TypeScript

With Visual Studio Code, detailed intellisense can be enabled for both TypeScript and JavaScript projects. In a TypeScript project, the settings would be stored in a tsconfig.json file. For a plain JavaScript project, we need to add a jsconfig.json file.

The default jsconfig.json file essentially tells Visual Studio Code what EMCAScript level we are running, and which directories to ignore parsing (node_modules) as that would make the editor unbearably slow.

```
        angular.ex
        $ctrl.refr  extend  (method) angular.IAngularStatic.extend(destination: any, ...
    }               ICompiledExpression
                    IExceptionHandlerService
    function refre  exports
        contractSe  text
```

https://code.visualstudio.com/docs/languages/javascript#_javascript-projects-jsconfigjson

The typings module with the configurations in typings.json helps the project track the latest Type definition files available for modules within the JavaScript project. Visual Studio Code is then able to use the TypeScript definition files to provide code intellisense service.

Typings have more benefits in a TypeScript project as TypeScript encourages developers to strongly type their code, and the intellisense service is more intelligent. But even in a plain old javascript project, Typings is useful and gives better information than VS Code is able to guess.

We utilize PnP JS Core to talk to SharePoint. It is built with typings definitions, which makes intellisense fairly robust.

```
    //http://  customListTemplate
    pnp.sp.we  lists  (property) Web.lists: Lists
        resol  Get the lists in this web
    }, reject  addMultilineText
});            getList
               list
pnp.sp.web.li
```

Gulp

Gulp is our chosen build system—a Gulp script is JavaScript script. It makes it easier to specify how we want to chain our operations or sometimes dive into a few simple functions. Various sets of our tools work with Gulp as Gulp extensions.

The entire build process runs on Gulp tasks, these are defined in gulpfile.js

```
gulp.task("build", ["lint", "webpack:build"]);

gulp.task("lint", () => {
	return gulp.src("./src/**/*.js")
		.pipe(eslint())
		.pipe(eslint.format());
});

gulp.task("webpack:build", function(callback) {
	// run webpack
	webpack(config, function(err, stats) {
		if(err) throw new gutil.PluginError("webpack:build", err);
		gutil.log("[webpack:build]", stats.toString({
			colors: true
		}));
		callback();
	});
});
```

Traditionally, Gulp would be used to trans-compile, minify, concatenate and uglify JavaScript files, but it relies on other modules to do this, and Webpack has taken over most of the responsibility.

Visual Studio Code understands Gulp and will happily run the Gulp tasks that we define.

>run build

Tasks: Run Build Task Ctrl+Shift+B

```
OUTPUT
         Asset     Size  Chunks        Chunk Names
    SPApp.html  543 kB       0
    chunk    {0} SPApp.html 526 kB [rendered]
        [0] ./~/html-webpack-plugin/lib/loader.js!./src/sp-app.ejs 3.21 kB {0} [built]
        [1] ./~/html-webpack-plugin/~/lodash/lodash.js 522 kB {0} [built]
        [2] (webpack)/buildin/module.js 251 bytes {0} [built]
[19:44:07] Finished 'webpack:build' after 6.24 s
[19:44:07] Starting 'build'...
[19:44:07] Finished 'build' after 4.68 µs
```

We can also type 'task ' to bring up a list of other Gulp tasks that can be run from Visual Studio Code.

task

build
deploy
lint
webpack:build
webpack:build-dev

gulp-eslint - eslint Checks the JavaScript

Particularly, we are looking for areas where a variable might be undefined. This is an extremely common issue where perhaps we've used the wrong casing, or we had renamed the variable, but had leftovers, or—and everyone's guilty of this—we've copied code from the Internet (or from another function) and haven't gone through and renamed everything.

Another error that eslint can pick up is when we've had mismatched brackets or braces. Usually, an advanced editor like VS Code will pick this up, but eslint will double check it for us.

There are some best-practice rules, such as no-alert, where it will warn us if we've left alert() calls for testing behind and it'll end up in production.

On style issues, we are lenient on issues related to linebreak characters, tabs and spaces. But on larger teams this may be dictated to avoid edit wars and huge merge nightmares.

Take particularly single or double quotes—we lean on using double quotes for all strings, as the JSON specification explicitly says only double-quotes can be used for member definitions.

We integrate eslint within Gulp as a lint task, running the extension gulp-eslint

Within Visual Studio Code, an extension is available for eslint so we can get syntax checking while we work on each file.

gulp-spsave

To really simplify getting our result files into SharePoint, a few Gulp tools have been built. The most common are gulp-spsync and gulp-spsave.

Gulp-spsync is designed for SharePoint Online, but has been forked to gulp-spsync-withcred that lets us use it for both Online and On-Premises.

Gulp-spsave is designed for both Online and On-Premises. The configuration options for both tools are similar and they work similarly. Gulp-spsave has a slight edge on the number of options as well as a rewritten stack in gulp-spsave that uses sp-request module; it copies files faster.

We started our tools on gulp-spsync, then to gulp-spsync-withcred so we can have one tool for both Online and On-Premises deployments. Finally, we moved over to gulp-spsave because it had more options and runs faster.

```
var gulp = require("gulp"),
    gutil = require("gulp-util"),
    o365 = require("./o365-user.js"),
    spsave = require('gulp-spsave');

gulp.task("deploy", ["webpack:build"], function(){

       return gulp.src('./dist/SiteAssets/**/*')
              .pipe(spsave({
                     "username": o365.username,
                     "password": o365.password,
                     "siteUrl": o365.site,
                     "folder": 'SiteAssets'
              }));
});
```

gulp-spsync	SPO	https://github.com/wictorwilen/gulp-spsync
gulp-spsave	SPO, On-Premises, *fast*	https://github.com/s-KaiNet/gulp-spsave
gulp-spsync-cred	SPO, On-Premises	https://github.com/estruyf/gulp-spsync-creds
robocopy	Require mapped drive	

Webpack Bundling

Running Webpack in Development Mode

In development, our solution is to use jQuery, Angular and Kendo UI (JavaScript and CSS) as external modules. In this configuration, the SPApp.html is generated and points to reference vendor libraries in a CDN.

```
<link rel="stylesheet" href="https://kendo.cdn.telerik.com/2016.2.714/styles/kendo.common-office365.min.css" />
<link rel="stylesheet" href=" https://kendo.cdn.telerik.com/2016.2.714/styles/kendo.office365.min.css" />
<script src="https://kendo.cdn.telerik.com/2016.2.714/js/jquery.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
<script src="https://kendo.cdn.telerik.com/2016.2.714/js/kendo.all.min.js"></script>


<script src="../SiteAssets/vendor.js"></script>
<script src="../SiteAssets/app.js"></script>

<div ng-app="app">
  <app>Loading...</app>
</div>
```

Additionally, we upload local fallback versions in case the CDN is unavailable.
http://docs.telerik.com/kendo-ui/intro/installation/cdn-service

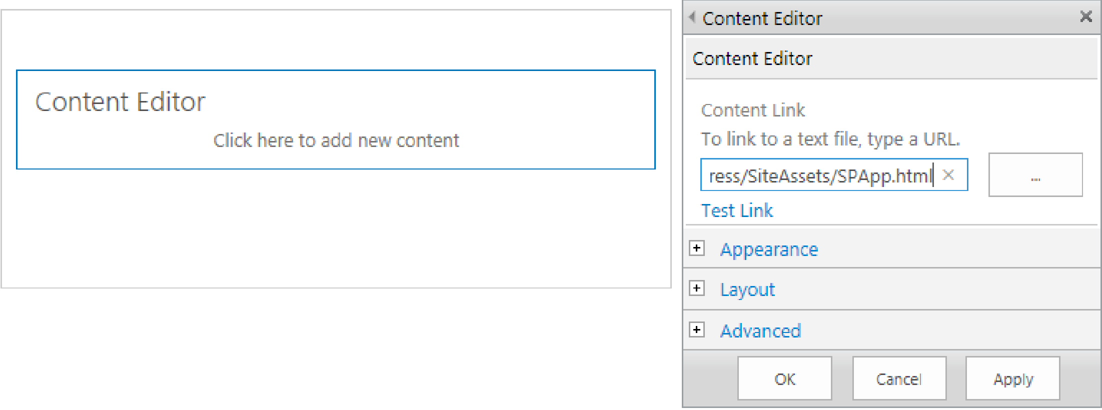
This means that Webpack is packaging only a vendor.js file for some remaining libraries we use, as well as the app.js containing all the application code that we wrote, along with the source map files for these.

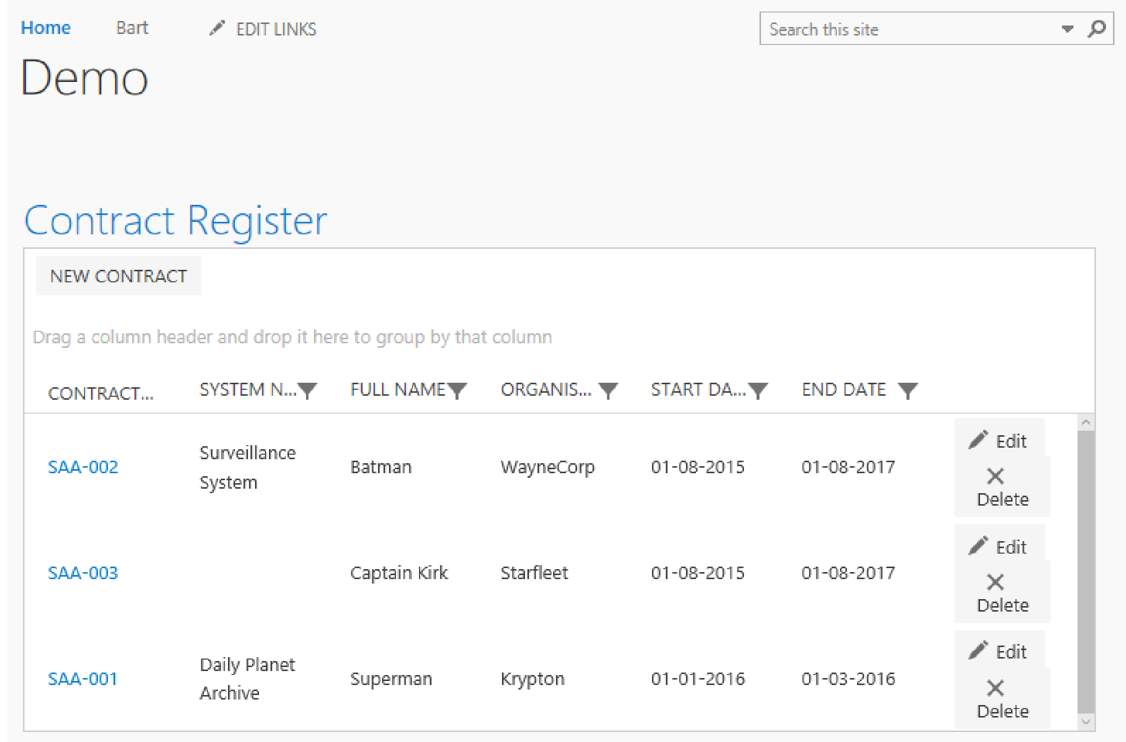
```
Side effects in initialization of unused variable AddFieldOptions [./~/sp-pnp-js/lib/sharepoint/rest/types.js:86,0]
Side effects in initialization of unused variable CalendarType [./~/sp-pnp-js/lib/sharepoint/rest/types.js:104,0]
Side effects in initialization of unused variable UrlFieldFormatType [./~/sp-pnp-js/lib/sharepoint/rest/types.js:109,0]
Side effects in initialization of unused variable PrincipalType [./~/sp-pnp-js/lib/sharepoint/rest/types.js:118,0]
Side effects in initialization of unused variable PageType [./~/sp-pnp-js/lib/sharepoint/rest/types.js:134,0]
Dropping unused variable r [./~/es6-promise/dist/es6-promise.js:130,0]
Condition left of && always true [./~/es6-promise/dist/es6-promise.js:949,0]
Dropping unused function lib$es6$promise$utils$$isMaybeThenable [./~/es6-promise/dist/es6-promise.js:19,0]
Child html-webpack-plugin for "SPApp.html":
        Asset     Size  Chunks       Chunk Names
    SPApp.html  543 kB       0
    chunk    {0} SPApp.html 526 kB [rendered]
        [0] ./~/html-webpack-plugin/lib/loader.js!./src/sp-app.ejs 3.21 kB {0} [built]
        [1] ./~/html-webpack-plugin/~/lodash/lodash.js 522 kB {0} [built]
        [2] (webpack)/buildin/module.js 251 bytes {0} [built]
[00:25:42] Finished 'webpack:build' after 10 s
[00:25:42] Starting 'deploy'...
[00:25:42] Finished 'deploy' after 8.25 ms
[00:25:45] spsave: app.js successfully uploaded to 'https://sharepointgurus365.sharepoint.com/sites/progress/SiteAssets'

[00:25:45] spsave: app.js.map successfully uploaded to 'https://sharepointgurus365.sharepoint.com/sites/progress/SiteAssets'
[00:25:46] spsave: SPApp.html successfully uploaded to 'https://sharepointgurus365.sharepoint.com/sites/progress/SiteAssets'
[00:25:47] spsave: vendor.js successfully uploaded to 'https://sharepointgurus365.sharepoint.com/sites/progress/SiteAssets'
[00:25:49] spsave: vendor.js.map successfully uploaded to 'https://sharepointgurus365.sharepoint.com/sites/progress/SiteAssets'

c:\Projects\contract-register>
```

The files are pushed to SharePoint.

Now, the HTML page can't run directly in SharePoint Online. SharePoint Online will actually not serve HTML pages; it will attempt to download them to the computer.

Instead, the HTML can be included into a blank page and run via the Content Editor webpart.

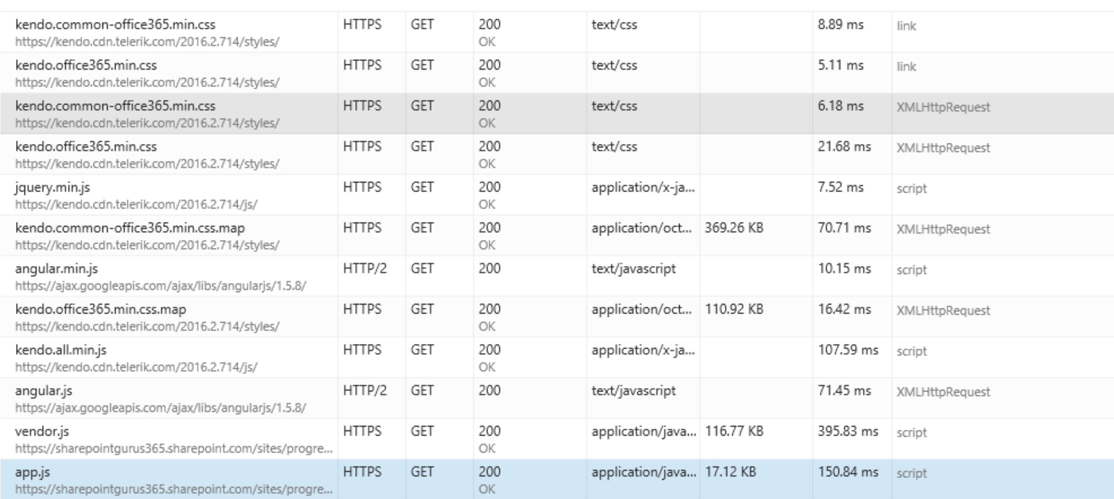




This is a one-time setup so we didn't include this in the Gulp scripts.

In the future, with the SharePoint Framework, we expect there will be a Gulp task that will deploy artefacts to SharePoint as well as register them with SharePoint APIs. We expect there will be small changes in this area as SharePoint Framework is released.

If we look at the network tab in the browser's F12 Development Mode, we can see a clearer picture of where the files are being loaded from.



Name	Protocol	Method	Status	Type	Size	Time	Initiator
kendo.common-office365.min.css https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	text/css		8.89 ms	link
kendo.office365.min.css https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	text/css		5.11 ms	link
kendo.common-office365.min.css https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	text/css		6.18 ms	XMLHttpRequest
kendo.office365.min.css https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	text/css		21.68 ms	XMLHttpRequest
jquery.min.js https://kendo.cdn.telerik.com/2016.2.714/js/	HTTPS	GET	200 OK	application/x-ja...		7.52 ms	script
kendo.common-office365.min.css.map https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	application/oct...	369.26 KB	70.71 ms	XMLHttpRequest
angular.min.js https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/	HTTP/2	GET	200	text/javascript		10.15 ms	script
kendo.office365.min.css.map https://kendo.cdn.telerik.com/2016.2.714/styles/	HTTPS	GET	200 OK	application/oct...	110.92 KB	16.42 ms	XMLHttpRequest
kendo.all.min.js https://kendo.cdn.telerik.com/2016.2.714/js/	HTTPS	GET	200 OK	application/x-ja...		107.59 ms	script
angular.js https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/	HTTP/2	GET	200	text/javascript		71.45 ms	XMLHttpRequest
vendor.js https://sharepointgurus365.sharepoint.com/sites/progre...	HTTPS	GET	200 OK	application/java...	116.77 KB	395.83 ms	script
app.js https://sharepointgurus365.sharepoint.com/sites/progre...	HTTPS	GET	200 OK	application/java...	17.12 KB	150.84 ms	script

The Kendo UI CSS, jQuery, Angular and Kendo JS files are loaded from CDN. Then a single vendor.js is loaded (this is where we packed additional vendor libraries into a single file) along with our app.js file—it is packed to only 17kb.

Webpack packs everything into modules—a highly unreadable mess, as one would expect.

app.js

```
("ThirdPartyOrganisationName"),e.fields.inBatch(o).addText("ThirdPartyContactEmail"),e.fields.inBatch(o).add("InternalOwner",
"SP.FieldUser",{FieldTypeKind:20}),e.fields.inBatch(o).addDateTime("ContractStartDate"),e.fields.inBatch(o).addDateTime
("ContractEndDate")]).then(function(){t.all([e.items.add({Title:"SAA-001",ThirdPartyContactFullName:"Superman",
ThirdPartyOrganisationName:"Krypton",ThirdPartyContactEmail:"super@man.com",SystemName:"Daily Planet Archive",InformationDescription:"",
InformationSensitivity:3,InternalOwnerId:r.userId,ContractStartDate:"2016-01-01",ContractEndDate:"2016-03-01"}),e.items.add(
{Title:"SAA-002",ThirdPartyContactFullName:"Batman",ThirdPartyOrganisationName:"WayneCorp",ThirdPartyContactEmail:"bat@man.com",
SystemName:"Surveillance System",InformationDescription:"City-wide surveillance system created through high-frequency sonar signals
captured from millions of cell phones, the power to visualize the locations of criminals throughout the city of Gotham. ",
InformationSensitivity:5,InternalOwnerId:r.userId,ContractStartDate:"2015-08-01",ContractEndDate:"2017-08-01"}),e.items.add(
{Title:"SAA-003",ThirdPartyContactFullName:"Captain Kirk",ThirdPartyOrganisationName:"Starfleet",SystemName:"",
InformationDescription:"",InformationSensitivity:1,InternalOwnerId:r.userId,ContractStartDate:"2015-08-01",ContractEndDate:"2017-08-01"}
)]).then(function(){n(e)})})["catch"](i),o.execute()})}var c=this;c.getItems=e,c.newItem=n,c.getItem=i,c.updateItem=o}t.exports=i;var
a=n(7),r=n(56);i.$inject=["$q"]},56:function(t,e){t.exports=_spPageContextInfo},57:function(t,e,n){function i(){}t.exports={template:n
```

But a quick Pretty-Print will help make more sense of it.

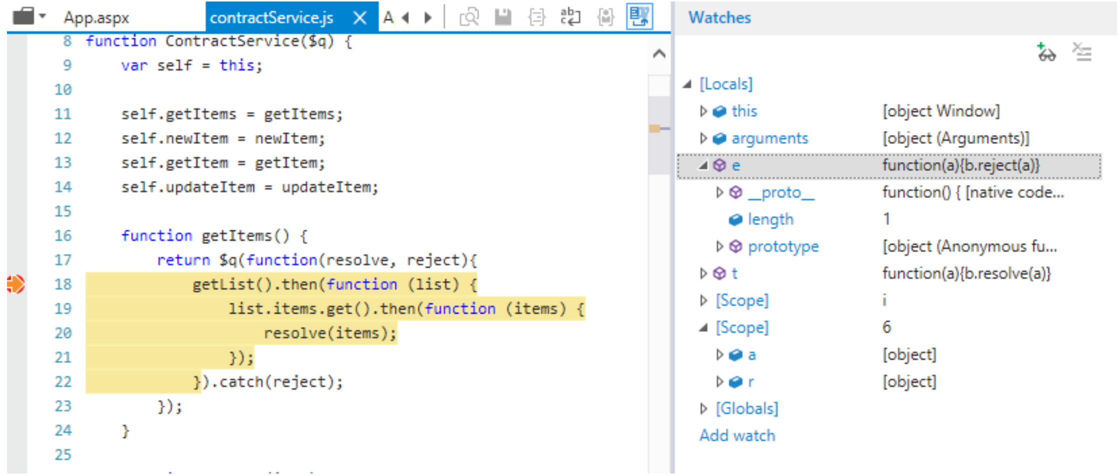
```
11 }, 6: function (t, e, n) {
12     function i(t) {
13         function e() {
14             return t(function (t, e) {
15                 l().then(function (e) {
16                     e.items.get().then(function (e) {
17                         t(e);
18                     });
19                 })["catch"](e);
20             });
21         }
22         function n(e) {
23             return t(function (t, n) {
24                 a.sp.web.lists.getByTitle("ContractRegister").items.add(e).then(function (e) {
25                     t(e.item);
26                 })["catch"](n);
27             });
```

Here – module 6 looks like our contract service.

And if we toggle the Source Map option in our browser debugger:

```
contractService.js   App.js   contractList.js   contractForm.js   contractForm.html

 9      var self = this;
10
11      self.getItems = getItems;
12      self.newItem = newItem;
13      self.getItem = getItem;
14      self.updateItem = updateItem;
15
16      function getItems() {
17          return $q(function(resolve, reject){
18              getList().then(function (list) {
19                  list.items.get().then(function (items) {
20                      resolve(items);
21                  });
22              }).catch(reject);
23          });
24      }
25
```

We are now able to set break points and debug our JavaScript even though it is minified in SharePoint.



Note: We can map the correct lines and set breakpoints, but remember that local variables have been renamed to simple "e", "t" variable names. While we may be expecting "items", it is actually "e" that we should be looking for.

Running Watch Mode

The gulp-spsave documentation already gives an example of how to use a Gulp task to monitor file changes and upload them automatically to SharePoint. This will save us from manually having to run the build/deploy task again. Since we use Webpack to bundle our files, we combine webpack's watch mode with the gulp-watch plugin to upload changes.

```
gulp.task("webpack:build-dev-watch", function () {
	//set webpack watch
	var devWatchConfig = Object.create(devConfig);
	devWatchConfig.watch = true;
	// run webpack
	webpack(devWatchConfig, function (err, stats) {
		if (err) throw new gutil.PluginError("webpack:build-dev-watch", err);
		gutil.log("[webpack:build-dev-watch]", stats.toString({
			colors: true
		}));
	});
});

gulp.task("deploy-watch", ["webpack:build-dev-watch"], function () {
	//need awaitWritefinish to prevent uploading twice
https://github.com/paulmillr/chokidar#api
	watch('./dist/SiteAssets/**/*', { awaitWriteFinish: true })
		.pipe(spsave(spSaveConfig));
});
```

The 'webpack:build-dev-watch' tasks sets 'watch = true' in the Webpack config. In addition, the dev build doesn't minify and uglify the JavaScript, which speeds up the compile time and gives a better debugging experience. The disadvantage here is that the output file sizes will be larger, so depending on the size and upload speed, we may want to use a minified/uglified build and rely on the source map.

In watch mode (and caching enabled) Webpack keeps each module in memory and only recompiles changed output files. Since all our source files are bundled in the app.js file, any change to, for example, index.js or contractservice.js will trigger Webpack to generate a new app.js, but it won't touch vendor.js or app.css files (if nothing has changed in the related files of course).

The Gulp file watcher monitors the "/dist/SiteAssets/" folder for new changes that are then uploaded by spsave every time Webpack rebuilds them.

Calling the 'deploy-watch' task will build all the output files once, upload them and after that only upload changes.

Running Webpack in Memory—the Webpack Dev Server

Webpack has a webpack-dev-server mode that will run and serve Webpack assets directly. What this can let us do is instead of injecting the app.js and vendor.js into the HTML page and upload it to SharePoint, we can inject localhost references to app.js and vendor.js, and have that uploaded to SharePoint.

Locally, the browser will connect the scripts served from webpack-dev-server with other scripts from SharePoint Online.

Webpack automatically watches the source directory we are working in, so if we update and save any of our work, Webpack will automatically rebuild the bundled modules and serve them all from memory, without need to write to file, then upload to SharePoint. This gives us extremely fast testing cycles.

Webpack-dev-server has an automatic refresh option with --inline mode, so when Webpack detects a change within the source code, it will even trigger the SharePoint page to refresh.

In the SharePoint Framework, Microsoft talks of a SharePoint Workbench tool that will allow us to test our customizations 'offline.' We expect the webpack-dev-server to be a related part of this offline story.

Running Webpack Dev Server with Hot Module Replacement

The ultimate craziness in the Webpack world is hot module replacement. Because Webpack loads each dependency as a separate module, it is able to watch for changes to individual modules and replace them at runtime.

As we make a change in our file, Webpack rebuilds the package and then replaces the module that's currently loaded in our webpage with the new code.

Because our HTML and CSS assets are also loaded into script by Webpack, it can change the template that's used to generate and bind HTML in the application too.

All this can happen without us having to refresh the browser; this allows JavaScript current scope to be maintained as the code is switched out.

Running Webpack for Production

For production, Webpack excels at packaging and removing modules from bundles that we aren't actually using. For both Angular, ngOfficeUIFabric and Kendo UI, as these libraries are built from components, Webpack bundling can actually detect which modules we are actually using within our code and will strip out modules that aren't used by us.

For example, if we aren't using Kendo UI ListView control it doesn't get bundled in the vendor.js file, making a custom vendor file that's a lot smaller.

In practice, we find the bundling more difficult to work with when we are testing or debugging issues with the controls. So while this is an exciting option, it should be reserved only for production code.

Getting Started

How to run the Contract-Register demo

1. You will need to have NodeJS installed
2. Fork from https://github.com/johnnliu/contract-register or download as Zip
3. In a Node command-line, go to the contract-register directory and run
    - > npm install
    - This installs all the defined packages within the package.json file
4. Run Gulp tasks
    - > gulp build
    - This will build the project
5. Deploy to your SharePoint
    - Update o365-user.js with your account details and site url
    - > gulp deploy
    - This will build and deploy the JavaScript to your SharePoint site's SiteAssets document library.
6. One-off
    - Create a page with a Content Editor webpart, and point it to the SiteAssets/SPApp.html file.

Summary

In summary, we built a modern Angular application with components using the latest work in PnP JS Core. We built and pack the source code with Webpack, and deploy it to SharePoint with Gulp.

We discuss our choices and describe what makes this set of tools work together so well, as well as potential pitfalls and gotchas one might need to be aware of.

Conclusion

These are the most exciting times as a developer, especially for a SharePoint developer. There are many new tools for modern web technology, and all of them are applicable to us to work in SharePoint On-Premises and SharePoint Online.

The new tools and web stack work well together, but there is no pressure to have to learn everything at once. We are learning, and sharing our learnings. We see new pieces and we try to figure out where they fit. We hope you will take what we've learned and run further with it and be successful.

About the authors

John Liu

John is a Senior Consultant based in Sydney for SharePoint Gurus. He specializes and blogs frequently on client-side scripting, custom development, workflows and Forms.

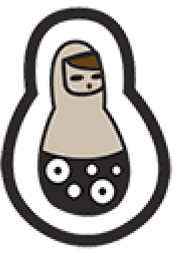
Originally from a technical background in ASP.NET, he made the jump to focus and work with numerous SharePoint projects since WSS. John loves to find ways to bring the latest ASP.NET and web technologies to the SharePoint world, applying the latest web developments to extend SharePoint's capabilities.

Bart Bouwhuis

Bart is a senior SharePoint Consultant at SharePoint Gurus and has focused on client side development since MOSS 2007.

He has implemented various tailored solutions for managing training, projects, assets and contracts within SharePoint and currently uses Kendo UI for the majority of his projects.

SharePoint Gurus



sharepointgurus

SharePoint Gurus is an award-winning consultancy based in Sydney. We specialize in improving productivity through configuring and developing Microsoft SharePoint technologies.

At SharePoint Gurus, we help companies understand what SharePoint is (and often isn't). Using our deep knowledge of the product and our years of practical experience, we will help you plan, configure and implement the right size solution on SharePoint On-Premises or SharePoint Online (Office 365).

Our goal is to provide unparalleled consulting services. Our specialist knowledge will speed up your deployment of SharePoint by helping you understand how it can be effectively implemented to achieve your business goals.

About Progress

Progress (NASDAQ: PRGS) is a global leader in application development, empowering the digital transformation organizations need to create and sustain engaging user experiences in today's evolving marketplace. With offerings spanning web, mobile and data for on-premises and cloud environments, Progress powers startups and industry titans worldwide, promoting success one customer at a time. Learn about Progress at www.progress.com or 1-781-280-4000.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw
For regional international office locations and contact information, please go to www.progress.com/worldwide

Progress is trademark or registered trademark of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.

© 2016 Progress Software Corporation and/or its subsidiaries or affiliates.
All rights reserved.
Rev 16/09 | 160325-0050