Multiple front-ends for Wordpress

Wordpress has had a REST API for several years now, the earliest work on what is now the REST API began in 2013 for version 1 and in 2015 for the current version 2.

This technology caught my attention for several reasons:

1. **It removes WordPress's reliance on PHP.** The REST API enables a much wider pool of developers to interact with the platform. We are no longer tied to PHP and can use any language we choose to build our front end
2. **It allows for new integrations beyond browsers.** Since we can use any language we want to create front ends for Wordpress content we are no longer limited to browser-based front ends. We can create native and hybrid apps without having to rely on what Automattic and the community have to offer (good if you're not that much into React)
3. **We can leverage the web ecosystem without the Wordpress gatekeepers**. Since we're only pulling content from Wordpress, its restrictive licensing and overzealous enforcement become less of an issue (as I understand it).

App Requirementss

This is what I would expect, at a minimum, a Wordpress based application to do:

- Display a list of Posts
- Display an individual Post
- Display all posts under a category
- Display a list of all Pages
- Display an individual Page

Starting from there it shouldn't beee too hard to add additional functionality for revistions, taxonomies and other elements provided by the API endpoints shown in the table below, taken from the wp-api handbook

Resource	Base Route
Posts	/wp/v2/posts

Resource	Base Route
Post Revisions	/wp/v2/revisions
Categories	/wp/v2/categories
Tags	/wp/v2/tags
Pages	/wp/v2/pages
Comments	/wp/v2/comments
Taxonomies	/wp/v2/taxonomies
Media	/wp/v2/media
Users	/wp/v2/users
Post Types	/wp/v2/types
Post Statuses	/wp/v2/statuses
Settings	/wp/v2/settings

Starting Point 1: Polymer Theme

A while back when I was experimenting with Polymer 1.0 I created a set of components to display a Wordpress blog contents using the REST API.

The following element, `wp-blog-list` is a Polymer element that will display the last 10 entries in my Publishing Project blog and format them according to the CSS built into the element.

```
<!-- Imports -->
<link rel="impor<link rel="import" href="../../elements.html">="wp-blog-l

  <template>
    <style is="custom-style">
      body {
  <template>
      body {
        font-family: Lato, Verdana, Helvetica, sans-serif;
        font-size: 16px; /* 1rem = 16px */
        font-weight: 300;
      }

      h1, h2, h3, h4, h5, h6 {
        clear: both;
        font-family: 'Handlee', cursive
        margin-bottom: .25em;
        text-rendering: optimizeLegibility;
      }

      'Handlee'h1 {
        font-size: 3em;
      }

      :host {
        --paper-card-header: {
          background-color: #3f51b5;
          color: white;
          font-family: Lato, Verdana, Helvetica, sans-serif;
          text-align: center;
        }
        ;
      }
    <!-- iron-ajax action -->jax auto url="http://rivendellweb.net/wp-json
      debounce-duration="300"></iron-ajax>

    <!-- local D</iron-ajax><!-- local DOM for your element -->container">
      <template is="dom-repeat" items="{{posts}}" as="post">

        <paper-card heading tab-index='0'>
          <h1>
            <marked-element markdown="[[post.title.rendered]]">
              <div class="markdown-html" tabindex="0"></div>
            </marked-element>
          </h1>
          <div class="post-meta">
            <p>created: [[post.date]]</p>
            <p>last update: [[post.modified]]</p>
            <p>Posted by: [[post._embedded.author.0.name]] in [[post._embe
          </div>
          <!-- closes post-meta --<template is="dom-repeat" items="{{posts
              <img src="[[post._embedded.wp:featuredmedia.0.media_details
            </a>

            <template is="dom-if" if="[[post.content.protected]]">

              <h1>Private Post</h1>

              <p>This is a protected post and the content is not available

            </template>
            <marked-element markdown="[[post.content.rendered]]">
              <div class="markdown-html" tab-index='0'></div>
            </marked-element>

          </div>
          <!-- closes card-content -->
        </paper-card>

      </template>
    </di<img src="[[post._embedded.wp:featuredmedia.0.media_details.sizes
      properties: {}
    });
  </script>
</dom-module>
</script>
    // element registration
    Polymer({
      is: "wp-blog-list",
      proper"wp-blog-list");
```

I'm not certain if the code will still work, but it was an interesting exercise and it may be worth updating it to Polymer 3 when it releases… I'm particularly curious about using lit-html in a component, if it's possible or not.

Starting Point 2: Handlebars Test

The second, most recent, example is a Handlebars template populated with the results of a fetch request to the Wordpress REST API.

This example is broken in two sections. The first one is the HTML portion that contains two items:

The first one is the HTML portion that contains:

- The place holder where we will place the content once it’s processed
- The template that shows the structure of the content to be rendered. It uses the `template` element to ensure the content will remain inert until instantiated

```
<div id="myContent"></div>

  <template id="post-list-template">

    {{#each posts}}
    <div class="post">
      <h1>{{title.rendered}}</h1>
      <div>
        {{{content.rendered}}}
      </div>
    </div>
    {{/each}}

  </template>
```

The second part is the Javascript that will fetch the JSON from the Wordpress API, convert it to JSON and then use the JSON to populate the template (one for each post returned by the API).

```
let myPosts = fetch(
  'https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?per_pag
)
```

```
.then(response => {
  return response.json();
})
.then(myJson => {
  let template = document.getElementById('post-list-template').innerHTM
  let renderPosts = Handlebars.compile(template);

  document.getElementById('myContent').innerHTML = renderPosts({
    posts: myJson
  });
})

.catch(err => {
  console.log("There's been an error getting the data", err);
});
```

You can see the working result in this Pen:

So, we can get data out of Wordpress using the REST API but there's much work to do. We'll build the application that will read and display the data and then spend time making it look pretty.

Getting the environment set

I've chosen to use Angular 5 and the soon to be released Angular 6 for the proof of concept.

I am not as familiar with newer versions of Angular as I'd like to be and want to make sure the tools don't become too much of an obstacle in the development process.

That said, let's get started.

Installing Angular CLI

Install the Angular CLI as a global module to gain access to the ng command. We'll use the command later.

```
npm install -g @angular/cli
```

Create the application

Using Angular CLI, I run the following command to generate the skeleton of the application:

```
ng new wp-angular
```

- ng is the CLI application anme
- new is the CLI command to run
- `wp-angular` is the name of the application we are creating

This will create the skeleton of the application.

```
cd wp-angular
```

Change to your application directory.

```
ng serve --open
```

And finally, run the ng `serve --`open command to start the built-in application server and automatically open the default browser to the application's index page.

So now we have a working Angular application running on the server. Let's start exploring what it'll take to add the REST backend and a front-end UI for it.

The Actual Code

I stopped working with Angular shortly after the Angular 2 release so this will be learning on the job and, as such, may not look right to Angular purists.

The idea is to, eventually, have the following API calls work with a Wordpress installation:

- **get/posts** to get a page of posts
- **get/posts/id** to get a single post by ID

Eventually I will want the following additional routes working for our content.

- **post/post** to add a post
- **put/post/id** to update an existing post
- **delete/post/id** to delete a post

Basic CRUD functionality at your service.

Creating the Wordpress API service

Rather than use the same call over and over we'll use a service to hold all our API calls in one place and where we can update them without having to go through all the elements the calls touch.

The command to generate the service is

```
ng generate service post
```

The command will generate a skeleton service for us to populate. The end result will look like the code below.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class PostService {
  private postsUrl = '<server>/wp-json/wp/v2/posts';

  constructor(private http: HttpClient) {}

  getPosts() {
    return this.http.get(this.postsUrl);
  }
}
```

TO incorporate the service to the application, you need to do the following in your `app.module.ts` file at the root of the application:

1. Import the service (**import { PostService } from './post.service';**)
2. Add PostService to the providers entry

In the code I'm working on I'm using one of my blogs, but it should work with any modern version of Wordpress (4.4 with the REST API plugin or 4.7 or later to use the API built on Wordpress Core) when self installed; don't know if it'll work with a Wordpress.com account.

The Wordpress API will return the 10 most recent posts. A future iteration of the service will also handle pagination so we can create links to other pages of content.

The Index Page: List of posts

To generate the listing of posts in the front page, we will create a `post-list` commponent. We'll leverage the Posts service we created in the previous step to pull in the data from our Wordpress instance.

```
ng generate component post-list
```

The component is broken into four parts:

- **posts-list.component.ts** is the Typescript file that holds the component's code
- **posts-list.component.html** holds the component's template
- **posts-list.component.css** contains all the styles that are specific to the component
- **posts-list.component.spec.ts** is a Typescript file that contains all the tests for the element. This is the file that will run when we execute ng `test` from the root of the application

We'll look at each of these files in turn.

Component Definition

The component definition for the post-list element does a few things:

- Imports items from the core Angular library
- Imports the Post service
- Uses the Component decorator to provide metadata about the component
  - What selector we'll use for the component
  - Where is the template for the element. In this case we've chosen to use an external file

    - Where are the style sheets for the component (you can have one or more)
- The class that defines the component

In the class we do a few things to make our page and the Posts service interact.

In the constructor, we pass an instance of Post Service as a private member. This makes Post Service available to the component.

When we initialize the component we call getPosts(), a function that will retrieve posts from the Wordpress instance we're using as CMS.

Finally we define the `getPosts()` function to pull the data from the REST endpoint, subscribe to it and then run the corresponding function if we succeed or fail.

Subscribe is part of Angular's Observables infrastructure. I don't think this particular application of observables merits using RxJS which would be the next step up in working with Observables.

```
import { Component, OnInit } from '@angular/core';
import { PostService } from '../post.service';

@Component({
  selector: 'posts-list',
  templateUrl: './posts-list.component.html',
  styleUrls: ['./posts-list.component.css']
})
export class PostsListComponent implements OnInit {
  public posts;

  constructor(private _postsService: PostService) {}

  ngOnInit() {
    this.getPosts();
  }

  getPosts() {
    this._postsService.getPosts().subscribe(
      data => {
        this.posts = data;
      },
      err => console.error(err),
      () => console.log(this.posts)
    );
  }
}
```

We’ll come back to the Service when we add a single post component and again when we revisit other parts of the Wordpress API.

Component Template

Angular templates use their own weird syntax and this component is no exception.

We create an outer wrapper element that will hold the posts we’ll generate with the template.

Each post will be an `article` element and here we use the first special feature of Angular templates: a directive. In this case we use `*ngFor="let post of posts"` as an attribute. This attribute will loop through all the available elements returned by the service.

We add the title of the post using a double mustache syntax, similar to Handlebars.js

To render the content we need to use an innerHTML binding attribute and pass it to a custom pipe that will prevent the content from being sanitized.

It would be tempting to write the content template like we wrote the title (and that was my first version). It looked like this

```
<section>
{{post.content.rendered}}
</section>
```

The problem was that Angular would escape all the HTML tags inside the content and that was not what I wanted to do. If we were taking in third party content this would be ok to filter but we are taking in content from a trusted source so we'll work on a version 2 so we can bypass the filter.

This is where property binding comes to the rescue. We bind the `innerHTML` attribute to the value of the post content and then we run it through a custom pipe that will prevent Angular from sanitizing the data before it's displayed in the template.

The final version of the template looks like this:

```html
<div class="wrapper">
  <article class="post" *ngFor="let post of posts">
    <h1>{{post.title.rendered}}</h1>

    <section
      class="content"
      [innerHTML]="post.content.rendered | safe: 'html'">
    </section>


  </article>
</div>
```

Before it will work you have two final things to do in `app.module.ts`:

1. Import the component
2. Add the module to the declaractions section of the `ngModule` decorator

Component Styles

The final piece of our component is the styling. The styles in this file are encapsulated to the element; they will not bleed out into our parent element or to other elements of the page it is in.

```css
.post {
  width: 80%;
  margin: 1em auto;
  padding: 1em 0;
  border: 2px solid rebeccapurple;
  border-radius: 5px;
}

.content {
  margin: 2em;
}
```

Creating a Custom Pipe

Only do this if you trust the data you're working with. Otherwise always sanitize the data and omit the safe pipe on your templates.

When we looked at the `posts-lists` template we saw the pipe used like this:

```
<section
  class="content"
  [innerHTML]="post.content.rendered | safe: 'html'">
</section>
```

The pipe will bypass different levels of sanitization. We can afford to do this because we trust the content being sent from Wordpress.

The pipe script will take the content and pass it through one of five different sanitizing options available in Angular.

```
import { Pipe, PipeTransform } from '@angular/core';
import { DomSanitizer, SafeHtml, SafeStyle, SafeScript,
  SafeUrl, SafeResourceUrl
} from '@angular/platform-browser';

@Pipe({
  name: 'safe'@angular/platform-browser'implements PipeTransform {
  constructor(protected sanitizer: DomSanitizer) {}
  public transform(
    value: any,
    type: string
  ): SafeHtml | SafeStyle | SafeScript | SafeUrl | SafeResourceUrl {
    switch (type) {
      case 'html':
        return this.sanitizer.bypassSecurityTrustHtml(value);
      case 'style':
        return this.sanitizer.bypassSecurityTrustStyle(value);
      case 'script':
        return this.sanitizer.bypassSecurityTru'html'pt(value);
      case 'url':
        return this.sanitizer.bypassSecurityTrustUrl(value);
      case 'resourceUrl':
        return this.sanitizer.bypassSecurityTrustResourceUrl(value);
      default:
        throw new Error('url'`Invalid safe type specified: ${type}`);
    }
  }
}
```

To add the pipe to the application, open `app.module.ts` and do the following:

1. Import the pipe
2. Add the pipe to the declaration section of your application

We can use this in addtion to the built-in pipes Angular provides: `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. There are other pipes discussed in the Angular API Guide.

Creating a Router

Part of creating this Wordpress-as-CMS consumer application is learning how to create a router for the application to know how to handle different parts of the application. This will also help with creating a single post component that will load the content from the CMS and then populate the template with a single post, rather than 10 of them at a time, the default for Wordpress.

to createe the module use the following command:

```
ng g module app-routing
```

It's important to note we're working with module, not a component, so the imports are different than those we've worked with before.

THe important part, for me, is the routes definition. As currently defined, there are three routes

1. A default route (`path=''`) that will match when the plain URL is entered. It redirects to the posts route
2. A posts route that will match both the default empty route and the route ending in posts
3. A wildcard route that will direct to an error component

The only other element to consider is the imports section of the ng`Module` decorator. Instead of importing the plain `RouterModule` it imports `RouterModule.forRoot()` to indicate that this is the master router for the application. There may be instances where we create more than one router and only the one at the top of the application would get the `forRoot()` attribute.

The current version of the router looks like this.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { PostsListComponent }
  from './posts-list/posts-list.component';
import { NotFoundComponent }
  from './not-found/not-found.component';

const routes: Routes = [{
    path: '',
    redirectTo: 'posts',
    pathMatch: 'full'
  },
  {
    path: 'posts',
    component: PostsListComponent
  },
  {
    path: '**',
    component: NotFoundComponent
  }];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  providers: []
})
export class AppRoutingModule {}
```

We'll add more routes as we had things to the app.

To integrate the router to the application, open `app.module.ts` at the root of the application and:

1. Import the router module
2. Add the router module to the imports section of your @ngModule decorator

Adding Individual Post Page

One of the things I realized early on is that I've always had a hard time with the concept of creating individual pages and this time it's not exception. Some of the challenges I see are:

1. How do we provide an id or a unique 'slug' for the page to link to?
2. How do I create a link that will pull the JSON data to populate a template rather than link to the external source?

Creating the single post template will require a new component and changes to our existing files. We'll point out what file we're changing when we do so.

The new component

As usual we’ll brake the component into it’s three main parts, the Typescript file containing the definition, the HTML file holding the templates and the styles in the CSS file.

The component definition leverages our posts service to retrieve a single post matching the ID we provide in the link and we get through the API.

I’ve also leveraged Angular’s [Activated Route]() to get the value of the ID returned by the API.

I’ve hit somewhat of a snag. Angular shows me an error in the console indicating that `this.route.params.value.id` doesn’t exist in the params observable.

When I first start the application in development mode it errors out and doesn’t compile. However, if I remove it and add it back the application compiles and runs without a problem.

```
import { Component, OnInit, Injectable } from '@angular/core';
import { ActivatedRoute } from "@angular/router";
import { PostService } from '../post.service';

@Component({
  selector: 'post-single',
  templateUrl: './post-single.component.html',
  styleUrls: ['./post-single.component.css']
})

@Injectable({
  providedIn: 'root',
})
export class PostSingleComponent implements OnInit {
  public post;
  public id;

  constructor(
    private _postService: PostService,
    private route: ActivatedRoute
  ) {}

  ngOnInit() {
    this.getPost(this.route.params.value.id + "?_embed");
  }

  getPost(id) {
    this._postService
      .getPost(this.route.params.value.id)
      .subscribe(
        data => {
          this.post = data;
        },
        err => console.error(err),
        () => console.log(this.post)
      );
  }
}
```

The template is very similar to the template for `post-list` the only difference is that we removed the *ngFor loop since we want to produce a single post, not a series of them.

```
<article class="post">
  <h1>{{post.title.rendered}}</h1>

  <section class="content" [innerHTML]="post.content.rendered | safe: 'htr
</article>
```

The CSS, again, is very smilar to the styles for `posts-list` since the template we are rendering individual items that are the same as those on the listing.

```
.post {
  width: 80%;
  margin: 1em auto;
  padding: 1em 0;
  border: 2px solid rebeccapurple;
  border-radius: 5px;
}

.content {
  margin: 2em;
}
```

We make one change to the `posts.list.component.html` template to give ourselves a way to access the individual posts. The change is just adding a link to the post using it's ID, otherwise it's identical to the template for `posts-listing`.

```
<h1><a href="posts/{{post.id}}">{{post.title.rendered}}</a></h1>
```

Basic post components are working. The rest, as they say, is gravy :).

Troubleshooting single post route issue

As documented in the creating single post section Angular is giving me an error that prevents running for the first time but works in subsequent runs of the application.

I'm wondering if this has to do with ActivatedRoute and the way it is pulling the content and nothing to do with the rendering of the page.

Links to other parts of the application

Discovery and Authentication

Discovery

Authentication

Creating an editing framework

Caching content

Pages

Categories

Injecting third party libraries: Prism.js

My blog posts make heavy use of Prism.js a syntax highlighter to make the code easier to read on the page.

- markup
- css
- clike
- javascript
- apacheconf
- bash
- csharp
- ruby
- css-extras
- docker
- git
- go
- graphql
- handlebars
- java
- json
- markdown
- nginx
- perl
- php
- python
- typescript

- scss
- yaml

And the following plugins

- toolbar
- command-line
- keep-markup
- show-language
- copy-to-clipboard

NPM Package

Code syntax highlighting with Angular and Prism.js

Moving to a framework: Bootstrap or Material Design?

Router Links

Working with response headers

Links and Resources

- WP API Handbook
- wp-api Node client
- Angular.JS
- The Ultimate Angular CLI Reference Guide
- Angular and RxJS: Adding a REST API Back End