

JavaScript APIs in WordPress

Adam Silverstein

@ROUNDEARTH

[GITHUB.COM/
ADAMSILVERSTEIN](https://github.com/adamsilverstein)



thanks everyone for coming to my talk. lets jump right in because we have a lot to cover in 30 minutes

i gave it a nice short title to get you in the door although I'm taking about a bit more than APIs

In fact we are going to cover a huge amount of tools in this talk and my goal is to overwhelm you with information and ideas. my hope is you will come out of this talk inspired to create something new and exciting

github.com/adamsilverstein/
wceu2018

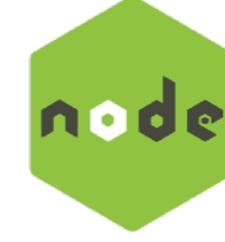
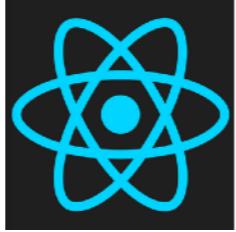
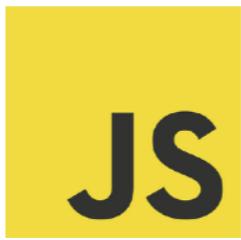
all of the links i mention in this talk, and the slideshow itself will be available later today at this github url which i'll also put up at the end of the talk
don't worry if you miss a detail or can't see the code samples as I'll be moving pretty quickly.

Learn JavaScript Deeply

Remember when Matt told us to learn JavaScript deeply a few years ago?

I hope you have taken that to heart with your themes and plugins and started using and learning more JavaScript.

Learn JavaScript Deeply



The DEEPLY part is learning the new build tools and frameworks that make up the modern JavaScript stack

Soon you may building a feature, theme or plugin that is primarily or entirely JavaScript



webpack

Speaking of building, i'm proud to say that thanks to Omar Reiss, WordPress recently introduced a web pack based build process to our development workflow

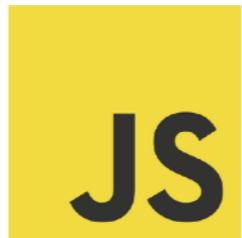
```
src/js
└── _enqueue
    ├── admin
    ├── deprecated
    ├── lib
    ├── vendor
    └── wp
        ├── customize
        ├── editor
        ├── media
        ├── utils
        └── widgets
└── media
```

| All the JavaScript source.
| Any script that ends up being enqueued.
| Procedural scripts ran in the admin.
| All deprecated scripts.
| All standalone lib scripts.
| All 3rd party deps that can't be managed with NPM.
| All scripts that assign something to the wp namespace.
| Anything under wp.customize.
| Anything under wp.editor.
| Anything under wp.media.
| Anything under wp.utils.
| jQuery widgets on the wp namespace.
| The media library.

this era bled us to reorganize our JavaScript in a sensible way while building it back into its legacy location for backwards compatibility

Of course, this is also sets up core for merging Gutenberg and its modern JavaScript coding practices

Hello WordPress



WordPress comes with many libraries and tools to help your JavaScript project and with the REST API and soon Gutenberg, developing for WordPress is increasingly a JavaScript task.

Hello `window.wp`

Historically as we have built new JavaScript features in core, we have added them to the global `wp` namespace or `window.wp`

```
> wp
< ▼ {emoji: {...}, heartbeat: {...}, autosave: {...}, ally: {...}, utils: {...}, ...} ⓘ
  ► Backbone: {Subviews: f, View: f}
  ► Uploader: f ( options )
  ► ally: {speak: f}
  ► ajax: {settings: {...}, post: f, send: f}
  ► apiRequest: f ( options )
  ► autosave: {getPostData: f, getCompareString: f, disableButtons: f, enableButtons: f, lo
  ► editor: {autop: f, removep: f, initialize: f, remove: f, getContent: f, ...}
  ► emoji: {parse: f, test: f}
  ► heartbeat: {hasFocus: f, connectNow: f, disableSuspend: f, interval: f, hasConnectionEr
  ► html: {attrs: f, string: f}
  ► mce: {views: {...}, View: f}
  ► media: f ( attributes )
  ► mediaelement: {initialize: f}
  ► oldEditor: {autop: f, removep: f, initialize: f, remove: f, getContent: f, ...}
  ► receiveEmbedMessage: f ( e )
  ► shortcode: f ( options )
  ► svgPainter: {init: f, setColors: f, findElements: f, paint: f, paintElement: f}
  ► template: f ( key )
  ► utils: {WordCounter: f}
  ▶ ... Object
```

The WP namespace is modular so different screens may have a different wp object and you can use the parts you need by enqueueing them

```
> wp
< ▼ {emoji: {...}, heartbeat: {...}, autosave: {...}, ally: {...}, utils: {...}, ...} ⓘ
  ► Backbone: {Subviews: f, View: f}
  ► Uploader: f ( options )
  ► ally: {speak: f}
  ► ajax: {settings: {...}, post: f, send: f}
  ► apiRequest: f ( options )
  ► autosave: {getPostData: f, getCompareString: f, disableButtons: f, enableButtons: f, lo
  ► editor: {autop: f, removep: f, initialize: f, remove: f, getContent: f, ...}
  ► emoji: {parse: f, test: f}
  ► heartbeat: {hasFocus: f, connectNow: f, disableSuspend: f, interval: f, hasConnectionEr
  ► html: {attrs: f, string: f}
  ► mce: {views: {...}, View: f}
  ► media: f ( attributes )
  ► mediaelement: {initialize: f}
  ► oldEditor: {autop: f, removep: f, initialize: f, remove: f, getContent: f, ...}
  ► receiveEmbedMessage: f ( e )
  ► shortcode: f ( options )
  ► svgPainter: {init: f, setColors: f, findElements: f, paint: f, paintElement: f}
  ► template: f ( key )
  ► utils: {WordCounter: f}
  ► ... - Object
```

This is the wp object when you load the post page - I typed wp into the console which is a great way to explore especially since we don't yet have a centralized place for JavaScript code documentation.

this talk will cover some of the more interesting and useful tools and hopefully inspire you to leverage them in your next project.

Documentation

make.wordpress.org/core/2018/01/31/js-docs-initiative-add-inline-docs-for-javascript/

atimmer.github.io/wordpress-jsdoc/wp.media.html

herregroen.github.io/wordpress-jsdoc/

Anton Timmermans is leading an effort to add inline documentation for all of our JavaScript which lets us generate a JSDoc guide, and ultimately create a developer handbook much the same way we currently generate our PHP code reference.

As a reminder, all these links will be available on my GitHub later today.

— — —



Oh! this slide has nothing to do with my presentation. I added it here to remind myself to take a breather and a sip of water

wp.editor

Add a rich HTML editor to your app from JavaScript

Lets dive in! OK - wp editor!

Wp.editor lets you transform a text area into a rich HTML editor using the same TinyMCE editors use in the post edit screen, including custom plugins you want to support.

Links:

https://codex.wordpress.org/Javascript_Reference/wp.editor

<https://make.wordpress.org/core/2017/05/20/editor-api-changes-in-4-8/>

wp.editor

```
▼ editor:  
  ► autop: f wpautop( text )  
  ► dfw: {activate: f, deactivate: f, isActive: f, on: f, off: f, ...}  
  ► getContent: f ( id )  
  ► initialize: f ( id, settings )  
  ► remove: f ( id )  
  ► removep: f pre_wpautop( html )  
  ► __proto__: Object
```

This is perfect for adding a rich editor to a meta box or a front end tool you are building in JavaScript. Again, I'll have links to all the documentation i could find for each object posted with my slides.

— — —

Links:

https://codex.wordpress.org/Javascript_Reference/wp.editor

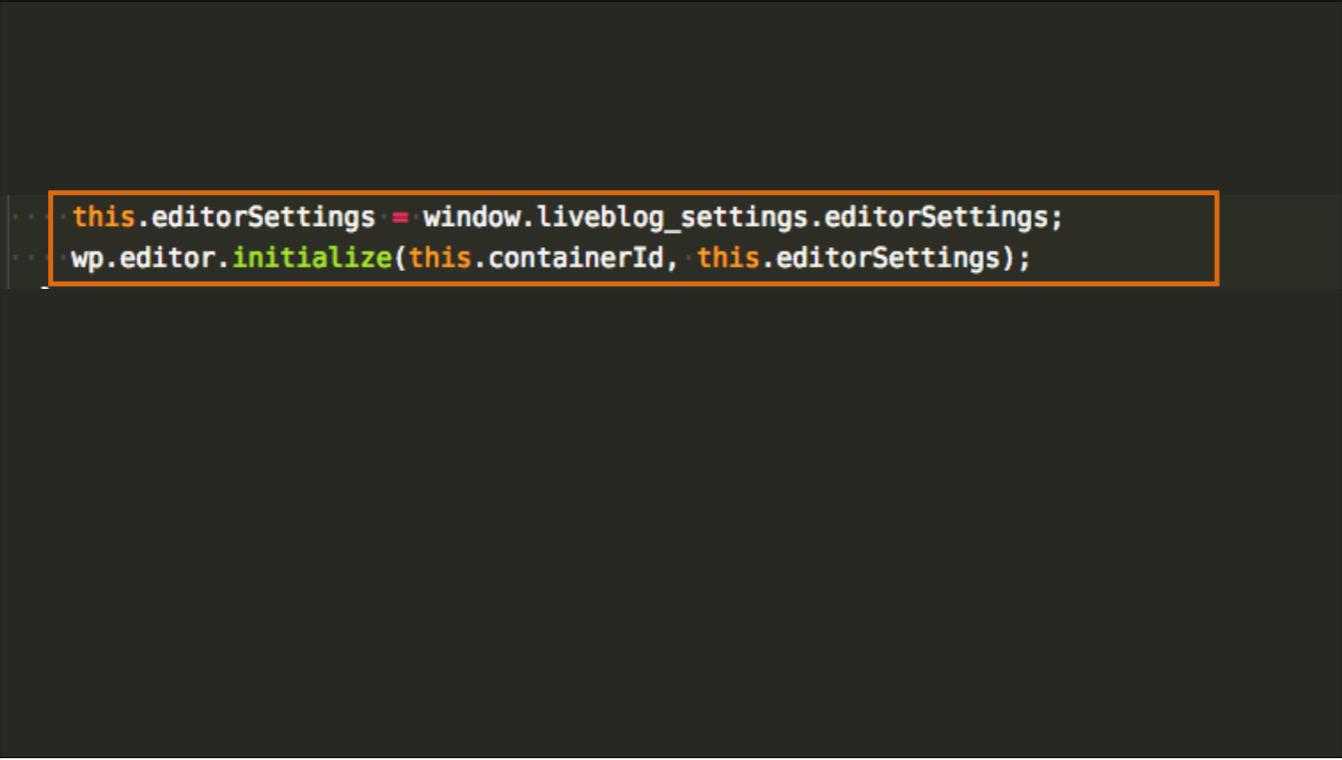
<https://make.wordpress.org/core/2017/05/20/editor-api-changes-in-4-8/>

```
1 import React, { Component } from 'react';
2
3 export const getTinyMCEContent = () => {
4   const currentEditor = tinymce.activeEditor;
5
6   return currentEditor ? currentEditor.getContent() : '';
7 };
8
9 export const clearTinyMCEContent = () => {
10   const currentEditor = tinymce.activeEditor;
11   currentEditor.setContent('');
12 };
13
14 class TinyMCEEditor extends Component {
15   constructor(props) {
16     super(props);
17     this.containerId = `live-editor-${Math.floor(Math.random() * 100000)}`;
18     this.editorSettings = window.liveblog_settings.editorSettings;
19     wp.editor.initialize(this.containerId, this.editorSettings);
20   }
21 }
```

Here is an example of a react component that uses wp.editor. I built this recently to enhance the liveblogging plugin from Automatic to optionally replace the react based editor uses by default

```
    render() {
      return <textarea className="liveblog-editor-textarea" id={this.containerId}>/>;
    }
  
```

Here is how it works - first in the component's render method, we output a text area for the editor onto the page



```
this.editorSettings = window.liveblog_settings.editorSettings;
wp.editor.initialize(this.containerId, this.editorSettings);
```

later, I call wp.editor to initialize the TinyMCE editor. This gives editors the same wp-editor experience they are used to in wp admin. I expect eventually we will have similar functionality for adding a standalone gutenberg editor in the same way.

— — —

<https://github.com/youknowriad/standalone-gutenberg>

wp.codeEditor

Add a code editor or syntax highlighting to a textarea.

wp codeEditor provides a code editor and syntax highlighting for text areas

make.wordpress.org/core/2017/10/22/code-editing-improvements-in-wordpress-4-9/

*CodeMirror is a versatile text editor implemented in
JavaScript for the browser.*

we included code mirror in core in wordpress 4.9 to enhance code editing in the theme and plugin editors and CSS editing in the customizer.

The screenshot shows the WordPress theme editor interface. At the top, it says "Twenty Seventeen: content-page.php (template-parts/page/content-page.php)". Below that is a dropdown menu "Select theme to edit: Twenty Seventeen" with a "Select" button. The main area is titled "Selected file content:" and contains the following PHP code:

```
1 <?php
2 /**
3  * Template part for displaying page content in page.php
4  *
5  * @link https://codex.wordpress.org/Template_Hierarchy
6  *
7  * @package WordPress
8  * @subpackage Twenty_Seventeen
9  * @since 1.0
10 * @version 1.0
11 */
12
13 ?>
14
15 <article id="post-<?php the_ID(); ?>" <?php post_class(); ?>>
16     <header class="entry-header">
17         <?php the_title( '<h1 class="entry-title">', '</h1>' ); ?>
18         <?php twentyseventeen_edit_link( get_the_ID() ); ?>
19     </header><!-- .entry-header -->
20     <div class="entry-content">
21         <?php
22             the_content();

```

To the right is a sidebar titled "Theme Files" with a tree view of theme files:

- Sidebar (sidebar.php)
- Single Post (single.php)
- template-parts
 - footer ▶
 - header ▶
 - navigation ▶
- page ▾
 - content-front-page-panels.php
 - content-front-page.php
 - content-page.php**
- post ▶
- README.txt
- test.txt

At the bottom left is a "Documentation: Function Name..." dropdown and a "Look Up" button. At the bottom center is a blue "Update File" button.

now in the theme editor you get a rich editor display with code highlighting.

you can use this same functionality in your own plugin or theme, wherever you want to to provide an editor or syntax highlighting of code.

Manage Ads.txt

Your Ads.txt contains the following issues:

Line 3: wrongdomain does not appear to be a valid subdomain
Line 6: Invalid record
Line 10: not-an-exchange does not appear to be a valid exchange domain
Line 10: Third field should be RESELLER or DIRECT
Line 10: f08c47fec0942 does not appear to be a valid TAG-ID

```
1 # This is a comment
2 contact=test@example.com
3 subdomain=wrongdomain
4 subdomain=sub.domain.com
5
6 Invalid record
7
8 # Records
9 google.com, pub-1234567890, DIRECT, f08c47fec0942fa0
10 not-an-exchange, pub-1234567890, INVALID, f08c47fec0942
```

Here is an example of using code mirror in our ads.txt plugin: we used wp.codeMirror to add line numbers and syntax highlight the ads.txt edit field, making it easier to spot potential errors.

```
<textarea class="widefat code" rows="25" name="adstxt" id="adstxt_content"><?php echo esc_textarea( $content ); ?></textarea>
```

The code for this is pretty straightforward... First we add a text area to the dom with any existing content

```
<textarea class="widefat code" rows="25" name="adstxt" id="adstxt_content"><?php echo esc_textarea( $content ); ?></textarea>

wp_enqueue_script( 'adstxt', esc_url( plugins_url( '/js/admin.js', dirname( __FILE__ ) ) ), array( 'jquery', 'wp-backbone', 'wp-codemirror' ), false, true );
wp_enqueue_style( 'code-editor' );
```

next we enqueue our javascript, adding wp-codemirror as a dependency, and also enqueueing the required code-editor styles

```
<textarea class="widefat code" rows="25" name="adstxt" id="adstxt_content"><?php echo esc_textarea( $content ); ?></textarea>

wp_enqueue_script( 'adstxt', esc_url( plugins_url( '/js/admin.js' ), dirname( __FILE__ ) ), array( 'jquery', 'wp-backbone', 'wp-codemirror' ), false, true );
wp_enqueue_style( 'code-editor' );

var editorArea = document.getElementById('adstxt_content');
editorOptions = { lineNumbers: true, mode: 'shell' };
editor = wp.CodeMirror.fromTextArea( editorArea, editorOptions );
```

Finally, in our javascript, we call wp.CodeMirror.fromTextArea on our text area, passing control options to turn it into a rich editor

wp.media

Leverage and extend the WordPress media library.

wp media lets you interact with the media library and extend it with custom functionality

there are many examples in plugins and tutorials about how to use wp.media especially for operations like inserting an image from the media library

wp.media

```
▼ media: f ( attributes )
  ► View: f ( options )
  ► ajax: f ( action, options )
  ► attachment: f ( id )
  ► audio: {coerce: f, defaults: {...}, edit: f, shortcode: f}
  ► coerce: f ( attrs, key )
  ► collection: f ( attributes )
  ► compare: f ( a, b, ac, bc )
  ► controller: {Region: f, StateMachine: f, State: f, Library: f, ImageDetails: f, ...}
  ► editor: {insert: f, add: f, id: f, get: f, remove: f, ...}
  ► embed: {coerce: f, defaults: {...}, edit: f, shortcode: f}
  ► events: {on: f, listenTo: f, off: f, stopListening: f, once: f, ...}
  ► featuredImage: {get: f, set: f, remove: f, frame: f, select: f, ...}
  ► fit: f ( dimensions )
  ► frame: N.d {views: wp.B...e.Subviews, states: b.Collection, options: {...}, cid: "view-frame-1", ...}
  ► frames: {}
  ► gallery: {coerce: f, attachments: f, shortcode: f, edit: f, setDefaults: f, ...}
```

Extending wp.media to add custom functionality can be tricky though and there is no good central documentation about how to extend it

wp.media

```
▶ frames: {}
▶ gallery: {coerce: f, attachments: f, shortcode: f, edit: f, setDefaults: f, ...}
▶ galleryDefaults: {itemtag: "dl", icontag: "dt", captiontag: "dd", columns: "3", l:
  isTouchDevice: false
▶ mixin: {mejsSettings: {...}, removeAllPlayers: f, removePlayer: f, unsetPlayers: f}
▶ model: {l10n: {...}, settings: {...}, Attachment: f, Attachments: f, Query: f, ...}
▶ playlist: {coerce: f, attachments: f, shortcode: f, edit: f, setDefaults: f, ...}
▶ post: f ( action, data )
▶ query: f ( props )
▶ selectionSync: {syncSelection: f, recordSelection: f}
▶ string: {props: f, link: f, audio: f, video: f, _audioVideo: f, ...}
▶ template: f ( e )
▶ transition: f ( selector, sensitivity )
▶ truncate: f ( string, length, replacement )
▶ video: {coerce: f, defaults: {...}, edit: f, shortcode: f}
▶ view: {l10n: {...}, settings: {...}, Frame: f, MediaFrame: f, Modal: f, ...}
▶ _galleryDefaults: {itemtag: "dl", icontag: "dt", captiontag: "dd", columns: "3", l:
```

media has a large number of views and models and it can be confusing to decipher what to extend.

Media Guide Plugin

*github.com/ericandrewlewis/
wp-media-javascript-guide*

My favorite guide to wp.media is this interactive plugin by Eric Lewis. Eric also gave a talk at WordCamp Philly 2014 you can find on [wordpress.tv](#)

--

<https://wordpress.tv/2014/09/25/eric-andrew-lewis-an-introduction-to-wp-media/>

The screenshot shows the WordPress admin interface with a sidebar on the left and a main content area on the right.

Sidebar (Left):

- Dashboard
- Posts
- Media
- Pages
- Comments
- Appearance
- Plugins (2)
- Users
- Tools
- Settings
- Gutenberg
- Media Guide** (highlighted in blue)

Main Content Area (Right):

WordPress Media Backbone Guide

Sections ▾

1. [Introduction](#)
2. [Old Media Modal](#)
3. [attachment_fields_to_edit](#)
4. [wp.Backbone.View](#)
5. [wp.media](#)
6. [wp.media\(\)](#)
7. [wp.media.controller.Region](#)
8. [wp.media.controller.State](#)
9. [wp.media.controller.StateMachine](#)
10. [wp.media.events](#)
11. [wp.media.view.MediaFrame.Post](#)
12. [wp.media.view.MediaFrame.Select](#)
13. [wp.media.view.Modal](#)
14. [wp.media.view.PriorityList](#)
15. [wp.media.view.UploaderWindow](#)

once you install the plugin, you will see a media guide in wp-admin with chapters covering the major wp.media components

WordPress Media Backbone Guide

Sections ▾

wp.Backbone.View

Extends Backbone.View.

Base view on top of which all views in WordPress are built on.

A Subview Manager is baked in via wp.Backbone.Subviews.

Example: Render a view with a subview

LIVE EXAMPLE [open in a new window](#)

Click to render the parent view

MARKUP

```
<div class="view-1-container"></div>
<button class="js--render-view-1">Click to render the parent view</button>
<script type="text/template" id="tmpl-view-1">
  A view template.
  <div class="Subview-Container"></div>
</script>
```

And since the guide is a plugin, it includes live samples you can interact with like this demo showing how to add an image upload button

Attachment taxonomies

wordpress.org/plugins/attachment-taxonomies/



For a great example of extending wp.media with custom functionality, check out Attachment Taxonomies by Felix Arntz.

The plugin adds categories and tags to the WordPress media library and lets you filter by them.

```
wp.media.view.AttachmentFilters.Taxonomy = wp.media.view.AttachmentFilters.extend({
    id: 'media-attachment-taxonomy-filters',
    createFilters: function() {
        var filters = {};

        if ( this.options.queryVar && this.options.allLabel ) {
            filters.all = {
                text: this.options.allLabel,
                props: {},
                priority: 1
            };
            filters.all.props[ this.options.queryVar ] = null;
        }
    }
});
```

Looking quickly at how it works, we see that the plugin follows the standard pattern to extend the core media objects with custom functionality.

```
wp.media.view.AttachmentFilters.Taxonomy = wp.media.view.AttachmentFilters.extend({  
    id: 'media-attachment-taxonomy-filters',  
    createFilters: function() {  
        var filters = {};  
  
        if ( this.options.queryVar && this.options.allLabel ) {  
            filters.all = {  
                text: this.options.allLabel,  
                props: {},  
                priority: 1  
            };  
            filters.all.props[ this.options.queryVar ] = null;  
        };  
    }  
});
```

Here, the code extends the core media AttachmentsFilters view, replacing its createFilters method with one that includes support for the media categories and tags.

```
wp.media.view.AttachmentsBrowser = wp.media.view.AttachmentsBrowser.extend({
    createToolbar: function() {
        wp.media.view.AttachmentsBrowser.__super__.createToolbar.apply( this,
            [
                {
                    title: 'Attachments',
                    subtitle: 'Select an attachment'
                }
            ]
        );
        var data = wp.media.taxonomies.data;
        for ( var i in data ) {
            this.toolbar.set( data[ i ].slug + 'FilterLabel', new wp.media.view.Button(
                {
                    title: data[ i ].label,
                    value: wp.media.taxonomies.l10n.filterBy[ data[ i ].slug ],
                    attributes: {
                        'for': 'media-attachment-' + data[ i ].slugId + '-filters'
                    },
                    priority: -72
                }
            ).render() );
            this.toolbar.set( data[ i ].slug + 'Filter', new wp.media.view.AttachmentFilter(
                {
                    controller: this.controller,
                    title: data[ i ].label
                }
            ).render() );
        }
    }
});
```

the plugin then replaces the core AttachmentsBrowser view completely

```
wp.media.view.AttachmentsBrowser = wp.media.view.AttachmentsBrowser.extend({
    createToolbar: function() {
        wp.media.view.AttachmentsBrowser.__super__.createToolbar.apply( this,
            [
                var data = wp.media.taxonomies.data;

                for ( var i in data ) {
                    this.toolbar.set( data[ i ].slug + 'FilterLabel', new wp.media.view.Button(
                        value: wp.media.taxonomies.l10n.filterBy[ data[ i ].slug ],
                        attributes: {
                            'for': 'media-attachment-' + data[ i ].slugId + '-filters'
                        },
                        priority: -72
                    )).render();
                    this.toolbar.set( data[ i ].slug + 'Filter', new wp.media.view.AttachmentFilterController(
                        controller: this.controller,
                        slug: data[ i ].slug
                    ) );
                }
            ]
        );
    }
});
```

notice how this replaces the original object, instead of creating a new object

```
this.toolbar.set( data[ i ].slug + 'FilterLabel', new wp.media.view.Label({
    value: wp.media.taxonomies.l10n.filterBy[ data[ i ].slug ],
    attributes: {
        'for': 'media-attachment-' + data[ i ].slugId + '-filters'
    },
    priority: -72
}).render() );
this.toolbar.set( data[ i ].slug + 'Filter', new wp.media.view.AttachmentFilters.Taxonomy({
    controller: this.controller,
    model: this.collection.props,
    priority: -72,
    queryVar: data[ i ].queryVar,
    terms: data[ i ].terms,
    id: 'media-attachment-' + data[ i ].slugId + '-filters',
    allLabel: wp.media.taxonomies.l10n.all[ data[ i ].slug ]
}).render() );
```

the AttachmentsBrowser view now adds the Taxonomy AttachmentFilters View we created in the previous step



ok, time for another breather

wp.heartbeat

*Periodic data transmission between client and server.
Use for locking, progress updates & semi live data.*

wp.heartbeat lets you constant communication between the client and the server

its Used for post locking and auto saves in core

its great when you want to display regular updates to users, for example an import progress bar, or a dashboard of scores or traffic.

wp.heartbeat

```
> wp.heartbeat
<- ▼ {hasFocus: f, connectNow: f, disableSuspend: f, interval
  ► connectNow: f connectNow()
  ► dequeue: f dequeue( handle )
  ► disableSuspend: f disableSuspend()
  ► enqueue: f enqueue( handle, data, noOverwrite )
  ► getQueuedItem: f getQueuedItem( handle )
  ► hasConnectionError: f hasConnectionError()
  ► hasFocus: f hasFocus()
  ► interval: f interval( speed, ticks )
  ► isQueued: f isQueued( handle )
```

like its name suggests, the heartbeat occurs at a steady pass, transmitting data back and forth between the client and the server

by default the heartbeat runs once every 60 seconds, and you can adjust the frequency with a filter

each heartbeat can send data to or from the server from one or more apps that have connected to the heartbeat

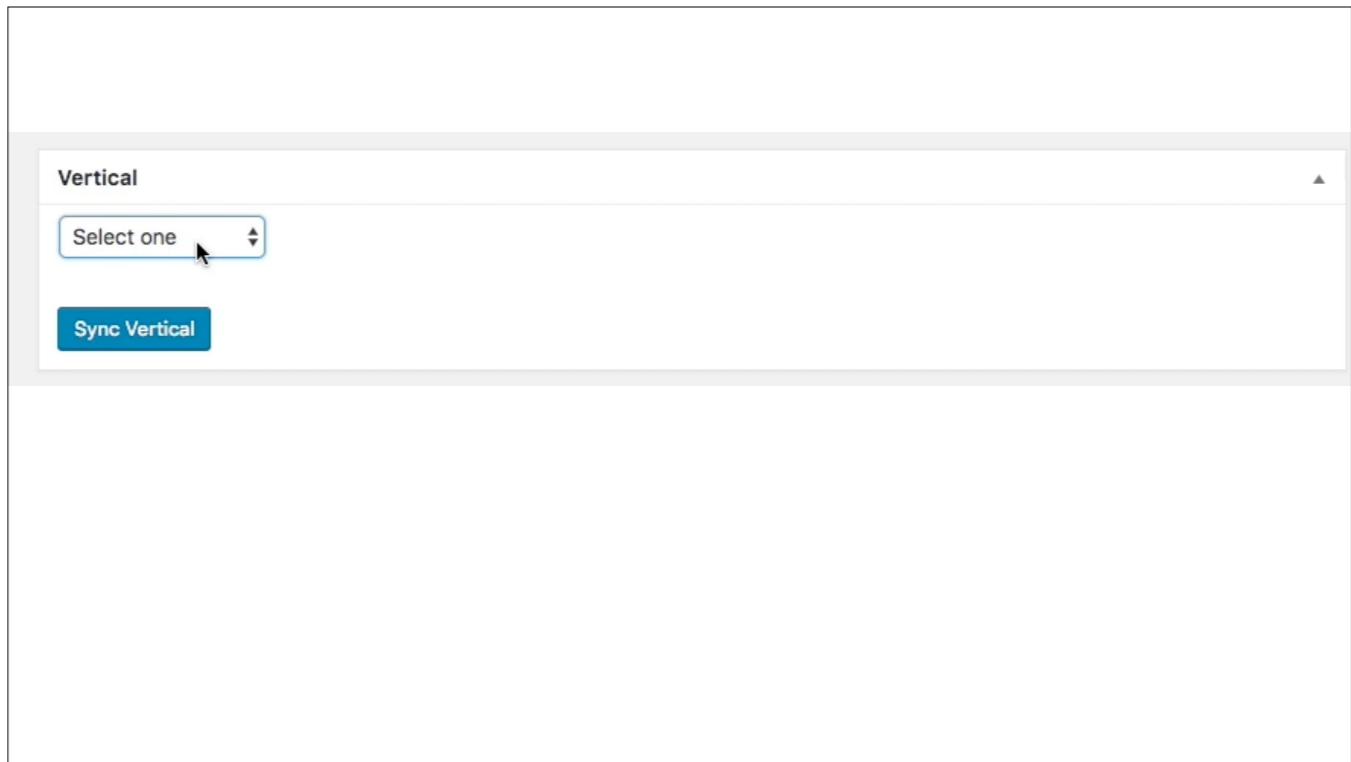
developer.wordpress.org/plugins/javascript/heartbeat-api/

heartbeat is well documented and very useful, but please note that some hosts disable the heartbeat api because it consumes additional resources

The screenshot shows a simple web form titled "Sync Single Post". It contains a single input field labeled "Post ID/URL" with a placeholder text area below it. Below the input field is a blue rectangular button with the white text "Sync Post". The entire form is enclosed in a light gray border.

In this example, we extended the PMC importing tool with a batch mode that asynchronously imports a group of posts in a category or vertical.

PMC tool: <https://github.com/Penske-Media-Corp/pmc-theme-unit-test>



the user selects the category to begin the sync which can several minutes to complete

the code uses heartbeat to transmit progress updates from the server to the client

the process is happening in the background so i can leave this page and when i return the progress updates will continue

```
/*
 * Init the Heartbeat API
 */
PMC_Sync.prototype.initHeartBeat = function() {
    api.heartbeat.interval('fast');
    $(document).on('heartbeat-send', _.bind(this.heartBeatSend, this));
    $(document).on('heartbeat-tick', _.bind(this.heartBeatTick, this));
};
```

here is the code that sets up the main events we need to handle to use heartbeat - there are two main events we need to listen for - heartbeat send and heartbeat tick

```
 /**
 * Callback to send data to the server.
 *
 * 'this' is bound to the instance of PMC_Sync
 * @see initHeartBeat
 * @param e
 * @param data
 */
PMC_Sync.prototype.heartBeatSend = function( e, data ) {
    data.sync_type = this.properties.type;
};
```

with our heartbeat send method, we attach any data we need to send to the server - in this case, the sync type data point

```
/**  
 * .Callback to check the heartbeat.  
 * @param e  
 * @param data  
 */  
PMC_Sync.prototype.heartBeatTick = function( e, data ) {  
    if ( true === this.properties.doingSync ) {  
        if ( true === data.in_progress ) {  
            window.console.log( 'Tick data', data );  
            this.properties.sync_data = data;  
            // Update the progress bar which will trigger the text update.  
            var value = ( this.properties.sync_data.processed.length / this.properties.sync_data.total ) * 100;  
            $( this.DOMElements.progressBar ).progressbar( 'value', value );  
        } else {  
            // We have no data, wait for next tick.  
            if ( false === data.in_progress ) {  
                this.syncingComplete( data );  
            } else {  
                console.log( 'Data not available, waiting for next tick', data );  
            }  
        }  
    }  
};
```

here is the haerBeatTick method which handles data returned from the server

```
  * @param {Object} data
  */
PMC_Sync.prototype.heartBeatTick = function( e, data ) {
    if ( true === this.properties.doingSync ) {
        if ( true === data.in_progress ) {
            window.console.log( 'Tick data', data );
            this.properties.sync_data = data;
            // Update the progress bar which will trigger the text update
            var value = ( this.properties.sync_data.processed.length /
                ( this.DOMElements.progressBar ).progressbar( 'value', va
        } else {
            // We have no data, wait for next tick.
            if ( false === data.in_progress ) {
                this.syncingComplete( data );
            } else {
                console.log( 'Data not available, waiting for next tick' );
            }
        }
    }
};
```

the progress data from the heartbeat is attached to the event, and is used to update the progress bar

```
function setup() {
    $n = function( $function ) {
        return __NAMESPACE__ . "\\$function";
    };
    add_filter( 'heartbeat_received', $n( 'filter_heartbeat_received' ), 10, 2 );
}

/**
 * Filter the spread sync heartbeat response.
 *
 * @param $response
 * @param $data
 *
 * @return mixed
 */
function filter_heartbeat_received( $response, $data ) {
    if ( isset( $data['sync_type'] ) ) {
        $current_vertical_sync = get_option( 'active_' . sanitize_key( $data['sync_type'] ) . '_sync', false ); // @codingStandardsIgnoreLine
        if ( ! $current_vertical_sync ) {
            return $response;
        }
        $response['total'] = count( $current_vertical_sync['to_sync'] ) + count( $current_vertical_sync['processed'] );
        $all_data = array_merge( $response, $current_vertical_sync );
        return $all_data;
    }

    if ( isset( $data['spreadSyncing'] ) ) {
        $current_spread_status = get_option( 'fivethirtyeight_' . $data['spreadSyncing'] );
        $current_spread_sync_status = get_option( 'active_spread_sync' );
        if ( ! $current_spread_status ) {
            return $response;
        }
    }
}
```

lets take a look at how that data is attached

```
function setup() {
    $n = function( $function ) {
        return __NAMESPACE__ . "\\$function";
    };

    add_filter( 'heartbeat_received', $n( '[filter_heartbeat_received]' )
}

/**
 * Filter the spread sync heartbeat response.
 * @param $response
 * @param $data
 */
```

we attach a callback to the heartbeat received filter

```
        */
        * Filter the spread sync heartbeat response.
        * @param $response
        * @param $data
        *
        * @return mixed
        */
    function filter_heartbeat_received( $response, $data ) {
        if ( isset( $data['sync_type'] ) ) {
            $current_vertical_sync = get_option( 'active_' . sanitize_key( $data['sync_type'] ) );
            @codingStandardsIgnoreLine.
            if ( ! $current_vertical_sync ) {
                return $response;
            }
            $response['total'] = count( $current_vertical_sync['to_sync'] );
            $all_data = array_merge( $response, $current_vertical_sync );
            return $all_data;
        }
    }
```

then when we see the sync type is passed, we return the appropriate data from the callback

The screenshot shows the Network tab of a browser's developer tools. A large number of requests are listed, all with the URL 'admin-ajax.php'. The requests are timestamped and show various parameters being sent. One specific request is highlighted with a gray background, showing its detailed Headers, Preview, Response, Cookies, and Timing. The Headers section includes common browser headers like Host, Origin, Referer, User-Agent, and X-Requested-With, along with custom parameters such as data[sync_type], interval, _nonce, action, screen_id, and has_focus. The Preview section shows the raw JSON data being sent. The Response section shows a large amount of JSON data, likely the result of the sync operation. The Cookies section shows session cookies. The Timing section shows the duration of each request.

here is how this looks in the browser... after starting the sync, the heartbeat begins receiving progress updates which are reflected in the UI

The screenshot shows the Network tab of the Chrome DevTools developer tools. The left pane lists several network requests, all of which have the URL 'admin-ajax.php'. The first request is highlighted with a grey background. The right pane displays the request details for the highlighted 'admin-ajax.php' request. The 'User-Agent' field shows 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36'. The 'X-Requested-With' field shows 'XMLHttpRequest'. The 'Form Data' section contains the following parameters: 'data[sync_type]: vertical', 'interval: 5', '_nonce: 7140104ee8', 'action: heartbeat', 'screen_id: tools_page_data', and 'has_focus: true'. At the bottom of the right pane, there is a summary: '91 requests | 151 KB transferred | Finish: 3.3 min | D...'. Below the summary, there are tabs for 'Console' (which is selected), 'Request blocking', and other developer tools. The bottom of the interface shows various status icons and a search bar.

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36

X-Requested-With: XMLHttpRequest

Form Data view source

data[sync_type]: vertical
interval: 5
_nonce: 7140104ee8
action: heartbeat
screen_id: tools_page_data
has_focus: true

91 requests | 151 KB transferred | Finish: 3.3 min | D...

⋮ Console Request blocking

▶ ⚡ top ▾ guten Default lev

because we want pretty real time updates, we set the rate at high or every 5 seconds

▼ Form Data [view source](#) [view URL encoded](#)

data[sync_type]: vertical
interval: 5
_nonce: 7140104ee8
action: heartbeat
screen_id: tools_page_data-import
has_focus: true

every heartbeat we send includes the sync_type data point that will tell our back end code what data we want back

```
imported: []
in_progress: true
objectId: "67"
objectName: "Sports"
▶ processed: [182641]
server_time: 1527649346
started: 1527649337
sync_type: "vertical"
▶ to_sync: [182591, 182523, 182515, 182505, 182284, 182426, 182328, 182297, 182252, 182226, 182088, 182096,...]
total: 30
updated: []
wp-auth-check: true
```

each response includes the list of processed and unprocessed post ids, so we can update the progress bar

wp_customize

Create live preview features for website customization.

The customizer lets you create rich curation and website setting adjustment interfaces that include live previewing. the customizer has always had a robust PHP API. Now we also have a full JavaScript API

<https://developer.wordpress.org/themes/customize-api/the-customizer-javascript-api/>

wp.customize

developer.wordpress.org/themes/customize-api/the-customizer-javascript-api/

make.wordpress.org/core/2017/11/01/improvements-to-the-customize-js-api-in-4-9/

github.com/xwp/wp-customize-featured-content-demo

Since WordPress 4.9 you can now do everything in the customizer. controls, sections, panels are now fully manageable in JavaScript - which enables more dynamic interactions

--

Weston wrote up a detailed blog post about using JavaScript to work with the customizer and I'll share a link in my slides

<https://make.wordpress.org/core/2017/11/01/improvements-to-the-customize-js-api-in-4-9/>

Theme Handbook

CHAPTERS	
Getting Started	▼
Theme Basics	▼
Template Files Section	▼
Theme Functionality	▼
Theme Options – The Customizer API	▲
Customizer Objects	
Tools for Improved User Experience	
The Customizer JavaScript API	
JavaScript/Underscore.js Rendered Custom Controls	
Advanced Usage	

Browse: Home / Theme Handbook / Theme Options – The Customize API

Theme Options – The Customize API

The Customize API (Customizer) is a framework for live-previewing any change to WordPress. It provides a unified interface for users to customize various aspects of their theme and their site, from colors and layouts to widgets, menus, and more. Themes and plugins alike can add options to the Customizer. The Customizer is the canonical way to add options to your theme.

The screenshot shows the WordPress Customizer interface on the left, displaying various theme customization options like Site Identity, Colors, and Menus. On the right, a sample page is shown with a bio and footer text.

Sample Page

This is an example page. It's different from a blog post because it will stay in one place and will show up in your site navigation (in most themes). Most people start with an About page that introduces them to potential site visitors. It might say something like this:

Hi there! I'm a bike messenger by day, aspiring actor by night, and this is my blog. I live in Los Angeles, have a great dog named Jack, and I like piña coladas. (And gettin' caught in the rain.)

...or something like this:

The XYZ Doohickey Company was founded in 1971, and has been providing quality doohickeys to the public ever

Fortunately the customizer APIs are very well documented and there are great examples of how to use it for a variety of use cases

Add Controls in JavaScript

```
var setting = new wp.customize.Setting( 'foo', value, {
    transport: 'postMessage', /* ... */
} );

wp.customize.control.add(
    new wp.customize.Control( 'foo', { /* ... */ } )
);
```

You can Create controls, sections, and panels and manage them fully in JavaScript

this means you can add custom controls as you need them, for example when the user navigates to a page in a specific category

wp.ajax

lkwdwrd.com/using-wp-ajax-async-requests/

wp.ajax may be getting a little old in, but it is still a reliable useful tool in the WordPress context. a simple helper lets you interact with wp_ajax callback you have added in PHP.

Lukes Woodward's tutorial is still the best source of information on wp.ajax



time for another breather

wp.api

*Helper client to quickly interact with the REST API;
automatically creates Backbone models and collections
for API endpoints, including custom post types.*

wp-api is great when you need to quickly interact with the REST API, especially if you want to leverage Backbone models and collections to manage your data

Enqueue

```
wp_enqueue_script( 'wp-api' );
```

All you do is enqueue it or make it a script dependency. It works by parsing the api schema to determine what routes are available.

Set up a post model

```
var post = new wp.api.models.Post( { 'id': 1 } );
```

Here is all the code I need to set up a post model

Request the post

```
post.fetch();
```

and since this is Backbone, a simple fetch command loads the post from the REST API endpoint.

```
> post.attributes
< ▼ Object {id: 1, date: "2017-02-16T04:06:43", date_gmt: "2017-02-16T04:06:43", author: 1, categories: Array(1), comment_status: "open", content: Object, date: "2017-02-16T04:06:43", date_gmt: "2017-02-16T04:06:43", excerpt: Object, featured_media: 0, format: "standard", guid: Object, id: 1, link: "http://demo.localhost/2017/02/16/hello-world/", meta: Array(0), modified: "2017-02-16T04:06:43", modified_gmt: "2017-02-16T04:06:43", ping_status: "open", slug: "hello-world", sticky: false, tags: Array(0), template: "", title: Object}
```

Here is the data returned by the REST API, now stored in the post model's attributes

```
    post.set( 'title', 'Updated title!' );  
    post.save();
```

Updating posts is just as easy using cookie based authentication assuming you are already logged into WordPress. The wp.api client automatically adds the required **nonce authorization headers** when making these requests

```
register_post_type(←
  ↪  'books', ←
  ↪  array(←
  ↪    ↪  'label'          => 'Books', ←
  ↪    ↪  'show_in_rest'   => true, ←
  ↪    ↪  'public'         => true, ←
  ↪    ↪  'publicly_queryable' => true, ←
  ↪    )←
  ↪ );←
  ↴
```

This works with custom post types and registered meta as well, as long as you set show_in_rest to true - for example given this books custom post type

```
| var book = new wp.api.models.Books();  
| book.fetch();
```

You can now fetch from the books model

```
var posts = new wp.api.collections.Posts();  
posts.fetch().done() {  
  » // Do something with the collection.  
}
```

Collections of every object type are available as well

```
post.getCategories();  
post.getTags();  
post.getAuthorUser();  
post.getFeaturedMedia();  
post.getRevisions();  
post.getDate();
```

post models also include helpers to get related data when it is available

wp.apiRequest

```
wp.apiRequest(  
  {  
    path: `/wp/v2/${ basePath }/${ post.id }`,  
    method: 'PUT',  
    data: toSend  
  } ).then( ( newPost ) => {  
  // do something with the newPost  
})
```

wp.apiRequest is a simple Lower level helper for making rest api requests. it returns a promise and is perfect for sending requests right to the api. like the wp.api client it handles nonces for cookie based authentication automatically

wp.utils.WordCounter, wp.sanitize, wp.a11y

There are many smaller utility objects available in WordPress including WordCounter for counting words, characters or sentences.

```
> wp.sanitize
<- ▼ {stripTags: f, stripTagsAndEncodeText: f} ⓘ
  ► stripTags: f ( text )
  ► stripTagsAndEncodeText: f ( text )
  ► __proto__: Object
```

wp.sanitize helps make it safer to output html in JavaScript (but has some limitations)

stripTags strips HTML tags

stripTagsAndEncodeText strips tags and converts HTML entities

```
≥ wp.a11y
↳ ▼ {speak: f} ⓘ
▶ speak: f speak( message, ariaLive )
▶ __proto__: Object
```

```
wp.speak( 'Hello Dolly' );
```

wp.a11y has only one method - it speaks the message you pass it to assistive screen readers.

announcing inline updates that may not be apparent to screen readers is critical in javascript apps when some asynchronous event occurs - for example when an auto search return additional results.

WordPress NPM Packages

let move on to NPM and we will circle back to wp.wordCount.

we have been moving code that are useful to wordpress developers into npm packages

packages are the new way of incorporating external modular code in JavaScript the old way was putting everything on a wp global

npmjs.com/org/wordpress

imagine a future app you are building that is no longer tied to wordpress,

The screenshot shows the npm search interface with the query 'wordpress'. The results page displays three packages:

- @wordpress/a11y**
Collection of JS modules and tools for WordPress development
aduth
published 3 months ago 1.0.7
- @wordpress/autop**
WordPress's automatic paragraph functions `autop` and `removewp`
aduth
published 22 days ago 1.0.6
- @wordpress/babel-plugin-makepot**
WordPress Babel i18n Plugin
netweb
published 12 days ago 1.0.1

you are building something with a purely JavaScript stack that works with WordPress, but might also work on a static site or a display kiosk.

@wordpress/npm-package-json-lint-config

WordPress npm-package-json-lint shareable config

netweb

published 12 days ago  1.0.0

@wordpress/scripts

Collection of JS scripts for WordPress development

gziolo

published 17 hours ago  1.2.0

@wordpress/url

WordPress URL utilities

garypendergast

published a month ago  1.1.0

@wordpress/wordcount

WordPress Word Count Utility

netweb

published 12 days ago  1.0.3

as WordPress packages become available on npm, you can include them directly into your JavaScript apps - making them more portable and not tied directly to running in WordPress

bundling from npm also moves you away from globals which represent a security vector because they can be tampered with.

Gutenberg packages

github.com/WordPress/gutenberg/issues/3955

gutenberg has begin the process of publishing their core apis to packages as well

	<ul style="list-style-type: none"><input checked="" type="checkbox"/> api-request (#7018)<input checked="" type="checkbox"/> blob (#6973) - extracted from <code>utils</code><input type="checkbox"/> blocks - depends on <code>wp.shortcode</code> and <code>window._wpBlocks</code><input type="checkbox"/> core-blocks<input checked="" type="checkbox"/> core-data (#7222)<input type="checkbox"/> components<input type="checkbox"/> editor<input type="checkbox"/> edit-post<input checked="" type="checkbox"/> data (#6828)<input checked="" type="checkbox"/> date (#6658)<input checked="" type="checkbox"/> deprecated (#6914) - extracted from <code>utils</code><input checked="" type="checkbox"/> dom (#6758) - extracted from <code>utils</code><input checked="" type="checkbox"/> element (#6756)<input checked="" type="checkbox"/> library-export-default-webpack-plugin (#6935)<input checked="" type="checkbox"/> plugins (#7235)<input type="checkbox"/> utils - to be extracted into smaller packages<input type="checkbox"/> viewport - depends on <code>components</code>
--	--

like everything Gutenberg, the project is proceeding at breakneck speed and some of the most important APIs like `wp.data` and `wp.element` are already available. using these packages means you can write javascript code that runs in both gutenberg and the classic editor

wp.wordcounter the old way:

PHP:

```
wp_enqueue_script( 'script', 'wc.js', array( 'word-count' ) );
```

JavaScript:

```
var counter = new wp.utils.WordCounter();
var numberOfWords = wp.count( 'Words to count', 'words', {} );
```

the old way of doing things is to use the dependency management built into wp_enqueue_script

first we add the script dependency to make sure the word count global is available when our JavaScript loads, then we use the global directly

wp.wordcounter the new way:

PHP: No PHP!

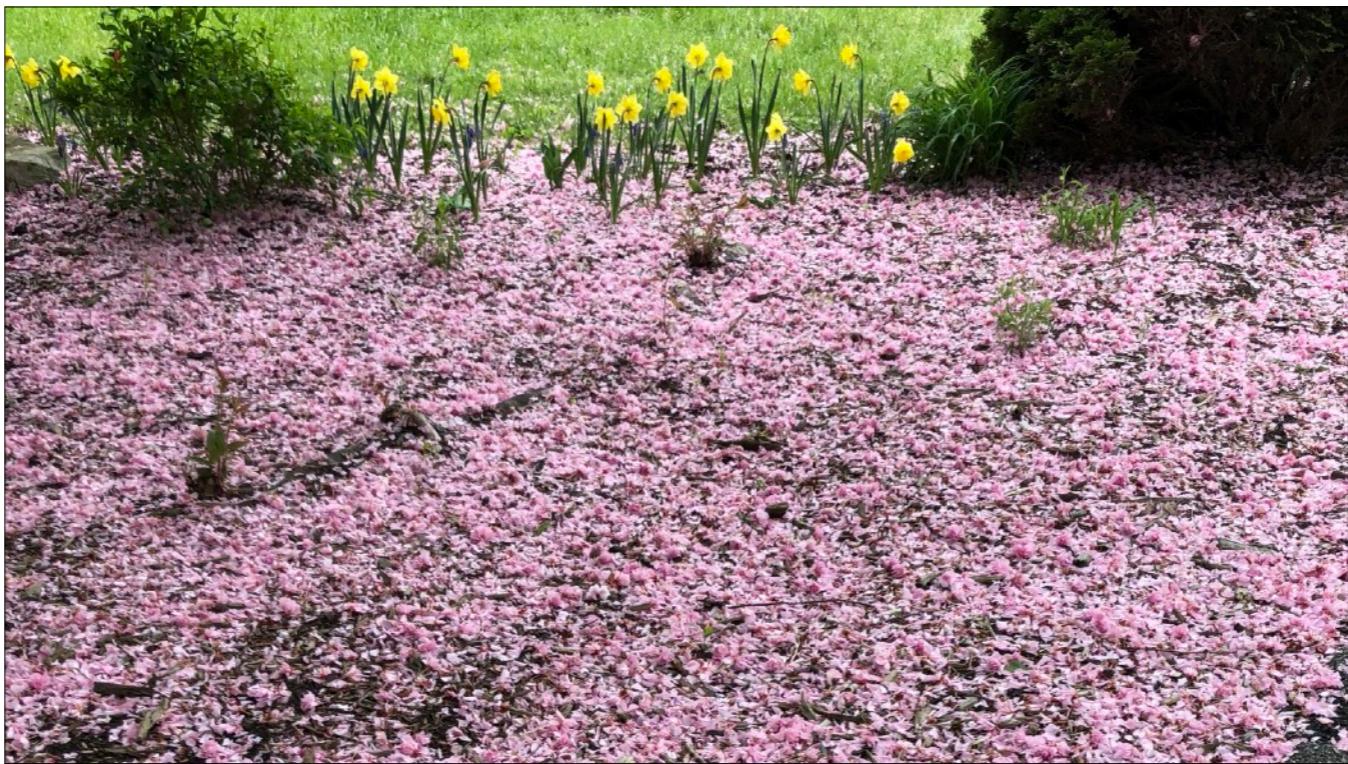
Command Line:

```
npm install @wordpress/wordcount --save
```

JavaScript:

```
import { count } from '@wordpress/wordcount';
const numberOfWorks = count( 'Words to count', 'words', {} );
```

the modern approach requires no PHP. instead, we add the wordpress/wordcount package to our project from the command line. then in our javascript we can import and use the count method directly. no more depending on the wp global



wp.hooks

A lightweight & efficient EventManager for JavaScript.

Ok, one final major api that is now available as a package and has been getting use and testing in Gutenberg

wp.hooks aims to be the equivalent to the php hooks we have currently in wordpress core

```
npm install @wordpress/hooks --save
```

You can add wp.hooks to your project by importing it from npm

```
npm install @wordpress/hooks --save
```

```
import { createHooks } from '@wordpress/hooks';

myObject.hooks = createHooks();
myObject.hooks.addAction(); //etc...
```

You can add wp.hooks to your project by importing it from npm

- `addAction('hookName', 'functionName', callback, priority)`
- `addFilter('hookName', 'functionName', callback, priority)`
- `removeAction('hookName', 'functionName')`
- `removeFilter('hookName', 'functionName')`
- `removeAllActions('hookName')`
- `removeAllFilters('hookName')`
- `doAction('hookName', arg1, arg2, moreArgs, finalArg)`
- `applyFilters('hookName', content, arg1, arg2, moreArgs, finalArg)`
- `doingAction('hookName')`
- `doingFilter('hookName')`
- `didAction('hookName')`
- `didFilter('hookName')`
- `hasAction('hookName')`
- `hasFilter('hookName')`

wp.hooks has all the same functions that you are familiar with in wordpress php hooks. you have add action and do action, add filter and apply filter, along with helpers like remove action and has action

```
/**  
 * Adds the hook to the appropriate hooks container.  
 *  
 * @param {string} hookName Name of hook to add  
 * @param {string} namespace The unique namespace identifying the callback in the form `vendor/plugin/function`.  
 * @param {Function} callback Function to call when the hook is run  
 * @param {?number} priority Priority of this hook (default=10)  
 */  
return function addHook( hookName, namespace, callback, priority = 10 ) {
```

these all work pretty much as you would expect them to if you are familiar with core php hooks - one difference is we currently require a namespace when adding a hook, which you need if you want to remove a hook later.

```
function addBackgroundColorStyle( props ) {
    return Object.assign( props, { style: { backgroundColor: 'red' } } );
}

wp.hooks.addFilter(
    'blocks.getSaveContent.extraProps',
    'my-plugin/add-background-color-style',
    addBackgroundColorStyle
);
```

hooks are now an experimental API in Gutenberg Filters let you extend block.

in this example I'm using the blocks.getSaveContent.extraProps
to attach an additional style to all blocks

dropping this code into your console you will see all the block backgrounds turn red

withFilters

`withFilters` is a part of [Native Gutenberg Extensibility](#). It is also a React [higher-order component](#).

Wrapping a component with `withFilters` provides a filtering capability controlled externally by the `hookName`.

Usage

```
/**  
 * WordPress dependencies  
 */  
import { withFilters } from '@wordpress/components';  
  
function MyCustomElement() {  
    return (  
        <div>  
            content  
        </div>  
    );  
}  
  
export default withFilters( 'MyCustomElement' )( MyCustomElement );
```

Any component can be made filterable by wrapping it in the `withFilters` higher order component in Gutenberg. Using filter is a great way to make your Gutenberg code more extensible.



Speaking of Gutenberg, lets talk about a couple of the major apis built in Gutenberg that are also becoming available as npm packages

The screenshot shows the WordPress.org website with a dark header. The header includes the WordPress logo, the text "WORDPRESS.ORG", a search bar with the placeholder "Search WordPress.org", and a magnifying glass icon. Below the search bar are navigation links: Showcase, Themes, Plugins, Mobile, Support, Get Involved, About, Blog, and Hosting. A blue button labeled "Download WordPress" is also visible. The main content area has a light background. On the left, there's a sidebar titled "Chapters" with a list of topics: Introduction, The Language of Gutenberg, Block API, Templates, Extensibility, Creating Block Types, Reference, and Outreach. The "Block API" topic is currently selected, indicated by a dropdown arrow icon. To the right of the sidebar, the main content area has a title "Introduction" and an "Edit" button. Below the title, there's a paragraph of text: "'Gutenberg' is the codename for the new WordPress editor focus. The goal of this focus is to create a new post and page editing experience that makes it easy for anyone to create rich post layouts. This was the kickoff goal:" followed by a larger text block: "The editor will endeavour to create a new page and post building experience that makes writing rich posts effortless, and has 'blocks' to make it easy what today might take shortcodes, custom HTML, or 'mystery meat' embed discovery."

everything in gutenberg has some documentation, and the gutenberg developer handbook is a great place to start learning

github.com/WordPress/gutenberg/tree/master/docs

block-api	Packages: Move element to packages maintained by Lerna (#6756)	27 days ago
blocks	Adding `ServerSideRender` information (#6838)	18 days ago
extensibility	Support opt-in visual styles for core blocks	11 days ago
outreach	Update meetups.md (#6829)	24 days ago
reference	Add support for Child Blocks (#6753)	15 days ago
block-api.md	Add API to configure icon colors (#7068)	7 days ago
blocks.md	Replace incorrect word in docs/blocks.md (#4235)	5 months ago
extensibility.md	Docs: Update folder structure and make all internal handbook links re...	2 months ago
final-g-wapuu-black.svg	Adds in logo by Cristel (#5202)	4 months ago
grammar.md	Docs: Auto-generate human-readable version of Gutenberg block grammar (...)	16 days ago
language.md	Docs: Update folder structure and make all internal handbook links re...	2 months ago
manifest.json	Fix rich-text markdown source (#7085)	10 days ago
outreach.md	Update outreach.md	10 months ago
readme.md	Docs: Gutenberg is the "new" editor focus, rather than the "2017" foc...	27 days ago
reference.md	Docs: Update folder structure and make all internal handbook links re...	2 months ago
templates.md	Add `my_` prefix to callback function for registering post type (#5150)	3 months ago

The GitHub repository also has a wealth of information and major components include their own documentation. getting involved in testing and contributing to Gutenberg is a great way to learn modern javascript practices

wp.plugins

*register custom fill components into
specific slots in Gutenberg*

an api for registering custom components into specific slots in the Gutenberg UI

Plugins API

The plugins API contains the following methods:

```
wp.plugins.registerPlugin( name: string, settings: Object )
```

This method registers a new plugin.

This method takes two arguments:

1. `name` : A string identifying the plugin. Must be unique across all registered plugins.
2. `settings` : An object containing the following data:
 - `render` : A component containing the UI elements to be rendered.

See [the edit-post module documentation](#) for available components.

Example:

```
const { Fragment } = wp.element;
const { PluginSidebar, PluginSidebarMoreMenuItem } = wp.editPost;
const { registerPlugin } = wp.plugins;

const Component = () => {
```

you'll find documentation about using `wp.plugins` in the [github repository](#), but it might be a bit out of date

```
const Component = () => (
  <Fragment>
    <PluginSidebarMoreMenuItem
      target="sidebar-name"
      icon="smiley"
    >
      My Sidebar
    </PluginSidebarMoreMenuItem>
    <PluginSidebar
      name="sidebar-name"
      title="My Sidebar"
    >
      Content of the sidebar
    </PluginSidebar>
  </Fragment>
);

registerPlugin( 'plugin-name', {
  render: Component,
} );
```

it allows you to render custom “fills” that get rendered into their correct location by Gutenberg. this is akin to how we would use a PHP action hook to output new UI elements in the classic editor.

```
> wp.editPost
< ▼ {...} ⓘ
  PluginPostPublishPanel: (...)
```

PluginPostStatusInfo: (...)

PluginPrePublishPanel: (...)

PluginSidebar: (...)

PluginSidebarMoreMenuItem: (...)

The complete list of fills keeps changing, and you can find them in the console by examining the wp.editPost global while in Gutenberg

wp.data

Manage and interact with state.

wp data is the way you can manage and interact with data in Gutenberg, and in your own gutenberg plugins.

```
npm install @wordpress/data --save
```

this should look familiar by now, wp.data is just an npm install away, and you can use it to write code that works in gutenberg, and also works outside gutenberg, for example, on the front end.

Data

WordPress' data module serves as a hub to manage application state for both plugins and WordPress itself, providing tools to manage data within and between distinct modules. It is designed as a modular pattern for organizing and sharing data: simple enough to satisfy the needs of a small plugin, while scalable to serve the requirements of a complex single-page application.

The data module is built upon and shares many of the same core principles of [Redux](#), but shouldn't be mistaken as merely [Redux for WordPress](#), as it includes a few of its own [distinguishing characteristics](#). As you read through this guide, you may find it useful to reference the Redux documentation — particularly [its glossary](#) — for more detail on core concepts.

Registering a Store

Use the `registerStore` function to add your own store to the centralized data registry. This function accepts two arguments: a name to identify the module, and an object with values describing how your state is represented, modified, and accessed. At a minimum, you must provide a reducer function describing the shape of your state and how it changes in response to actions dispatched to the store.

```
const { data, apiRequest } = wp;
const { registerStore, dispatch } = data;
```

`wp.data` will look familiar if you have used Redux, with a similar data flow. The `wp.data` API builds on this to provide a complete set of state management tools for developers

Riad Benguella

Thoughts and Projects

riad.blog/2018/06/07/efficient-client-data-management-for-wordpress-plugins/

The API is now solidifying, and for the latest information on wp.data, I recommend reading Riad Benguealla's recent blog post

Riad explains the latest wp.data api features including how to retrieve, update and respond to changes in Gutenberg Data. Also, how to register a custom data "namespace" to store your own app's data and finally how to use wp.data to fetch WordPress Data via the REST API.

wp.element

wp.element is, quite simply, an abstraction layer atop React

and with that we come to our final api of the talk, wp.element which is an abstraction layer atop React

```
npm install @wordpress/element --save
```

include it in your javascript app by importing it from npm. an application built to render with wp.element will function in both gutenberg and the classic editor, or anywhere!

```
<div id="greeting"></div>
<script>
function Greeting( props ) {
    return wp.element.createElement( 'span', null,
        'Hello ' + props.toWhom + '!'
    );
}

wp.element.render(
    wp.element.createElement( Greeting, { toWhom: 'World' } ),
    document.getElementById( 'greeting' )
);
</script>
```

here is a simple example of using wp.element. if you have used react before this will look familiar to you - its exactly like react and in fact calls react underneath the hood

```
<div id="greeting"></div>
<script>
function Greeting( props ) {
    return wp.element.createElement( 'span', null,
        'Hello ' + props.toWhom + '!'
    );
}

wp.element.render(
    wp.element.createElement( Greeting, { toWhom: 'World' } ),
    document.getElementById( 'greeting' )
);
</script>
```

importantly the createElement and Render methods map directly to those methods in react.

if you are new to react but know jQuery, this code is very similar to doing DOM construction and rendering using the \$.html() function.



ok wrapping up I hope i've left you with a few takeaways:

WordPress comes with a large amount of built in JavaScript tools you can leverage in your plugins and themes

learning javascript deeply means learning the frameworks and build tools that make up the modern javascript ecosystem

leveraging npm , JavaScript apps can use a single codebase to build features that work inside and outside of WordPress

Get involved in contributing to Gutenberg to learn modern JavaScript deeply!

That's all!

*github.com/adamsilverstein/
wceu2018*



adam@10up.com • @roundearth



Thats all folks. I'm Adam Silverstein and I work for 10up where we make a better web thats maybe even fun. And yes, we are hiring!

I'll be around later in the community room, so please come talk to me if you want to get involved in contributing to Gutenberg or have questions about what I covered.