Developing With Wordpress

I've been working with WordPress since version 2.6, released in 2005 and have been seriously creating content with WordPress since 2015 or so.

WordPress 5.0 and the Gutenberg editor have changed the way we do development. It is no longer enough to know HTML, CSS, Javascript and PHP.

Whether you agree with the new editor or not, if you want to stay current and learn how to code Gutenberg blocks you need to learn React and be familiar with the WordPress-specific layer that sits on top of the React code for the blocks.

This book covers different ways to create WordPress themes and expand functionality:

- **Child Themes**: Using an existing theme as the base and customizing the child theme gives you the flexibility of creating your own templates without loosing the updates the parent theme brings
- **Gutenberg blocks**: blocks provide a way to customize the content without necessarily changing the underlying theme. They also alow bundling the content customizatio with your theme
- **Creating themes from scratch using Underscores**: Creating a theme from scratch gives you more flexibility but it also requires more knowledge of the internals of Wordpress and things like the template hierarchy and how different pieces of WordPress interact with one another
- **Creating themes using a theme framework**: Frameworks like Genesis or Thesis give you a lot of freedom without having to start completely from scratch

Along the way the book will sprinkle additional content dealing with, among other things:

- Planning your theme
- Automation with Gulp
- Web Fonts and Variable fonts
- An Introduction to SASS/SCSS, why would we use it instead of vanilla CSS and how it integrates with CSS
- Scripting for your theme

Planning your project

One of my favorite phrases when it comes to project planning is ***"plan the flight and flight the plan"***.

In the context of a theme, it means that we should plan what we want our theme to do and what items we should create and then stick to the plan and don't deviate from it unless something unexpected happens or the requirements change.

Some of the things I consider when planning a new theme:

1. What type of site am I creating?
    1. What types of pages do we need?
    2. Do the type of pages require new layouts?
    3. How do we leverage existing WordPress functionality versus building our own?
2. Who is our target audience
    1. At the client site?
    2. End users?
        1. These may be the same people at the client site or a completely different audience
3. To Gutenberg or not to Gutenberg
    1. Do we stick with default blocks, third party libraries or build our own?
4. What technology do I want to use?
    1. What browsers are we targeting?
        1. Some times this decision impacts the choice of tools and technologies we use
        2. Mobile or desktop first?
    2. Javascript Development
        1. What version?
            1. if we choose to implement the latest and greatest how much will we have to transpile using `Babel` and `preset-modules`?
        2. Functionality
            1. What are we trying to do with JS?
            2. Can we do it with CSS?
    3. CSS and Styling
        1. Grid and Flexbox

        1. Awesome tech but not supported in older browsers
        2. What is the alternative?
    2. SASS/SCSS instead of plain CSS?
    3. Scripting
        1. What kind do we need?
        2. Typescript or vanilla JS?
        3. Third-party scripts?
            1. Loaded from CDN or bundled as part of the theme?
    4. Do we need a build system?
        1. WebPack based solution?
        2. Is Gulp OK?
5. Accessibility Testing
    1. General Testing
    2. WordPress specific, if any
6. Building the theme or app

WordPress Child Themes

Every so often I want to explore new WordPress themes for my blog without having to start from scratch using Underscores. At the same time I want the flexibility of customizing the theme without loosing the changes every time I update the theme.

That's where child themes come in. They allow developers to customize an existing theme independently of the parent and without loosing the changes when we update the parent theme.

Process

Creating a child theme is a four-step process:

1. Create a folder inside the `wp-content/themes` directory
2. Create a `style.css` stylesheet and add the theme boilerplate
3. Create a `functions.php` file and add the code to enqueue parent and child stylesheets
4. Activate the theme

For this post we'll use Twenty Twenty as the parent theme to take advantage of the new Gutenberg editor (whether it works or not is a subject for another post).

Create child theme folder

Depending on how you do development creating the theme folder can be done either through a GUI (Windows Explorer or Finder) or through a terminal (Linux, Mac or WSL for Windows)

I work on Mac's terminal so, starting on the root of the WordPress installation I run the following command

```
mkdir -p wp-content/themes/twentytwenty-rivendellweb/
```

Change to the directory you just created.

```
cd wp-content/themes/twentytwenty-rivendellweb/
```

Create a stylesheet: `style.css`

The core of the child theme is the `style.css` stylesheet and the comment block that defines the theme.

The example below is what I used for my Twenty Twenty child theme. Individual fields are explained below.

```
/*
 Theme Name:   Twenty Twenty Rivendellweb
 Theme URI:    https://github.com/caraya/2020-child
 Description:  Twenty Twenty Child Theme for The Publishing Project
 Author:       Carlos Araya
 Author URI:   https://publishing-project.rivendellweb.net/
 Template:     twentytwenty
 Version:      1.0.0
 License:      MIT
 License URI:  https://opensource.org/licenses/MIT
 Tags:         light, dark, two-columns, right-sidebar, responsive-layout
 Text Domain:  twentytwentyrivendellweb
*/
```

Theme name (***REQUIRED***). This is the name that will show up for your theme in the WordPress admin screen

Theme URI. This points to the website or demonstration page of the theme at hand. **This or the author's URI must be present in order for the theme to be accepted into the WordPress directory**.

Description This description of your theme will show up in the theme menu when you click on "Theme Details".

Author. This is the author's name.

Author URI. Author's website.

Template (***REQUIRED***). This is the name of the parent theme, meaning its folder name. Be aware that it is case-sensitive, and if you don't put in the right information, you will receive an error message

Version. This displays the version of your child theme. Using semver is usually a good way to version your theme

License. This is the license of your child theme. WordPress themes in the directory are usually released under a GPL license.

I don't think that the GPL is a good license to use due to its viral nature, the fact that after 10 years and cotacting FSF (see GPL and WordPress: Developer beware) I still don't understand the interaction between GPL and commercial software, and the fact that there are GPL compatible licenses like MIT, contrary to what Matt says.

However, since WordPress Core is licensed under GPL it may make more sense to have only one license throughout the codebase. It ends up being a matter of preference.

The Free Software Foundation maintains a list of free software licenses that are compatible with the GPL in case that the GPL is too restrictive for your taste.

License URI. This is the address where your theme's license is explained. The URI must match the license you choose too use.

Tags. The tags help others find your theme in the WordPress directory. Thus, if you include some, make sure they are relevant.

Text domain. This part is used for internationalization and to make themes translatable. This should fit the "slug" of your theme.

Enqueue parent styles

The recommended way of enqueuing the parent theme stylesheet currently is to add a `wp_enqueue_scripts` action and use wp_enqueue_style() in your child theme's `functions.php`.

The following example adds both the parent and the child theme stylesheets.

```php
<?php
function my_theme_enqueue_styles() {
  $parent_style = 'pare'parent-style'// This is 'twentytwenty-style' for t

  wp_enqueue_style( $parent_style, get_template_directory_uri() . '/style
  wp_enqueue_style( 'child-style',
    get_stylesheet_directory_uri() . '/style.css',
    array( $parent_style ),
    wp_get_theme()->get('Version')
  );
}
add_action( 'wp_enqueue_scripts', 'my_theme_enqueue_styles' );
```

In this example, each theme has a single stylesheet. If a theme has more than one stylesheet you're responsible to enqueue the styles in the right order so the theme will continue working.

A solution is to concatenate the stylesheets during the build step or using SASS imports.

Activate the new theme

Now we can activate the theme from the Administrator interface. Go to `Appearance > Themes`, select the theme that we just created and activate it.

If everything worked out OK, there should be no difference between the parent and child themes.

We can now start working with our child theme and change it to our heart's content and everything should work the same.

Once WordPress detects you're working with a child theme it will no longer warn you about changing the theme's CSS but it will still suggest you use the theme customizer.

Conclusion

We've just looked at the basics on creating and customizing a child theme for WordPress.

Since the release of WordPress 5.0 and the Gutenberg editor the development process has changed significantly. This post barely scratches the surface of what you can do.

General Concepts for child themes and themes from scratch

Once we have a working theme we can look at ways to enhance it. This chapter will look at three ways to enhance a WordPress theme:

- Using third-party scripts and tools
- Customizing with Gutenberg blocks
- Using CSS to modify the theme appearance
- Modifying PHP templates

Third-part scripts

> ### Warning
>
> This example, and any other function that adds scripts and stylesheets to pages of a WordPress installation that convert to AMP, is likely to run afoul of AMP validation.
>
> If you want to use an AMP compliant syntax highlighter, I'd suggest you look at Syntax-highlighting Code Block (with Server-side Rendering) by Weston Router. I haven't tested if this will work with the Classic Editor plugin or if it's Gutenberg only.

We can create multiple `wp_enqueue_scripts` actions and use wp_enqueue_script() and wp_enqueue_style() to add tools in your child theme's `functions.php` in a similar way to how we added the stylesheets for the theme to work. We can also add multiple scripts and stylesheets to a single `wp_enqueue_script` function if we want to consolidate all our additions into a single action.

For example, let's say that we want to enqueue Prism core and styles for our theme rather than use a plugin.

The file names for the scripts and the stylesheets in the example do not correspond to the real names. In an ideal world, I'd create brand new downloads and organize them inside the theme as needed.

```
<?php
function rivendellweb_enqueue_prism() {
  wp_enqueue_style( 'prism_style', get_template_directory_uri() . '/prism/
  wp_enqueue_script( 'prism_script',
    get_template_directory_uri() . '/prism/prism-core.js' );
}
add_action( 'wp_enqueue_scripts', 'rivendellweb_enqueue_prism' );
```

The usual performance caveats of adding scripts and stylesheets apply. If you need to, use Scripts To Footer and Scripts-To-Footer Exclude AMP to move the scripts to the footer excluding AMP-related scripts.

This should improve performance.

Overriding the parent theme: CSS

The first means to change the parent theme is to use CSS in the child theme to override the parent's CSS.

One thing that has changed since I last played with child themes is that WordPress is now based on blocks so you can't just override the specific class.

The following code sets the default content width to 960 pixels.

I had a hard time parsing the first item of the rule. It reads as:

Select any element that has the class `entry-content` that has any children that *do not have* any of `alignwide`, `alignfull`, `alignleft`, `alignright`, or `is-style-wide` classes

We also pick the elements with class, `post-meta-wrapper` or `post-meta`

```
.entry-content > *:not(.alignwide):not(.alignfull):not(.alignleft):not(.a
.post-meta-wrapper,
```

```
.post-meta {
    max-width: 960px;
}
```

Again, more research is needed, particularly when working with Gutenberg blocks and what additional classes they introduce to the different components of the page.

Overriding the parent theme: PHP

The hardest way to modify the parent theme is to change the PHP templates to create a new structure or create new templates to represent new types of content.

The basic template file gives the new template a name, `Project Template` and tells WordPress what type of content to use.

```
<?php
/*
Template Name: Project Template
Template Post Type: post, page
*/
get_template_part( 'singular' );
'singular'
```

Once we have the basic theme, we can leverage template partials to further customize the template. Inside the partials we can write custom HTML or template tags to customize the behavior of the template.

The example below changes the header of all pages that don't use the project template. If the page uses the project template then the custom header will not be used, we should provide a fallback with the default template so the site will look as it did before we made the changes...

```
<?php
  if (! is_page_template(array('temp'templates/template-project.php'{
  ?>
```

```
  <header id="site-header" class="header-footer-group" role="banner">
    <p class="blog-title"><strong><a href="<?php bloginfo('url''url'?>"><

    ___PHP4___
```

Likewise, we customize the footer template to remove all content from the footer so we can fully customize it with Gutenberg blocks or we can edit it and customize it with PHP, HTML and CSS.

```
<?php
if ( ! is_page_template( array('temp'templates/template-project.php' {
?>
    <footer id="site-footer" role="contentinfo" class="header-footer-grou
    <!-- Build footer content here -->
    </footer></footer><!-- #site-footer -->
<?php } ?>
    <?php wp_footer(); ?>

    </body>
</html>
```

See Template Files in the WordPress Theme Handbook.

You can also see a fully worked version of the code in these examples on Github at https://github.com/caraya/2020-child/

Adding custom fonts

With custom web fonts we can add our own branding and identity to the site and play with typography and Poen Type typographical features.

We can add web fonts in two different ways:

- Using fonts from a third-party font services like Google Fonts or Typekit / Adobe Fonts
- Include the fonts in your theme and host and host them locally

Third-party font services: Google Fonts and Adobe Fonts

Font services like Google Fonts and Adobe Fonts (formerly known as Typekit) offer large variety of fonts to use and easy way to install and use them on your web projects.

Using these fonts with WordPress is slightly different than how we'd use them on a regular website.

Instead of linking them in the head of the page, we enqueue them just like we do with other scripts and stylesheets.

The following example enqueues three different kinds of fonts to the WordPress site.

1. The first one is a regular font from Google Fonts
2. Second and third examples use variable fonts from Google Fonts using their experimental fonts API
3. The third example loads a font from Typekit

We then use a single `add_action` function to add all the enqueued styles to the queue.

```
function rivendellweb_webfonts() {
  // Google fonts version 1:
```

```
  // regular font
  wp_enqueue_style('google_uncial_font',
   'https:'google_uncial_font'//fonts.googleapis.com/css?family=Uncial+Ant
  // Google fonts version 2:
  // variable font'google-heptaslab-variable-font',
  'https://fonts.googleapis.c'google-heptaslab-variable-font'//fonts.goog
  // Google fonts version 2
  // variable font', 'https://fonts.googleapis.com/css2?family=Crimson+Pro
  // Typekit / Adobe Fonts.typekit.net/cmb8gbt.css';)
}

add_action('wp_enque';)
}

add_action('//use.typekit.net/cmb8gbt.css';)
}

add_action('wp_enqueue_scripts', 'rivendellweb_webfonts');
```

Working with hosted fonts make it easier to use the fonts but adds llatency as the browser has to make one roundtrip per stylesheet cued, parse the stylesheet and load the font or fonts specified

Hosting your own

If you'll host the best way to do it is to create a stylesheet with the font-face declarations and then enqueue it like you would any other font.

> Follow these steps to enqueue and use local fonts on yoour site:

1. Download the fonts to your local drive
2. Use the fonts you downloaded in step 1 to generate a @font-face kit at Font Squirrel, if one doesn't come already bundled with the fonts
3. Inside your theme directory create a fonts directory
4. Copy the font files and license (if provided) to the fonts directory
5. Copy the CSS code for the fonts in your fonts directory
6. Enqueue the fonts stylesheet you created in the previous step

This is a simplified version of the full stylesheet generated by Font Squirrel's

Webfont Generator for the Open Sans font

```
@font-face {
  font-family: 'open_sans'open_sans'url('opensans-bold-webfont.woff2') fo
        'woff2'url('opensans-bold-webfont.woff')('woff');
  font-weight: 700;
  font-style: normal;
}

@font-f'woff'@font-faceamily: 'open_sans';
    src: url('opensans'open_sans'url('opensans-bolditalic-webfont.woff2')
    font-weight: 700;
    font-style: italic;

}
@font-face {
  font-fami'opensans-bolditalic-webfont.woff'@font-face-webfont.woff2') fo
    ') format('url('opensans-italic-webfont.woff2') format('woff2'),
    url('opensans-italic-webfont.woff')t-face {
  font-family: 'open_sans';
  src: url('opensans-regular'open_sans'@font-faceat('woff2'),
    url('opensans-regular'woff2'url('opensans-regular-webfont.woff2') for
    url('opensans-regular-webfont.woff') format('woff');
  font-weight: normal;
  font-style: normal;
}
```

How we enqueue the stylesheet will also change because we're using adding a local file and not a full URL.

To get the path to the stylesheet we use the get_stylesheet_directory_uri WordPress function and concatenate it with the final directory path and the name of the stylesheet.

```
function rivendellweb_webfonts() {
  wp_enqueue_style('typekit_fonts', get_stylesheet_directory_uri() . '/fo
}
```

```
add_action('wp_enqueue_scripts', 'rivendellweb_webfonts');
```

Variable Fonts and Economies of scale

Variable fonts present another interesting opportunity to do more with less. They combine mutliple fonts and font faces into a single file by interpolatings the values from the different fonts to create a range.

One of the most intriguing Variable fonts I've seen is Recursive. It offers the following axes for you to experiment with:

- **Monospace**: Sans (natural-width) or Mono (fixed-width)
- **Casual**: Linear to Casual
- **Weight**: Light to ExtraBlack
- **Slant**: 0 to -15 degrees
- **Italic**: always roman, auto, or always italic

Some of these axes use standard CSS to express the property values. One example on Recursive is the weight axis that is represented with the `font-weight` rule.

Other axes like Casual and Monospace cannot be expressed with existing CSS. That's where CSS custom properties and the `font-variation-settings` come into play.

If you change any values using `font-variation-settings` you have to update all of them, otherwise they will be reset to their default values.

```
/*
  Define default values in :root selector
  using CSS custom properties
*/
:root {
  --mono: 0;
  --casl: 0;
```

```
  --wght: 400;
  --slnt: 0;
  --ital: 0;
}

/*
  Use default values to define variation settings with CSS variables.
  Use standard CSS attributes where possible.
*/
body {
  font-weight: var(--wght);
  font-variation-settings:
  "MONO" var(--mono),
  "CASL" var(--casl),
  "slnt" var(--slnt),
  "ital" var(--ital);

  "MONO"/* other styles for body go here */
}
```

This single variable font replaces 2 fonts and 5 styles; the 4 styles of the main font (regular, bold, italic and bold italic) and one style for monospaced code listings.

```
:root {
  --mono: 0;
  --casl: 0;
  --slnt: 0;
  --ital: 0;

  @font-face {
    font-family: 'Recursive';
    src: 'path/to/font/file/recursive.woff2'
          forma'Recursive'riations');
    font-weight: 300 1000;
  }
}
```

```
body {
  font-family: "Recursive";
  font-weight: var(--wght);
  font-variation-settings:
  "MONO" var(--mono),
  "CASL" var(--casl),
  "slnt" var(--slnt),
  "ital" var(--ital);

  "Recursive"/* other styles for body go here */
}
```

Recursive allows things that are not possible without adding fonts that are purpose specific. For this project it has replaced 6 different fonts that I would have normally loaded:

- Regular, italic, bold and bold italic sans serif font for copy text
- Monospaced regular font for code snippets
- Casual font for headings using a custom axis

Variable fonts also allow for things I didn't know were possible with fonts and we can incorporate these awesome things into our themes and design processes.

See Variable fonts & the new Google Fonts API for a deeper look at variable fonts.

Rather than enqueue a separate script I've used the `@font-face` rule to load the font in the `style.css` main stylesheet.

Responsive typography

Variable Fonts give you a lot of flexibility while losing support for older browsers and operating systems. They reduce the number of font files required to render content and they give you options that are difficult or not possible with traditional web fonts.

Quoting Jason Pamental's The evolution of typography with variable fonts: an introduction:

> As described by John Hudson, a variable font is a single font that acts as many: all the variations of width and weight, slant, and even italics can be contained in a single, highly efficient and compressible font file. What's more: the format (which is technically part of the OpenType 1.8 specification) is completely extensible. The type designer has complete control over what axes are used, their ranges, and even the definition of new axes. There are currently 5 'registered' axes (width, weight, slant, italics, and optical sizing), but the designer can vary any axis they choose. Some examples include the height of ascenders and descenders, text grade, even serif shape. The possibilities are nearly limitless.

Why use them

Variable fonts improve performance in several ways. They reduce the number of HTTP connections we have to make for Font assets, it makes the fonts smaller overall (the one file you download may be larger but it's one as opposed to 4 for each traditional font you work with).

They also allow for things that were very difficult or impossible to do before. We can animate the axes if we set them up properly giving us additional flexibility.

We'll explore Variable fonts using Recursive as the single font for a WordPress-based site. Along the way, we'll talk about responsive typography, based on the Work of Jason Pamental, and how to work with older browsers.

Loading variable fonts

WordPress strongly suggests that you enqueue third-party scripts and stylesheets for use with a WordPress theme. However, when creating a theme from scratch we don't need to enqueue the main stylesheet and that's where we'll make all our variable fonts additions.

We'll cover both methods below.

Modifying an existing theme

Assuming that we've created a stylesheet to load the font using `@font-face` syntax and all the style that override the default font size then all it takes is to enqueue the stylesheet.

We've discussed how to enqueue local stylesheets so I won't go into details about how the code below works, I'll just show the end product.

```
function rivendellweb_enqueue_local_fonts() {
    wp_enqueue_style( 'local_styles',
            get_stylesheet_directory_uri() . '/css/recursive-styles.css' )
}
add_action( 'wp_enqueue_scripts', 'rivendellweb_enqueue_local_fonts' );
```

From Scratch

When building a theme from scratch the rules change slightly. We're not adding new resources to the theme but we're changing the existing CSS to match our design.

There is no enqueueing necessary as we're working with the default styles for the theme. We'll look at how to do it in the next section.

Example: Recursive Font from scratch

The following code will build a responsive-typography stylesheet using Recursive.

We first load the font using the `@font-face` rule with some changes to accommodate the variable fonts.

We use two different formats to support different syntaxes for the format for the attribute.

Attributes like `font-weight`, `font-style` and `font-stretch` take two values indicating the lower and upper boundaries for the particular axis.

Finally, we use `font-display: swap` to tell the browser to swap the font once it's loaded.

```
@font-face {
  font-family: "Recursive"Recursive VF"   url('./fonts/recursive.woff2')
      format('woff2 supports variations'),
    'woff2 supports variations'url('./fonts/recursive.woff2')
```

```
      format('woff2-variations');
    font-weight: 300 1000;
    font-display: swap;
}
```

The next block defines variables with the default values for each of the axes that the font makes available. We'll make extensive use of these variables elsewhere in the document.

```
:root {
  --recursive-mono: 0;
  --recursive-casual: 0;
  --recursive-weight: 400;
  --recursive-slant: 0;
  --recursive-italic: 0.5;
}
```

This default selector adds the default font family and default values using the variables defined in the previous block.

font-variation-settings allows you to add the custom axes with variables.

The uppercase axes, **MONO** and **CASL**, are custom axes that will only work with Recursive.

The lowercase axes, **slnt** and **ital** are predefined axes. The reason why we don't use the equivalent CSS property is that they both match the same property so we'd have to use either one but we can't use them together.

```
* {
  font-family:  "Recursive VF",
                Verdana,
                sans-serif;
  font-weight: var(--recursive-weight);
  font-variation-settings:
    "MONO" var(--recursive-mono),
    "CASL" var(--recursive-casual),
```

```
    "slnt" var(--recursive-slant),
    "ital" var(--recursive-italic);
}
```

The rest of the code in this post is taken and adapted from FF Meta Variable Font Demo, a pen from Jason Pamental.

We first add another `:root` block with CSS Custom Properties / Variables too define the values that we want to work with.

This is a simplified version that considers only p and h1 elements. The full version has additional entries for h2 through h4.

This code only deals with font size, line height, and their relationship when the screen size changes using media queries. It also takes advantage of Fontface Observer to add styles for when the font fails to load.

```
:root {
  /* Breakpoint variables */
  --bp-small: 24.15;
  --bp-medium: 43.75;
  --bp-large: 60.25;
  --bp-xlarge: 75;
  /* Paragraph variables */
  --p-line-height-min: 1.25;
  --p-line-height-max: 1.4;
  --p-font-size-min: 1.0;
  --p-font-size-max: 1.25;
  /* H1 variables */
  --h1-line-height-min: 1.1;
  --h1-line-height-max: 1.1;
  --h1-font-size-min: 2.5;
  --h1-font-size-max: 4;
  --h1-vf-wght-multiplier-s: 0.75;
  --h1-vf-wght-multiplier-m: 0.75;
  --h1-vf-wght-multiplier-l: 0.75;
}
```

The default rule for paragraphs sets the size to 16px, the font size to 400 and the line-height to 1.

All the media queries play with what values to use and how to combine them.

```
p, li {
  font-size: calc( var(--p-font-size-min) * 1rem );
  font-weight: var(--recursive-weight);
  line-height: var(--p-line-height-min);
}
@media screen and (min-width: 24.15em) {
  p, li {
    line-height: calc(( var(--p-line-height-min) * 1em ) + ( var(--p-line-
  }
}
@media (min-width: 60.25em) {
  p, li {
    font-size: calc(( var(--p-font-size-min) * 1em ) + ( var(--p-font-size
    line-height: var(--p-line-height-max);
  }
}
@media (min-width: 75em) {
  p, li {
    font-size: calc( var(--p-font-size-max) * 1em );
  }
}
```

We do something similar with h1 with the corresponding h1 variables and one additional change.

We leverage the `.fonts-failed` class generated by FontFace Observer and style elements when our variable font is not available.

```
h1 {
  font-weight: calc( var(--recursive-weight) * var(--h1-vf-wght-multiplier
  font-size: calc( var(--h1-font-size-min) * 1em );
  font-style: normal;
  line-height: var(--h1-line-height-min);
```

```
}
.fonts-failed h1 {
  font-family:  Georgia,
                "New Times Roman",
                serif;
  margin: 2em 0;
  letter-spacing: -.5px;
}
"New Times Roman"@media screen and (min-width: 24.15em) {
  h1 {
    line-height: calc(( var(--h1-line-height-min) * 1em ) +
      ( var(--h1-line-height-max) - var(--h1-line-height-min) ) * ((100vw
    font-size: calc(( var(--h1-font-size-min) * 1em ) + ( var(--h1-font-s
  }
}
@media screen and (min-width: 43.75em) {
  h1 {
    font-weight: calc( var(--recursive-weight) * var(--h1-vf-wght-multipl
  }
  .fonts-failed h1 {
    letter-spacing: normal;
  }
}
@media (min-width: 75em) {
  h1 {
    font-size: calc( var(--h1-font-size-max) * 1em );
    font-weight: calc( var(--recursive-weight) * var(--h1-vf-wght-multipl
    line-height: var(--h1-line-height-max);
  }
  .fonts-failed h1 {
    letter-spacing: -1px;
  }
}
```

Yes, this is a lot of code but it will keep text readable and easy to change. Whenever we need to change something, we change the corresponding variables at the top.

One of the Recursive font's custom axes is Casual. I use it to create distinctive

headers in combination with both Slant and Italic axes.

The code looks something like this:

```
h1.casual {
  --recursive-casual: 1;
  --recursive-slant: -15;
  --recursive-italic: 1;
}
```

We've done the same thing with styles. This is the modified styles for Prism.js used on my project.

We change the font to monospaced and add slashed 0 to fully distinguish them from lowercase and uppercase o.

```
code[class*="language-"],
pre[class*="language-"] {
    --recursive-mono: 1;
    --recursive-zero: "zero" on;
    color: #657b83;
    font-family:  "Recursive VF",
                Consolas, Monaco,
                'Andale Mono',
                'Ubuntu Mono',
                monospace;
    font-size: 1.1em;
    text-align: left;
    white-space: pre;
    word-spacing: normal;
    word-break: normal;
    word-wrap: normal;
    line-height: 1.5;
    tab-size: 4;
    hyphens: none;
}
```

Considering Subsetting Fonts

We can decrease the impact fonts have in load with font subsetting where you reduce the number of glyphs available on your font to what's actually used on the page or pages you're working on.

Good news is that all modern browsers support unicode-range font subsetting

With a WordPress theme we have to limit subsetting to elements that we as theme developers control or to the minimal set of characters for the languages that we're targeting or to the Latin character set.

My preferred tool for font subsetting is Glyphhanger from the Filament Group

What I particularly like about the tool is its simplicity. The following command allows you to create a font-subset for all the characters used on a site.

Once the site is built I can do something like this to subset the fonts I'm using based on the glyps used on the site.

```
glyphhanger http://localhost:5000 \
  --subset=font/location/*.woff2
```

This will use Puppeteer to generate the subsets and the subset font files to use.

This is the basic use for Glyphhanger. Check the Glyphhanger Github repository for more information about what it can do.

Gutenberg Blocks

Another way to customize a theme is to create theme-specific blocks that will render theme-related content.

Before we build Gutenberg blocks we need to decide whether we want to use Gutenberg with the theme or not.

Gutenberg is a new editor that replaces the long-standing TinyMCE editor used in WordPress with a block-based architecture, just like a page builder.

> The Gutenberg WordPress editor is a new page builder that is being designed to integrate with WordPress core. Gutenberg will add content blocks and page builder-like functionality to every up-to-date WordPress website. When in use, it will replace TinyMCE as the default content editor. With Gutenberg, content is added in blocks of various types from the WordPress backend.
>
> The Gutenberg WordPress Editor: 10 Things You Need to Know

There are several third-party block libraries that we can leverage on our themes if we choose to use the editor. Some of the most popular gutenberg addon plugins are:

- Gutenberg Blocks
- CoBlocks
- Stackable
- Atomic Blocks.
- Advanced Gutenberg

We can also create custom Gutenberg blocks to fit the type of content we want to create and that we're too lazy to figure out how to do without blocks.

Building Gutenberg blocks

- https://developer.wordpress.org/block-editor/developers/themes/theme-support/

- https://developer.wordpress.org/block-editor/developers/themes/block-based-themes/

Themes from scratch: Underscores

Themes from scratch: Genesis

Conditional tags

- Conditional Tags
- List of Conditional Tags

Wordpress without themes: The REST API

The introduction of the REST API gives WordPress developers a much wider canvas to play with.

Using WordPress as a headless CMS we can build the front-end with whatever framework or library we choose or need. This is particularly useful when you're integrating WordPress with an existing site or appication that already uses a given library or framework.

In this example we'll use plain Javascript and object literals (available in ES2015 or later) to build components that will fetch the last 5 posts of a blog as an example of what you can do with the API.

The following code snipet uses fetch to retrieve the last 5 posts of a WordPress blog along with embedded/related data and log the result to console. This will be the basis of our experiment.

Things to be aware of:

After we retrive the code we have to check if the response was retrieved successfully, indicated by the OK value in the response. We do this because Fetch will not fail (and trigger the catch block) when the response code is in the 400 (not found) or 500 (server error) series.

Keep your target browsers in mind. Different browsers support different features and that's important because native templates are one of those

The examples in this section make unauthenticated requests. This means we can't retrieve editable content or upload data using this system. We'll discuss this in detail in Authenticating API requests [See page 33]

```
fetch('https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?_er
  .then((response) => {
```

```
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then((myJson) ='Network response was not ok'// Do what we want with the
    console.log(myJson);
  })
  .catch((error) => {
    console.error('There has been a problem with your fetch operation:', e
  });
```

Once we have retrieved the posts, we can use the components to populate our templates or custom elements.

Authentication

As mentioned in the ***things to be aware of*** box at the begining of the chapter, we've kept most of the examples in this section without authentication.

Theme testing

https://codex.wordpress.org/Theme_Unit_Test

https://wordpress.org/plugins/theme-check/

Annotated Bibliography

WordPress Themes

Underscores: A starting theme for WordPress
: This is my favorite starter theme for when Child Themes are not enough and I want to work creating material that I can't do by using child themes

The Beginner’s Guide to Creating a Theme With Underscores

Studio Press, makers of the Genesis Framework
A Beginner's Guide to the Genesis Framework for WordPress

WordPress Gutenberg Editor

The Gutenberg WordPress Editor: 10 Things You Need to Know

Variable Fonts

W3C CSS Fonts Module 4 Specification (editor’s draft)
W3C Github issue queue
Microsoft Open Type Variations introduction
Microsoft OpenType Design-Variation Axis Tag Registry

Variable Fonts Playground

Wakamai Fondue
: A site that will tell you what your font can do via a simple drag-and-drop inspection interface

Axis Praxis
: The original variable fonts playground site

V-Fonts.com
: (a catalog of variable fonts and where to get them)

Font Playground
: (another playground for variable fonts with some very unique approaches to user interface)

WordPress REST API

WordPress REST API Handbook
: The REST API Handbook provides an introduction to the REST API

A JavaScript Client for the WordPress REST API
: wp-api provides an abstraction for the REST API

WordPress REST API Sample JavaScript App Code
WordPress REST API JavaScript Demo Code
: The site and the application code

A Quick Start Guide To The WordPress REST API

Headless WordPress

How Smashing Magazine Manages Content: Migration From WordPress To JAMstack

How To Create A Headless WordPress Site On The JAMstack