1. Fundamental Web Development Concepts

1.1 Demonstrate knowledge of HTML and CSS.

Asset libraries

The asset library system exists to make it possible for Drupal to:

- Include only the CSS and JavaScript required for a specific page
- Vary what is included depending on the content of the page

An asset library is a YAML data structure inside a THEMENAME.libraries.yml file that specifies one or more CSS and JavaScript files, and their settings, bundled together under a uniquely identified library name. Once the library has been defined adding it to a page, or attaching it to a particular type of element, is done in the same fashion regardless of the contents of the library. This means there is now one unified mechanism for adding CSS and JavaScript whether it's being added in a module or a theme.

The general steps for loading assets (CSS/JS) are:

1. Save the CSS or JS to a file.
2. Define a "library", which can contain both CSS and JS files.
3. "Attach" the library to a render array in a hook.

But in the case of themes, there is an alternative to step 3: themes can choose to load any number of asset libraries on all pages.

How to define a library

Add a *.libraries.yml file to the root of your module or theme folder:

```
retro:
  version: 1.0
  css:
    theme:
      css/theme/fonts.css: { weight: 10 }
  js:
    js/rainbow-headings.js: {}
  dependencies:
    - retro/rainbow

rainbow:
```

```
  version: 1.0
  remote: https://github.com/xoxco/rainbow-text
  license:
    name: MIT
    url: https://github.com/xoxco/Rainbow-Text/blob/master/README.md
    gpl-compatible: true
  js:
    js/rainbow.js: {}
  dependencies:
    - core/jquery
```

Ways of adding a library to page(s)

Globally from the .info file of a theme using the libraries key

```
libraries:
  - retro/global
```

Using a preprocess function

```
/**
* Implements hook_preprocess_HOOK() for page.html.twig.
*/
function retro_preprocess_page(array &$variables): void {
  $path = \Drupal::service('path.current')->getPath();
  if ($path == '/node') {
    $variables['#attached']['library'][] = 'retro/retro';
  }
}

/**
* Implements hook_preprocess_HOOK() for node.html.twig.
*/
function retro_preprocess_node(array &$variables): void {
  if ($variables['node']->bundle() == "article") {
    $variables['#attached']['library'][] = 'retro/retro';
  }
}
```

On a twig template using **attach_library**

```
{# Only attach our retro library if this is an 'article' node. #}
{% if node.bundle == 'article' %}
  {{ attach_library('retro/retro') }}
{% endif %}
```

Extend or alter an existing library

To extend a library we can use **libraries-extend** in ***.info.yml** file of a theme.

```
libraries-extend:
  # Name of the library you want to extend.
  core/drupal.dropbutton:
    # A list of one or more libraries you want included each time the parent is
    # used.
    - icecream/dropbutton
```

To alter an existing library we can use libraries-override in the ***.info.yml** file to the theme.

```
libraries-override:
  core/drupal.dropbutton:
    css:
      component:
        misc/dropbutton/dropbutton.css: css/my-dropbutton.css
```

If we want to remove an asset instead of replacing it, on the right side we can use **false** instead of an asset path.

Attributes and classes in twig templates

Attribute objects (**Drupal\Core\Template\Attribute**) are used to store one or more attributes for an HTML tag and provide theme developers with helpful utility methods for modifying the value(s) of any attribute. Attribute objects can be printed just like any other variable in a Twig template, but are unique in that when printed the object itself is smart enough to output the attribute name/value pairs in the correct format.

Given an Attribute object with the following key/value pairs:

```
attributes:
```

```
  classes:
    - node
    - node--article
  id: node-42
  data-custom: a string of custom data
```

Printing it out in a a template (NO spaces needed between the object and the tag name, the spaces will be added automatically)

```
<div{{ attributes }}></div>
```

will generate valid html like the following:

```
<div class="node node--article" id="node-42" data-custom="a string of
custom data"></div>
```

Working with the attributes object

Add or remove (a) class(es)

```
{%
  set classes = [
    'my-class',
    'my-other-class',
    'my-special-class',
  ]
%}
<div{{ attributes.addClass(classes).removeClass('my-special-class')
}}></div>
```

Set or remove attributes

```
<div{{ attributes.setAttribute('id', 'myID') }}>
<div{{ attributes.setAttribute('data-bundle', node.bundle) }}>
<div{{ attributes.removeAttribute('id') }}>
```

Check for the existence of a class

```
{% if attribute.hasClass('myClass') %}
  {# do stuff #}
{% endif %}
```

Any of these dot methods can be chained.

Using Twig's without filter

When printing out individual attributes to customize them within a Twig template, you can use the without filter to prevent attributes that have already been printed from being printed again:

```
<div class="{{ attributes.class }} my-custom-class"{{
attributes|without('class') }}>
```

Responsive web design

Responsive image styles

The core Responsive Image module provides responsive image styles.

They are a mapping between breakpoints and image styles.

They can work with breakpoints defined in your theme or in the responsive image style settings.

Once the responsive image style is defined, it can be used in the display settings for Image fields.

Responsive image use cases

The three main uses cases for responsive images are:

1. **Viewport sizing:** providing different sizes of the same image based on the size of the viewport, but without changing aspect ratio, orientation, cropping, or content
2. **Art direction:** providing different image sources that may have alternate content or a different aspect ratio or orientation
3. **Display-density:** providing 1.5x or 2x images for higher display-density user devices.

For UC #1 and #3, the resulting markup uses the srcset and sizes attribute in an <img> element. In the #2 use case, <picture> and <source> elements are generated in conjunction with the <img> tag.

For more details on how to configure responsive image styles in Drupal to accomplish any of these 3 use cases: https://drupalize.me/tutorial/responsive-image-style-use-cases?p=0

More details about responsive images in web

https://cloudfour.com/thinks/responsive-images-part-10-conclusion/

Breakpoints in Drupal

In Drupal, a site's breakpoints can be exposed to other modules and themes in YAML format inside a breakpoints configuration file (either in a theme or in a module).

Example from Olivero (olivero.breakpoints.yml file):

```yaml
olivero.sm:
  label: Small
  mediaQuery: 'all and (min-width: 500px)'
  weight: 0
  multipliers:
    - 1x
olivero.md:
  label: Medium
  mediaQuery: 'all and (min-width: 700px)'
  weight: 1
  multipliers:
    - 1x
olivero.lg:
  label: Large
  mediaQuery: 'all and (min-width: 1000px)'
  weight: 2
  multipliers:
    - 1x
olivero.xl:
  label: X-Large
  mediaQuery: 'all and (min-width: 1300px)'
  weight: 3
  multipliers:
    - 1x
```

These breakpoints are defined and used in Olivero's css files.

Each breakpoint machine name should start with the extension name, a dot and the name, as in the examples above.

Optionally, a group can be added in the middle. Also the group key can be used to add the group a friendly name to display in the admin.

By exposing your site's breakpoints in this way, other modules (such as Responsive Images module) can create features that make use of breakpoints.

Layouts

Layout builder

Layout builder consists of two core modules:

- **Layout Discovery:** An API module that defines what a layout is and handles layout registration and rendering.
- **Layout Builder:** Drag-and-drop UI for creating flexible layouts built on top of Layout Discovery API.

Each layout consists of:

- Sections
- Blocks inside the sections: These are not the same as standard Drupal blocks. They might be content type fields, drupal blocks or custom blocks created with the Layout builder UI.

A layout can be associated with a specific display for a content type. In order to use layout builder for a content type display, we go to manage display on the content type, enable the use of layout builder and then click on “Manage Layout”.

Also a layout can be applied to a specific node on a content type. In order to do that, we need to select the "Allow each content item to have its layout customized" checkbox in the manage display admin.

Once you select the option, when you go to edit a node of that content type, it will have a new “Layout” tab where you’ll be able to modify the layout specifically for that node.

https://drupalize.me/course/learn-drupals-layout-builder
https://www.drupal.org/docs/develop/development-tools

1.2 Identify JavaScript and jQuery programming concepts.

1.3 Demonstrate the use of Git for version control.

2. Site Building

2.1 Demonstrate ability to create and configure content types with appropriate fields and field settings for building basic data structures.

2.2 Demonstrate ability to configure display modes for building custom form and view modes for core entities.

2.3 Demonstrate ability to create and use taxonomy vocabularies and terms for classification and organization of content.

2.4 Demonstrate ability to configure block types, manage blocks library and configure block layouts.

2.5 Demonstrate ability to build main and alternative navigation systems by using menus.

2.6 Demonstrate ability to create and configure views for building content list pages, blocks, and feeds.

2.7 Demonstrate ability to use configuration management capabilities for exporting site configurations.

2.8 Demonstrate ability to build multilingual websites using core multilingual capabilities.

3. Front-end development (theming)

3.1 Demonstrate ability to create a custom theme or sub theme.

3.2 Demonstrate knowledge of theming concepts.

3.3 Demonstrate ability to use Twig syntax.

3.4 Demonstrate ability to build or override Twig templates for defining layout content.

3.5 Demonstrate ability to write template pre-process functions for overriding custom output.

4. Back-end development (coding)

4.1 Demonstrate ability to write code using core and object-oriented PHP.

4.2 Demonstrate ability to develop custom modules using Drupal API for extending Drupal functionality.

4.3 Demonstrate ability to store and retrieve data using code.

4.4 Demonstrate ability to work with other essential APIs.

4.5 Demonstrate ability to write code using Drupal coding standards.

4.6 Demonstrate ability to analyze and resolve site performance issues arising from site configuration or custom code.

4.7 Demonstrate ability to analyze and resolve security issues arising from site configuration or custom code.

4.8 Demonstrate ability to write a test using the core testing framework.