RESPONSIVE WEB DESIGN

LESSON 05

SWAFE-01

OVERVIEW

- In the early days of web design, pages where built to target a particular screen size
- If users had different screen sizes than expected:
    - Unwanted scrollbars
    - Overly long line lengths
    - Poor use of space
- As more diverse screen sizes became available, the concept of responsive appeared
- Responsive web design allow web pages to alter layout and appearance to suits different screen widths and resolutions

It is important to understand that

RESPONSIVE WEB DESIGN

IS NOT A SEPARATE TECHNOLOGY

It is a term used to describe an approach to web design

WHAT NEEDS TO BE RESPONSIVE?

- Containers – Document divisions, sections, articles
- Text – Headings
- Media – Images, Video players

MOBILE-FIRST DESIGN

- Design with mobile users in focus
- Identify most the important content to present
- Make it easy to navigate
- There is a difference between a mobile-first design and a mobile-reponsive design
- Start with a very basic design and gradually add more complexity

VIEWPORT & MEDIA QUERIES

OVERVIEW

- The user's visible area of a web page
- The `viewport` meta tag instructs the device to set the width to the device width
    - Why is this needed? Devices lie about their width!
    - iPhone set viewport to 960px
- Layout that kicks in at specific breakpoints will never kick in
- Override with `width` property to set device width

THE VIEWPORT META TAG

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MediaBreakpoints</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

examples/lesson05-reactive-web-design/projects/media-breakpoints/src/index.html

- `width=device-width` tells the browser to set the viewport to the actual device width
- `initial-scale=1` tells the browser scale the document to 100% of it's intended size

MEDIA QUERIES

- Media queries adapt web applications depending on the various device characteristics and parameters
- In CSS, use `@media` rule to conditionally apply styles
- Media types
  - `all` –suitable for all devices
  - `screen` –intended for screens
  - `print` –intended for print
  - `speech` -intended for speech synthesizers
- Logical operators
  - `not` – negates the query (must be used with media type)
  - `and` – combines multiple queries into one (also used to combine media types with media features)
  - `only` –apply only styles if entire query matches (useful when writing backwards compatible queries)

MEDIA FEATURES

- Media features describe specific characteristics of the user agent
- `orientation` –specifies the viewport orientation, can be the following values:
  - `portrait` The viewport is in a portrait orientation, i.e., the height is greater than or equal to the width
  - `landscape` The viewport is in a landscape orientation, i.e., the width is greater than the height
- `width` and `height`
  - Can be prefixed with `min-` and `max-`
- Check out the documentation for complete list of available media features

MEDIA QUERIES

```scss
@media screen and (max-width: 425px) {
  html, body {
    background-color: lime;
  }
  p {
    background-color: tomato;
  }
}

@media screen and (min-width: 426px) and (max-width: 768px) {
  p {
    background-color: steelblue;
  }
}

@media screen and (min-width: 769px) {
  p {
    background-color: skyblue;
  }
```

examples/lesson05-reactive-web-design/projects/media-breakpoints/src/app/app.component.scss

FLEXBOX

Run `ng serve --project flexbox` in `examples/lesson05-reactive-web-design`

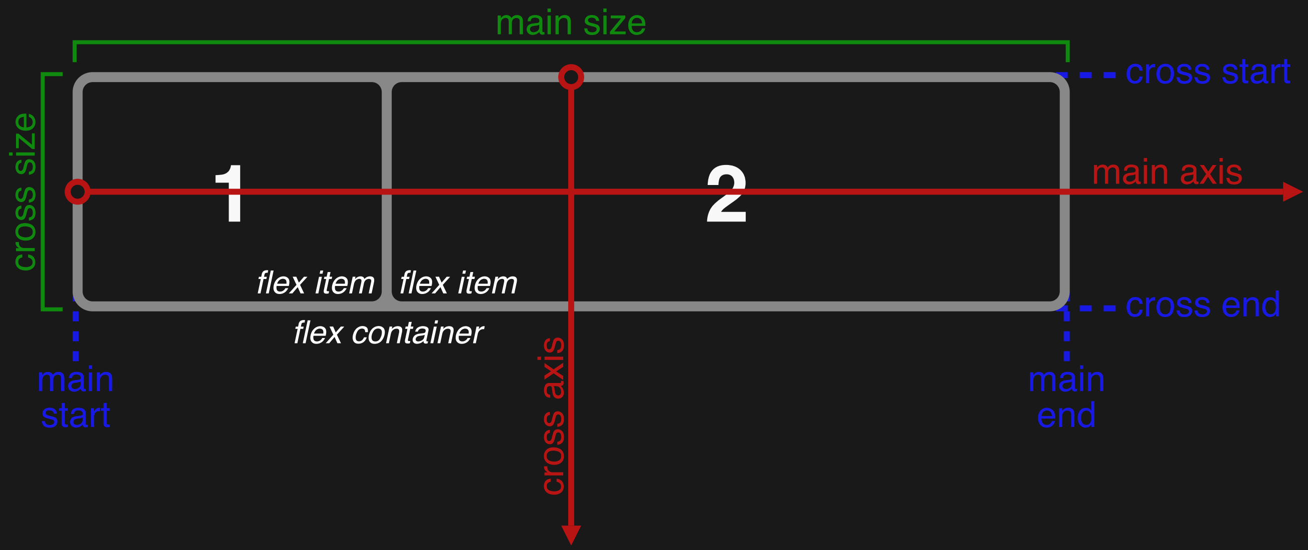
OVERVIEW

- Flexbox is a one-dimensional layout model concerned with one dimension at a time
- An element with the `display` property set to `flex` is a flex container
- An child element in a flex container is called a flex item
- The two axes of flexbox
  - Main axis – defined by the `flex-direction` property
  - Cross axis – runs perpendicular to the main axis

THE TWO AXES OF FLEXBOX

- The main axis is defined by `flex-direction`
- It has four possible values:
  - `row` –orientation matches the inline axis of the current writing mode
  - `row-reverse` –same as `row`, but `main-start` and `main-end` are switched around
  - `column` –orientation matches the block axis of the current writing mode
  - `column-reverse` –same as `column`, but `main-start` and `main-end` are switched around

FLEX CONTAINER



https://www.w3.org/TR/css-flexbox/images/flex-direction-terms.svg

FLEX ITEMS

- All direct children of a flex container becomes a flex item
- Flex items have three properties:
  - `flex-grow` –Positive free space. Causes items to take up more space
  - `flex-shrink` —Negative free space. Causes items to take up less space
  - `flex-basis` —Size before shrink and grow. Defines the size space items leaves as available space
- Often expressed with the shorthand notation: `flex: <grow> <shrink> <basis>`

WRAPPING

- Flexbox is designed as a one-dimensional layout
    - Layout items as rows
    - Layout items as columns
- Flexbox can also wrap items, creating a multi-line container
- Line wrapping is controlled by the `flex-wrap` property, which can have the following values
    - `nowrap` It items are too wide to fit the container, they will overflow it
    - `wrap` An item wraps to a new line if they there is not enough available space to place it on the current line
    - `wrap-reverse` Same as `wrap` but will start at `main-end`
- Can be combined with `flex-direction`

ALIGNMENT

- A key feature of Flexbox is the ability to align and justify items to the main- and cross-axes
- It enabled propper vertical alignment
- Offers the following properties
    - `justify-content`–aligns items on the main-axis
    - `align-items`–aligns items on the cross-axis
    - `align-self`–aligns individual items on the cross-axis

`justify-content`

- Align items across the main-axis
- Available values for `justify-content`
  - `flex-start` –items aligns to `main-start`
  - `flex-end` –items aligns to `main-end`
  - `center` –items are horizontally centered in the container
  - `space-between` –items are placed with even space between each other. First item is aligned with main-start and last item is aligned with `main-end`
  - `space-around` —items are placed with even space between each other. First and last item will have half-size space between `main-start|end`

`align-items` AND `align-self`

- Align items across the cross-axis
- Available values for `align-items`
  - `stretch`–streches a flex children vertically
  - `flex-start`–items aligns to cross-start
  - `flex-end`–items aligns to cross-end
  - `center`–items are vertically centered in the container
  - `baseline`–items are placed after largest distance between its baseline and its cross-start
- Use `align-self` to override values set with `align-items`
  - Additional possible value `auto`—defers cross-axis alignment control to the value of `align-items` (initial value)

CSS GRID LAYOUT

Run `ng serve --project grid` in `examples/lesson05-reactive-web-design`

OVERVIEW

- CSS Grid layout introduces a two-dimentional grid system to CSS
- Terminology
  - Grid line –are the horizontal and vertical dividing lines
  - Grid track –the space between two grid lines
  - Grid cell –the intersection a grid row and a grid column
  - Grid area –consists of one or more adjacent grid cells
- Tracks is a generic term for a grid row or grid column
  - Fixed and flexible track sizes
  - Item placement
  - Creation of additional tracks to hold content
  - Alignment control
  - Control of overlapping content

PROPERTIES

- Template properties are used to define tracks
  - `grid-template-columns`–defines line names and track sizing functions of grid columns
  - `grid-template-rows`–defines line names and track sizing functions of grid rows
- Functions
  - `repeat()`–represent a repeated fragment of tracks
  - `minmax(min, max)`–defines a size range greater than or equal to `min` and less than or equal to `max`
- Unit `fr` defines flexible space in terms of a fraction of the available leftover space

GRID

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(3, 1f
  gap: 10px;
  grid-auto-rows: minmax(100px, auto)
  background-color: darkkhaki;
}

.one {
  background-color: aqua;
  grid-column: 1 / 3;
  grid-row: 1;
}

.two {
  background-color: cornflowerblue;
  grid-column: 2 / 4;
  grid-row: 1 / 3;
  opacity: 0.75;
```

examples/lesson05-reactive-web-design/projects/grid/src/app/grid/grid.component.scss

NAMED AREAS

```scss
#grid {
  display: grid;
  grid-template-areas: "head head"
                       "nav  main"
                       "foot foot";
  grid-template-columns: 1.5fr 4fr;
  grid-template-rows: 60px calc(100vh

}
#grid > header {
  grid-area: head;
  background-color: tomato;
}
#grid > nav {
  grid-area: nav;
  background-color: crimson;
}
#grid > main   {
  grid-area: main;
```

examples/lesson05-reactive-web-design/projects/grid/src/app/named-areas/named-areas.component.scss

RELATIONSHIP OF LAYOUT METHODS

- Grid and flexbox
  - One-dimensional vs. two-dimentional layout
    - Control the layout by row OR column? Use `flexbox`
    - Control the layout by row AND column? Use `grid`
  - Content out or layout in?
    - Flexbox is content out
    - Grid is layout in
- Other layout methods
  - Simple positioning
    - Relative, absolute, fixed, sticky
  - Float positioning
    - Remove elements from the normal flow and float around them

TYPICAL USE CASES

- Navigation
- Split navigation
- Centering items
- Card layout pushing footer down
- Form controls

CSS PREPROCESSING

OVERVIEW

- Extends Cascading Style Sheets (CSS) by providing paradigms known from conventional programming languages
- Preprocessors
  - Sass
    - Two syntaxes: Sass, Sassy Cascading Style Sheets (SCSS)
    - Biggest difference between Sass and SCSS is curly brackets and semicolon
  - Less – Leaner Style Sheets
- They ALL compiles into CSS

SCSS

- Variables
- Nesting
- Partials
- Modules
- Mixins
- Extend/Inheritance
- Operators

VARIABLES

- Assign a value to a name that begins with '`$`'
- Refer to that value instead of the value itself
- A variable declaration is written `<variable>: expression;`
- Default values
  - Allows configuration of variables in modules
  - Use the `!default` to set default values
  - Load module with `@use <url> with (<variable>: <value>, <variable>: <value>, ...`

NESTING

- HTML has a clear nested and visual hierarchy, CSS does not
- Sass enables nesting of CSS selector
  - Follows the same visual hierarchy as HTML
  - A great way to organize CSS code and make it more readable
- Beware that overly nested rules will produce over-qualified CSS
  - Hard to maintain
  - Generally considered bad practice

PARTIALS & MODULES

- Partials
  - Partial Sass files are snippets of CSS code that is included in other Sass files
  - Partial files are named with a leading underscore, e.g. `_base.scss`
    - The underscore tells Sass that it is a partial file and should not compiled
  - A great way to modularize CSS code, that makes it easier to maintain
- Modules
  - Modules can be loaded into other files with the `@use` rule
  - Access mixins and functions with a namespace based on filename
  - Using a module will include the generated CSS in the compiled output

MIXINS

- Some things in CSS are a bit tedious to write
- Mixins lets you make groups of CSS delarations and reuse them
- You can pass in values to make it even more flexible
- Use the `@mixin` directive to create a mixin, e.g. `@mixin theme($theme <value>)`
- A good use for mixins is for vendor prefixes
    - Browser vendors add prefixes to experimental or nonstandard CSS properties to prevent breaking code
- Use mixins in CSS declarations with `@include` followed by the name of the mixin

EXTEND/INHERITANCE

- Keep Sass code very DRY
- Use `@extend` to share CSS properties from one selector to another
- `Placeholder` classes optimizes compiled CSS output
    - Only included if extended
    - This keeps the output neat and clean
- Use with care
    - You can create unintended selectors if extending a nested selectors
    - Watch out for combining unrelated selectors in compiled CSS output

OPERATORS

- Doing math in CSS can be very helpful
- Sass has a handful of standard math operators: `+`, `-`, `*`, `/`, and `%`
- Operations take pixel values and easily convert them to percentages

BASE

```scss
$base-color: #2d3142 !default;
$accent-color: tomato !default;

@font-face {
  font-family: 'RobotoMono-Bold';
  src: url('./assets/fonts/roboto-mono/static/RobotoMono-Bold.ttf');
};

@mixin text-shadow($font-family: 'Lobster', $font-size: 2em) {
  font-family: $font-family;
  font-size: $font-size;
  color: $base-color;
  text-shadow: 3px 3px $accent-color;
}

h1 {
  @include text-shadow
}
```

examples/lesson05-sass/_base.scss

HEADER COMPONENT

INPUT

```scss
@use './base';

.h1-roboto-mono {
  @include base.text-shadow(
    'RobotoMono-Bold',
    3em
  )
}
```

examples/lesson05-sass/header.component.scss

OUTPUT

```css
@font-face {
  font-family: "RobotoMono-Bold";
  src: url("./assets/fonts/roboto-mon
}
h1 {
  font-family: "Lobster";
  font-size: 2em;
  color: #2d3142;
  text-shadow: 3px 3px tomato;
}

.h1-roboto-mono {
  font-family: "RobotoMono-Bold";
  font-size: 3em;
  color: #2d3142;
  text-shadow: 3px 3px tomato;
}
```

examples/lesson05-sass/out/header.component.css

NAVIGATION BAR COMPONENT

INPUT

```scss
@use './base' with (
  $base-color: #cccccc,
  $accent-color: #6699ff
);

.wrapper {
  background-color: #333333;
  padding: 10px 20px;
  h1 {
    margin: 0;
  }
}
```

examples/lesson05-sass/navigation-bar.component.scss

OUTPUT

```css
@font-face {
  font-family: "RobotoMono-Bold";
  src: url("./assets/fonts/roboto-mon
}

h1 {
  font-family: "Lobster";
  font-size: 2em;
  color: #cccccc;
  text-shadow: 3px 3px #6699ff;
}

.wrapper {
  background-color: #333333;
  padding: 10px 20px;
}

.wrapper h1 {
  margin: 0;
```

examples/lesson05-sass/out/navigation-bar.component.css

ANIMATIONS

Run `ng serve --project animations` in `examples/lesson05-reactive-web-design`

OVERVIEW

- CSS animations makes it possible to animate transitions from one style to another
- Three key advantages to CSS animations
    - Easy and simple to implement
    - The animations runs well under moderate system load.
    - Simple animations often perform poorly when written in JavaScript
    - The browser optimizes performance and efficiency

ANIMATION PROPERTIES

- Animation has the following sub-properties:
  - `animation-name`–name of the `@keyframes` at-rule
  - `animation-duration`–time the animation should take to complete one cycle
  - `animation-timing-function`–the timing of the animation defined by a keyframes acceleration curve
  - `animation-delay`–time between the load and the beginning of animation sequence
  - `animation-iteration-count`–how many times should animation sequence repeat
  - `animation-fill-mode`–how are styles applied before and after animation sequence
  - `animation-play-state`–pause and resume animation sequence

ANIMATION

```scss
.block-animation {
  animation-name: round;
  animation-duration: 5s;
  animation-fill-mode: forwards;

  @keyframes round {
    from {
      border-radius: 0;
    }
    to {
      border-radius: 50%;
    }
  }
}

.radius {
  border-radius: 50px;
}

```

examples/lesson05-reactive-web-design/projects/animations/src/app/animations/animations.component.scss

TRANSITIONS

- CSS transitions provide a way to control animation speed when changing CSS properties

TRANSITION PROPERTIES

- Transitions has the following sub-properties:
  - `transition-property` –specifies the name of the CSS properties to which transistions should be applied
  - `transition-duration` –specifies over which duration the transition should occur
  - `transition-timing-function` –specifies how intermediate property values are computed
  - `transition-delay` –specifies the time from when property is changed to when transistion begins

TRANSITIONS

```scss
.container {
  display: flex;
  align-items: center;
  justify-content: center;
  width: 500px;
  height: 500px;
  border: 1px solid lightgray;
  div {
    background-color: tomato;
    width: 100px;
    height: 100px;
    transition-property: transform, b
    transition-duration: 2s, 2s, 2s;
  }
  div:hover {
    transform: rotate(360deg);
    background-color: lightgreen;
    border-radius: 50px;
  }
```

examples/lesson05-reactive-web-design/projects/animations/src/app/transitions/transitions.component.scss

ANIMATIONS VS. TRANSITIONS

- Animations
  - Explicit–property changes are defined with keyframes
  - Use `animation` for complex animation sequences
- Transitions
  - Implicit–the browser handles the property changes
  - Use `transition` for simple animation sequences
- Check out the shorthand notation in the documentation and the example code

WRAP-UP

- View ports and media queries
- Flexbox and Grid layout
- CSS preprocessing
- Animations