SuperHi-Advanced-Week4

http://guides.superhi.com/advanced/good-times/#/0

Good Times

We look at more scroll events, animations, 3d perspective effects and taking card payments using Stripe.

What we'll learn…

- Exploring fully responsive grid layouts
- Using CSS toolkits to write cleaner code
- Consistent layout and sizing using a design system
- Scrolling events on our page
- Taking card payments using Stripe
- 3D perspective image effects

Outlining text with a stroke

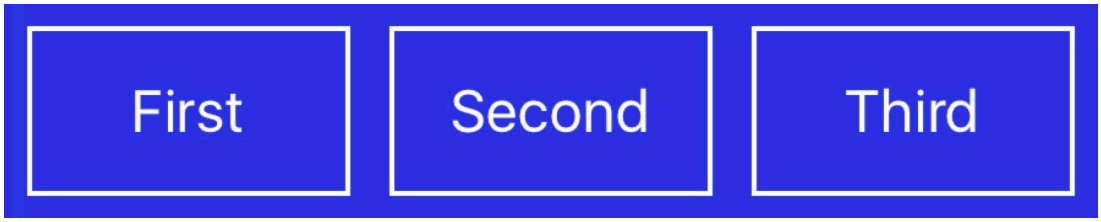
```
.heading {
  color: #fff;
  -webkit-text-fill-color: transparent;
  -webkit-text-stroke-width: 1px;
  -webkit-text-stroke-color: #fff;
}
```

Using pure CSS we can apply a stroke to text on our page. It only really works well for large display text, and the idea is that we use `fill-opacity` along with some modern `-webkit-text` features to apply it.
The `-webkit` part of the code is called a vendor prefix. It allows us to use modern CSS features that haven't been adapted into all browsers yet without breaking older versions.

Grids using basscss

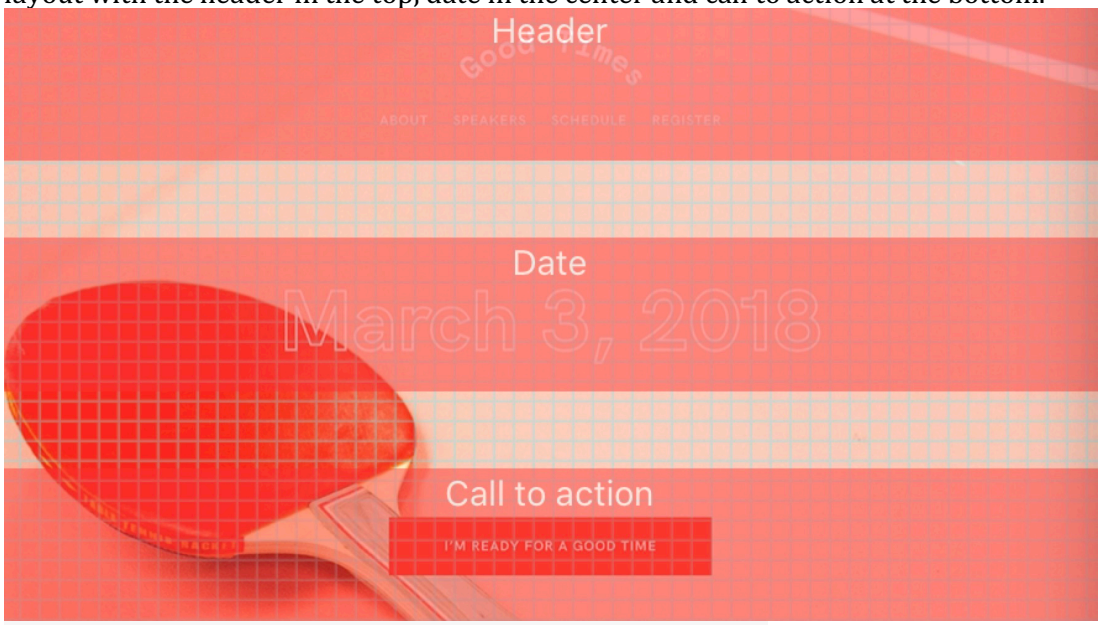
http://basscss.com
Basscss is very much like Tachyons, and we're going to use it's 12 column grid classes to help us out a bit.
The main difference is that we are going to use classes like `.col-2` and `.col-6` (these would give us 2 of 12 and 6 of 12 columns.

```
<div class="flex">
  <div class="col-4">First</div>
  <div class="col-4">Second</div>
  <div class="col-4">Third</div>
</div>
```



Hero section using flexbox

Using flexbox along with some helper classes from Tachyons, we can achieve our hero element layout with the header in the top, date in the center and call to action at the bottom.




```
<div class="flex flex-column min-vh-100 justify-between">
  <header class="header"></header>
  <div class="date"></div>
  <div class="call-to-action"></div>
</div>
```

Creating a revealing header

When we scroll we want the fixed header to show and reveal at the top of the page. To do this we're going to create a special duplicate fixed header which has a class of `.header-desktop`

```
// here we grab our original header
const header = document.querySelector('.header')
// and then we grab our desktop header
const desktopHeader = document.querySelector('.header-desktop')
// we duplicate the header contents into our desktop header (so we don't have to write it twice)
desktopHeader.innerHTML = header.innerHTML
inView('.header')
  // when our hedaers exits the viewport, make the desktop one visible
  .on('exit', el => desktopHeader.classList.add('visible'))
  // when it enters again, hide it
  .on('enter', el => desktopHeader.classList.remove('visible'))
```

Styling our fixed header

```
.header-desktop {
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  /* move it up by 100% of its height so it' s off the page */
  transform: translateY(-100%);
  /* give it a nice transition */
  transition: all 0.5s ease;
  /* force it to display on top of other elements */
  z-index: 4;
  /* a subtle shadow so it pops out a bit */
  box-shadow: 0 0 4px 0px rgba(65, 65, 65, 0.5);
}
/* adding the visible class shifts it down so it' s visible */
.header-desktop.visible {
  transform: translateY(0);
}
```

The idea of what we're going to do to show and hide our header is to add a class that toggles its `translate` position so it's visibly hiding or showing on the page.

```
@media (max-width: 800px) {
  .header-desktop {
    /* using !important to force this rule to apply regardless */
    display: none !important;
  }
}
```

We'll also add a media query to hide it on screen widths below `800px` as our navigation is a bit clunky for mobile.

Making our grid responsive

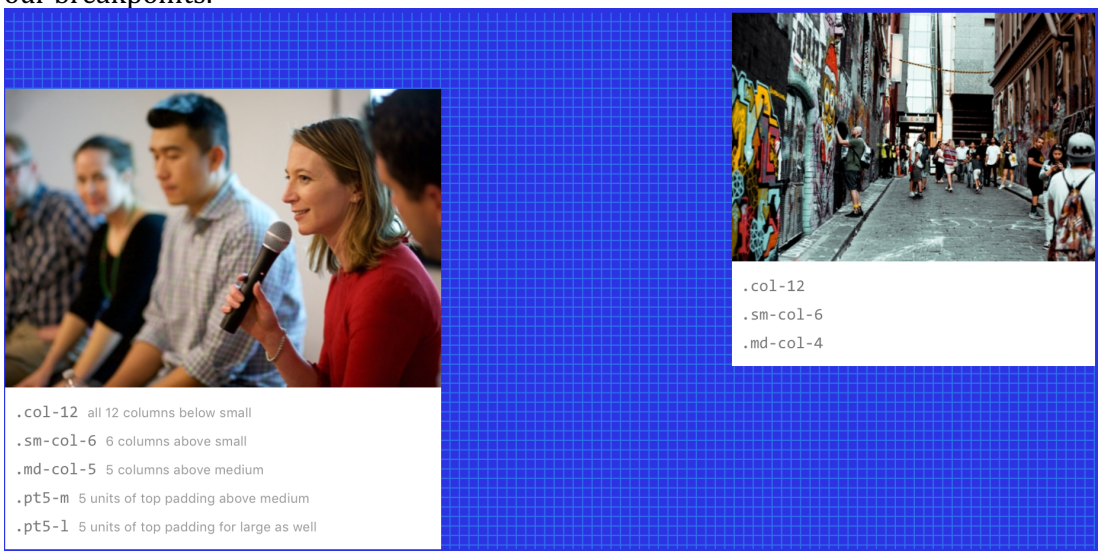
Although this looks like quite a lot of class names we get a fully responsive grid system as a result. We don't have to worry about writing any CSS, we just need to think about our columns and that each row adds up to 12.

```
<div class="flex flex-wrap">
  <div class="col-6 sm-col-4 md-col-3"></div>
  <div class="col-6 sm-col-4 md-col-3"></div>
  <div class="col-6 sm-col-4 md-col-3"></div>
  <div class="col-6 sm-col-4 md-col-3"></div>
</div>
```

- `col-6` takes up 6 of 12 columns
- `sm-col-4` above our small (30em/480px) breakpoint, it will take up 4 of 12 columns
- `md-col-3` above our medium (60em/960px) breakpoint, it will take up 3 of 12 columns

Laying out our images

Using the basscss grid helper classes we'll lay out our images into columns that vary in width at our breakpoints.



Adding indents to our columns

We added a couple of helper classes to our basscss grid to help with indenting the columns at higher width sreens. For our small breakpoint we'll indent by 1 column, and then 2 above medium.

Variables in CSS

Something really useful we can take advantage of is variables in our CSS. They're a relatively new feature that allow us to declare something once (e.g. our orange color) and use it in multiple places.

```
/* we declare our variable inside a :root selector */
:root {
  /* give the variable a name and value (needs two -- in front) */
  --orange: #ff3c15;
}
body {
  /* use the variable with var() and its name inside */
  color: var(--orange);
}
section {
  /* use it again! */
  border: solid 1px var(--orange);
}
```

Smooth horizontal scrolling boxes

We can make boxes that scroll horizontally using a combination of flexbox and overflow. Another thing we can do is to style up a custom scrollbar too.

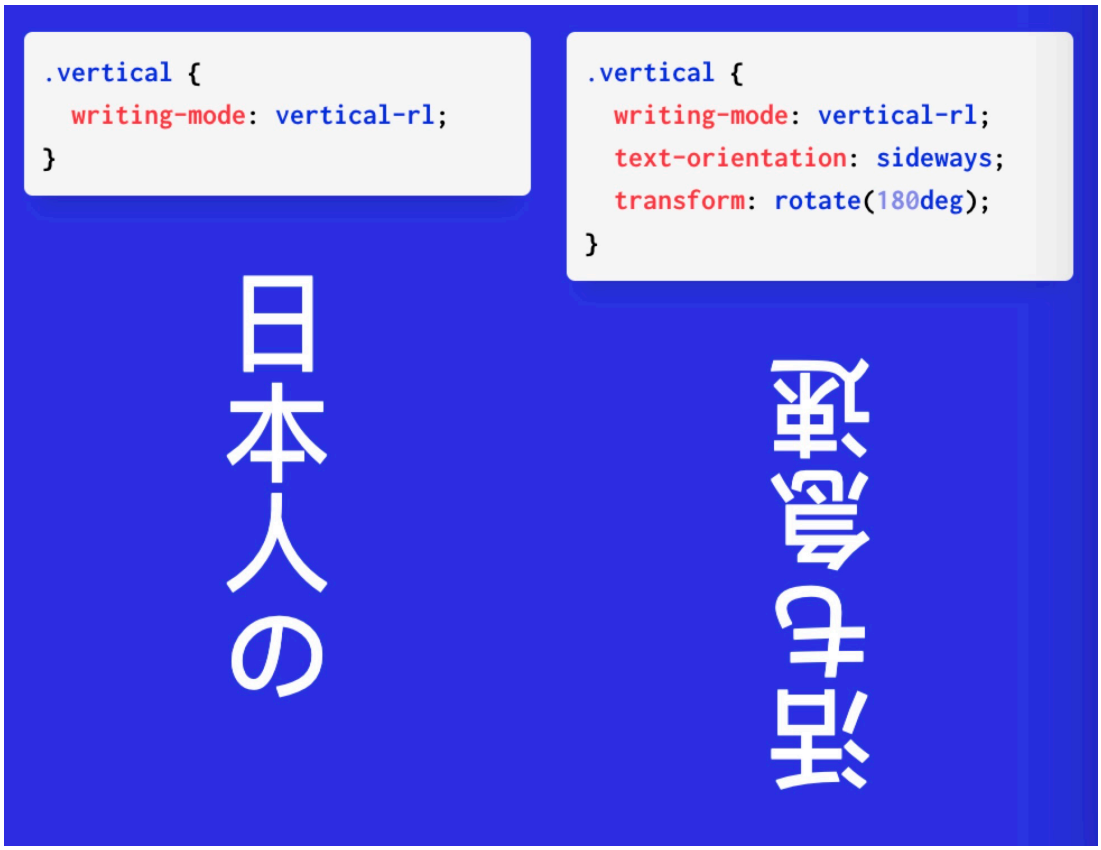
```
<!-- use a combination of flex and overflow: auto; -->
<div class="flex overflow-auto scroller">
  <div>We</div>
  <div>Can</div>
  <div>Make</div>
  <!-- and so on... -->
</div>

.scroller {
  /* this gives smooth scrolling on touch devices */
  -webkit-overflow-scrolling: touch;
}
/* we style up the scrollbar background */
.scroller::-webkit-scrollbar {
  width: 12px;
  height: 14px;
  background-color: transparent;
}
/* styles for the scrollbar itself */
.scroller::-webkit-scrollbar-thumb {
  background: #FDF035;
}
```

Vertical text

Using CSS `writing-modes` which are designed for languages that read vertically like Japanese. This is also useful if you want to create vertical text and avoid confusing 90° rotations!

```
.vertical {
  writing-mode: vertical-rl;
}
```

```
.vertical {
  writing-mode: vertical-rl;
  text-orientation: sideways;
  transform: rotate(180deg);
}
```



Perspective hover effect on images

https://micku7zu.github.io/vanilla-tilt.js/

Using vanilla-tilt.js gives us a really cool 3d perspective when we hover on images. It plays with the `rotateX` and `rotateY` transform properties.

```
<img src="image.jpg" class="tilt">
VanillaTilt.init(document.querySelectorAll(".tilt"), {
 max: 25,
 speed: 400
});
```

Adding `will-change: transform;` to an element improves performance and keeps things super snappy.

Payment form flip effect

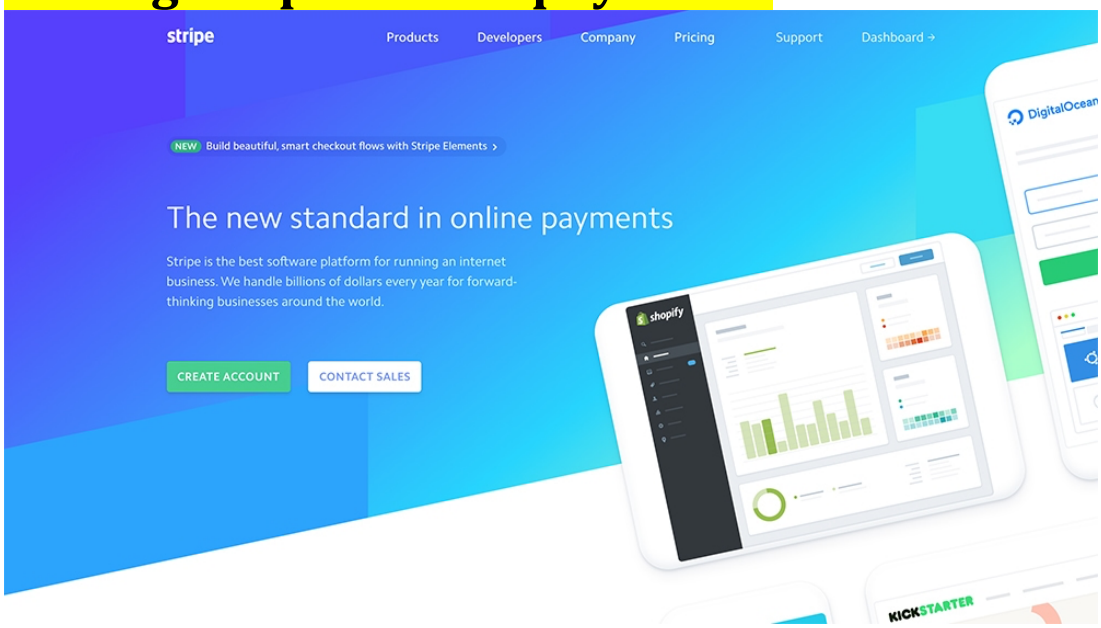
This great technique has been adapted from David Walsh's article to use CSS grids to stack the `.front` and `.back` on top of each other.

https://davidwalsh.name/css-flip

```
<div class="flip-container">
 <div class="flipper">
  <div class="front">Front face</div>
  <div class="back">Back face</div>
 </div>
</div>

/* entire container, keeps perspective */
.flip-container {
 perspective: 1000px;
}
/* flip the pane when hovered */
.flip-container.flipped .flipper {
 transform: rotateY(180deg);
}
/* flip speed goes here */
.flipper {
 transition: 0.6s;
 transform-style: preserve-3d;
 position: relative;
 display: grid;
 place-items: center;
}
/* hide back of pane during swap */
.front, .back {
 backface-visibility: hidden;
```

```
  grid-area: 1/1;
}
/* front pane, placed above back */
.front {
  z-index: 2;
  /* for firefox 31 */
  transform: rotateY(0deg);
}
/* back, initially hidden pane */
.back {
  transform: rotateY(180deg);
}
```

Adding Stripe to take payments



Stripe is a payment provider that allows developers to take card payments with ease. We've set up a test Stripe account and backend to test things with. The username and password are in the`copy.txt`.

Creating our payment form

Using the HTML structure from the Stripe documentation we can add the card form to our page with relative ease. We also need to add our own inputs for collecting our user's `name` and `email`.

```
<!-- include the stripe library on our page -->
<script src="https://js.stripe.com/v3/"></script>
<!-- our form action points to our backend -->
<form action="https://api.superhi.com/api/test/orders" id="payment-form">
  <!-- the name attributes on inputs match to our data -->
  <input name="name" placeholder="Name" required>
  <!-- we use the email type here and make the fields required -->
  <input name="email" placeholder="Email" type="email" required>
```

```
 <!-- stripe will mount the card inputs here -->
 <div id="card-element"></div>
 <!-- any validation errors are inserted here -->
 <div id="card-errors" role="alert"></div>
 <!-- a button to submit the form with -->
 <button>
  Buy ticket for $100 AUD
 </button>
</form>
```

Taking the payment using Stripe

Stripe makes payments by securely taking the card details and creating a `token` with them (a token is just a long code). With this `token` we can then send it to our backend to complete the payment.

```
const form = document.getElementById('payment-form');
form.addEventListener('submit', (event) => {
 event.preventDefault();
 stripe.createToken(card).then((result) => {
  if (result.error) {
   // Inform the user if there was an error
   errorElement.textContent = result.error.message;
  } else {
   // Send the token to your server
   stripeTokenHandler(result.token);
  }
 });
});
```

Here we grab the form element and listen for the `submit` event on it. Once it submits, we stop the default form action using `preventDefault()`.
Using the `card` data we create a token using Stripe, and if it's succesful we run the `stripeTokenHandler` function to finalise the payment with our backend.

Using fetch to send data

```
// we post to the url of our backend
fetch('https://api.superhi.com/api/test/orders', {
 // we tell fetch to POST instead of GET
 method: 'POST',
 // tell it we are sending across json data
 headers: {
  'Content-Type': 'application/json'
 },
 // here we send the data across using JSON.stringify to make it secure
 body: JSON.stringify({
  order: {
   name: 'Lawrence',
   email: 'lawrence@superhi.com'
   stripe_token: 'tok_1BPwruIWwy1ke4X7LR5ULl3a',
  }
 })
```

```
})
 // convert the response into json
 .then(response => response.json())
 .then(data =>
   // do something with the data if it's successful
 )
 .catch(error =>
   // if there's an error, do something with it
 )
```

Using `fetch` we send across the order information to our backend to complete the payment. The data needs to look like this:

```
{
 order: {
  name: 'Alan Partridge',
  email: 'alan@peartree.co.uk',
  stripe_token: 'tok_1BPwruIWwy1ke4X7LR5ULl3a'
 }
}
```

If it's successful we'll get back some data about the order, otherwise it'll give us back an error.