9 Creating a virtual credit card

This chapter covers

- Using Flexbox and `position` in layout
- Working with background images and sizing
- Loading and applying local fonts
- Using transitions and the `backface-visibility` property to create a 3D effect
- Working with additional styles such as the `text-shadow` and `border-radius` properties

As we saw in chapter 3, animation in CSS opens lots of opportunities to create interactive web experiences. In chapter 3, we used animations to give users the sense that something was happening in the background as they waited for a task to complete. Now we'll use animation to respond to users' interactions and create a flip ef-

fect for a credit card image. On one side, the animation will show the front of a credit card; on hover or on click for mobile devices, it will flip to show the back of the credit card.

This effect is useful to users, as we're re-creating what their credit cards may look like, showing which information from the cards they need to enter when buying something online, such as the expiration date or the security code. Animation is a way to represent something in real life by re-creating it for the web. This project goes hand in hand with the one in chapter 8, in which we designed a checkout cart.

We'll also explore styling images to set the background of the credit card and icons on the card. We'll use the CSS Flexbox Layout Module for the layout, as well as styling properties such as shadows, colors, and border radius. By the end of the chapter, our layout will look like figure 9.1.

Figure 9.1 Final output of the front and back of the credit card

As we go through the project, feel free to try customizing it to match your style. Try a different background image or typeface, for example. This project is a great opportunity to tweak the styling to suit your style. Let's get started.

.1 Getting started

Our HTML is made up of two main parts. Within the overall section representing the virtual card are a front side and a back side. You can find the starting HTML in the `chapter-09` folder of the GitHub repository (**http://mng.bz/Bm5g**), on CodePen (**https://codepen.io/michaelgearon/pen/YzZKMKN**), and in the following listing.

Listing 9.1 Project HTML

```
<section class="card-item">                                          ①
  <section class="card-item__side front">                            ②
    <div class="card-item__wrapper">
      <div class="card-item__top">                                   ③
        <img src="chip.svg" class="card-item__chip" alt="card chip">  ③
        <div class="card-item__type">                                ③
          <img src="logo.svg" alt="Card Type" class="card-item__typeImg"  ③
```

```
➥ height="37" width="152">                                    ③
        </div>                                                 ③
      </div>
      <div class="card-item__number">                          ④
        <div>1111</div>
        <div>2222</div>
        <div>3333</div>
       <div>4444</div>
      </div>
      <div class="card-item__content">                         ⑤
        <div class="card-item__info">
          <div class="card-item__holder">Card Holder</div>
          <div class="card-item__name">John Smith</div>
        </div>
        <div class="card-item__date">
          <div class="card-item__dateTitle">Expires</div>
          <div class="card-item__dateItem">02/22</div>
        </div>
      </div>
    </div>
  </section>
  <section class="card-item__side back">
    <div class="card-item__band"></div>
    <div class="card-item__cvv">
      <div class="card-item__cvvTitle">CVV</div>
      <div class="card-item__cvvBand">999</div>
      <div class="card-item__type">
        <img src="card-type.svg" class="card-item__typeImg"
➥ height="30" width="50">
      </div>
    </div>
  </section>
</section>
```

① The container for the whole credit card

② The container for the front of the card

③ The section for the top front of the card

④ The section for the middle front of the card, showing the card number

⑤ The section for the bottom front of the card, showing the expiration date and cardholder name

We also have some starting CSS to change the background color to a light blue and increase the margin at the top of the page, as shown in the following listing.

Listing 9.2 Starting CSS

```
* {
  box-sizing: border-box;
}
body {
  background: rgb(221 238 252);
  margin-top: 80px;
}
```

We're using the universal selector that we looked at in chapter 1 to set the `box-sizing` value for all HTML elements to `border-box` . This selector has two values:

- `content-box` —This setting is the default value for calculating the width and height of an element. If the `content-box` height and width are `250px`, any borders or padding will be added to the final rendered width. Given a border of `2px` all around, for example, the final rendered width would be `254px`.
- `border-box` —The difference between this value and `border-box` is that if we set the element height to `250px`, any borders and padding will be included in this specified value. The `content-box` will reduce as the padding and border increase.

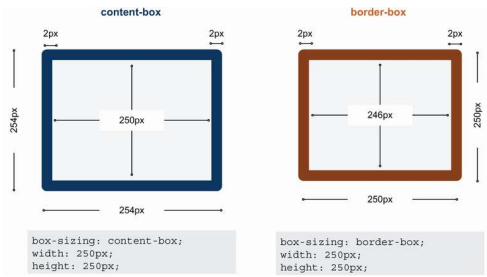

Figure 9.2 The effect of `box-sizing` on element size

Figure 9.2 shows an example. Our starting point looks like figure 9.3.

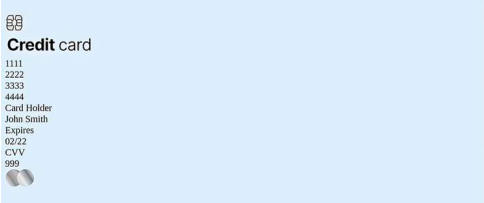

Figure 9.3 Starting point

.2 Creating the layout

Both the front and the back have a class name of `card-item__side`. The front also has a second class assignment of `front`, and the back has a second class of `back`. Having two class names—one that's identical on both sides and a second, different one—allows us to assign styles that are common to both sides using the `.card-item__side` selector (the class they have in common) and styles that are unique to a side in their individual rules of `.front {}` or `.back {}`.

Let's start by centering the card on the screen. The first step is setting the height and width of the card to a maximum width of `430px` and a fixed height of `270px`. We're also setting its position to `relative`, which will be useful when we place the back of the card on top of the front to create the flip effect later in this chapter (section 9.5).

The final piece is setting the left and right margins of the card to `auto` to center the card horizontally in the browser window. To do this, we use the `.card-item` selector to create the rule shown in the following listing.

Listing 9.3 Container styling

```
.card-item {
  max-width: 430px;
  height: 270px;
  margin: auto;
  position: relative;
}
```

Figure 9.4 shows the updated positioning.

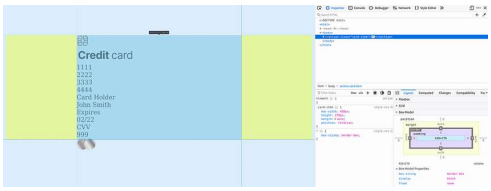

Figure 9.4 Centered credit card

9.2.1 Sizing the card

Now that we've set a maximum width and height for the card, we want to ensure that the front and back faces fill the entire space available to them within their parent container (the card). Therefore, we'll assign a height and width of `100%` to both sides of the cards by using the class selector `.card-`

`item__side` , as shown in the following listing.

Listing 9.4 Container shared between the front and back

```
.card-item__side {
  height: 100%;
  width: 100%;
}
```

With this piece of code added, our card faces (front and back) expand to match the size of its parent container, as figure 9.5 shows.

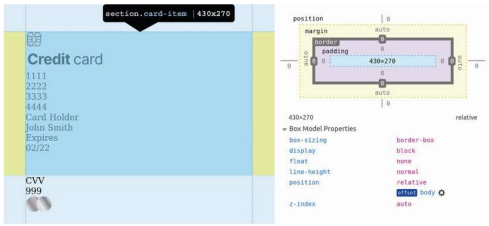

Figure 9.5 The card faces (front and back) match the parent's container size.

9.2.2 Styling the front of the card

For the front of the card, we have three main sections (figure 9.6):

- The top of the card has two images, one showing the chip and the other showing the type of credit card (such as Visa or MasterCard).

- In the middle is the card number, which is spread evenly across the width of the card.
- At the bottom are the cardholder's name and the card's expiration date. These elements are on opposite ends.

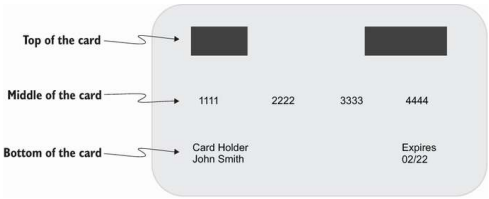

Figure 9.6 A wireframe of the front of the card

Before we start styling the individual parts of the front of the card, let's give the card face some padding so that the contents aren't positioned right up against the edge. We'll give them some breathing room. The following listing shows the code.

Listing 9.5 Container styling for the front of the card

```
.front {
  padding: 25px 15px;
}
```

Remember that in the styles originally provided with the project, we set the `box-sizing`

of all elements to `border-box`. With the added padding, we see that changing the `box-sizing` didn't increase the dimensions of the card face `<section>`; rather, it decreased the space available to the content (figure 9.7).

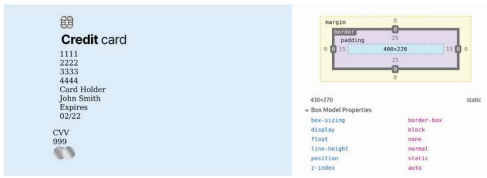

Figure 9.7 Card with added padding and box model diagram

Top of the card

We're using Flexbox for the layout of the card. As we've learned, Flexbox is likely to be the best choice for placing items in a single-axis layout. Also, we need to take advantage of the extra functionality Flexbox gives us with spacing and alignment—functionality that float doesn't give us.

NOTE For details on the CSS Flexbox Layout Module and its associated properties, check out chapter 6. Chapter 7 covers float.

With these facts in mind, we'll set the top of the card to have a `display` property value of

`flex` and set the alignment so that the tops of the elements align. The default property of `align-items` is `stretch`, which increases the heights of the `flex` items so that their heights match that of the tallest element in the set.

We don't want this distortion, though; we want the elements to be aligned vertically to the tops of the items. So we'll set the `align-items` property to `flex-start`. Then we'll set the `justify-content` property to `space-between`, which distributes the elements evenly along the axis, creating a gap between the two elements and placing them at the extreme edges of the card.

We'll give the top some margin and padding to position them further relative to the edge of the card. Then we'll increase the width of the chip to `60px`. As this image is an SVG, we can increase its size without affecting its quality. Because we're manipulating only the width and haven't altered the default height, the image's height will scale proportionally by default. The following listing shows the rules used to style the top portion of the card.

Listing 9.6 Layout for the top front of the card

```
.card-item__top {
  display: flex;
  align-items: flex-start;
  justify-content: space-between;
  margin-bottom: 40px;
  padding: 0 10px;
}
.card-item__chip {
  width: 60px;
}
```

Our updated card looks like figure 9.8.

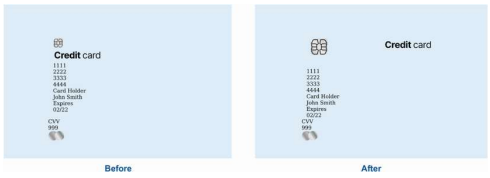

Figure 9.8 Styled top portion of the card

Middle of the card

In the middle front of the card, we find the card number. Again, we use a `display` property value of `flex`, with `justify-content: space-between` distributing the number groups evenly across the card's width. We also add padding and margin to add space between the numbers and the elements around them, as shown in the following listing.

Listing 9.7 Layout for the middle of the front of the card

```
.card-item__number {
  display: flex;
  justify-content: space-between;
  padding: 10px 15px;
  margin-bottom: 35px;
}
```

Figure 9.9 shows our number groups distributed evenly across the width of the card.

Figure 9.9 Evenly distributed numbers

Bottom of the card

In the bottom front of the card, we have two elements: cardholder name and card expiration date. As we did in the top and middle of the card, we want to separate the bits of information and place them at opposite edges of the card.

We'll follow the same pattern of using Flexbox, `justify-content`, and `padding` to place the elements. We don't need any margin this time, however. The following listing shows the rule we'll use.

Listing 9.8 Layout for the bottom front of the card

```
.card-item__content {
  display: flex;
  justify-content: space-between;
  padding: 0 15px;
}
```

Figure 9.10 shows the updated layout. Next, we'll position the elements on the back of the card.

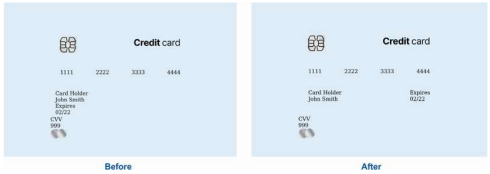

Figure 9.10 Layout for the front of the card

9.2.3 Laying out the back of the card

The layout for the back includes the security code number and a semitransparent band (the magnetic strip), as shown in figure 9.11. Let's start with the semitransparent back strip.

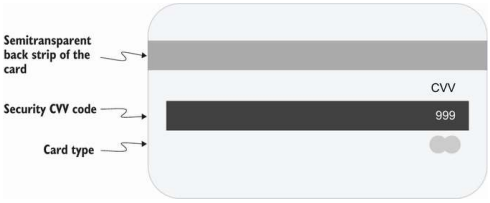

Figure 9.11 A wireframe of the back of the card

The strip has a class of `card-item__band`. We want to make it `50px` in height and position it `30px` from the top of the card. We'll use the `height` property to indicate how tall it should be. Even though the `<div>` is empty, it automatically takes the full width available to it because `<div>`s are block-level elements.

To move the strip down rather than keep it at the top of the back of the card, we'll add some padding to the back of the card itself. We can't give it margin, because it would push against the previously existing content (in the top card) rather than the top edge of the back.

Although we'll manage most of the theming later in this chapter, let's add the background color now so that we can see what we're doing (listing 9.9). The background is dark blue at 80% opacity, which will allow some of the background image we place on the card to show through.

Listing 9.9 Positioning the strip

```
.back { padding-top: 30px }
.card-item__band {
  height: 50px;
  background: rgb(0 0 19 / 0.8);
}
```

Now our strip looks like figure 9.12.

Figure 9.12 Styled strip on back of card

Security code

The security code has the letters *CVV* above it and a white band (usually intended for the user's signature) that includes the security code. Both the letters and the numbers are right-justified and nested inside a `<div>` with a class name of `card-item__cvv`.

For the letters *CVV*, because we don't need to distribute elements across the width of the card, we don't need to use Flexbox. Aligning the text to the right by using the `text-align` property is sufficient to accomplish the task. But we'll use Flexbox on the white band that contains the security numbers, not because it's needed to right-

justify the text but because it makes vertically aligning the content inside the band much easier. Let's start by giving the `card-item__CVV` container some basic styles: `padding` to space elements and the `text-align` property so that our text will place itself on the right of the card, as shown in the following listing.

Listing 9.10 Positioning the text

```
.card-item__cvv {
  text-align: right;
  padding: 15px;
}
```

With the container taken care of (figure 9.13), we can style the letters and security code individually.

Figure 9.13 Aligning the text

For the *CVV* letters, all we need to do is give this text some margin and padding to offset it from the right edge and away from the number below. Because we want the number to be inside a white band of a spe-

cific height, we'll use the `height` property with a value of `45px`. To align the text vertically in the middle of the box, instead of trying to calculate the amount of vertical padding necessary based on the text size, we'll use Flexbox with an `align-items` property value of `center`. We'll still use padding to separate the text from the right edge of the box, however.

Because Flexbox's default property value for `justify-content` is `flex-start` (which would reposition our text to the right of the box), we need to assign it a value of `flex-end` explicitly so that the elements within (the text) stay to the right. The following listing shows the CSS we use to style *CVV* and the security code.

Listing 9.11 Layout for the back of the card

```
.card-item__cvvTitle {
  padding-right: 10px;
  margin-bottom: 5px;
}
.card-item__cvvBand {
  height: 45px;
  margin-bottom: 30px;
  padding-right: 10px;
  display: flex;
  align-items: center;
  justify-content: flex-end;
```

```
    Background: rgb(255, 255, 255);
  }
```

At this point, our card looks like figure 9.14.

Figure 9.14 Elements positioned on the card

The card is starting to take shape. Now we need to apply the background image to both the front and back, as well as the colors and typography. These steps will make a huge difference and get us one step closer to the final look.

.3 Working with background images

Our credit card needs to have some sort of background image. To add one, we'll use the `background-image` property. The image could be in any format that's valid for the web.

9.3.1 Background property shorthand

When setting the background for an element, we can set each related property independently

(`background-image` , `background-size` , and so on) or can use the shorthand `background` property. We're going to use the following properties and values:

- `background-image: url("bg.jpeg")`
- `background-size: cover`
- `background-color: blue`
- `background-position: left top`

If we use the shorthand `background` property, our declaration ends up being `background: url("bg.jpeg") left top / cover blue;`. The URL to the image is truncated here to make the code easier to read and discuss, but it'll be required in its entirety in our code to retrieve the image, as we'll do several times in this chapter. Figure 9.15 breaks down the property value.

Figure 9.15 Shorthand `background` property

Notice that we're using a `background-size` property value of `cover` . We're using this setting so that the browser will calcu-

late the optimal size the image should be to cover the entire element while still showing as much of the image as possible without distortion. If the image and our element don't have the same aspect ratio, the excess image will be clipped. If we don't want any part of the image to be clipped, we can use `contain` instead. Figure 9.16 shows examples of using `cover` and `contain`.

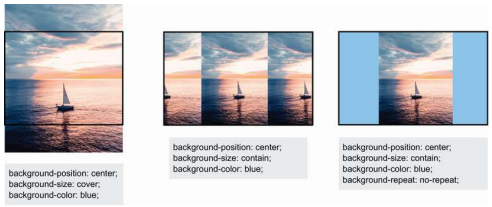

Figure 9.16 Examples of `background-cover`

Although we use a `background-size` of `cover`, we still include a background color. When both an image and background color are provided, the image always appears on top of the color. We may want to do this for multiple reasons. If the image were smaller than the element or transparent, for example, including a background color would provide a uniformly colored background behind the image. It would also provide something for the browser to display while the image is load-

ing or if the image fails to load. We don't have to provide this value in our project, but having a color that differs from the page's background helps distinguish the card from the page itself, making it a good fallback position if the image fails to load. Because we want the front and back of the card to have the background image, we'll update our `.card-item__side` rule, which affects both the front and back of the card, as shown in the following listing.

Listing 9.12 Background image for the front and back of the card

```
.card-item__side{
  height: 100%;
  width: 100%;
  background: url("bg.jpeg") left top / cover blue;
}
```

With the background image applied (figure 9.17), we can focus on styling the text.

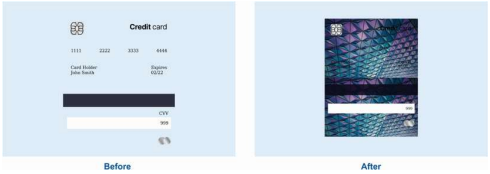

Figure 9.17 Background image added to both the front and back of the card

9.3.2 Text color

Now that we have the background image in place, we notice that the text is difficult to read, so we'll change it from black to white by updating our `.card-item` selector. Listing 9.13 shows our updated `.card-item` rule.

Color contrast and background images

Verifying that color contrast is accessible when text overlaps an image is notoriously difficult and requires manual testing. In many cases as the window is resized, the content reflows, and where the text overlaps, the image changes. One technique to ensure that contrast is always sufficient is to test the text color against both the lightest and darkest points of the image.

Also worth mentioning, and as clearly demonstrated in this project, the busier the image is, the more difficult achieving good readability becomes.

Listing 9.13 Setting the container color

```
.card-item {
  max-width: 430px;
```

```
    height: 270px;
    margin: auto;
    position: relative;
    color: white;
  }
```

By updating this rule, we've made all text on the card white (figure 9.18). Our security code is on a white background, however, so we need to update its rule to change its text color to something darker.

Figure 9.18 Text color changed to white

To change the color of the text, we'll update the `.card-item__cvvBand` rule (listing 9.14), which currently gives us the white band and positions the security code within. We'll change the text color to a dark blue-gray.

Listing 9.14 Back-of-card white background

```
  .card-item__cvvBand {
    background: white;
    height: 45px;
```

```
    margin-bottom: 30px;
    padding-right: 10px;
    display: flex;
    align-items: center;
    justify-content: flex-end;
    color: rgb(26, 59, 93);
  }
```

With the visibility of our security code restored (figure 9.19), let's turn our attention to the two text elements on the front of the card: *Card Holder* and *Expires*.

Figure 9.19 Restored security code

In terms of information, these two pieces of text are there only to label the elements with which they're paired, so they're less important than the actual name and date. To diminish their importance visually, we'll decrease their opacity (listing 9.15) to render them mildly translucent and decrease their brightness. In section 9.4, when we handle the typography, we'll diminish their size for the same reason.

Listing 9.15 Styling the labeling text

```
.card-item__holder, .card-item__dateTitle {
  opacity: 0.7;
}
```

At this point, the final appearance of the card is coming through (figure 9.20). We've styled the layout, format, images, and colors. But we still need to adjust the typography and create the main effect: the flip on hover. The next step is looking at the fonts.

Figure 9.20 Diminished text opacity

.4 Typography

For other projects, we used the free online resource Google Fonts to load the fonts we needed. We did this by linking to the Google Fonts application programming interface (API), requesting the fonts we needed, and then setting the property value to the font family we're using. But in some cases, we may want to load our font files ourselves rather than depend on an API or a content distribution network (CDN).

WARNING Like images and other forms of media, fonts are

subject to licensing. Always make sure that you have the appropriate licenses, regardless of how a font is being imported (API, CDN, or locally hosted) before using it on a website or in an application. When in doubt, ask your legal team!

Both approaches have benefits and drawbacks. Neither is overwhelmingly better than the other, so the choice comes down to the needs of the project we're working on.

The benefits of using local or self-hosted fonts include

- We don't have to depend on a third party.
- We have more control in terms of cross-browser support and performance optimization, which can make the font load time faster than that of a third-party font.

Drawbacks include

- We have to do our own performance optimization.
- The user won't already have the font cached.

The advantages of using fonts hosted by a third party include

- The user may already have the font cached on their device.
- Importing is easier.

Drawbacks include

- We need to make an extra call to fetch the font file.
- There are privacy concerns about what the third party is tracking.
- The service can discontinue the font at any time.

To load our own fonts from our local project folder, we need to create `@font {}` at-rules to define and import the fonts we want to use. To understand this at-rule, let's start by looking at font formats.

9.4.1 @font-face

Fonts can come in a few file types. Some well-known ones are

- *TrueType (TTF)*—Supported by all modern browsers; not compressed
- *Open Type (OTF)*—Evolution of TTF; allows for more characters such as small caps and old-style figures

- *Embedded Open Type (EOT)*—Developed by Microsoft for the web; supported only by Internet Explorer (obsolete because Internet Explorer has been end-of-lifed)
- *Web Open Font Format (WOFF)*—Created for the web; is compressed; includes metadata within the font file for copyright information; and is recommended by the World Wide Web Consortium (**https://www.w3.org/TR/WOFF2**)
- *Web Open Font Format 2 (WOFF2)*—Continuation of WOFF; 30% more compressed than WOFF
- *Scalable Vector Graphic (SVG)*—Created to allow embedding glyph information in SVGs before web fonts became widespread

When you select a font type to use, we generally recommend using WOFF or WOFF2.

NOTE Only recently have we been able to rely on WOFF2 files without having to upload multiple font formats. You can still find a lot of outdated information about fonts on the web. A trick that helps is looking at when the information was pub-

lished—the more recently, the better.

When dealing with fonts, we know from previous chapters that we need to import each weight we want to use. The same is true for dealing with fonts locally: each variation (weight and style) needs to be included in the project individually unless we use a variable font.

Variable fonts are fairly new. Rather than having each style in a separate file, all the permutations are included inside a single file. So if we wanted regular, bold, and semibold, we could import only one file instead of three, and we'd have access not only to those three font weights, but also to everything from thin to extra-bold. Italics may not be in the same file; in some typefaces, the italic glyphs are different from those of the non-italic versions.

For our project, we want to load three fonts: Open Sans normal, Open Sans bold, and Open Sans Italic. These fonts are variations within the same family. Open Sans has both static and variable font versions. The variable version separates italic and reg-

ular styles into two separate files. For our non-italic needs, because we're loading multiple weights, we'll use the variable version.

For italic, however, we're going to use only one weight: regular. It doesn't make sense to load the variable font version for that weight. Because the variable font includes all the information necessary to cross the full gamut of weights, it's significantly larger (314.8 KB) than the file that holds only one weight (17.8 KB). For performance reasons, it makes sense to stick with the static version.

For each font, we need to create a separate `@font-face` rule. This at-rule defines the font and includes where the font is being loaded from, what its weight is, and how we want it to load.

First, we declare the `@font-face { }` rule. Inside the curly braces, we'll define its characteristics and behavior, including four descriptors:

- `font-family` —The name we use to refer to our font when we apply it to an element via the `font-family` property.

- `src` —Where the font is being loaded from. This descriptor takes a comma-delimited list of locations to fetch the font from and what format to expect from each source. The browser will go down the list, starting with the first one, until it fetches the font successfully.
- `font-weight` —What weight this particular font file represents. In the case of variable fonts, we'll include a range.
- `font-display` —Dictates how the font is loaded. We'll use the descriptor value `swap` . Fonts are load-blocking, in that the browser will wait until they're loaded before moving on to load other resources. `swap` limits the amount of time the font is allowed to be load-blocking. If the font isn't done loading when that period is over, the browser will move on to load other resources and finish applying the font whenever the font is done loading. This setting allows content to be shown and the user to interact with the interface even if the font is not available yet.

Listing 9.16 shows both of our rules, which must be added *at the top of the stylesheet.* Also, with a few exceptions, a rule can't be declared inside an existing rule. `.myClass { @font-face { ... } }` wouldn't work, for example. One exception is the `@supports` at-rule, which we expand on in the next section.

Listing 9.16 Declaring our fonts

```
@font-face {
  font-family: "Open Sans";                                          ①
  src: url("./fonts/open-sans-variable.woff2")                       ②
       format("woff2-variations");                                   ②
  font-style: normal;
  font-weight: 100 800;                                              ③
  font-display: swap;
}

@font-face {
  font-family: "Open Sans";                                          ①
  src: local("Open Sans Italic"),                                    ④
       url("./fonts/open-sans-regular.woff2") format("woff2"),       ⑤
       url("./fonts/open-sans-regular.woff") format("woff");         ⑥
  font-style: italic;
  font-weight: normal;                                               ⑦
  font-display: swap;
}
```

① The name we'll use to refer to the font

② If the browser can load the variable font, where to get the font from

③ The font will support any font size from 100 to 800.

④ Checks the device to see whether it has the font loaded locally

⑤ Tries to load woff2 format

⑥ If woff2 isn't supported, loads woff

⑦ Declaring that the font weight for this file is normal (same as 400)

After this code is applied, there's no change in the user interface; the `font-family` being used is still the browser's default because we haven't applied the fonts to any of our elements yet. We also want to create a fallback in case the browser doesn't support variable fonts. Before we apply the font to our elements, let's look at browser support.

9.4.2 Creating fallbacks using @supports

Because variable fonts are fairly new, and because not everyone is good at running updates on their devices, we'll include a fallback in case variable fonts aren't supported by a user's browser. For this pur-

pose, we'll use the `@supports` at-rule. This rule allows us to check whether the browser supports a particular property and value, and allows us to write CSS that gets applied only if the provided condition is met.

Our feature query will be `@supports not (font-variation-settings: normal) { ... }`. Because our query has the keyword `not` before the condition, the styles it contains will be applied when the condition is *not* being met. In other words, if the browser doesn't support variable font behaviors, we want to load the static version.

Inside the `@supports` at-rule, which we place at the top of our file, we include the `@font-face` rules for both weights of the normal style version we want to include (listing 9.17). We also create an `@supports (font-variation-settings: normal) { }` rule, this time without the `not`. In this second at-rule for browsers that do support variable fonts, we move the two rules we created in section 9.4.1. This way, we load the variable fonts only if they're supported by the browser and prevent the file from being loaded

if the browser doesn't support variable fonts.

Listing 9.17 Fallback for browsers that don't support variable fonts

```
@supports (font-variation-settings: normal) {          ①
  @font-face {                                         ②
    font-family: "Open Sans";                          ②
    src: url("./fonts/open-sans-variable.woff2")       ②
➥ format("woff2-variations");                         ②
    font-weight: 100 800;                              ②
    font-style: normal;                                ②
    font-display: swap;                                ②
  }                                                    ②
}

@supports not (font-variation-settings: normal) {      ③
  @font-face {                                         ④
    font-family: "Open Sans";
    src: local("Open Sans Regular"),
         local("OpenSans-Regular"),
         url("./fonts/open-sans-regular.woff2") format("woff2"),
         url("./fonts/open-sans-regular.woff") format("woff");
    font-weight: normal;
    font-display: swap;
  }

  @font-face {                                         ⑤
    font-family: "Open Sans";
    src: local("Open SansBold"),
         local("OpenSans-Bold"),
         url("./fonts/open-sans-regular.woff2") format("woff2"),
         url("./fonts/open-sans-regular.woff") format("woff");
    font-weight: bold;
    font-display: swap;
  }
}
```

① Applies styles when variable fonts are supported

② Our previously created rule for the variable font, moved into the at-rule

③ Applies styles when variable fonts aren't supported

④ Rule for normal style, font weight regular (400)

⑤ Rule for normal style, font weight bold (700)

With our fallback added, let's update our body rule to apply Open Sans to our project (listing 9.18). Although we added fallbacks for loading the font, we'll still include `sans-serif` in the `font-family` property value in the body rule in case our font files fail to load.

Listing 9.18 Applying the fonts to our project

```
body {
  background: rgb(221, 238, 252);
  margin-top: 80px;
  font-family: "Open Sans", sans-serif;
}
```

When the font is applied, we see that our text has been updated to use Open Sans rather than the browser default (figure 9.21). Now we can edit our indi-

vidual elements for font weight and style.

Figure 9.21 Open Sans applied to the project

9.4.3 Font sizing and typography improvements

Starting with the front of the card, we'll increase the font size of the numbers and make them bold. We'll add to our existing rule, as shown in the following listing.

Listing 9.19 Boldfacing and increasing the size of the numbers

```
.card-item__number {
  display: flex;
  justify-content: space-between;
  padding: 10px 15px;
  margin-bottom: 35px;
  font-size: 27px;
  font-weight: 700;
}
```

Figure 9.22 shows our styled numbers.

Figure 9.22 Styled numbers

Moving on to the text below the numbers, we want to decrease the size of *Card Holder* and *Expires*. We'll set their `font-size` to `15px` and increase the size and `font-weight` of the name and date, as shown in the following listing.

Listing 9.20 Cardholder information and expiration-date typography

```
.card-item__holder, .card-item__dateTitle {    ①
  opacity: 0.7;                                ①
  font-size: 15px;                             ①
}                                              ①

.card-item__name, .card-item__dateItem {       ②
  font-size: 18px;                             ②
  font-weight: 600;                            ②
}                                              ②
```

① Card Holder and Expires

② Name and expiration date

With the text elements on the front of the card taken care of (figure 9.23), let's turn our attention to the back.

Figure 9.23 Typography of the front of the card

On the back, we need to update the security code to be in italics. We'll update our existing rule with `font-style: italic`, as shown in the following listing.

Listing 9.21 Italicizing the card security number

```
.card-item__cvvBand {
  background: white;
  height: 45px;
  margin-bottom: 30px;
  padding-right: 10px;
  display: flex;
  align-items: center;
  justify-content: flex-end;
  color: #1a3b5d;
  font-style: italic;
}
```

Now that our card is styled (figure 9.24), we're ready to apply the flip effect.

Figure 9.24 Completed typography styles

.5 Creating the flipping-over effect

Next, we'll create the flipping-over effect for devices that support the `hover` interaction. We'll start by adjusting the position to overlay the back of the card on top of the front. Then we'll use the `backface-visibility` and `transform` properties to place the card. To animate the change, we'll use a transition.

9.5.1 Position

To achieve the flip effect, we stack the card faces on top of one another via the `backface-visibility` property. Then we'll toggle which side is shown. When we use the `backface-visibility` property and expose the back side, we perform a rotation on the horizontal axis; therefore, we need to invert the back so that its contents are mirrored. Imagine taking a piece of tracing paper and drawing an image on the back. When we look at the front, the image that appears through it from the back is mirrored. That effect is what we're building here. The CSS we use to stack front and back and then flip the back is in listing 9.22. We place

our code inside a media query that checks whether the browser has `hover` functionality. We want to have the flip effect only on devices that support `hover`. For devices that don't (such as mobile phones), we'll show the front and the back at the same time.

Listing 9.22 Positioning the back over the front

```
@media (hover: hover) {
  .back {
    position: absolute;
    top: 0;
    left: 0;
    transform: rotateY(-180deg);      ①
  }
}
```

① Flips the card

Earlier in this chapter, we set the `position` property value to `relative` in our `.card-item` rule. Using relative positioning on a parent or ancestor element goes hand in hand with the fact that we're setting the `position` property value of the back of our card to `absolute`. The top and left positions of `0` will be the top-left section with the `card-item` class (the container that holds the two card faces).

Whenever we use `position: absolute`, we take the element out of the regular flow of the page and can set a specific position on the page on which to place the element. The position is calculated based on the closest ancestor with a `position` value of `relative`. If none is found, the top left will be the top-left corner of the page.

What gets a bit confusing here is that if no values are set to position the element (`top`, `left`, `right`, `bottom`, or `inset`), the element is placed wherever it normally would lie but takes up no space in the flow. The height and width of the element are also affected. If a value is provided in the CSS, the element maintains that value; otherwise, it takes up only as much room as it needs. Even if it's a block-level element, it no longer takes up the full width available to it. Furthermore, if the width is set using a relative unit such as percentage, it will be calculated against the element to which it's relative. Figure 9.25 shows some scenarios for using `position: absolute`.

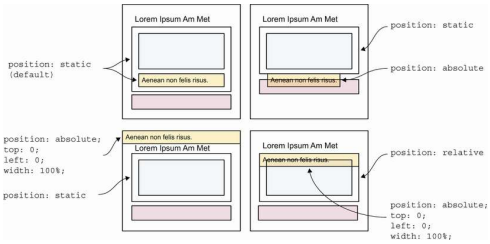

Figure 9.25 Absolute positioning

With our CSS applied (figure 9.26) and the back of the card flipped and on top of the front, we can apply the `backface-visibility` property.

Figure 9.26 Back of the card positioned on top of front and flipped

9.5.2 Transitions and backface-visibility

Up to now, we've looked at objects in 2D space—in other words, a flat perspective. We've looked at width and height but not depth. Now we'll consider that third dimension.

With the back flipped, we need it to be hidden unless the user is hovering over the card. We have two sides, the second of which has a `transform: rotateY(-180deg)` declaration (the back). In a 3D space, there-

fore, that side is facing away from us. If we set the `backface-visibility` property value to be `hidden` on both sides, whichever side is facing away from us is hidden.

Our back, which currently faces away from us, is hidden. If we rotate the entire card, the back faces us, and the front is hidden. Figure 9.27 diagrams how our CSS and HTML interact to create the flip effect.

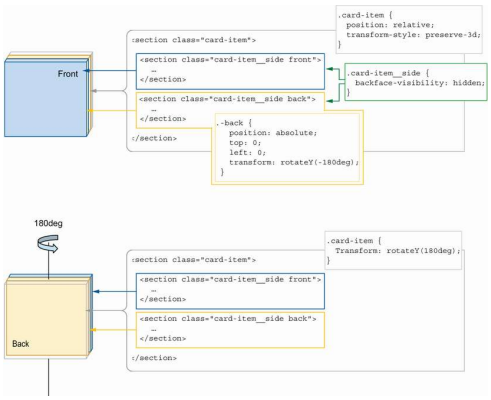

Figure 9.27 The `backface-visibility` property applied to our use case

In our CSS, we add the following rules and properties to our media query (listing 9.23). They instruct the card to hide the side if it's facing away from us and to rotate the entire card 180-degrees on the y-axis on hover. Notice a property that we haven't talked about yet: `transform-style`, to which we've given a value of `preserve-3d`. Without this prop-

erty, the flip won't work. It tells the browser that we're operating in 3D space rather than 2D space, establishing the concept of a front and a back.

Listing 9.23 Hiding the back and exposing it on `hover`

```
@media (hover: hover) {
  ...
  .card-item {
    transform-style: preserve-3d;      ①
  }
  .card-item__side {
    backface-visibility: hidden;       ②
  }
  .card-item:hover {                   ③
    transform: rotateY(180deg);        ③
  }                                    ③
}
```

① Instructs the browser to operate as though we were in 3D space

② Hides the side facing away from us

③ On hover, flips the entire card around to expose the back side

With our hover functionality exposing the back of our card (figure 9.28), we need to add the animation to make it look more like a card flip. Notice that the back is no longer mirrored.

Figure 9.28 Card default state and on `hover`

Currently, when we hover over the card, the back is shown instantaneously. We want to make it look as though the card is actually being flipped.

9.5.3 The transition property

To animate the card flip, we'll use a transition. You may recall from chapter 5 that transitions are used to animate the change of CSS. In this case, we'll animate the change in the rotation of the card by adding a transition declaration to the `card-item` (container that holds the two faces). We'll also add a condition to our media query.

Because this animation is motion-heavy, we want to make sure to respect our users' settings. Therefore, we'll add a `prefers-reduced-motion: no-preference` condition to our media query, as shown in the following listing.

Listing 9.24 Transitions and `transform`

```
@media (hover: hover) and (prefers-reduced-motion: no-preference) {
  ...
  .card-item {
    transform-style: preserve-3d;
    transition: transform 350ms cubic-bezier(0.71, 0.03, 0.56, 0.85);
  }
  ...
}
```

Our animation, which takes 350 milliseconds, affects the `transform` property (the rotation) and is present only for users who don't have `prefers-reduced-motion` set to `reduce` on their devices. Figure 9.29 shows the progression of the animation, and figure 9.30 shows the user interface when the user has `prefers-reduced-motion` enabled.

Figure 9.29 Animation over time

For our timing function, we used a `cubic-bezier()` function. Next, let's take a closer look at what this function represents.

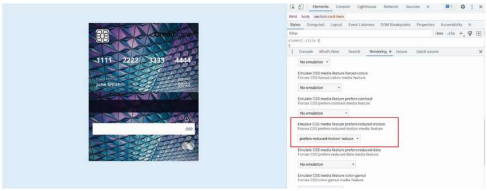

Figure 9.30 `prefers-reduced-motion: reduce` emulation in Chrome DevTools

9.5.4 The cubic-bezier() function

The Bézier curve is named after French engineer Pierre Bézier, who used these curves on the bodywork of Renault cars (**http://mng.bz/d1NX**). A Bézier curve is composed of four points: P_0, P_1, P_2, and P_3. P_0 and P_3 represent the starting and ending points, and P_1 and P_2 are the handles on the points. Point and handle values are set with x and y coordinates (figure 9.31).

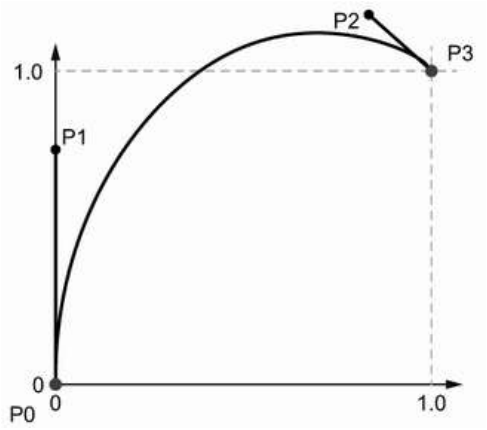

Figure 9.31 Points and handles on the Bézier curve

In CSS, we need to worry about only the handles because the P_0 and P_3 values are set for us to

`(0, 0)` and `(1, 1)`, respectively. By manipulating the curve, we change the acceleration of the animation. In CSS, our function takes four parameters that represent the `x` and `y` values of P_1 and P_2: `cubic-bezier(x1, y1, x2, y2)`, where the `x` values must remain between `0` and `1`, inclusive.

The premade timing functions we used in previous chapters for both our transitions and our animations have `cubic-bezier()` values by which they can be represented (figure 9.32).[1]

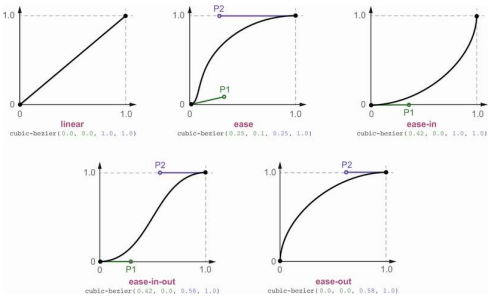

Figure 9.32 Predefined curves

Writing our own `cubic-bezier()` functions to animate our designs can be tedious. Luckily, online tools such as **https://cubic-bezier.com** allow us to see the curve and determine the values (figure 9.33).

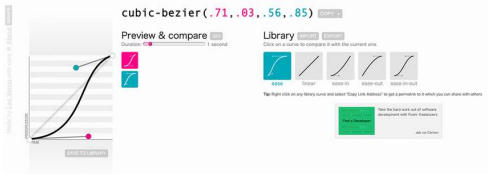

Figure 9.33 An example `cubic-bezier()` function from cubic-bezier.com

We can also see the `cubic-bezier()` in some browser developer tools, such as those of Mozilla Firefox (figure 9.34).

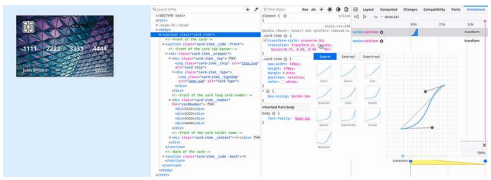

Figure 9.34 Firefox DevTools curve details

With our animation completed, let's add some finishing touches to our project.

.6 Border radius

Most credit cards have rounded corners, so we're going to round ours as well. We'll also round the corners of the white CVV box on the back of the card.

Adding rounded corners to a user interface can be a balancing act. We'll add rounded corners to the card to make it look more natural and realistic. Sharp corners can come across as aggressive, but overuse of rounded corners can make an

interface look too soft and playful, which may not work in all cases. The "correct" amount of curve is design-specific. To make our card look more realistic, we'll add the following CSS.

Listing 9.25 Adding `border-radius`

```
.card-item__side {          ①
  height: 100%;
  width: 100%;
  background: url("bg.jpeg") left top / cover blue;
  border-radius: 15px;
 }
.card-item__cvvBand {       ②
  background: white;
  height: 45px;
  margin-bottom: 30px;
  padding-right: 10px;
  display: flex;
  align-items: center;
  justify-content: flex-end;
  color: #1a3b5d;
  font-style: italic;
  border-radius: 4px;
}
```

① The card

② White CVV band

With the rounded corners, our card looks like figure 9.35.

Figure 9.35 Rounded corners on the card and CVV band

.7 Box and text shadows

In chapter 4, we looked briefly at the `drop-shadow` value, which can be applied to the `filter` property for image filters. Another way to apply a shadow to an element is via the `box-shadow` property, which applies a shadow to the element box.

9.7.1 The drop-shadow function versus the box-shadow property

We may be wondering about the difference between the `drop-shadow` filter property and the `box-shadow` property. Both have the same base set of values, but the `box-shadow` property has an additional two nonmandatory values: `spread-radius` and `inset`.

The benefit of using a filter with the `drop-shadow` property on images is that when we're using a filter, the shadow is applied to the alpha mask rather than the bounding box. So if we have a

PNG or SVG image, and that image has transparent areas, the shadow is applied around that transparency. If we add a `box-shadow` to the same image rather than the filter, the shadow is applied only to the outer image container (figure 9.36).

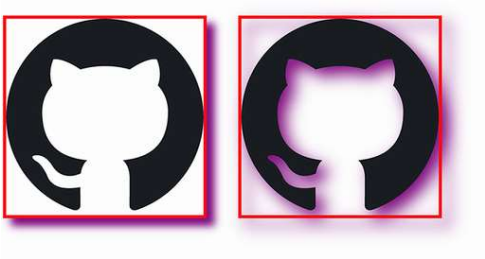

Figure 9.36 Comparing `box-shadow` (left) and `drop-shadow` (right)

To reinforce the 3D effect on the card and make the card appear to be floating, we're going to give our card a shadow. Because we're concerned only about giving the bounding area of the card a shadow, we can use the `box-shadow` property, which will give the project a sense of depth and further emphasize that something is on the back. The shadow will be large, soft, and fairly transparent. To achieve that effect, we'll add `box-shadow: 0 20px 60px 0 rgb(14 42 90 / 0.55);` to our `.card-item__side` rule. Our updated rule looks like the following listing.

Listing 9.26 Using `box-shadow` on our card

```
.card-item__side {
  height: 100%;
  width: 100%;
  background: url("bg.jpeg") left top / cover blue;
  border-radius: 15px;
  box-shadow: 0 20px 60px 0 rgb(14 42 90 / 0.55);
}
```

Figure 9.37 shows our updated card.

Figure 9.37 Added shadow to make the card appear to be floating

9.7.2 Text shadows

We can also add shadows to text. If we applied a `box-shadow` to text, the shadow would be applied to the box containing the text, not to the individual letters. To add a shadow to the letters, we use the `text-shadow` property, which has the same syntax as the `box-shadow` property. We'll use this property on the front of the card to lift the text from the background. We need to add this property to our `.front`

rule, as shown in the following listing.

Listing 9.27 Text shadow for all the text elements on the front of the card

```
.front{
  padding: 25px 15px;
  text-shadow: 7px 6px 10px rgb(14 42 90 / 0.8);
}
```

Figure 9.38 shows the card before and after.

Figure 9.38 Before and after adding the `text-shadow`

Although the effect is subtle, the added shadow makes the numbers pop out a bit. It's worth noting that this effect is best used with finesse and sparingly, as it can easily impede readability rather than help it.

.8 Wrapping up

The last detail we need to handle deals with users who aren't interacting with the flip effect but are viewing both sides of the card at the same time (devices that don't have hover capabilities, such as phones and

tablets, and users with a `prefers-reduced-motion` setting). Currently when both sides are displayed, there's no space between the card faces. So let's add some margin to the bottom of the faces to separate them, as shown in the following listing.

Listing 9.28 Separating the card faces

```
.card-item__side {
  height: 100%;
  width: 100%;
  background: url("bg.jpeg") left top / cover blue;      ①
  border-radius: 15px;
  box-shadow: 0 20px 60px 0 rgb(14 42 90 / 0.55);
  margin-bottom: 2rem;
 }
```

① URL truncated for legibility

On a Moto G4 device, our card looks like figure 9.39.

Figure 9.39 Our project on a mobile device

With this last addition, our project is complete. Using a combination of media queries, shadows, positioning, and transitions, we created a realistic-looking card (figure 9.40).

Figure 9.40 Finished project

ummary

- We can alter the box model's behavior through the `box-sizing` property.
- The `background` property value `cover` allows us to show as much of a background image as possible while still covering the full element.
- Although fonts come in a range of formats, for the web we need only the WOFF and WOFF2 formats.
- Fonts can be static or variable.
- We use the `@font-face` at-rule to define where and how fonts are imported and how they should behave.
- The `@font-face` at-rule needs to be at the top of the stylesheet.

- The `@supports` at-rule allows us to create styles specific to a browser's functionality.
- The `backface-visibility` property used in conjunction with `transform-style: preserve-3d` creates a flip effect.
- The `cubic-bezier()` function defines how our elements will animate over time.
- The `box-shadow` property allows us to add a shadow to an element's box.
- `text-shadow` rather than `box-shadow` is the property we use to add a shadow to individual letters of text.

---

[1] *Architecting CSS: The Programmer's Guide to Effective Style Sheets,* by Martine Dowden and Michael Dowden (2020, Apress).