5 Summary cards with hover interactions

This chapter covers

- Clipping static background images using the `background-clip` property
- Using transitions to reveal content on hover
- Using media queries to choose styles based on device capabilities and window size

Summary cards are used for a range of purposes, whether that be showing a preview for a film, buying a property, previewing a news article, or (in this chapter) showing a list of hotels. Usually, a summary card contains a title, description, and a call to action; sometimes, it also contains an image. Figure 5.1 shows the cards we'll create in this project.

Figure 5.1 Finished product

The cards will be placed in a single line, using the CSS Grid Layout Module for layout. Each card will have its own background image, with the content placed on top. If the user is viewing the card on a device that supports hover and has a screen at least 700 pixels wide, they'll be able to see the title and then hover over the card, which will reveal the short description and an orange call-to-action button for contrast with the black background (figure 5.2).

Figure 5.2 Hover effect on finished product

For users whose devices don't support hover or have a screen less than 700 pixels wide, we'll show all the information without hover so that

the user experience isn't affected (figure 5.3).

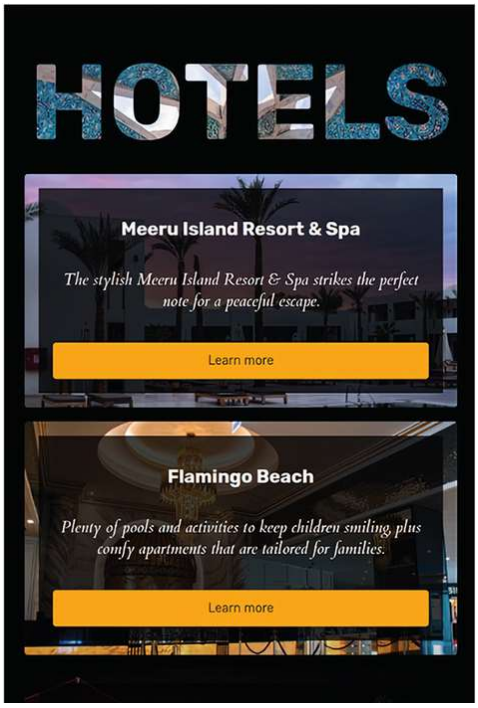

Figure 5.3 Finished product on small or touch devices that can't handle the hover state

The other piece of this project is the header, which we want to make stand out and have some visual interest. To do this, we'll explore the `background-clip` property and see how we can clip an image around the text.

.1 Getting started

Listing 5.1 and listing 5.2 include our starting CSS and HTML for the page that we'll build on in this chapter. To follow along as we style the page, you can download the starting

HTML and CSS from the GitHub repository at **http://mng.bz/KlaO** or from CodePen at **https://codepen.io/michaelgearon/pen/vYpaQPO**.

The mobile and desktop experiences will use the same HTML and stylesheet. Similarly to what we did in chapter 4, we'll use media queries to alter the styles based on browser size and capabilities.

Listing 5.1 shows our starting HTML. Each card is wrapped in a `<section>` element and includes its title (`<h2>`), description (`<p>`), and call to action (`<a>`).

Listing 5.1 Starting HTML

```
<header>
  <h1>Hotels</h1>                                    ①
</header>
<main>
  <section class="flamingo-beech">                   ②
    <div>
      <h2>Meeru Island Resort & Spa</h2>             ③
      <p>The stylish Meeru Island ...</p>            ④
      <a href="#">Learn more</a>                     ⑤
    </div>
  </section>                                         ⑥
  ...

</main>
```

① Page title

② Start of the first summary card

③ Card title

④ Card description (shown only on hover when browser allows)

⑤ Card call to action

⑥ End of first summary card

Our starting CSS (listing 5.2) includes some base styles to set up our page. For the body, we're increasing the margin by 40 pixels and adding padding of 20 pixels to all four sides. We're using Google Fonts—this time the font family Cardo, regular weight, italicized version—for the description of each card. For the headers, we'll use Rubik in both regular and bold weights. This font is a good choice because it combines good readability with rounded edges, providing a sense of informality that works well with the Cardo font. Notice that when we're loading multiple Google Fonts, we can combine the imports into one request.

Listing 5.2 Starting CSS

```
@import url("https:/ /fonts.googleapis.com/css?
➥ family=Cardo:400i|Rubik:400,700&display=swap");      ①

body {
  margin-top: 40px;
  padding: 20px;
}
```

① One request to load both the Cardo and Rubik fonts

As we begin styling our project, our page looks like figure 5.4.

Figure 5.4 Starting point

.2 Laying out the page using grid

A good starting point is reviewing the layout of our cards and the web page as a whole. We need to consider three aspects of our layout:

- The header and main content

- The container for the cards
- The content within the cards

We'll use the CSS Grid Layout Module for layout in all three use cases.

NOTE The CSS Grid Layout Module allows us to place and align elements across both the vertical and horizontal axes in a system of columns and rows. Check out chapter 2 to find out how this module works.

To lay out the elements on our page, we'll start by creating the styles for narrow screens and edit the layout as we build up to larger screen sizes by using media queries.

5.2.1 Layout using grid

Our layout consists of two landmarks: `<header>` and `<main>`, which are immediate children of `<body>` (listing 5.3). By giving the `<body>` a `display` property with a value of `grid`, we'll be affecting the position of the `<header>` and `<main>` elements.

Listing 5.3 Starting HTML

```
<body>
  <header>    <!-- title -->
  </header>
   <main>    <!-- cards -->
   </main>
</body>
```

Next, we use the `place-items` property to center the elements on the page. This property is a shorthand way to combine declaring values for the `align-items` and `justify-items` properties. We'll set its value to `center`, aligning all the items in the middle of their respective rows and columns. The following listing shows our updated `body` rule.

Listing 5.4 Positioning the `<header>` and `<main>` elements

```
body {
  display: grid;
  place-items: center;
  margin-top: 40px;
  padding: 20px;
}
```

Notice that we haven't defined any `grid-template-rows`, `grid-template-columns`, or `grid-template-areas`. By default, when none of these areas is declared, the browser

creates a one-column grid with as many rows as there are elements to position. In our case, we have two elements: `<main>` and `<body>`. Therefore, our grid has one column and two rows (figure 5.5).

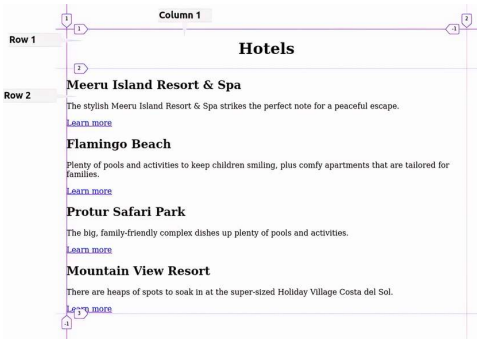

Figure 5.5 One-by-two grid

The widths of the `<header>` and `<main>` are altered by being within the grid to take only as much horizontal space as their content requires. Because the `<header>` has narrow content (the `<h1>` containing the word *hotel*), the page title centers itself on the page. The `<main>` element takes the full width available to it because the description of Flamingo Beach (in the second card) needs the full width and even wraps. If we extend the width of the screen further, we see that the `<main>` element also centers itself (figure 5.6).

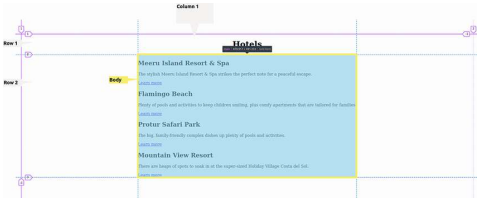

Figure 5.6 Centered `main` on a wide screen

We'll also rely on the default functionality of the grid and omit defining rows and columns because we want to keep the cards stacked on narrow screens. To add space between cards, we include a gap of `1rem`. We also restrict the width of the `<main>` element to a maximum 1024 pixels to prevent our cards being too spaced out on wide screens after we align them horizontally on wide screens (section 5.2.2). Our updated CSS, shown in the following listing, keeps the cards stacked but adds a 1-rem gap between cards (figure 5.7).

Figure 5.7 Grid applied to `<main>`

Listing 5.5 Positioning the cards on narrow screens

```
main {
  display: grid;
  max-width: 1024px;
  grid-gap: 1rem;
}
```

5.2.2 Media queries

At the moment, our cards are stacked vertically—the default behavior in most cases with HTML elements. This layout makes sense on mobile devices, which have rather narrow screens. For desktop screens, however, because the browser window can be much wider, we can take advantage of the horizontal space by using media queries. We can define some media queries to adjust the layout:

- If the window width is greater than or equal to 700 pixels, we adjust the grid to have two equal-size columns and set the height of each section to exactly 350 pixels.

- At 950 pixels, we adjust the layout again to have four equal-size columns overriding the `grid-template-columns` value set in the preceding media query. The `height` property value remains 350 pixels because the condition for the preceding media query `(min-width: 700px)` is still being met.

If neither of the requirements for these media queries is met (when the browser window is less than 700 pixels wide), the cards will be stacked vertically in a single column. The following listing shows the two media queries being created.

Listing 5.6 Layout for the cards

```
@media (min-width: 700px) {      ①
  main {
    grid-template-columns: repeat(2, 1fr);
  }
  main > section {
    height: 350px;
  }
}
@media (min-width: 950px) {      ②
  main {
    grid-template-columns: repeat(4, 1fr);
  }
}
```

① Media query to determine whether the browser window is at least 700 pixels wide. If so, the styles within the query are used.

② Second media query to determine whether the browser window is at least 950 pixels wide. If so, this query overrides the preceding query and sets the grid to four columns wide.

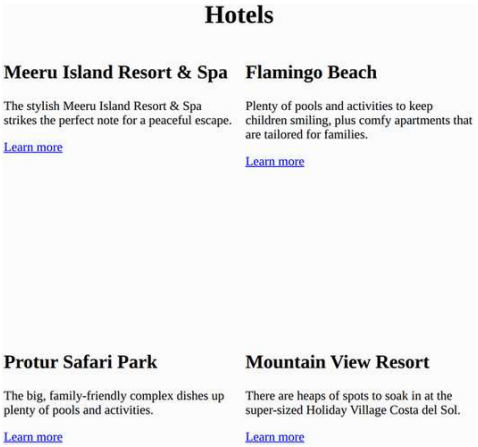

Figure 5.8 Layout on a screen 800 pixels wide

Figure 5.8 and figure 5.9 show the output in browser windows that are 800 and 1000 pixels wide, respectively.

Figure 5.9 Layout on a screen 1000 pixels wide

With our layout in hand, let's focus on styling our content, starting with the header. We're going to change the font for our `<h1>` element and look at how to use an image to color our text.

.3 Styling the header using the background-clip property

The title of this page—Hotels—could be more interesting visually. One way to liven it up could be to set a nice vibrant color and update the font family to something modern. Another way is to apply a background image to the text. These changes are possible through two experimental properties: `background-clip` and `text-fill-color`.

Experimental properties

Some properties' browser support may be value-specific. The `background-clip` property is one of those. This property is supported in all major browsers without a vendor prefix for all its possible values except `text`, which still required a vendor prefix in

Microsoft Edge and Google Chrome when this book was written (**https://caniuse.com/?search=background-clip**).

Experimental properties should be used with care because they often have non-standard implementations. For more details about experimental properties, please refer to chapter 3.
We can reduce the risks from `background-clip: text` being an experimental property by setting a fallback color value so that if these two properties don't work, the user will see the text without the background image.

5.3.1 Setting the font

The first step is to update the `font-family`, `weight`, and `size`, as well as transform the text to `uppercase`. The following listing shows these changes.

Listing 5.7 Header typography

```
h1 {
  font: 900 120px "Rubik", sans-serif;      ①
  text-transform: uppercase;
}
```

① Shorthand font property

We used the shorthand `font` property. The first value sets the `weight`, which in this case is `heavy`. The second value is the font size (`120px`), followed by the `font-family` we want to use. If this font can't be loaded, we fall back to a `sans-serif` font.

We transformed the text to uppercase through styling rather than by writing it all in uppercase letters within the HTML. Using all uppercase characters can affect accessibility, as some screen readers may interpret all caps as an acronym and read the letters individually. If we set the text to uppercase through CSS, we're styling the text only visually; the characters can be mixed-case.

Moreover, we're in a unique position with only one page to style. In a traditional project, our styles would most likely be applied to multiple pages. By adjusting our casing in our styles, we help ensure consistency throughout our website or application.

It's also worth noting that we should use all capitals sparingly, as that format can affect the readability of the content. Now our header looks like figure 5.10.

Figure 5.10 Applied typography styles to headers

5.3.2 Using background-clip

Now we'll use an image to color the letters, essentially applying a background image to the letters themselves. The first thing we need to do is set a background image on the `<h1>` element. To ensure that the image covers the entirety of the `<h1>` element, we assign the `background-size` property a value of `cover`. This value automatically calculates the width and height the image needs to make sure that the image covers the entire element.

Next, we manipulate the image to apply only to the letters, rather than the entire `<h1>` element. This step is where the `background-clip` property

comes into play. This property defines, based on the box model, which part of the element the background should cover. In our case, we'll give it a value of `text` because we want the image to show behind the letters. This property with the value of `text` still requires a browser prefix for WebKit-based browsers (Chrome, Edge, and Opera), so we also include the prefixed property for compatibility with those browsers.

Currently, our text is black, preventing the image from showing through. We must make the letters transparent so as not to obscure the image we set as our text background. The `text-fill-color` property allows us to set the color of the text. This property is similar to `color`, but if both properties are set, `text-fill-color` supersedes `color`. Because `text-fill-color` also requires a vendor prefix (for both WebKit- and Mozilla-based browsers), we can use the `color` property as a fallback in case the image doesn't load or any of the experimental properties fails.

We're using `text-fill-color` instead of using the `color` property with a value of `transparent` because we'll use the `color` to create a fall-back in case `background-clip` doesn't work in a user's browser. We set its value to `white` because we'll add a black background to our page later in this chapter. That way, if `background-clip` fails or isn't supported, our text will still be visible to the user; it will be white instead of having the image coloring it. The following listing shows our updated header class.

Listing 5.8 `background-clip` text code

```
h1 {
  text-transform: uppercase;
  font: 900 120px "Rubik", sans-serif;
  background: url(background: url("bg-img.jpg");    ①
  background-size: cover;
  -webkit-background-clip: text;                    ②
  background-clip: text;                            ②
  -moz-text-fill-color: transparent;                ③
  -webkit-text-fill-color: transparent;             ③
  color: white;                                     ④
}
```

① Adds the background image

② Clips the background to be applied only behind the text

③ Makes the text transparent to allow the image to show through

④ Fallback color

When using prefixes, we add the `-moz-` and `-webkit-` properties before the non-prefixed version if an nonprefixed version is available. This allows the browser to make sure it's using the nonexperimental version when it becomes available.

With our header styled (figure 5.11), the next task is styling the cards. We'll focus on styling the cards without the hover effect first and then create our media query for handling cards on wide screens that support hover.

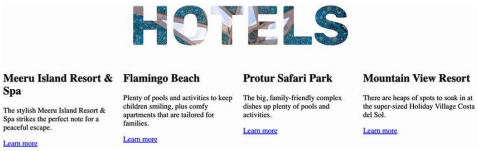

Figure 5.11 Background image clipped to the heading

.4 Styling the cards

Each card is created with an outer `<section>` element that has a background image and an inner `<div>`, which we'll

give a background color to keep our text legible over the image. Within that `<div>` is the actual content. The following listing shows our card structure in isolation from the rest of the HTML.

Listing 5.9 Card HTML in isolation

```
<section class="meeru-island">                  ①
  <div>                                         ②
    <h2>Meeru Island Resort & Spa</h2>          ③
    <p>The stylish Meeru Island Resort...</p>   ③
    <a href="#">Learn more</a>                  ③
  </div>
</section>
```

① Outer card container. Each section has a class name based on the hotel it describes.

② Content container

③ Content

To style each part of the card, we'll work from the outside in, styling the container for each card, followed by the container for the content, and finally the content itself.

5.4.1 Outer card container

The outer container is the element that gets the background

image. Each section gets an image for its hotel or resort. We'll select each section individually by its class name. Then we'll assign each of the sections a background image, as shown in the following listing.

Listing 5.10 Adding background images

```
.meeru-island {
  background-image: url("1.jpg");
}
.flamingo-beech {
  background-image: url("2.jpg");
}
.protur-safari {
  background-image: url("3.jpg");
}
.mountain-view {
  background-image: url("4.jpg");
}
```

With the background images added (figure 5.12), let's configure some general styles that apply to all the sections.

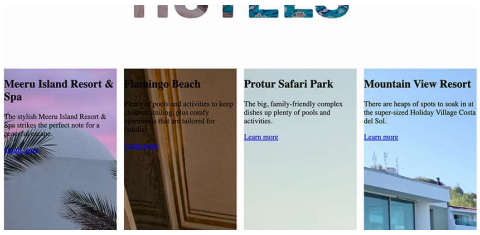

Figure 5.12 Card background pictures

We can see that the images aren't properly centered and don't showcase the hotels and resorts well. We can adjust the size of the images by using the `background-size` property. We set this property to `cover` to maximize the amount of the picture being shown without leaving any whitespace visible if the aspect ratio of the image differs from that of our card. We also add a `background-color` of `#3a8491` (turquoise) as a fallback. Finally, we add a `border-radius` to the card to curve our corners and soften our edges. Listing 5.11 shows our container styles.

Listing 5.11 Card container styles

```
main > section {
  background-size: cover;
  background-color: #3a8491;
  border-radius: 4px;
}
```

With our outer container addressed (figure 5.13), let's move on to the content container.

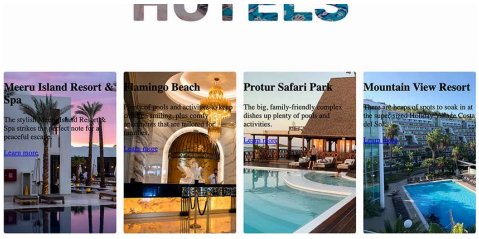

Figure 5.13 Styled outer card container

5.4.2 Inner container and content

Currently, our text isn't readable; the dark text is hard to read against the image background and also close to the edge of the outer container. To improve readability, we'll give our inner container a `background-color` of `rgba(0, 0, 0, .75)`, which is black with some transparency. We'll also change the text color to `whitesmoke` and center it. By not using pure black or pure white in our design, we achieve a softer feel for our overall composition.

With the added background color, we add `1rem` of padding with our content container to keep the text away from the edge of our dark background and `1rem` of margin to leave a gap between the edge of the picture and the beginning of the background. Finally, we adjust the `font-size`, `font-`

`weight`, `line-height`, and `font-family` of our text inside our card. The following listing shows the CSS.

Listing 5.12 Card content styles

```
main > section > div {                          ①
  background-color: rgba(0, 0, 0, .75);         ①
  margin: 1rem;                                 ①
  padding: 1rem;                                ①
  color: whitesmoke;                            ①
  text-align: center;                           ①
  font: 14px "Rubik", sans-serif;               ①
}                                               ①

section h2 {                                    ②
  font-size: 1.3rem;                            ②
  font-weight: bold;                            ②
  line-height: 1.2;                             ②
}                                               ②

section p {                                     ③
  font: italic 1.125rem "Cardo", cursive;       ③
  line-height: 1.35;                            ③
}                                               ③
```

① Card content container

② Card header

③ Card content

With our styles applied (figure 5.14), the last piece of content that needs styling is our link.

Figure 5.14 Card inner container and typography

Because our link serves as a call to action, getting users to look at more information about the hotel or resort, we want to make it bold and flashy (listing 5.13). To achieve this end, because the majority of our elements inside our cards are rather dark, we'll give the link a bright yellow-ish-orange (`#ffa600`) background and change its text color to almost black. We'll also add padding. But because a link is an inline element by default, we'll want to change its `display` property's value to `inline-block` so that the padding will affect the height of the element.

Listing 5.13 Link styles

```
a {
  background-color: #ffa600;
  color: rgba(0, 0, 0, .75);
  padding: 0.75rem 1.5rem;
  display: inline-block;
  border-radius: 4px;
  text-decoration: none;
}
```

```
a:hover {
  background-color: #e69500;
}

a:focus {
  outline: 1px dashed #e69500;
  outline-offset: 3px;
}
```

To match our cards, we'll give the links a `border-radius` of `4px` and finally handle `hover` and `focus`. Instead of underlining, which we'll remove, on `hover` we'll darken the background color slightly, and on `focus` we'll add a dashed outline offset from the link by 3 pixels. Figure 5.15 shows our styled links.

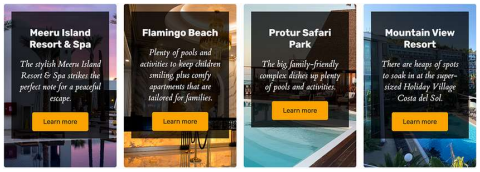

Figure 5.15 Styled links

Not having all of the links aligned horizontally is a bit odd and doesn't seem to be organized. To have all the links aligned, we'll use `grid` once again. We'll give our inner container a `display` value of `grid` and set our `grid-template-rows` value to `min-content auto min-content`, at the same time setting the

height of the inner container to `100%` minus the padding and margin we allotted to it (figure 5.16).

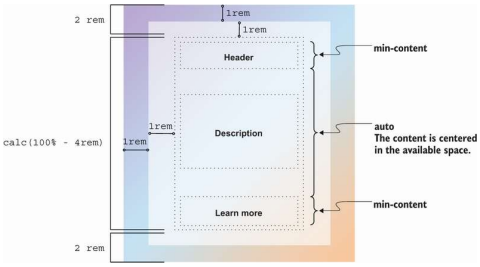

Figure 5.16 Aligning card elements horizontally

Earlier in this chapter, we gave the inner container a `margin` of `1rem` and `padding` of `1rem`, meaning that the height it needs to take up the full height of the space provided is equal to `100%` minus `4rem` (1 rem of padding and 1 rem of margin at the top and the same at the bottom, equaling 4 rems total). To achieve this effect in CSS, we use the `calc()` function to do the math for us, assigning `calc(100% - 4rem)` to the `height` property. The combination of defined rows (`grid-template-rows: min-content auto min-content`) and set height creates a layout in which the header and link take only as much room as they need and the middle sec-

tion (the paragraph element) gets what is left.

Finally, to center the paragraph content vertically in the middle of the card, we use the `align-items` property with a value of `center` and remove the bottom margin automatically added by the browser to the `<h2>`. If we left the margin at the bottom of the header, we'd have more room at the top of the paragraph than at the bottom because `min-content` takes the margin included on an element into account. Because the link at the bottom of the card has no margin, there would be a disproportionate amount of whitespace above the paragraph compared with below it. The following listing shows our layout adjustments.

Listing 5.14 Inner container layout adjustments

```
main > section > div {
  background-color: rgba(0, 0, 0, .75);
  margin: 1rem;
  padding: 1rem;
  color: whitesmoke;
  text-align: center;
  height: calc(100% - 4rem);
  display: grid;
  grid-template-rows: min-content auto min-content;
```

```
    align-items: center;
  }

  section h2 {
    font-size: 1.3rem;
    font-weight: bold;
    line-height: 1.2;
    margin-bottom: 0;
  }
```

This last adjustment finishes our card layout (figure 5.17). Next, we'll focus on showing and hiding parts of the content for devices that are wide enough (width greater than or equal to 700 pixels) and have hover capabilities.

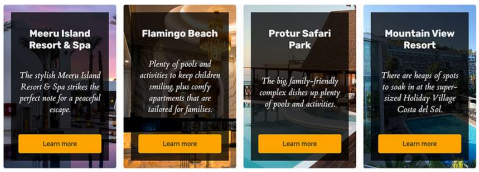

Figure 5.17 Styled cards

.5 Using transitions to animate content on hover and focus-within

To start, we need to create a media query that checks whether the device supports the `hover` interaction, whether the browser window is at least 700 pixels wide, and whether our user has `pre-`

`fers-reduced-motion` enabled on their machine.

Reduced-motion preference

Some users want to opt out of motion-heavy animations. They can do this by enabling a setting on their devices that is conveyed to the browser via the `prefers-reduced-motion` property. We want to make sure that we respect our users' settings. Therefore, we'll state that the setting isn't set (has a value of `no-preference`) as part of our query determining whether to animate our content. For more information about `prefers-reduced-motion`, refer to chapter 3.

Our media query is `@media (hover: hover) and (min-width: 700px) and (prefers-reduced-motion: no-preference) { }`. Notice that we can chain multiple parameters that need to be met for the CSS in the query to be applied.

To hide everything but the header, we'll shift the content down to the bottom of the card by using the `transform` property with a value of

`translateY()`. The `translateY()` value allows us to move content vertically outside the flow of the page; the content around the element being moved is unaffected by the movement and won't reposition itself or get out of the way.

To calculate the distance that the element needs to move, we'll use the `calc()` function again. We'll move the header down by the height of the card (`350px`) minus `8rem` (the top margin of the container + top padding of the container + size of the header), as shown in the following listing.

Listing 5.15 Hiding the non-header content

```
@media (hover: hover) and (min-width: 700px) and
➥ (prefers-reduced-motion: no-preference) {
  main > section > div {
    transform: translateY(calc(350px - 8rem));
  }
}
```

The inner portion of the card is moved down, as shown in figure 5.18.

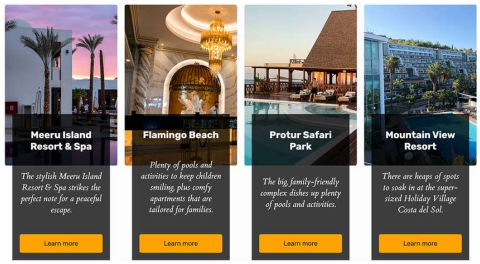

Figure 5.18 Moving the content down

Because we're going to animate showing the content when the user stops hovering over the section, we don't want the trailing content at the bottom to remain: if the user hovers on the content bleeding out of the picture, the content will move upward into the picture, lose the hover, and then move back down. This behavior will repeat, creating a flicker. Therefore, we'll set a height of `5rem` for our inner container and hide the overflow when the paragraph and link are hidden.

Notice that in the second card, a little bit of the paragraph content will still be visible when the content should be hidden, so we'll also hide the nonheader content by using opacity when it shouldn't be seen. Additionally, we'll move that content down `1rem` by using `translateY()`, which will give it a bit of motion

when we animate it back in on `hover`.

All together, the CSS used to hide the content and shorten the inner container appears in the following listing. To select all the content that isn't the header, we can use the `:not()` pseudo-class.

Listing 5.16 Hiding the non-header content

```
@media (hover: hover) and (min-width: 700px) and (prefers-reduced-motion:
➥ no-preference) {                                    ①
  main > section > div {                               ②
    transform: translateY(calc(350px - 8rem));         ②
    height: 5rem;                                      ②
    overflow: hidden;                                  ②
  }                                                    ②
  main > section > div > *:not(h2) {                   ③
    opacity: 0;                                        ③
    transform: translateY(1rem);                       ③
  }                                                    ③
}
```

① Media query

② Moves and shortens the inner content container

③ Hiding all the non-<h2> content

The `not()` pseudo-class allows us to filter selectors. In this case, we want to target anything that isn't an `<h2>`.

Figure 5.19 diagrams the process.

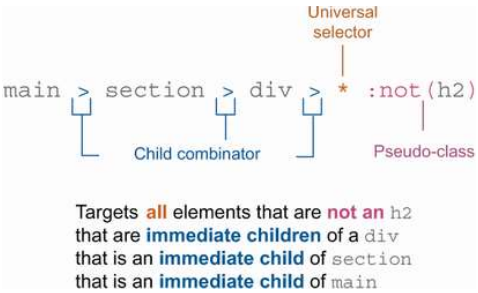

Figure 5.19 Selecting anything that isn't an `<h2>` inside the inner container

Now that the content is hidden (figure 5.20), we can focus on showing it again.

Figure 5.20 Hiding content

To show the content again, we need to undo everything we did to hide it on both `hover` and `focus`. Because we're not removing the links from the Document Object Model (DOM), they're hidden only visually; programmatically, they still exist, and a user can tab to a link via the keyboard. As a result, we need to show the content both when the user hovers over the card and when a link gains focus. Because we want to act on an

ancestor (the content container) when a child (the link) is in focus, we can use the `:focus-within` pseudo-class. This pseudo-class allows us to apply styles conditionally based on whether a descendant of the element is currently in focus.

So when either the link is in focus or the section is being hovered over, we move the container back into place by setting the `translateY()` parameter to `0` (no vertical displacement) and setting the height of the inner container to `350px` (height of the outer container) minus `4rem` (total of the vertical padding and margin of the container). We also need to reinstate the paragraph and link, the opacity of which was set to `0` and which had been moved down by `1rem`.

We'll finish our `hover` and `focus-within` effect by adding a transition to elements being shown and hidden. Because we have predefined states that we're changing between and want the animation to run only once, when the change occurs, we don't

need to use keyframes. We can simply instruct the CSS to animate all the changes when they happen, using the `transition` property with a value of `all 700ms ease-in-out`. All the changes will be animated; the animation will take 700 milliseconds to complete; and the animation will start slow, accelerate, and then slow again before completing. The following listing shows our `hover` and `focus-within` CSS.

Listing 5.17 Showing content on `hover` and `focus-within`

```
@media (hover: hover) and (min-width: 700px) and
➥ (prefers-reduced-motion: no-preference) {
 main > section > div {
  transform: translateY(calc(350px - 8rem));
  height: 5rem;
  overflow: hidden;
  transition: all 700ms ease-in-out;        ①
 }
 div > *:not(h2) {
  opacity: 0;
  transform: translateY(1rem);
  transition: all 700ms ease-in-out;        ①
 }
 section:hover div,                         ②
 section:focus-within div {                 ③
  transform: translateY(0);
  height: calc(350px - 4rem);
 }

 section:hover div > *:not(h2),             ④
 section:focus-within div > *:not(h2){      ⑤
```

```
    opacity: 1;
    transform: translateY(0);
   }
  }
```

① Animates the changes

② On section hover, moves container back into place

③ On section focus-within, moves container back into place

④ On hover, moves all non-<h2> elements inside the container back into place with full opacity

⑤ On section focus-within, moves all non-<h2> elements inside the container back into place with full opacity

With these changes applied (figure 5.21), all that's left to do to complete the project is set the background on our page.

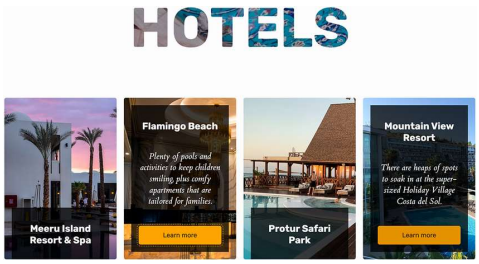

Figure 5.21 The `hover` and `focus-within` effect

To make the pictures pop, we'll add a dark gray, almost black background to the entire page. To apply the background color, we'll add the `background` property with a value of `#010101` to our existing `body` rule, as shown in the following listing.

Listing 5.18 Adding the background

```
body {
  display: grid;
  place-items: center;
  margin-top: 40px;
  padding: 20px;
  background: #010101;
}
```

Figures 5.22, 5.23, and 5.24 show our finished project at various screen sizes.

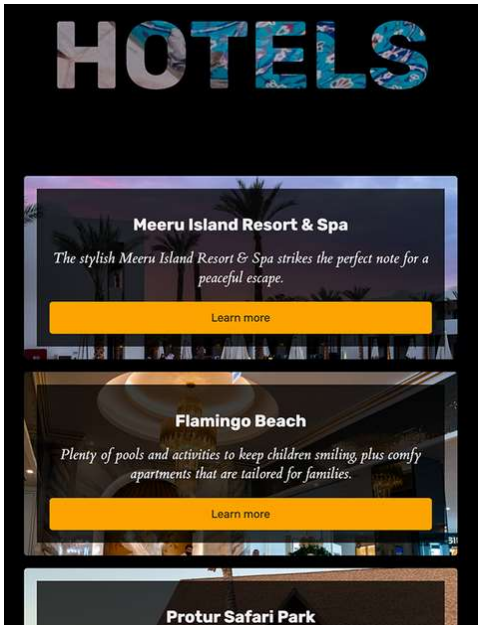

Figure 5.22 Project in window 600 pixels wide

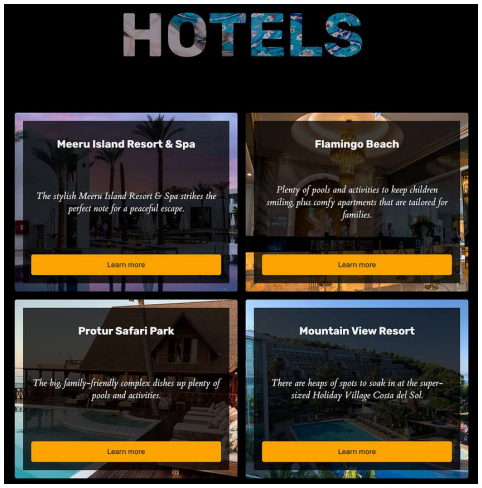

Figure 5.23 Project in window 850 pixels wide

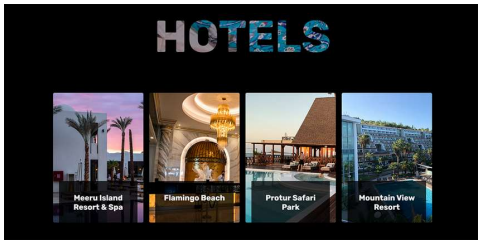

Figure 5.24 Project in window 1310 pixels wide with `prefers-reduced-motion` enabled

ummary

- Grid can be used for entire layouts or individual elements within the layout.
- The `text-transform` property can change text to uppercase without affecting the accessibility of the content.
- Use `text-transform: uppercase` sparingly, not on large areas of content.
- The `background-clip` property with a value of `text` can clip a background image around the text.
- The `background-clip` property with a value of `text` still needs to be prefixed, and this property can change while it's being implemented.
- We can use a media query to check whether a device supports `hover` and adjust our layout so that it prevents the user from seeing the content if their device doesn't support `hover`.
- We can chain multiple conditions in the same media query by using `and`.

- We can use `prefers-reduced-motion` in our media query to respect user preferences regarding animations and motion.
- The `:not()` pseudo-class represents elements that don't match a list of selectors.
- `translateY()` will move content vertically without affecting reflow.
- We can use the `transition` property to animate style changes between states.
- To apply styles conditionally based on an element's descendant being in focus, we use the `focus-within` pseudo-class.