HTML

The example included for this part is a basic HTML webpage without any CSS styling. It obviously does not look very good, but that is a topic for the CSS section! The web is built on HTML. All other technologies eventually convert their own code into HTML in the browser to render all the content that ends up on the webpage. Each HTML tag (<one of these>) denotes a different element, and the closing tag has a forward slash to show the end of the element. It's also possible to embed CSS or JavaScript into the HTML (more on that later). It's pretty much essential to know HTML if you would like to develop a website, because so many things are built on it—even if you all but never write pure HTML, other frameworks use it and output it, so knowing how it works (and its quirks) is still important.

CSS

The example for this part is the same basic HTML webpage, but with CSS this time. It looks a bit better, because that's what CSS's job is. CSS lets developers style webpages, meaning changing the look and format of the page. With more recent CSS editions, there is also a good amount of calculation that is done under the hood to get the layout on the page correct—this is through options like the flex layout or grid layout. Just like HTML, any technology that helps with styling on the web sits on top of CSS. This can be things like tailwindcss or bootstrap—at the end, everything you write gets compiled into CSS so the user's browser can render it. It is possible to add more logic to CSS, and some developers have made CSS-only games for fun, but it is not very practical or easy to do.

JavaScript

The webpage example from the last two sections now has a little bit of JavaScript (JS). JS lets developers add infinitely more logic to webpages, and even on its own, is good enough to do the majority of basic tasks on the web. JS is a programming language, just like Python, C, Java, or anything else. Unlike those languages, it is also optimized for the web and can be run on any modern browser. Developers can use JS to update, add, or remove HTML code on a page, including CSS styling. It can also do any other arbitrary functions before doing that. In my example, all the JS does is implement some basic logic to show a clock on the page, show whether or not each academic building is open, or update an HTML section to show some information about housing selection. Pretty much all web frameworks are built on JS, like React, NextJS, Angular, etc. These frameworks add additional functionality on top of JS for specific use cases—but at the end of the day, they all still compile to JS!

SSH, SCP, and SSH Keys

In the new example, I show some ways I have used SSH and SCP before. I also show a bit on how to make an SSH key and how it can make SSHing or SCPing easier. SSH, which stands for secure shell, and SCP, which stands for secure copy, are ways to securely access another computer, either to run shell commands on it or copy files to it. This is extremely useful, as often in web development, we are not running websites from our own computers and are instead using a hosting provider that has servers somewhere else. SSH keys are a way to verify your identity when you log into your server. When you generate an SSH key, the program generates a public and private key pair. The private key should never be shared with anyone or ever uploaded anywhere, as it is the "key" to the "lock" on everything you use SSH for. The public key, however, *should* be shared anywhere, as it generates the "locks" that the private key can unlock. In public key cryptography, the public key can encrypt information, while the private key can decrypt information encrypted by the public key it is paired to. Servers that use SSH and SCP use this fact to verify that the owner of the private key paired to the public key they were given is the one trying to log in. In practice, using SSH keys is just a more secure and faster way to log in to remote servers, since no passwords are involved. Just keep the private key safe!!

React and Components

React is a framework built on top of JS that lets developers build websites using "components." These components are small JS (or TypeScript, TS) files that contain all the JS or TS and HTML/CSS code needed to render a small piece of a website. Each component can also import other components, so more complex components can be built in pieces from smaller ones. The folders and files for a React project can get somewhat big, so the example I provided is just two of the more important parts of a React project: an App file and a component (a Button, in this case). In the example, the App file imports the button, passes it some parameters (like color), gives it the text to display on the button, and determines what it should do when clicked. The Button component has all of the logic to deal with these inputs. Working with just HTML and JS can be cumbersome quickly, and can involve a lot of code to do things that shouldn't need a lot of code—like making a new component or modifying the state of another component. React makes it a bit easier to have components "talk" to each other and update automatically when other parts of the website change or new data arrives. It also has a lot of tools that people have built for it, and a lot of support in documentation and previous code. LLMs also happen to be better at using React versus other web frameworks because it is so ubiquitous.

Bootstrap

Bootstrap is a framework for CSS that makes it easy to make good-looking, simple websites. It provides a wide variety of premade components to mix and match as desired. You can build a website out of these components, with not a lot of tweaking, and have it look pretty good. A lot of basic websites use it, so you start to notice it when you find one that does. To me, this makes the components look a little bit dated. Anyways, the example for this part is a NavBar component that we made in class and tweaked a bit to have a Bates logo and connect to some JS logic that counts how many times you click a part of the navbar. To use Bootstrap, all you really have to do is import it via npm, copy some examples from the documentation, and modify as needed. The biggest downside is that it is somewhat tough to change the styling of the components, since you don't have easy access to the source code used to make the components. However, it is super easy to pull in some simple components that will be recognizable to the user and have them work and look decent out of the box.

TailwindCSS

Tailwind is another CSS framework that makes it very easy to style webpages. Unlike Bootstrap, it does not use premade components, and instead provides premade CSS classes that do almost everything you would need to with CSS. Because the classes are very concise, and because it also has tools to combine classes and support for mobile-first design and dark mode, it is better than inline CSS, as it is less convoluted while still showing you right in the HTML what each element will generally look like. It's super easy to add or remove classes in the HTML, and in a development environment, the elements will update quickly, so you can see the change quickly. It's also useful when borrowing premade components from other sources (ShadCN, for example) because you can easily change their styling without having to learn anything new or dig into too many hidden implementation details. The example I provided was from one of the tutorials we did. It shows how you can use Tailwind to make a simple design on a webpage. One other nice thing about Tailwind is that the developers have named classes in ways that make sense, and whose importance corresponds to how short each class is—so more important classes have shorter names and are easier to type quickly. I personally really like Tailwind and find it much easier to use than Bootstrap and much more modular. All it takes to be good with Tailwind is to get familiar with the class names.

ShadCN

ShadCN is a bit like Bootstrap but for TailwindCSS. It is a library of pre-made components that you can combine to make a website or a larger component. One key difference is that it uses Tailwind, though. Using Tailwind lets you very easily change the styling of the components, because

all of it is inline in the component styling. I personally found ShadCN to be better-looking than Bootstrap as well. It's pretty easy to install and works the same as Bootstrap in that regard. My example is a pie chart I made when prototyping the final project that uses some ShadCN components.

Krug's Design Principles

The main design principle Krug talks about is also the title of the book—"Don't Make Me Think." The idea is to make things as straightforward as possible and stick to conventions or things that intuitively make sense. When the book was released in the 2000s, this made a lot of sense at the time, as there was still a lot of customization on the web, and people were titling things like add-to-cart buttons with strange names that only made sense in the very specific context they were implemented in. The book also talks about a lot of conventional design principles, like visual hierarchy. One of the most important things in modern design is simply the fact that people only spend a few seconds looking at a website before assessing it and then making the decision to leave or stay—so first impressions matter! The example for this part is some screenshots from the assignment where we had to find some websites that talked about water quality data. Some of them have bad design, which definitely don't use Krug's rules, like the Irish and USGS ones, and one (the Korean one) only implements a few of Krug's rules, but still looks good and usable.

Accessibility

Accessibility is one of the most important parts of designing for the web, and with a lot more tools available to help developers do so, there are not a lot of reasons not to do it. We talked about how websites should be accessible via screen readers and keyboards, meaning you should be able to "tab" through the whole website. There should also be things like image descriptions so screen readers know what an image is about. We also talked about colors for those with colorblindness (like myself) and having captions on anything with audio. The web is for anyone, so everyone should be able to use it. In order to do this, web standards have made available things like ARIA labels, which give screen readers a bit more information about what the element is. In my example of a component from the final project, the aria labels are commented to show them, and the visual layout of the whole component is colorblind friendly. Nowadays, with the advent of LLMs and AI, it's pretty simple to just request the model to prioritize accessibility when making a website. It's even simpler to have it fixed after the fact. No reason not to do it!

Figma

Figma is a bit like photoshop or GIMP, but targeted specifically for web design. It lets you mock up a website using tools reminiscent of the ones you'd have when actually implementing a website. The example I provided is the mockup of our final project compared with the actual implementation. Using Figma allowed us to iterate quickly without having to build out a full model. It's also useful for getting stakeholder input and approval before any code needs to be written. Like a lot of other topics in here, tools like Cursor and even individual models are gaining the ability to directly tap into Figma and start work on implementing a Figma design. We also talked about how this tool is pretty much the standard in industry at the moment, especially since it provides a lot of features to work as part of a team—collaboration in real time, sharing assets, managing users, etc. I found Figma to be super useful for web development, so I will probably use it again.

Cursor

Cursor is a fork of VS Code (VSC) that adds a significant suite of AI features, including agentic ones. The most talked about feature and one of the most useful (in my opinion), is the agent mode. In this mode, the model you select will be able to directly interact with the codebase. It can get context from the files in the repo, search the internet, or read screenshots or specific things you send it. Then, based on your instructions, it will go out and write and implement all the code for your request. Because it is still pretty new, there are a lot of quirks to be aware of, and it can often even depend on which model you use. I found that managing context, and not giving it too many things to think about, was one of the most important things to remember when using it. By far the biggest advantage is that it can generate a lot of pretty solid code very quickly, and only seems to be getting better. If you want an MVP or a concept, there is not a lot of reason NOT to use Cursor or a similar agentic AI-powered platform. Many people have concerns about using AI code in production, which mostly revolve around security and the small details that maybe a human coder could catch and fix, but an AI coder couldn't. The example for this section is the GridView for the final project, which was mostly built with Cursor and GPT-4.1. I also included a screenshot of the chat log window that shows how interacting with the model usually goes.

Google Firebase

Google Firebase is a backend database that is run by Google and whose core features are free for anyone with a Google account. You can store structured data in it and use it for things like user authentication or just a simple database. They have made implementing it in the frontend very easy, so it is not difficult to learn and actually get started with. There are a lot of different tools that provide databases like this, but Firebase is just very easy to use. Because it is widespread, easy to use,

and well-documented, LLMs are also good at using it. The example I included was written by GPT-4.1 in Cursor and downloads the data from Firebase and stores it in the browser of the user as JSON. This would be for our final project. Colby and I started using Firebase initially, so there is a screenshot there of of the web UI with some of our tests. It's also very easy to pull from the database when needed, and not all at once, but this implementation is just to show the basics of how to use it.

Questions

1. **What was your experience in using React+Vite for building web apps compared to "rolling your own" using HTML, CSS, & JavaScript?**
    a. I found using React+Vite to be easier than doing everything the old-fashioned way with HTML/CSS and JS. The only exception is very small websites that don't need to be interactive and are just there to provide information. Even then, using tools like Tailwind would still be useful. Using React reduces the amount of stuff in a single JS or HTML file by using components, which I think makes it easier to work with. I also liked how it was simple to have other components update and "react" to changes in the rest of the website. I did not explore Vite a whole lot, but it seemed very easy to use as well, and included all of the things necessary to run React—so great in my book!

2. **What are the primary takeaways you had from reading Krug's book and your corresponding analysis of sites?**
    a. Krug's book had a lot of good information and put to words a lot of the things I intuitively felt when using the internet. His main point is that users shouldn't have to think too much when using your website. For example, if you click a button and are confused about the results of that button, then the website is poorly designed. This means that you don't have to reinvent the wheel when designing websites, and familiar design is often better for usability, since users are already familiar with how the website should work. One of the worst things you can do is use familiar design language, but have the results of interacting with those design elements be different than what is conventionally accepted. He also talks a lot about how people barely spend any time looking at websites before they decide to stay or leave. In the web world, first impressions matter a lot. I notice all of these things when I visit a new website (which actually isn't that often these days as it was when Krug was writing his book), and I often leave and look for a new website if it has a weird layout or too much going on.

3. **What is the importance of accessibility (give a few explicit examples), and what steps can (and should) you take in assessing the accessibility of your site?**

a. Accessibility is super important for web design. Everyone should have access to the web, and numerous tools are available to help developers create more accessible websites. One of the things we talked about was designing websites for use with screen readers, or for use by colorblind people. In terms of screen readers, so many tools are available for web developers, including ARIA roles and keyboard navigation tools that enable keyboard-only users to navigate a website. This can include things like navigating to the main content or showing an accessibility menu when you hit tab for the first time. Especially with LLMs these days, it's very easy to have a model implement ARIA roles on your website and include features like alt text for images. For colorblind users, when comparing two different things, we discussed ensuring that color is not the only distinguishing feature, or at least that brightness serves as the distinguishing factor rather than hue. For deaf or hard-of-hearing users, we also talked about having captions available for anything that depends on sound.

4. **In what ways did the different design sprints, and use of Figma, help you in thinking about what an end product should look like and how it should function?**
    a. I thought that the design sprints were extremely useful in quickly iterating and evaluating our products as we developed them. Having stakeholders provide input into the project and approve it was also very useful. I think using Figma was also a great way to do this, as it meant we didn't have to build a whole website—we could simply mock it up and still achieve the same effect. With each round of input, we could change or add things to our designs. Because Figma is also intended for web development, it was helpful to have everything set up in a way that could be built on the web. With tools like Photoshop, you may not be entirely sure that you can actually build what you've designed. However, Figma is designed in a way that makes it easier to implement on the web, as long as you can design it in Figma in the first place. People often take an iterative approach when building software or websites, so mirroring that approach in the design phase helps to unify the whole process as well.

5. **What takeaways do you have from working with AI/LLMs through Cursor (or similar) in building web applications?**
    a. I found working with AI and LLMs through Cursor to be extremely useful in building web applications. I think this is because code for the web is so ubiquitous on the internet that there is a lot of data available to the models, and they have become quite good at coding for the web, especially in popular frameworks like React and Tailwind. We briefly discussed in class how some believe that nearly all code will be written by AI in the very near future. After using it a few times this semester, I really agree with that perspective and think that it will only continue to improve. Even in just the time that we had on the final project, a new model of GPT was released (GPT-4.1), and I saw a huge improvement in using GPT-4.1 versus Claude 3.7. That was just a few weeks' time from when I first started using Cursor to

when it had already shown significant improvement. I felt that although it was doing a lot of the coding itself, knowing how software is designed and understanding (generally) how the technologies it uses work helps you direct it better. It's like being a manager more than being a programmer—good managers usually have at least a little bit of experience with the technology the programmers they manage use.

6. **What was your favorite thing (or deemed most useful) that we covered this semester, and why?**
    a. Because I already had some experience with HTML, CSS, and JavaScript before taking this course, I found the material on React and the various CSS frameworks that tend to go with it to be the most interesting thing we covered this semester. I feel like there is a lot of possibility with these technologies, and I'd heard about them before when people talk about web development, but I was not as familiar with them. I generally enjoyed using React, and I felt that there is a lot of depth to it without it being too convoluted. I can see how it simplifies managing large projects, and I completely understand why Meta/Facebook developed it for a platform as large as theirs. I also liked the little bit that we talked about back-end with Google Firebase. Personally, I find the back-end to be my favorite part of web development. And even though we didn't use it very much, I thought that the problems you have to solve when doing back-end development were more personally interesting compared to the problems you have to solve when doing front-end development.

7. **What do you wish we had covered, or had covered in more detail, and why?**
    a. Jumping off from the last question, definitely the backend and database stuff. I think it would be interesting to discuss the various approaches to building the backend for a website. I've personally used Flask (as you know), so I think it would be interesting to discuss Flask further and explore other options for backend development. I would have personally preferred that we spent a little less time on the front-end and perhaps made the course a bit more balanced, so by the end, you could have a fully functional website with both a front-end and a back-end. While I find design interesting, I think the logic and problem-solving aspects of the back end are more fun. So I just wish we had a little bit more time to cover that. It might even make more sense to have two web development courses, the first one where you talk about front-end and the second one where you talk about back-end, so that way if people are more interested in just the front-end they could take one course, but if they're interested in the full stack then they could take both courses.