Developing Full Stack Next.js Web Applications

Dr. Jose Annunziato

Table of Contents

Chapter 1 - Building Next.js User Interfaces with HTML

The ***World Wide Web*** is a collection of dynamic and static documents hosted in a global network called the ***Internet***. Computers connected to the Internet implement a ***Client-Server*** architecture where ***client applications***, such as ***browsers***, request ***web pages*** formatted in ***HTML*** (***HyperText Markup Language***) from ***server*** computers. Client browsers and servers communicate using ***HTTP*** (***HyperText Transport Protocol***), a network communication protocol specifically designed for transporting ***HTML*** content meant to be parsed and rendered in web browsers.

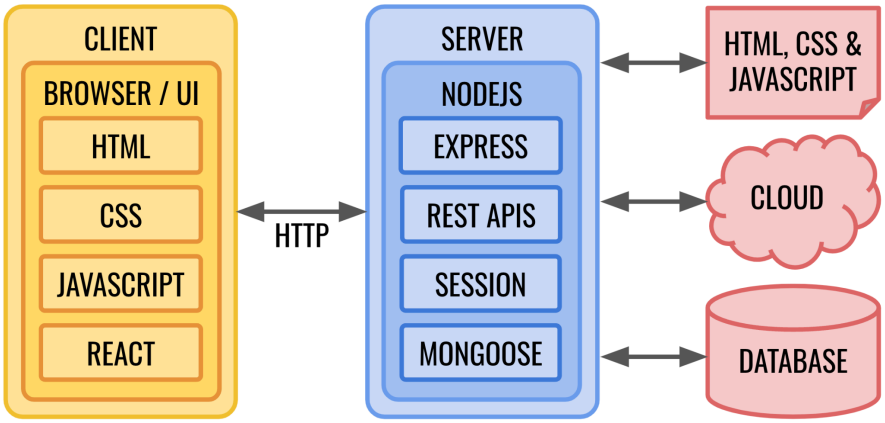


Figure 1.1 - The Client Server Architecture

Web pages consist of plain text documents formatted with ***HTML***, a dialect of ***XML*** (***eXtensible Markup Language***). ***HTML*** is a computer language used to format the content displayed in web pages, such as foreground and background color, white spaces, text alignment, font, lists, tables, and forms. Clients ***request*** HTML documents from servers using ***Uniform Resource Locator***s (***URL***s), which are strings uniquely identifying the address of the server and the location of the document in that server. Servers locate the requested documents and ***respond*** with the content. HTML documents are parsed by browsers, which create in-memory object representations of the pages called the ***DOM*** (***Document Object Model***). The ***DOM*** consists of a hierarchical data structure representing the ***HTML*** document, where each node in the hierarchy is configured to render the content in a particular style. In this chapter, we will learn how to use HTML to format web pages and create user interfaces.

JavaScript is a programming language most famous for running within browsers, programmatically manipulating the ***DOM*** and controlling what browsers render on the screen. HTML documents reference JavaScript files to be downloaded from servers and executed within the browser, implementing dynamic, interactive user interfaces. ***JavaScript*** can also run outside of browsers to implement general-purpose programs. In later chapters, we will use it to implement server-side logic, including API routes and database interactions with ***MongoDB***.

Next.js is a powerful ***React*** framework designed to simplify building ***full-stack web applications***, including client-side user interfaces for ***Single Page Applications*** (***SPA***s) executing in browsers, as well as ***server-side rendering*** (***SSR***), ***static site generation*** (***SSG***), and API endpoints. Next.js applications consist of React components—JavaScript functions or classes—that calculate and render user interfaces based on user inputs, data structures, and server resources. Next.js enhances React by providing built-in routing, data fetching, and optimization features, making it ideal for creating scalable, performant web applications. Next.js applications are web applications that leverage web infrastructure and technologies like HTML, CSS, JavaScript, and can interact with backend resources such as MongoDB databases hosted via HTTP servers.

This chapter describes how to install and configure a local development environment for building Next.js applications. Development is done in the local environment and then shared in a remote source repository such as ***GitHub***, to make the

application publicly available on the web. The source in ***GitHub*** can then be deployed to a remote server hosted on ***Vercel*** which is optimized for Next.js and provides seamless serverless deployment. This chapter introduces creating a Next.js application and explores building user interfaces using HTML and JavaScript. Various HTML elements are described to render user interface content, such as headings, paragraphs, lists, tables, and form elements. All sections in this chapter contain exercises that introduce basic HTML elements and concepts, giving an opportunity to learn and practice HTML skills. The exercises provide detailed instructions to successfully accomplish the tasks. Make sure to complete all exercises described in the book.

The ***Kambaz*** sections in each chapter contain exercises that ask readers to build a fully functional web application inspired by a popular ***Learning Management System*** (***LMS***) with a similar name. The exercises provide sample code and requirements but deliberately leave out steps where the reader is expected to experiment and discover how to implement the requirements using the skills learned in prior sections. This chapter focuses on using plain ***HTML*** within Next.js components to implement a draft, rough prototype of various ***Kambaz*** screens, which at first won't look like the target screenshots. Later chapters will continue working on the ***Kambaz*** application, introducing ***Cascade Style Sheets*** (***CSS***) to style the Web pages so they look more like the screen shots provided, and integrating MongoDB for data persistence.

1.1 Learning Objectives

By the end of this chapter, you will be able to:

- Understand the fundamentals of HTML and how it structures web content.
- Set up a development environment for Next.js applications.
- Create and organize Next.js components using JSX.
- Use Chrome DevTools to inspect and manipulate the DOM.
- Implement various HTML elements such as headings, paragraphs, lists, tables, and images.
- Build interactive web forms with different input types.
- Use CSS to style web pages and improve user interfaces.
- Implement navigation in a Next.js single-page application (SPA) using built-in routing.
- Develop a structured approach to building user interfaces in Next.js, including an introduction to server-side features.

1.2 Setting Up the Development Environment

This section walks through several exercises to become familiar with ***HTML*** in the context of Next.js applications. First a development environment is set up, enabling you to practice ***HTML*** exercises within Next.js components. Copy the examples into your IDE as instructed and confirm that the code renders and behaves as described. Make sure to implement all the exercises in the order they are presented, as later exercises assume you have completed earlier ones.

1.2.1 Installing Node.js

Node.js is a JavaScript runtime that allows executing JavaScript from a computer console. It is essential for developing Next.js applications, as it powers the local development server, handles dependencies via ***npm*** (***Node Package Manager***), and enables server-side features like API routes. Navigate to https://nodejs.org/, download the latest ***LTS*** (***Long Term Support***) version of Node.js for your operating system (recommended: version 22.x or later as of 2025), and install it on your local computer. Restart your computer if prompted, and confirm Node.js is installed by typing the command ***node -v*** in a console or terminal. The output should display the installed version (e.g., v22.4.0). Your version might vary slightly, but ensure it is at least 18.18 or later, as required by Next.js.

```
node -v
v22.4.0
```

Installing Node.js adds the ***npm*** and ***npx*** build tools to run, test, and package ***Node.js*** projects, similar to build tools such as ***mvn*** for ***Java*** and ***pip*** for ***Python*** projects. Later chapters describe how to create ***HTTP Severs*** with Node.js to implement ***RESTful Web API***s and integrate databases such as ***MongoDB***. For now Node.js will be used to create and host a React user interface.

1.2.2 Installing an Integrated Development Environment (IDE)

While you can use any text editor, ***Visual Studio Code*** (***VS Code***) is highly recommended for Next.js development due to its excellent support for JavaScript, React, and extensions like the official Next.js extension for debugging and ***IntelliSense***. Download and install VS Code from https://code.visualstudio.com. Once installed, open VS Code and install useful extensions such as "***ESLint***" for code linting, "***Prettier***" for formatting, and "***React Developer Tools***" for enhanced React support. Open a terminal within VS Code (via Terminal > New Terminal) to run commands seamlessly.

1.2.3 Creating a Next.js Application

React has become one of the most popular ***JavaScript*** libraries for building Web user interfaces. Using ***npx***, a ***Node.js*** tool part of the ***Node.js*** installation, create a ***React*** project where we will be learning all about Web development. From the ***home*** directory (~) of your computer, create a directory for this semester, and then another directory under that for this course. Below are examples of creating directories from the home directory of the file system. On ***macOS***, start the ***Terminal*** application. On a ***Windows OS***, start the ***console*** application or ***PowerShell***. On either OS, type the following to create the directory for the current year and term, e.g., ***~/2049/winter/webdev***.

```
cd  ~                           # navigate to your home directory in your file system
mkdir  2049                     # creates a directory for the current year, e.g., year 2049
mkdir  2049/winter              # creates a directory for the current term, e.g., winter inside 2049
mkdir  2049/winter/webdev       # creates a directory called webdev inside winter/2049
cd     2049/winter/webdev       # navigates to the directory webdev inside winter/2049
```

Feel free to choose other places in the file system making sure all directory and file names are all lowercase, do not have spaces in them, and are inside directories that also meet these criteria. With Node.js installed, you can create a new Next.js project using the ***create-next-app*** tool, which sets up a boilerplate app with best practices. From the terminal (or VS Code's integrated terminal) run the following command:

```
npx create-next-app@latest
```

If this is the first time creating a ***Next.js*** project, you'll be asked to install the ***create-next-app*** package. If this is the case, accept to proceed.

```
Need to install the following packages:
create-next-app@15.3.5
Ok to proceed? (y)
```

Name the project ***kambaz-next-js***.

```
? What is your project named? › kambaz-next-js
```

Choose all the defaults

```
✔ What is your project named? ... kambaz-next-js
✔ Would you like to use TypeScript? ... No / Yes
✔ Would you like to use ESLint? ... No / Yes
```

```
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like your code inside a `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to use Turbopack for `next dev`? … No / Yes
✔ Would you like to customize the import alias (`@/*` by default)? … No / Yes
Creating a new Next.js app in /Users/jannunzi/neu/nextjs/kambaz-next-js.
```

Wait for the project to install.

```
Using npm.

Initializing project with template: app-tw

Installing dependencies:
- react
- react-dom
- next

Installing devDependencies:
- typescript
- @types/node
- @types/react
- @types/react-dom
- @tailwindcss/postcss
- tailwindcss
- eslint
- eslint-config-next
- @eslint/eslintrc

added 339 packages, and audited 340 packages in 36s

137 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
Initialized a git repository.

Success! Created kambaz-next-js at /Users/jannunzi/neu/nextjs/kambaz-next-js
```

Next.js will create the project in a directory called ***kambaz-next-js***. ***Change directory*** (***cd***) to the project directory and run the project using npm run dev as shown below.

```
cd   kambaz-next-js
npm  run dev

> kambaz-next-js@0.1.0 dev
> next dev --turbopack

   ▲ Next.js 15.3.5 (Turbopack)
   - Local:        http://localhost:3000
   - Network:      http://192.168.1.177:3000

 ✓ Starting...
 ✓ Ready in 845ms
```

To see the Web application running, use Google Chrome to navigate to the URL displayed on the console, e.g., http://localhost:3000. Confirm the ***NEXT.js*** logo render the browser as shown below. Stop the Web application by typing Ctrl+C. Although the ***Next.js*** application can be run from a terminal, prefer running it from within an IDE such as ***Visual Studio Code*** (***code***). Using ***code***, open the ***kambaz-next-js*** directory and start the application from the terminal window at

the bottom of ***code***. It the terminal is not already showing, display it from the menu ***View***, ***Terminal***. Although other browsers and IDEs are acceptable, prefer using ***Google Chrome*** and ***Visual Studio Code***.

Figure 1.2 - Default Next.js project running in a browser

1.2.4 Creating Next.js Pages with React Components

React is a ***JavaScript*** library for building dynamic ***web user interfaces*** (***UI***). In Next.js, which is built on React, user interfaces are created using ***JavaScript*** (or ***TypeScript***) functions or classes called ***components***. These components compute and return the content of the user interface as HTML-like code snippets, often incorporating logic for interactivity, data fetching, or state management. The syntax that combines JavaScript and HTML is called ***JSX*** (***JavaScript XML***), and files typically use the ***.jsx*** extension. JSX allows you to seamlessly embed HTML tags within JavaScript code, making it intuitive to describe UI structures programmatically.

For enhanced type safety and scalability—especially in larger applications—***TypeScript*** is recommended. TypeScript is a superset of JavaScript that adds static typing, helping catch errors early during development. Next.js supports TypeScript out-of-the-box, and files use the ***.tsx*** extension (as shown in examples below). Throughout this course, we'll use TypeScript to leverage its benefits in a full-stack context.

In your VS Code ***EXPLORER*** window on the left, navigate to the ***app*** directory and do all work inside the ***app*** directory. To practice JSX syntax in Next.js, create a new page component for Lab 1 in a file ***app/Labs/Lab1/page.tsx*** containing the source code below:

app/Labs/Lab1/page.tsx	*http://Localhost:3000/Labs/Lab1*

```
export default function Lab1() {
  return (
    <div id="wd-lab1">
      <h2>Lab 1</h2>
    </div>
);}
```

Lab 1

The ***Lab1*** function above defines a default-exported React component, which Next.js treats as a page when placed in a folder like ***app/Labs/Lab1/page.tsx***. The function returns an ***HTML div element*** (***division element***) that contains an ***h2 heading element*** with the text ***Lab 1***. The ***HTML*** code snippet will render a heading of size 2 introducing the topic of the first chapter. The file-system routing automatically makes this accessible at /Labs/Lab1 in your app. Save the file and ensure your development server is running (npm run dev). Confirm you can navigate to the ***Lab1*** page by pointing your browser to ***http://localhost:3000/Labs/Lab1***.

The Next.js project uses a default styling based on Tailwind.css which we'll discuss in the next chapter. For now, comment out the global ***cascading style sheet*** (***CSS***) so we can see the raw, default style of the Web pages implemented in this course. Later chapters will discuss how to style Web pages with ***CSS***. In ***app/layout.tsx***, comment out the import statement as shown below

app/layout.tsx

```
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
// import "./globals.css";                                    // comment out this line
...                                                           // leave the rest of this file alone
```

1.3 Introduction to HTML

HTML (HyperText Markup Language) is a specialized dialect of ***XML (eXtensible Markup Language)*** designed for structuring and formatting plain text content so that web browsers can interpret and render it with specific styles, layouts, and interactivity. In the context of Next.js applications, HTML elements are embedded within JSX (JavaScript XML) syntax inside React components, allowing you to define user interfaces declaratively while leveraging JavaScript for dynamic behavior. This section introduces some of the most common and simple HTML elements, often referred to as "tags," and explains how they contribute to building web pages.

Consider the following HTML snippet, which styles the text "Labs" as a level-1 heading, rendering it in large, bold letters:

```
<h1>Labs</h1>
```

In code, ***<h1>*** and ***</h1>*** are called ***tags***: The <h1> is the ***opening tag***, and </h1> is the ***closing tag***. The text "Labs" between them is the ***body*** or content of the tag. Tags add semantic meaning to the content—for example, h1 indicates a top-level heading, which browsers interpret by applying default styles like larger font size and bold weight.

When a browser parses this HTML (or JSX in a Next.js component), it creates an in-memory representation called the ***DOM (Document Object Model)***—a hierarchical tree structure where each tag becomes a ***node***. The DOM determines how the browser renders the page on the screen, and JavaScript (including React in Next.js) can manipulate it dynamically for updates without full page reloads.

We often use the terms "tag" and "element" interchangeably, but there's a subtle distinction: A ***tag*** refers to the textual syntax in your code (e.g., <h1>), while an ***element*** is the broader concept, encompassing the tag, its attributes, body, and the resulting DOM node. For practical purposes in this course, either term works, but understanding this helps when debugging with browser tools like ***Chrome DevTools***.

1.3.1 Structuring Web Content with the HTML Heading and Div Tags

Text documents are often broken up into several sections and subsections. Each section is usually prefaced with a short title or ***heading*** that attempts to summarize the topic of the section it precedes. For instance this paragraph is preceded by the heading ***Heading Tags***. The font of the section headings are usually larger and bolder than the plain text and their subsection headings. This document uses headings to introduce topics such as HTML Documents, HTML Tags, Heading Tags, etc. HTML ***heading tags*** can be used to format plain text so that it renders in a browser as large headings. There are 6 heading tags for different sizes: ***h1***, ***h2***, ***h3***, ***h4***, ***h5***, and ***h6***. Tag ***h1*** is the largest heading and ***h6*** is the smallest heading.

Another commonly used HTML element is the ***div*** tag (division tag), which is a generic container used to group other elements together. Unlike heading tags, which have specific visual and semantic purposes, ***div*** tags do not inherently carry any meaning or formatting. They serve as a way to organize content on the page, often making it easier to apply styles or

layout rules using CSS. For example, you can use a to group a heading, a paragraph, and an image together as a single unit within a webpage. To practice using the heading and div tags, create several headings and subheadings to introduce the topics covered in this chapter. In the ***Lab1*** component copy the HTML below and confirm that the browser refreshes with the new content as shown below on the right. Note how the text surrounded by the ***<h2>*** tag is larger and bolder than the text surrounded by the ***<h3>*** tag, and both are larger than the text that has no tags around it. The ***app/Labs/Lab1/page.tsx*** document will contain all the exercises for this chapter. Be sure to complete all the exercises described in this chapter following the instructions in the order they are listed.

app/Labs/Lab1/page.tsx

```
export default function Lab1() {
  return (
    <div id="wd-lab1">
      <h2>Lab 1</h2>
      <h3>HTML Examples</h3>
      <div id="wd-h-tag">
        <h4>Heading Tags</h4>
Text documents are often broken up into several sections and
subsections. Each section is usually prefaced with a short
title or heading that attempts to summarize the topic of the
section it precedes. For instance this paragraph is preceded by
the heading Heading Tags. The font of the section headings are
usually larger and bolder than their subsection headings. This
document uses headings to introduce topics such as HTML
Documents, HTML Tags, Heading Tags, etc. HTML heading tags can
be used to format plain text so that it renders in a browser as
large headings. There are 6 heading tags for different sizes:
h1, h2, h3, h4, h5, and h6. Tag h1 is the largest heading and
h6 is the smallest heading.
      </div>
      {/* do the next exercise here */}
    </div>
);}
```

How the browser renders

Labs

Lab 1

HTML Examples

Heading Tags

Text documents are often broken up into several sections and subsections. Each section is usually prefaced with a short title or heading that attempts to summarize the topic of the section it precedes. For instance this paragraph is preceded by the heading Heading Tags. The font of the section headings are usually larger and bolder than the plain text and their subsection headings. This document uses headings to introduce topics such as HTML Documents, HTML Tags, Heading Tags, etc. HTML heading tags can be used to format plain text so that it renders in a browser as large headings. There are 6 heading tags for different sizes: h1, h2, h3, h4, h5, and h6. Tag h1 is the largest heading and h6 is the smallest heading.

1.3.2 Formatting Text with the HTML Paragraph Tag

Browsers ignore white spaces such as extra spaces, tabs and newlines. To add space between different paragraphs, use the paragraph tag ***<p>*** to wrap the text and add vertical spacing. To practice using the paragraph tag, copy the code here on the right to the end of the ***page.tsx***. Below is another example of how the browser renders HTML text on the left column. Note how the browser ignores line breaks and other white space formatting such as tabs, new lines and extra spaces, and content just flows from left to right and then wraps when there's no more horizontal space.

```
<div id="wd-p-tag">
  <h4>Paragraph Tag</h4>
  <p id="wd-p-1">
This is a paragraph. We often separate a long set
of sentences with vertical spaces to make the text
easier to read. Browsers ignore vertical white
spaces and render all the text as one single set
of sentences. To force the browser to add vertical
spacing, wrap the paragraphs you want to separate
with the paragraph tag  </p>
  {/* continue here */}
</div>
```

This rendering behavior is referred to as ***inline layout behavior*** or just ***inline***. Inline content flows from left to right horizontally the whole width of its parent container and then wraps vertically when there's no more horizontal space. Add the content below on the left and confirm that it renders as shown below on the right.

app/Labs/Lab1/page.tsx

```
export default function Lab1() {
  return (
    <div id="wd-lab1">
      <h2>Lab 1</h2>
      <h3>HTML Examples</h3>
      <div id="wd-h-tag"> ... </div>
      <div id="wd-p-tag">
        <h4>Paragraph Tag</h4>
```

How the browser renders

This is the first paragraph. The paragraph tag is used to format vertical gaps between long pieces of text like this one. This is the second paragraph. Even though we

```
        <p id="wd-p-1"> ... </p>
This is the first paragraph. The paragraph tag is used to format vertical gaps
between long pieces of text like this one.

This is the second paragraph. Even though there is a deliberate white gap
between the paragraph above and this paragraph, by default browsers render
them as one contiguous piece of text as shown here on the right.

This is the third paragraph. Wrap each paragraph with the paragraph tag to
tell browsers to render the gaps.
      </div>
    </div>
  );
}
```

added a deliberate gap between the paragraph above and this paragraph, by default browsers render them as one contiguous piece of text as shown here on the right. This is the third paragraph. Wrap each paragraph with the paragraph tag to tell browsers to render the gaps.

Apply the paragraph tags as shown below to let the browser know we want to keep the vertical spacing. Make sure to include the ***id*** attributes with the values shown. Confirm the paragraphs now render as shown below on the right. Note how now there's a gap between the paragraphs. Both the paragraph and heading tags add vertical space and is referred to to as ***block layout behaviour*** or just ***block***. By controlling the ***inline*** and ***block*** layout, all sorts of useful layouts can be achieve.

app/Labs/Lab1/page.tsx

```
      <div id="wd-p-tag">
        <h4>Paragraph Tag</h4>
        <p id="wd-p-1"> ... </p>
        <p id="wd-p-2">
This is the first paragraph. The paragraph tag is used to format
vertical gaps between long pieces of text like this one.
        </p>
        <p id="wd-p-3">
This is the second paragraph. Even though there is a deliberate white
gap between the paragraph above and this paragraph, by default
browsers render them as one contiguous piece of text as shown here on
the right.
        </p>
        <p id="wd-p-4">
This is the third paragraph. Wrap each paragraph with the paragraph
tag to tell browsers to render the gaps.
        </p>
      </div>
```

How the browser renders

This is the first paragraph. The paragraph tag is used to format vertical gaps between long pieces of text like this one.

This is the second paragraph. Even though there is a deliberate white gap between the paragraph above and this paragraph, by default browsers render them as one contiguous piece of text as shown here on the right.

This is the third paragraph. Wrap each paragraph with the paragraph tag to tell browsers to render the gaps.

1.3.3 Listing Content with the HTML Ordered List Tag

List elements are used to create lists of related items. There are two types of lists: ***ordered*** and ***unordered***. Ordered list elements are useful for listing items in a particular order. Here's a list of steps to make pancakes. Add the content on the left after the paragraph exercise. Confirm the browser renders as shown on the right.

app/Labs/Lab1/page.tsx

How the browser renders

```
export default function Lab1() {
  return (
    <div id="wd-lab1">
      <h2>Lab 1</h2>
      <h3>HTML Examples</h3>
      <div id="wd-h-tag"> ... </div>
      <div id="wd-p-tag"> ... </div>
      <div id="wd-lists">
        <h4>List Tags</h4>
        <h5>Ordered List Tag</h5>
        How to make pancakes:
        1. Mix dry ingredients.
        2. Add wet ingredients.
        3. Stir to combine.
        4. Heat a skillet or griddle.
        5. Pour batter onto the skillet.
        6. Cook until bubbly on top.
        7. Flip and cook the other side.
        8. Serve and enjoy!
      </div>
    </div> );}
```

Paragraph Tag

This is a paragraph. We often separate a long set of sentences with vertical spaces to make the text easier to read. Browsers ignore vertical white spaces and render all the text as one single set of sentences. To force the browser to add vertical spacing, wrap the paragraphs you want to separate with the paragraph tag

This is the first paragraph. The paragraph tag is used to format vertical gaps between long pieces of text like this one.

This is the second paragraph. Even though there is a deliberate white gap between the paragraph above and this paragraph, by default browsers render them as one contiguous piece of text as shown here on the right.

This is the third paragraph. Wrap each paragraph with the paragraph tag to tell browsers to render the gaps.

List Tags

Ordered List Tag

How to make pancakes: 1. Mix dry ingredients. 2. Add wet ingredients. 3. Stir to combine. 4. Heat a skillet or griddle. 5. Pour batter onto the skillet. 6. Cook until bubbly on top. 7. Flip and cook the other side. 8. Serve and enjoy!

Note that in the HTML text on the left explicitly included the numbers 1., 2., etc., but the formatting is lost when the browser renders it on the right. Instead of rendering a list of items, each in its own line, they are instead all rendered on the same line. Use the ordered list tag to achieve the desired format. The ordered list tag actually consists of a pair of tags. The ***
*** and ***
*** denote the beginning and end of the list. The ***
*** and ***
*** denote the content of an item in the list. Here's the same example from earlier, but now applying the ordered list tags to achieve the intended formatting.

app/Labs/Lab1/page.tsx

```
<div id="wd-lists">
  <h4>List Tags</h4>
  <h5>Ordered List Tag</h5>
  How to make pancakes:
  <ol id="wd-pancakes">
    <li>Mix dry ingredients.</li>
    <li>Add wet ingredients.</li>
    <li>Stir to combine.</li>
    <li>Heat a skillet or griddle.</li>
    <li>Pour batter onto the skillet.</li>
    <li>Cook until bubbly on top.</li>
    <li>Flip and cook the other side.</li>
    <li>Serve and enjoy!</li>
  </ol>
</div>
```

How the browser renders

List Tags

Ordered List Tag

How to make pancakes:

1. Mix dry ingredients.
2. Add wet ingredients.
3. Stir to combine.
4. Heat a skillet or griddle.
5. Pour batter onto the skillet.
6. Cook until bubbly on top.
7. Flip and cook the other side.
8. Serve and enjoy!

Remove the unnecessary numbers 1, 2, 3, etc. and add the ***<ol>*** and ***<li>*** elements as shown above and confirm the list renders as shown on the right. Create a new ordered list describing favorite recipes or yours. Include it in an ordered list whose ID is "***wd-your-favorite-recipe***" and it should have at least 3 steps to make your recipe.

app/Labs/Lab1/page.tsx

```
<div id="wd-lists">
  <h4>List Tags</h4>
  <h5>Ordered List Tag</h5>
  How to make pancakes:
  <ol id="wd-pancakes"> ... </ol>
  My favorite recipe:
  <ol id="wd-your-favorite-recipe">
    {/* complete on your own */}
  </ol>
</div>
```

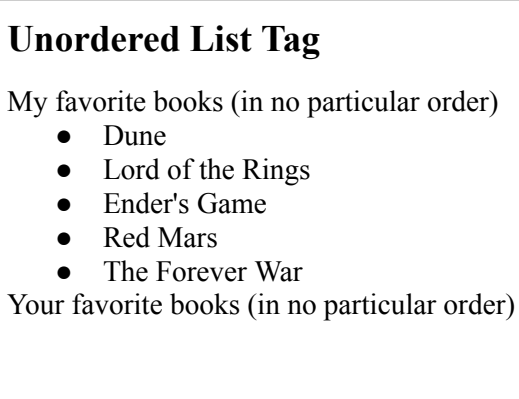
1.3.4 Listing Content In No Particular Order with the HTML Unordered List Tag

Unordered list elements are similar to ordered lists with the difference that the items are not numbered and instead bullets decorate each line item. The unordered list tag is ***<ul>***, but the list item tag is still ***<li>*** as shown below. Unordered lists are great for displaying a list of items in no particular order. Here's an example of an unordered list of my favorite books in no particular order. Add the example HTML code below after the end of the previous ordered list exercise and then confirm the browser renders as shown below. Add an unordered list that contains at least 3 of your favorite books.

app/Labs/Lab1/page.tsx

```
My favorite recipe:
<ol id="wd-my-favorite-recipe">{/* complete on your own */}</ol>
<h5>Unordered List Tag</h5>
My favorite books (in no particular order)
<ul id="wd-my-books">
  <li>Dune</li>
  <li>Lord of the Rings</li>
  <li>Ender's Game</li>
  <li>Red Mars</li>
  <li>The Forever War</li>
</ul>
Your favorite books (in no particular order)
<ul id="wd-your-books">
  {/* complete on your own */}
</ul>
```

How the browser renders



1.3.5 Tabulating Data with the HTML Table Tags

HTML began as a tool for formatting and sharing research papers and their results amongst scientists. Documents often contained data points captured as results of some experiments. Each data point might have several attributes associated such as speed, temperature, and location. A common way to display or visualize these results was formatted as a data table with a row for each data point and a column for each attribute. The ***<table>*** tag allows formatting data into a table of rows and columns. For instance, consider capturing grade results for several quizzes someone might have taken over a semester. These might be captured using the following table.

Quiz	Topic	Date	Grade
Q1	HTML	2/3/21	85
Q2	CSS	2/10/21	90
Q3	JavaScript	2/17/21	95
Average			90

Several things to note:

1. The first row is formatted as headings for each column
2. There are 3 data points, one for each quiz, one in each row
3. Data under the same column is of the same data type
4. The last row is formatted as a footer
5. The three first columns of the last row are merged into a single cell and unlike the 3 data rows. Rows also can span to merge into a single cell.

HTML ***table*** tag can contain additional tags to format the data as follows:

- ***table*** - declares the start of a table
- ***tr*** - declares the start of a row
- ***td*** - declares a table data cell
- ***thead*** - declares a row of headings
- ***tbody*** - declares the main data content rows of the table
- ***tfoot*** - declares a row as a footer
- ***th*** - declares a table cell as a heading

To practice using ***table*** tag, copy the HTML below to the end of ***page.tsx***. The code implements the table shown earlier. Ignore the comments on the right.

app/Labs/Lab1/page.tsx

```
export default function Lab1() {
  return (
    <div id="wd-lab1">
      <h2>Lab 1</h2>
      <h3>HTML Examples</h3>
      ...
      <div id="wd-tables">
        <h4>Table Tag</h4>
        <table border={1} width="100%">              <!-- declares the table, sets border and width -->
          <thead>                                    <!-- declares the table heading section -->
            <tr>                                     <!-- declares the headings row -->
              <th>Quiz</th>                          <!-- declares heading for first column -->
              <th>Topic</th>                         <!-- declares heading for second column -->
              <th>Date</th>                          <!-- declares heading for third column -->
              <th>Grade</th>                         <!-- declares heading for fourth column -->
            </tr>
          </thead>
          <tbody>                                    <!-- declares the table's main content -->
            <tr>                                     <!-- declares the first row -->
              <td>Q1</td>                            <!-- declares data for first row, first column -->
              <td>HTML</td>                          <!-- declares data for first row, second column -->
              <td>2/3/21</td>                        <!-- declares data for first row, third column -->
              <td>85</td>                            <!-- declares data for first row, fourth column -->
            </tr>
            <tr>                                     <!-- declares the second row -->
              <td>Q2</td>                            <!-- declares data for second row, first column -->
              <td>CSS</td>                           <!-- declares data for second row, second column -->
              <td>2/10/21</td>                       <!-- declares data for second row, third column -->
              <td>90</td>                            <!-- declares data for second row, fourth column -->
            </tr>
            <tr> ... </tr>                           <!-- declares the third row -->
          </tbody>
          <tfoot>                                    <!-- declares the table footer section -->
            <tr>                                     <!-- declares the footer row -->
              <td colSpan={3}>Average</td>           <!-- declares data spans 3 columns -->
              <td>90</td>                            <!-- declares data for fourth column -->
            </tr>
          </tfoot>
        </table>
      </div>
    </div>
  );
}
```

Content in a table cell can be aligned at the top of the cell ***td*** using the ***valign*** attribute set to ***"top"***, e.g., ***<td valign="top">***. Content can also be aligned horizontally in a table cell with the ***align*** attribute, e.g., ***<td align="rigth">*** aligns the content to the right. This course has 10 quizzes. Add the 7 quizzes that are missing in the table. Use quiz names such as ***Q3***, ***Q4***, through ***Q10***. Make up different dates and scores.

1.3.6 Image Tag

The image tag is used to render pictures in HTML documents. The images can be anywhere on the internet, or a local image document in the local file system.

```
<img src="my-picture.jpg"          <!-- Use img tag to embed pictures in HTML documents.
     width="200px"                 src attributes references image file either locally or remotely. width and height
     height="300px"                attributes configure the image size. If only width or height is provided, the
/>                                 other scales proportionally -->
```

To practice using the image tag, copy the code below to ***page.tsx***. The first image tag embeds an image from a remote server. The second one assumes there's a local image file called ***teslabot.jpg*** in ***public/images/teslabot.jpg***. Search for ***Tesla Bot*** on the internet, and download an image that looks similar to the one shown below. Name the image ***teslabot.jpg***.

```
<div id="wd-images">
  <h4>Image tag</h4>
  Loading an image from the internet: <br />
  <img id="wd-starship" width="400px"
src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg" />
  <br />
  Loading a local image:
  <br />
  <img id="wd-teslabot" src="/images/teslabot.jpg" height="200px" /></div>
```

1.3.7 Creating Web Forms

Form tags are useful for entering data. Let's take a look at the most common ones: ***form***, ***input***, ***select***, ***textarea***, ***radio***, ***checkbox***.

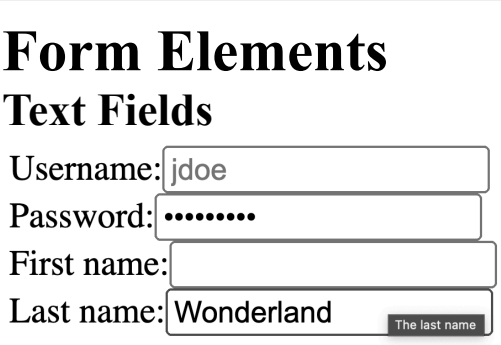
1.3.7.1 Creating Text Input Fields

Text fields are the most common form elements allowing entering a single line of text.

```
<input type="text"                  <!--    use input tag's text type to declare a single line input field text is
       placeholder="hint"           default if type is left out. Use placeholder and title to give a hint of what
       title="tooltip"              information you're expecting. Optionally initialize the value of the field with
       value="COMEDY"/>             value attribute-->
```

To practice using text fields, add the following code snippet at the end of ***page.tsx***. It creates a set of input fields for entering some personal information. The ***label*** tags below associate descriptive text with each form element by setting a ***label***'s ***forName*** attribute to the ***id*** attribute of the related form field.

```
<div id="wd-forms">
  <h4>Form Elements</h4>
  <form id="wd-text-fields">
    <h5>Text Fields</h5>
    <label htmlFor="wd-text-fields-username">Username:</label>
    <input placeholder="jdoe" id="wd-text-fields-username" /> <br />
    <label htmlFor="wd-text-fields-password">Password:</label>
    <input type="password" value="123@#$asd" id="wd-text-fields-password" />
    <br />
    <label htmlFor="wd-text-fields-first-name">First name:</label>
    <input type="text" title="John" id="wd-text-fields-first-name" /> <br />
    <label htmlFor="wd-text-fields-last-name">Last name:</label>
    <input type="text" placeholder="Doe"
           value="Wonderland"
           title="The last name"
           id="wd-text-fields-last-name" />
    {/* copy rest of form elements here  */}
  </form>
</div>
```



1.3.7.2 Creating Multi-Line Input Fields with Textarea

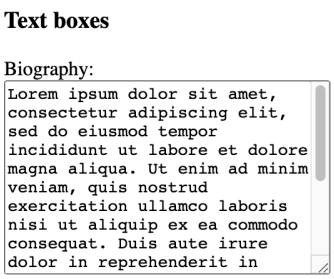
The ***textarea*** tag is useful for entering long form text such as someone's biography data, or a blog post.

```
<textarea cols="20" rows="25"
          placeholder="Biography"
          title="tooltip">Some text
</textarea>
```

```
<!--    use textarea tag for long form text configure its width and
height with attributes cols and rows. Use placeholder and tooltip to
give hints. Note default value is in tag's body -->
```

To practice using the ***textarea*** tag, add the following example to the end of ***page.tsx***. It creates a ***textarea*** useful for entering a biography. Get a sample of the dummy text at https://www.lipsum.com.

```
<h5>Text boxes</h5>
<label>Biography:</label><br/>
<textarea id="wd-textarea" cols={30} rows={10}>Lorem ipsum dolor sit
amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.</textarea>
```



1.3.7.3 Creating Interactive Buttons with HTML

Buttons allow invoking actions executed by the browser. To practice creating buttons, copy the code below at the end of ***page.tsx***. Confirm that clicking the button displays an alert box with an optimistic message.

```
<h5 id="wd-buttons">Buttons</h5>
<button type="button"
        onClick={() => alert("Life is Good!")}
        id="wd-all-good">
  Hello World!
</button>
```

1.3.7.4 Creating Exclusive Choices with HTML Radio Buttons

Radio buttons allow selecting a single choice from multiple alternative options

```
<input type="radio"
       name="NAME1"
       value="OPTION1"/>
<input type="radio"                {/* use the input tag's radio type to declare radio buttons */}
       name="NAME1"                {/* group radio buttons by using the same name accross several buttons */}
       value="OPTION2" checked/>   {/* use checked to set the checkbox's as initially checked  */}
```

To practice using radio buttons, add the following example at the end of ***page.tsx***. The ***label*** tags allow associating a piece of text with an input field by setting its ***htmlFor*** attribute to the ***id*** of the corresponding input field. For instance, the ***<label htmlFor="wd-radio-comedy">Comedy</label>*** is associated with ***<input id="wd-radio-comedy"/>*** because the ***label***'s ***htmlFor*** attribute has the same value as the ***id*** attribute of the ***input*** field. Now, when clicking on the label, it's as if having clicked on the associated input field. In the case of a ***checkbox***, the value toggles between ***checked*** or not. In the case of a ***text*** input field, the field gets focus. To make radio buttons ***mutually exclusive***, group them by setting their ***name*** attributes to the same value. When a radio button is clicked in the same group, all the other radio buttons are unselected.

```
<h5 id="wd-radio-buttons">Radio buttons</h5>

<label>Favorite movie genre:</label><br />

<input type="radio" name="radio-genre" id="wd-radio-comedy"/>
<label htmlFor="wd-radio-comedy">Comedy</label><br />

<input type="radio" name="radio-genre" id="wd-radio-drama"/>
<label htmlFor="wd-radio-drama">Drama</label><br />

<input type="radio" name="radio-genre" id="wd-radio-scifi"/>
<label htmlFor="wd-radio-scifi">Science Fiction</label><br />
<input type="radio" name="radio-genre" id="wd-radio-fantasy"/>
<label htmlFor="wd-radio-fantasy">Fantasy</label>
```

1.3.7.5 Creating Multiple Selections with HTML Checkboxes

Checkboxes allow selecting multiple choices

```
<input type="checkbox"                  {/* use the input tag's checkbox type to declare a checkbox */}
       name="NAME2"                     {/* use the same name value to group multiple checkboxes    */}
       value="OPTION1" checked/>        {/* use checked attribute to select the checkbox by default */}
<input type="checkbox"
       name="NAME2"
       value="OPTION2"/>
<input type="checkbox"
       name="NAME2"
       value="OPTION3" checked/>
```

To practice using checkboxes, add the following example to the end of ***page.tsx***. It creates a set of checkbox buttons to select all the favorite movie genres, which there might be more than one.

```
<h5 id="wd-checkboxes">Checkboxes</h5>
<label>Favorite movie genre:</label><br/>

<input type="checkbox" name="check-genre" id="wd-chkbox-comedy"/>
<label htmlFor="wd-chkbox-comedy">Comedy</label><br/>

<input type="checkbox" name="check-genre" id="wd-chkbox-drama"/>
<label htmlFor="wd-chkbox-drama">Drama</label><br/>

<input type="checkbox" name="check-genre" id="wd-chkbox-scifi"/>
<label htmlFor="wd-chkbox-scifi">Science Fiction</label><br/>

<input type="checkbox" name="check-genre" id="wd-chkbox-fantasy"/>
<label htmlFor="wd-chkbox-fantasy">Fantasy</label>
```

1.3.7.6 Creating Dropdown Menus with HTML Select and Option

Dropdowns are useful for selecting one or more options from a list of possible values. The default version displays a set of values from which one can choose a single value.

```
<select>                                        <!--    Wrap several option tags in a select tag.
   <option value="VAL1">Value 1</option>        Optionally provide option's value, otherwise the
   <option value="VAL2" selected>Value 2</option>   option's text is the value of the select element.
   <option value="VAL3">Value 3</option>        Optionally use selected attribute to select default.
</select>                                               -->
```

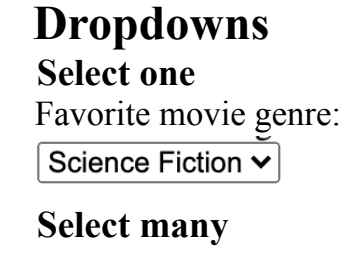
Adding the optional ***multiple*** attribute converts the dropdown into a list of options that can be selected.

```
<select multiple>                                   <!--    Alternatively use attribute multiple to
   <option value="VAL1" selected>Value 1</option>   allow selecting more than one option.    Use
   <option value="VAL2">Value 2</option>            ctrl+click to select more than one option -->
   <option value="VAL3" selected>Value 3</option>
</select>
```

Practice using the ***select*** tag by adding the following code snippet to the end of ***page.tsx***. It creates a dropdown and a list of options as shown (styling might differ).

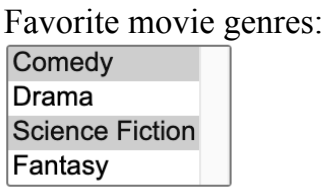
```
<h4 id="wd-dropdowns">Dropdowns</h4>

<h5>Select one</h5>
<label  htmlFor="wd-select-one-genre"> Favorite movie genre: </label><br/>
<select id="wd-select-one-genre">
   <option value="COMEDY">Comedy</option>
   <option value="DRAMA">Drama</option>
   <option selected value="SCIFI">
       Science Fiction</option>
   <option value="FANTASY">Fantasy</option>
```



```
</select>

<h5>Select many</h5>
<label  htmlFor="wd-select-many-genre"> Favorite movie genres: </label><br/>
<select multiple id="wd-select-many-genre">
   <option value="COMEDY" selected> Comedy          </option>
   <option value="DRAMA">           Drama           </option>
   <option value="SCIFI"  selected> Science Fiction </option>
   <option value="FANTASY">         Fantasy         </option>
</select>
```



1.3.7.7 Other HTML Field Types

The input tag's ***type*** attribute has several other possible values: ***date***, ***email***, ***number***, and ***range*** which configure the ***input*** tag to handle different data formats. To practice these other formats add the following example under the last input field worked on earlier, but inside the ***form*** tag. The fields should look as shown below on the right (your styling might differ).

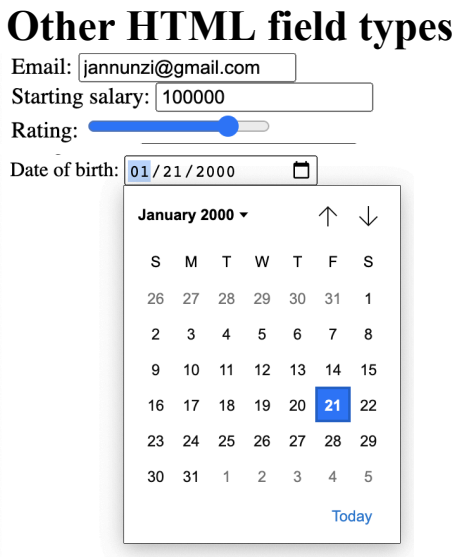
```
<h4>Other HTML field types</h4>

<label htmlFor="wd-text-fields-email"> Email: </label>
<input type="email"
       placeholder="jdoe@somewhere.com"
       id="wd-text-fields-email"/><br/>

<label htmlFor="wd-text-fields-salary-start"> Starting salary:</label>
<input type="number"
       value="100000"
       placeholder="1000"
       id="wd-text-fields-salary-start"/><br/>

<label htmlFor="wd-text-fields-rating"> Rating: </label>
<input type="range"
       value="4"
       max="5"
       placeholder="Doe"
       id="wd-text-fields-rating"/><br/>

<label htmlFor="wd-text-fields-dob"> Date of birth: </label>
<input type="date"
       value="2000-01-21"
       id="wd-text-fields-dob"/><br/>
```



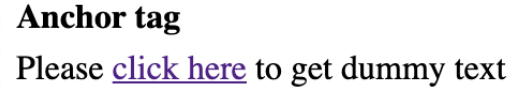
1.3.8 Implementing Navigation with the HTML Anchor Tag

The anchor tag renders text encouraging users to click on it to navigate to other websites or pages.

```
<a href="aa.com">              <!-- Use the href attribute to refer to the location of the website or other page in the
American Airlines</a>          same website. Click on the body text to navigate -->
```

To practice using anchor tags, add the following example at the end of ***page.tsx***. It creates a hyperlink that navigates to ***lipsum.com***, a website that contains dummy text. Create another link to your code repository on GitHub. Click on the links and confirm the navigation works. Give the anchor an ID value of ***wd-github***.

```
<h4>Anchor tag</h4>
Please
<a href="https://www.lipsum.com" id="wd-lipsum">click here</a>
to get dummy text<br/>
```



1.3.9 Implementing Navigation

Implementing navigation in ***Single Page Applications*** (***SPA***s) is different than with anchor tags since in SPAs don't navigate away from the current webpage, but instead show or hide screens and components based on the URL. For instance, there might be screens ***Signin*** and ***Profile*** mapped to URL ***path***s, or ***route***s ***/signin*** and ***/profile***. When the ***URL*** contains the corresponding path, then SPA applications will display the corresponding content.

As you work through the various chapters, we're going to work on various exercises discussing different topics. To practice some more navigation, create placeholders for upcoming lab exercises and implement navigation to each of the labs. Create the following ***Lab2*** and ***Lab3*** components as shown below.

app/Labs/Lab2/page.tsx

```
export default function Lab2() {
  return (
    <div>
      <h2>Lab 2</h2>
    </div>
);}
```

app/Labs/Lab3/page.tsx

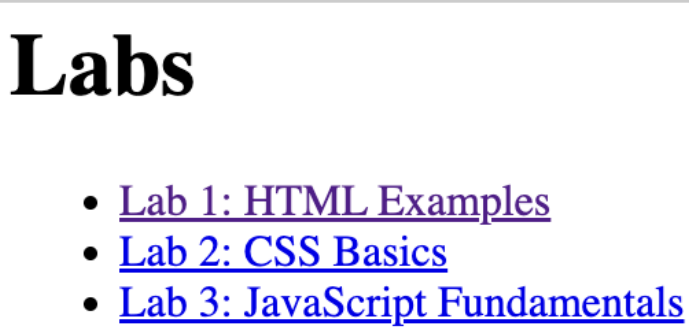
```
export default function Lab3() {
  return (
    <div>
      <h2>Lab 3</h2>
    </div>
);}
```

Navigation can be implemented with HTML ***Anchor*** tags as shown earlier, but Next.js provides the ***Link*** component which is a better option optimized for navigating within the web pages of the same web application. The ***Labs*** page shown below illustrates how to implement ***hyperlinks*** to navigate to the various ***Lab*** exercises. Confirm that navigating to http://localhost:3000/Labs displays the ***Labs*** page shown below. Also confirm that clicking the links does actually navigate to the corresponding lab pages.

app/Labs/page.tsx

```
import Link from "next/link";
export default function Labs() {
 return (
   <div id="wd-labs">
     <h1>Labs</h1>
     <ul>
       <li>
         <Link href="/Labs/Lab1" id="wd-lab1-link">
           Lab 1: HTML Examples </Link>
       </li>
       <li>
         <Link href="/Labs/Lab2" id="wd-lab2-link">
           Lab 2: CSS Basics </Link>
       </li>
       <li>
         <Link href="/Labs/Lab3" id="wd-lab3-link">
           Lab 3: JavaScript Fundamentals </Link>
       </li>
     </ul>
   </div>
);}
```

http://localhost:3000/Labs



1.3.10 Implementing Layouts

In Next.js, layouts are a powerful feature that allow developers to create consistent and reusable UI structures across multiple pages or routes. By utilizing the ***layout.tsx*** file within the app directory, you can define a shared layout that wraps specific pages or child routes, ensuring a uniform appearance and functionality. The code below demonstrates a ***LabsLayout*** component that integrates a ***table of contents*** (***TOC***) as a sidebar navigation, alongside the main content rendered via the ***children*** prop. The TOC component, defined in ***TOC.tsx***, generates a list of navigational links, which are embedded into the layout to create a cohesive user experience across the "Labs" section of the application.

app/Labs/TOC.tsx

```
import Link from "next/link";
export default function TOC() {
 return (
   <ul>
     <li>
       <Link href="/Labs" id="wd-lab1-link">
         Home </Link>
     </li>
     <li>
       <Link href="/Labs/Lab1" id="wd-lab1-link">
         Lab 1 </Link>
     </li>
     <li>
       <Link href="/Labs/Lab2" id="wd-lab2-link">
         Lab 2 </Link>
     </li>
     <li>
       <Link href="/Labs/Lab3" id="wd-lab3-link">
         Lab 3 </Link>
     </li>
   </ul>
);}
```

app/Labs/Layout.tsx

```
import { ReactNode } from "react";
import TOC from "./TOC";

export default function LabsLayout({
 children,
}: Readonly<{ children: ReactNode }>) {
 return (
   <table>
     <tbody>
       <tr>
         <td valign="top" width="100px">
           <TOC />
         </td>
         <td valign="top">{children}</td>
       </tr>
     </tbody>
   </table>
);}
```

The ***children*** prop in the ***LabsLayout*** component is dynamically populated with the content of the ***page.tsx*** file located in the ***app/Labs*** directory. In this case, the ***page.tsx*** file defines the ***Labs*** component, which renders a main page for the "***Labs***" section containing a list of links to individual the lab page. When the ***/Labs*** route is accessed, the ***LabsLayout*** wraps this content, placing the ***TOC*** component in a sidebar column and the ***Labs*** component's content in the main column, ensuring a consistent layout across all pages under the ***/Labs*** route. Confirm that navigating to http://localhost:3000/Labs shows the content below.

Figure 1.3.10 - Labs with layout

1.4 Prototyping the React Kambaz User Interface with HTML

So far the exercises in this chapter have served to practice various aspects of ***HTML*** and gain confidence with building user interfaces. The following sections will put those skills to use by building ***Kambaz***, a Web site based on a popular ***Online Learning Management System*** (***OLMS***). This chapter will focus on implementing some simple, prototype versions of the more common screens and later chapters will incrementally improve on the prototype. Create a new directory ***src/Kambaz*** and do all work in there.

1.4.1 Implementing the Kambaz Landing Page

A ***landing page*** is the default screen, or entry point of a Website or application. Create the ***Kambaz*** component that will serve as the entry point to the ***Kambaz*** application. Create the component in ***app/Kambaz/page.tsx*** as shown below.

app/Kambaz/page.tsx

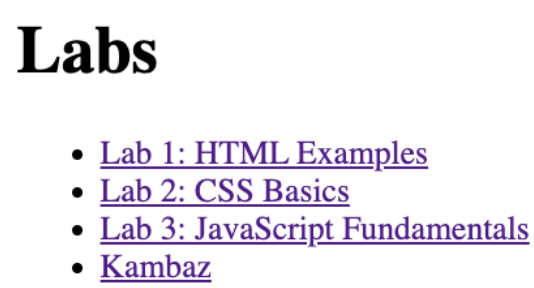
```
export default function Kambaz() {
  return (
    <div id="wd-kambaz">
      <h1>Kambaz</h1>
    </div>
);}
```

To make the ***Kambaz*** page the default page, remove the root ***app/page.tsx*** and rename the ***Kambaz*** directory by surrounding the name with parenthesis like so: ***(Kambaz)***. Confirm that navigating to http://localhost:3000 displays the ***Kambaz*** page. Add the new ***Kambaz*** page to the ***TOC.tsx*** and the ***Labs*** page to navigate to the default ***Kambaz*** page. Confirm that navigating to the ***Labs*** page now shows links to the ***Kambaz*** page. Also confirm that clicking the new ***Kambaz*** links actually navigate to the ***Kambaz*** page.

app/Labs/TOC.tsx

```
import Link from "next/link";
export default function TOC() {
 return (
   <ul>
     <li>
       <Link href="/Labs" id="wd-lab1-link">
         Home </Link> </li>
     ...
     <li>
       <Link href="/" id="wd-lab3-link">
         Kambaz </Link> </li>
   </ul>
);}
```

http://Localhost:3000/Labs



1.4.2 Implementing the Kambaz Account Screens

The ***Account Screens*** provide users access to their personal information and all related data such as courses they are registered for and courses they might be teaching. Users use the ***Sign Up*** screen to register with the application. They can then use the ***Sign In*** screen to identify themselves and access their ***Profile*** screen to view and edit personal information. This section describes how to create the ***Sign in***, ***Sign up***, and ***Profile*** screen illustrated below.

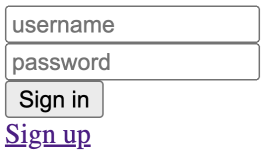


Figure 1.4.2.a
The Sign in screen

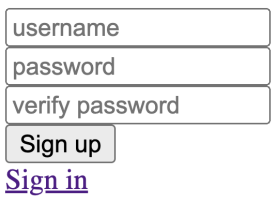


Figure 1.4.2.b
The Sign up screen

Figure 1.4.2.c
The Profile screen

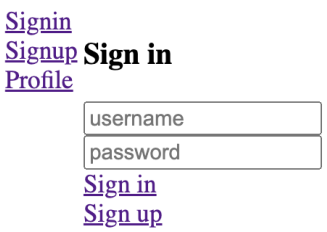


Figure 1.4.2.d
Account Navigation

1.4.2.1 Implementing the Kambaz Sign In Screen HTML Layout

Signing into an application usually consists of providing credentials such as a ***username*** and ***password***. The following component below declares input fields where users can enter their ***username*** and ***password*** illustrated in Figure 1.4.2.a.

app/(Kambaz)/Account/Signin/page.tsx

```
import Link from "next/link";
export default function Signin() {
 return (
   <div id="wd-signin-screen">
     <h3>Sign in</h3>
     <input placeholder="username" className="wd-username" /> <br />
     <input placeholder="password" type="password" className="wd-password" /> <br />
     <Link href="Profile" id="wd-signin-btn"> Sign in </Link> <br />
     <Link href="Signup" id="wd-signup-link"> Sign up </Link>
   </div>
);}
```

Create an ***Account*** screen that redirects to the ***Signin*** screen so that it's the default screen. Confirm that navigating to http://localhost:3000/Account does indeed redirect to http://localhost:3000/Account/Signin.

app/(Kambaz)/Account/page.tsx

```
import { redirect } from "next/dist/client/components/navigation";

export default function AccountPage() {
 redirect("/Account/Signin");
}
```

In the ***Kambaz*** page, redirect to the ***Signin*** screen so that it's also the default screen. Confirm that navigating to http://localhost:3000 does indeed redirect to http://localhost:3000/Account/Signin.

app/(Kambaz)/page.tsx

```
import { redirect } from "next/navigation";
export default function Kambaz() {
  redirect("/Account/Signin");
  return (
    <div id="wd-kambaz">
      <h1>Kambaz</h1>
    </div>
  );
}
```

1.4.2.2 Implementing the Kambaz Sign Up Screen HTML Layout

Implement a ***Signup*** screen component as shown below and illustrated in Figure 1.4.2.b. Import the ***Signup*** screen in the ***Account*** screen right below the ***Signin*** screen. Confirm that the ***Kambaz*** screen displays the ***Account*** screen and renders the ***Signin*** and ***Signup*** screens.

app/(Kambaz)/Account/Signup/page.tsx

```
import Link from "next/link";
export default function Signup() {
  return (
    <div id="wd-signup-screen">
      <h3>Sign up</h3>
      <input placeholder="username" className="wd-username" /><br/>
      <input placeholder="password" type="password" className="wd-password" /><br/>
```

```
      <input placeholder="verify password"
             type="password" className="wd-password-verify" /><br/>
      <Link  href="Profile" > Sign up </Link><br />
      <Link  href="Signin" > Sign in </Link>
    </div>
);}
```

1.4.2.3 Implementing the Kambaz Profile Screen HTML Layout

Implement the ***Profile*** screen as shown below and illustrated in Figure 1.4.2.c. Include the new screen in the ***Account*** screen below the ***Signup*** screen. Confirm that the ***Kambaz*** screen shows the ***Account*** screen with the ***Signin***, ***Signup***, and ***Profile*** screens.

app/(Kambaz)/Account/Profile/page.tsx

```
import Link from "next/link";
export default function Profile() {
  return (
    <div id="wd-profile-screen">
      <h3>Profile</h3>
      <input defaultValue="alice" placeholder="username" className="wd-username"/><br/>
      <input defaultValue="123"   placeholder="password" type="password"
             className="wd-password" /><br/>
      <input defaultValue="Alice" placeholder="First Name" id="wd-firstname" /><br/>
      <input defaultValue="Wonderland" placeholder="Last Name" id="wd-lastname" /><br/>
      <input defaultValue="2000-01-01" type="date" id="wd-dob" /><br/>
      <input defaultValue="alice@wonderland" type="email" id="wd-email" /><br/>
      <select defaultValue="FACULTY" id="wd-role">
        <option value="USER">User</option>       <option value="ADMIN">Admin</option>
        <option value="FACULTY">Faculty</option> <option value="STUDENT">Student</option>
      </select><br/>
      <Link href="Signin" > Sign out </Link>
    </div>
);}
```

1.4.2.4 Implementing Navigation Between the Kambaz Account Screens

The ***Account*** screen navigation currently shows all three screens: ***Signin***, ***Profile***, and ***Signup***, which makes no sense. Add routes to the ***Account*** and ***Kambaz*** screens so that these screens only show when users navigate to them. In the ***Account*** screen, add the following routes so that the ***Signin*** screen is the default.

Also confirm that clicking the ***Sign in*** button navigates to the ***Profile*** screen and clicking ***Sign Out*** in the ***Profile*** screen navigates back to the ***Sign in*** screen. Further confirm that you can navigate between the ***Sign Up*** and the ***Sign in*** screen.

1.4.2.5 Implementing an Navigation Sidebar for the Account Screens

Create an ***Account Navigation*** component providing navigation links to the ***Sign in***, ***Sign up***, and ***Profile*** screens as shown below.

app/(Kambaz)/Account/Navigation.tsx

```
import Link from "next/link";
export default function AccountNavigation() {
 return (
   <div id="wd-account-navigation">
     <Link href="Signin"> Signin </Link> <br />
     <Link href="Signup"> Signup </Link> <br />
     <Link href="Profile"> Profile </Link> <br />
   </div>
);}
```

In the ***Account*** screen add a ***table*** to layout the ***AccountNavigation*** component on a column on the left and the routes on the right side column as shown below. Confirm that the ***Sign in*** screen is the default screen as shown in Figure 1.4.2.d. Also confirm that clicking the links in the ***Account Navigation*** sidebar actually navigates to the screens

app/(Kambaz)/Account/layout.tsx

```
import { ReactNode } from "react";
import AccountNavigation from "./Navigation";
export default function AccountLayout({ children }: Readonly<{ children: ReactNode }>) {
 return (
   <div id="wd-kambaz">
     <table>
       <tbody>
         <tr>
           <td valign="top">
             <AccountNavigation />
           </td>
           <td valign="top" width="100%">
             {children}
           </td>
         </tr>
       </tbody>
     </table>
   </div>
);}
```

1.4.3 Implementing the Kambaz Dashboard Screen

The ***Kambaz Dashboard*** lists courses as illustrated below, giving access to courses a user is enrolled in as a student and courses a faculty is teaching.

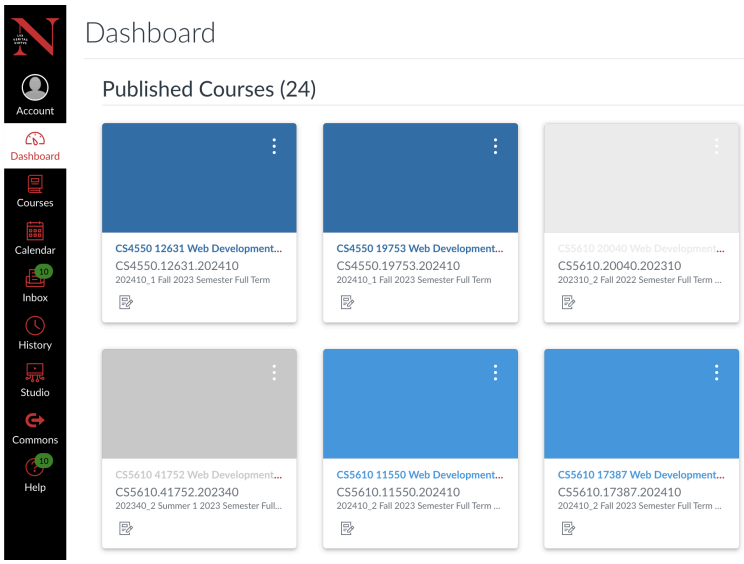


Figure 1.4.3 - The Dashboard Screen

Clicking on a course navigates to that specific course. In a new ***page.tsx*** file under ***app/(Kambaz)/Dashboard***, create the ***Kambaz Dashboard*** displaying at least 7 courses. Use the ***Dashboard/page.tsx*** file shown below as an example. Feel free to come up with additional courses. Download images for each course and save them to ***public/images***. The code below uses an image of the React logo.

app/(Kambaz)/Dashboard/page.tsx

```
import Link from "next/link";
import Image from "next/image";
export default function Dashboard() {
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h2 id="wd-dashboard-published">Published Courses (12)</h2> <hr />
      <div id="wd-dashboard-courses">
        <div className="wd-dashboard-course">
          <Link href="/Courses/1234" className="wd-dashboard-course-link">
            <Image src="/images/reactjs.jpg" width={200} height={150} />      {/* Dashboard Title */}
            <div>                                                             {/* Published Courses */}
              <h5> CS1234 React JS </h5>                                      {/* Course 1 */}
              <p className="wd-dashboard-course-title">
                Full Stack software developer
              </p>
              <button> Go </button>
            </div>
          </Link>
        </div>                                                                {/* Course 2 */}
        <div className="wd-dashboard-course"> ... </div>                      {/* Course 3 */}
        <div className="wd-dashboard-course"> ... </div>                      {/* Add at least 7 courses in total */}
      </div>
    </div>
);}
```

The dashboard implemented in this chapter does not have to look exactly like the screen shot shown above. Later chapters will revisit this screen and apply ***CSS*** to style it as shown in the screen shot.

Change the ***Sign in*** link in the ***Sign In*** screen so that it navigates to the new ***Dashboard*** screen. Confirm that signing in navigates to the ***Dashboard***. Confirm you can still navigate to the ***Account*** screens with the ***Account Navigation Sidebar***.

app/(Kambaz)/Account/Signin/page.tsx

```
    ...
    <div id="wd-signin-screen">
      <h3>Sign in</h3>
      <input className="wd-username" placeholder="username" /> <br />
      <input className="wd-password" placeholder="password" type="password" /> <br />
      <Link id="wd-signin-btn" href="/Dashboard"> Sign in </Link> <br />
      <Link id="wd-signup-link" href="Signup"> Sign up </Link>
    </div>
```

1.4.3.1 Implementing the Kambaz Navigation Sidebar

The first column of the ***Kambaz Dashboard*** contains the ***Kambaz Navigation*** sidebar, a list of hyperlinks (anchors) to navigate to different parts of the ***Kambaz*** application. Users can navigate to the ***Dashboard*** by clicking ***Dashboard*** on the ***Kambaz Navigation*** sidebar. Implement the ***Kambaz Navigation*** sidebar in ***app/(Kambaz)/Navigation.tsx***, shown here on the right. Use the code below as an example. Import all necessary components.

app/(Kambaz)/Navigation.tsx

```
import Link from "next/link";
export default function KambazNavigation() {
  return (
    <div id="wd-kambaz-navigation">
      <a href="https://www.northeastern.edu/" id="wd-neu-link" target="_blank">Northeastern</a><br/>
      <Link href="/Account" id="wd-account-link">Account</Link><br/>
      <Link href="/Dashboard" id="wd-dashboard-link">Dashboard</Link><br/>
      <Link href="/Dashboard" id="wd-course-link">Courses</Link><br/>
      <Link href="/Calendar" id="wd-calendar-link">Calendar</Link><br/>
      <Link href="/Inbox" id="wd-inbox-link">Inbox</Link><br/>
      <Link href="/Labs" id="wd-labs-link">Labs</Link><br/>
    </div>
);}
```

Northeastern
Account
Dashboard
Courses
Calendar
Inbox
Labs

In a ***Kambaz*** layout, use the ***table*** element to layout the ***Kambaz Navigation*** and the ***Dashboard*** into a single row with two columns. Put the ***Kambaz Navigation*** sidebar on the left column, and the ***children*** on the right. Confirm that the ***Kambaz*** application renders as shown below.

app/(Kambaz)/Layout.tsx

```
import { ReactNode } from "react";
import KambazNavigation from "./Navigation";
export default function KambazLayout({ children }: Readonly<{ children: ReactNode }>) {
 return (
   <table>
     <tbody>
       <tr>
         <td valign="top" width="200">  <KambazNavigation /> </td>
         <td valign="top" width="100%"> {children}           </td>
       </tr>
     </tbody>
   </table>
);}
```

1.4.4 Implementing the Courses Screen

Clicking a course in the ***Dashboard*** navigates to the ***Courses*** screen displaying a course's content as shown below.

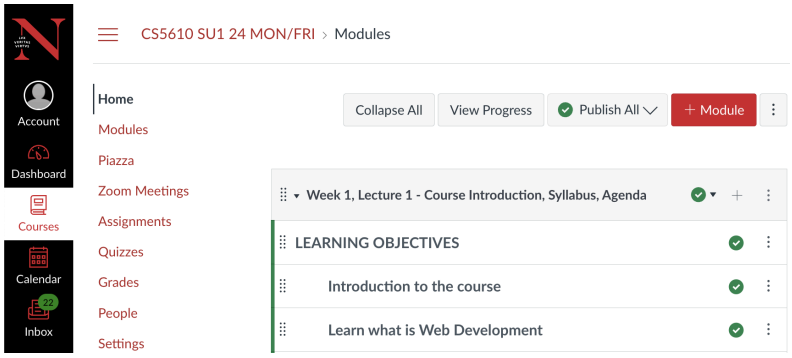


Figure 1.4.4 - The Courses Screen

Create a placeholder for the ***Course*** screen in a new file ***app/(Kambaz)/Courses/[cid]/page.tsx***.

app/(Kambaz)/Courses/[cid]/page.tsx

```
export default function Courses() {
  return (
    <div id="wd-courses">
      <h2>Course 1234</h2>
    </div>
);}
```

1.4.4.1 Implementing the Kambaz Course Navigation Sidebar

When navigating to the ***Courses*** screen, a second column displays a ***Course Navigation*** sidebar to navigate to various screens related to that course. In ***app/(Kambaz)/Courses/[cid]/Navigation.tsx*** create the ***CourseNavigation*** sidebar as shown below.

app/(Kambaz)/Courses/[cid]/Navigation.tsx

```
import Link from "next/link";
```

```
export default function CourseNavigation() {
  return (
    <div id="wd-courses-navigation">
      <Link href="/Courses/1234/Home" id="wd-course-home-link">Home</Link><br/>
      <Link href="/Courses/1234/Modules" id="wd-course-modules-link">Modules
        </Link><br/>
      <Link href="/Courses/1234/Piazza" id="wd-course-piazza-link">Piazza</Link><br/>
      <Link href="/Courses/1234/Zoom" id="wd-course-zoom-link">Zoom</Link><br/>
      <Link href="/Courses/1234/Assignments" id="wd-course-quizzes-link">
          Assignments</Link><br/>
      <Link href="/Courses/1234/Quizzes" id="wd-course-assignments-link">Quizzes
        </Link><br/>
      <Link href="/Courses/1234/Grades" id="wd-course-grades-link">Grades</Link><br/>
      <Link href="/Courses/1234/People/Table" id="wd-course-people-link">People</Link><br/>
    </div>
  );}
```

Home
Modules
Piazza
Zoom
Assignments
Quizzes
People

Then in the ***Courses*** component, use a ***table*** to display the sidebar on the left and the ***Routes*** on the right as shown below. For now, the ***Routes*** are just placeholders for several screens containing learning materials, assignments, grades, etc. Each screen will be implement later in the chapter.

app/(Kambaz)/Courses/[cid]/layout.tsx

```
import { ReactNode } from "react";
import CourseNavigation from "./Navigation";
export default async function CoursesLayout(
  { children, params }: Readonly<{ children: ReactNode; params: Promise<{ id: string }> }>) {
 const { cid } = await params;
 return (
   <div id="wd-courses">
     <h2>Courses {cid}</h2>
     <hr />
     <table>
       <tbody>
         <tr>
           <td valign="top" width="200"> <CourseNavigation /> </td>
           <td valign="top" width="100%"> {children} </td>
         </tr>
       </tbody>
     </table>
   </div>
);}
```

1.4.5 Implementing the Kambaz Modules Screen

When a course is clicked in the ***Dashboard***, the application navigates to the ***Home*** screen of that course by default. The ***Home*** screen displays a list of modules that contain a course's material. The screen shot below on the left illustrates the first module ***Week 1, Lecture 1***, as well as providing links to ***READING*** material and ***SLIDES***. This list of modules is the same content also available in the ***Modules*** shown below. Implement the ***Modules*** screen first and then reuse it to implement the ***Home*** screen. The screen consists of three columns containing the ***Kambaz Navigation*** sidebar in the first column, a ***Course Navigation*** sidebar in the second column, and the ***Modules*** in the third column. The ***Kambaz*** and ***Course Navigation*** sidebars are already implemented. Focus now on a prototype of the ***Modules*** as shown below on the right.

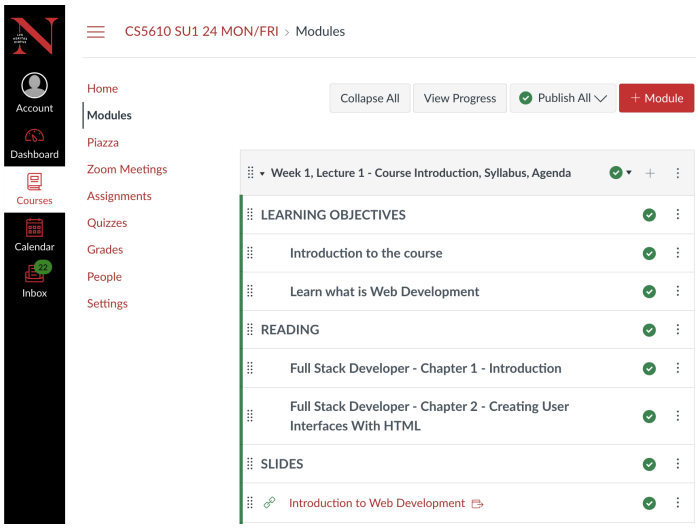


Figure 1.4.5.a - The Modules Screen Screenshot

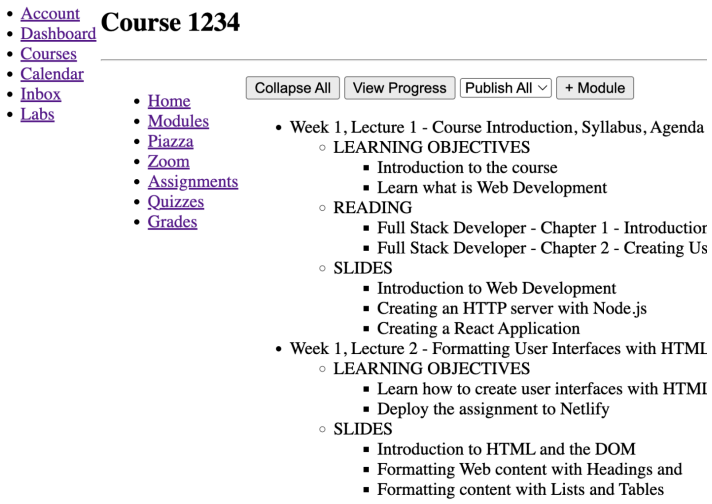


Figure 1.4.5.b - Our implementation of the Modules Screen before styling

The ***Modules*** screen can be implemented as a set of nested lists where the top level list consists of a list of ***modules***. Each module item can contain a nested list of ***lessons***. Each lesson in turn can contain a list of different content items. The code sample below illustrates how nested lists could be used to implement the ***Modules***.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
export default function Modules() {
  return (
    <div>
      {/* Implement Collapse All button, View Progress button, etc. */}
      <ul id="wd-modules">
        <li className="wd-module">
          <div className="wd-title">Week 1</div>
          <ul className="wd-lessons">
            <li className="wd-lesson">
              <span className="wd-title">LEARNING OBJECTIVES</span>
              <ul className="wd-content">
                <li className="wd-content-item">Introduction to the course</li>
                <li className="wd-content-item">Learn what is Web Development</li>
              </ul>
            </li>
          </ul>
        </li>
        <li className="wd-module">
          <div className="wd-title">Week 2</div>
        </li>
        <li className="wd-module">
          <div className="wd-title">Week 3</div>
        </li>
      </ul>
    </div>
);}
```

1.4.6 Implementing the Kambaz Course Home Screen

Clicking on a course in the ***Dashboard*** navigates to the course's ***Home*** screen shown below on the left. The ***Home*** screen contains four columns containing the ***Kambaz*** and ***Courses Navigation*** sidebars in the first and second column, the ***Modules*** in the third column, and a ***Course Status*** sidebar on the last column. The content for the first three columns has already been implemented. To implement the ***Home*** screen, implement the ***Course Status*** sidebar shown here on the right. Combine the sidebar with the other existing components to create the ***Home*** screen as shown below on the right.

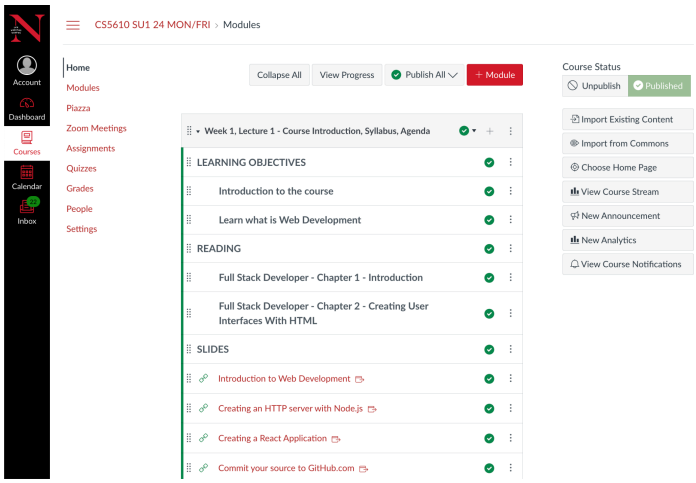


Figure 1.4.6.a - The Home Screen Screenshot

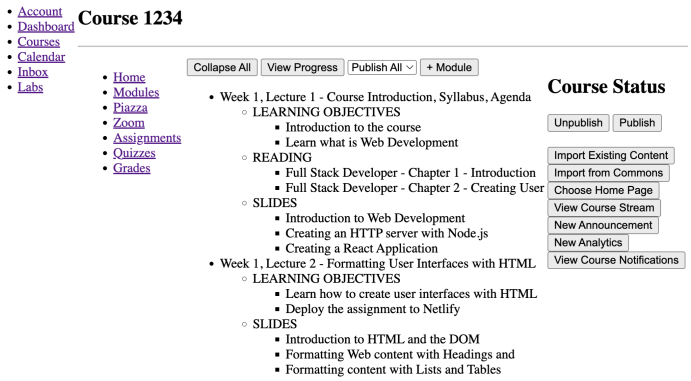


Figure 1.4.6.b - Our implementation of the Home Screen before styling

Complete the ***Course Status*** sidebar on your own in a new file in ***app/(Kambaz)/Courses/Home/Status.tsx***.

app/(Kambaz)/Courses/[cid]/Home/Status.tsx

```
export default function CourseStatus() {
  return (
    <div id="wd-course-status">
      <h2>Course Status</h2>
      <button>Unpublish</button> <button>Publish</button>
      {/* Complete on your own */}
      <button>View Course Notifications</button>
    </div> );}
```

Combine the ***Course Status*** with the ***Modules*** to create the ***Home*** screen as shown below.

app/(Kambaz)/Courses/[cid]/Home/page.tsx

```
import Modules from "../Modules/page";
import CourseStatus from "./Status";
export default function Home() {
 return (
   <div id="wd-home">
     <table>
       <tbody>
         <tr>
           <td valign="top" width="70%"> <Modules /> </td>
           <td valign="top"> <CourseStatus /> </td>
         </tr>
       </tbody>
     </table>
   </div>
);}
```

Replace the ***Home*** heading placeholder in the ***Courses*** screen with the actual ***Home*** component as shown below.

app/(Kambaz)/Courses/[cid]/page.tsx

```
import { redirect } from "next/navigation";

export default async function CoursesPage({ params, }: { params: Promise<{ cid: string }>; }) {
 const { cid } = await params;
 redirect(`/Courses/${cid}/Home`);
}
```

Confirm that navigating to a course from the dashboard displays the ***Home*** screen similar to the one shown below.

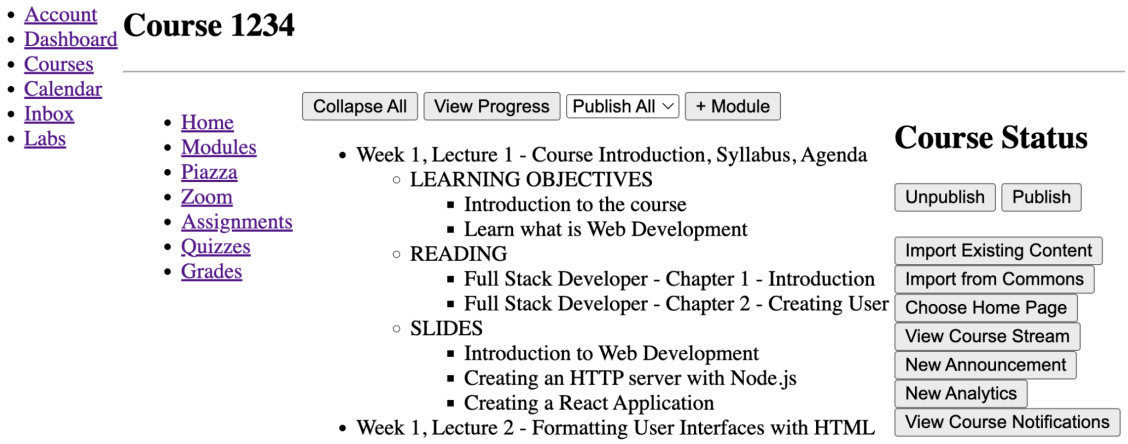


Figure 1.4.6.c - The Home Screen

1.4.7 Assignments Screen (On Your Own)

So far this chapter has described how to create various ***Kambaz*** screens in detail. To confirm your skills, this section challenges you to implement a couple of screens on your own. Some sample code is provided as suggestions, but feel free to ignore the code and write your own.

The ***Assignments Screen*** lists all the assignments students are required to complete throughout a course. A screenshot of the assignments screen is shown below on the left. To navigate to the ***Assignments Screen***, users navigate to the course from the ***Dashboard***, and then click on the ***Assignments*** link in the ***Course Navigation*** sidebar. Assignments are grouped into categories such as ***ASSIGNMENTS***, ***QUIZZES***, ***EXAMS***, and ***PROJECT***, and are shown in the third column as shown below. Here only the ***ASSIGNMENTS*** category is shown and required. Implement a component called ***Assignments*** shown below on the right as a prototype of the ***Assignments*** screen on the left.

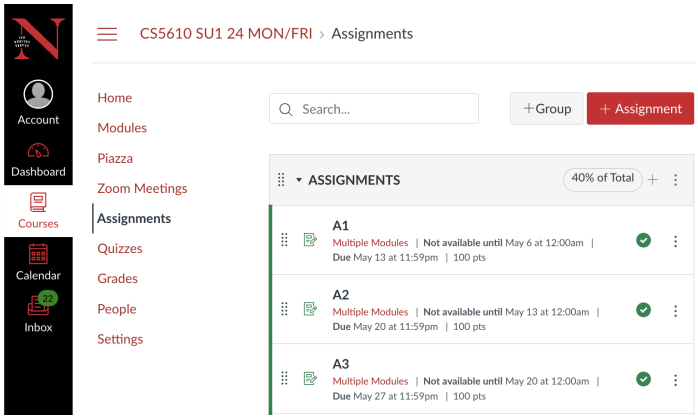


Figure 1.4.7.a - The Assignments Screen Screenshot

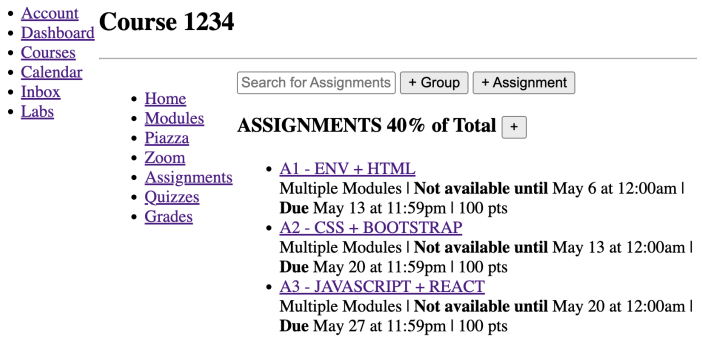


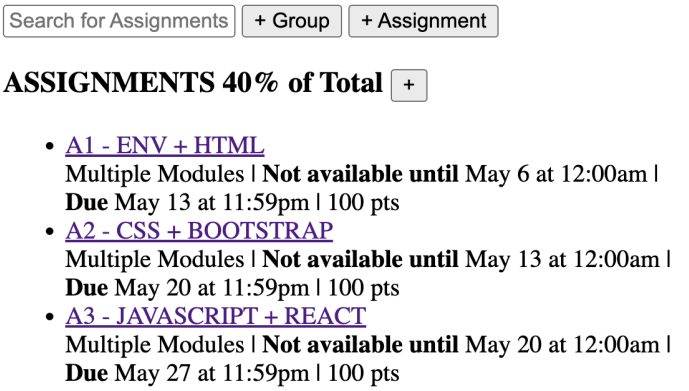
Figure 1.4.7.b - Our implementation of the Assignments Screen before styling

Feel free to use the following starter code to implement the ***Assignments*** component. If you prefer to build your own version from scratch, feel free to ignore the code provided. Be aware that if you decide to use the code provided, later chapters will build on the code from prior chapters. Your own implementation in earlier chapters might not be compatible with code in later chapters. Make the component look as shown below on the right. Make sure to keep the ***id*** and

className attributes provided and all ***li*** (line items) in the ***ul*** (unordered list) must have the ***className*** set to ***wd-assignment-list-item***.

app/(Kambaz)/Courses/[cid]/Assignments/page.tsx

```
export default function Assignments() {
  return (
    <div id="wd-assignments">
      <input placeholder="Search for Assignments"
             id="wd-search-assignment" />
      <button id="wd-add-assignment-group">+ Group</button>
      <button id="wd-add-assignment">+ Assignment</button>
      <h3 id="wd-assignments-title">
        ASSIGNMENTS 40% of Total <button>+</button> </h3>
      <ul id="wd-assignment-list">
        <li className="wd-assignment-list-item">
          <a href="/Courses/1234/Assignments/123"
             className="wd-assignment-link" >
            A1 - ENV + HTML
          </a> </li>
        <li className="wd-assignment-list-item">
          {/* Complete On Your Own */}
        </li>
      </ul>
    </div>
);}
```



1.4.8 Assignment Editor Screen (On Your Own)

When clicking the title of an assignment in the ***Assignments*** screen, ***Kambaz*** navigates to the ***Assignment Editor*** screen where faculty can edit the assignment's details such as the title, points, and due date as shown below on the left. In ***app/(Kambaz)e/Courses/[cid]/Assignments/[aid]/Editor.tsx*** create the ***AssignmentEditor*** component that implements a prototype of the ***Assignment Editor*** screen as shown below on the right.

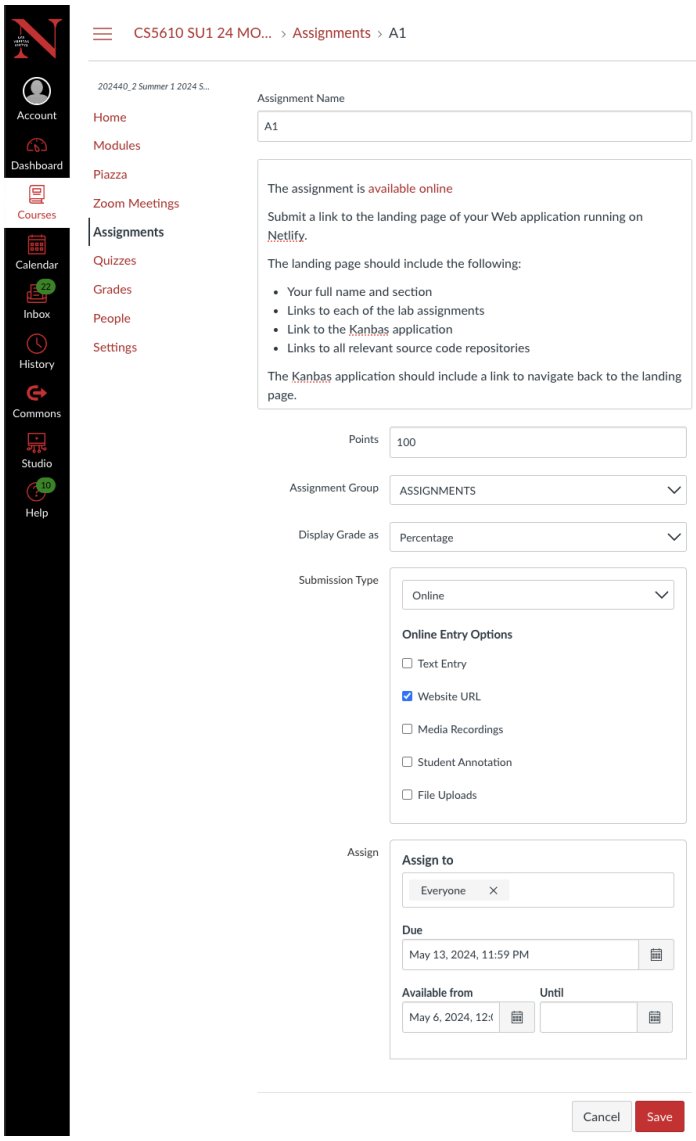


Figure 1.4.8.a - The Assignment Editor Screen Screenshot

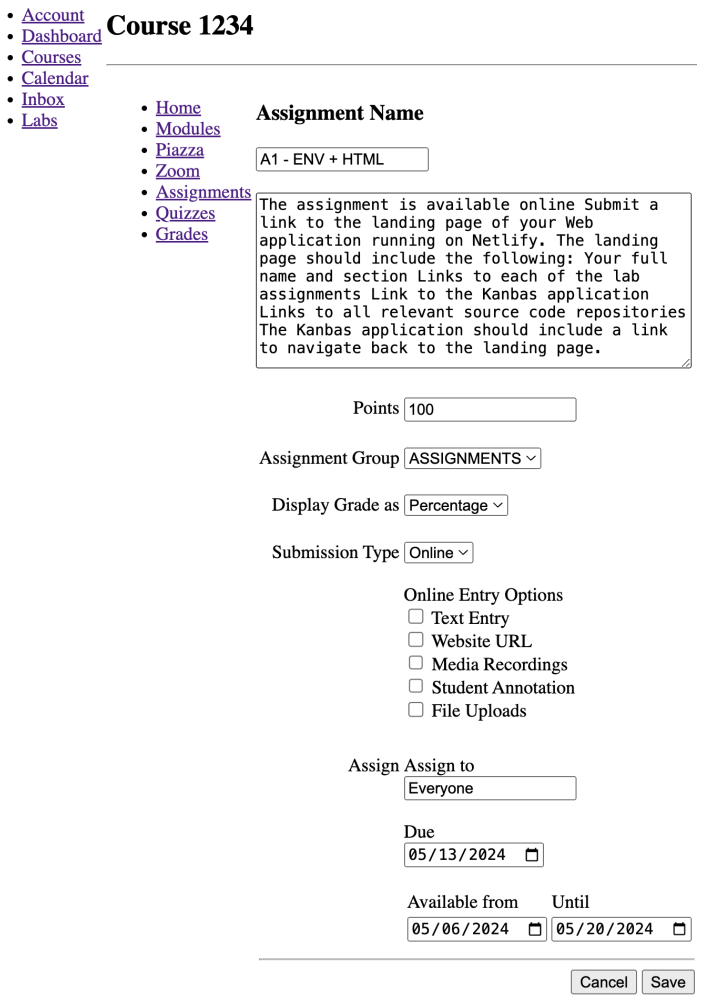


Figure 1.5.8.b - Our implementation of the Assignment Editor Screen before styling

Feel free to use the following starter code to get going with the implementation of the ***Assignment Editor*** screen.

app/(Kambaz)/Courses/[cid]/Assignments/[aid]/page.tsx

```
export default function AssignmentEditor() {
  return (
    <div id="wd-assignments-editor">
      <label htmlFor="wd-name">Assignment Name</label>                    {/* clicking this label */}
      <input id="wd-name" value="A1 - ENV + HTML" /><br /><br />          {/* selects this field */}
      <textarea id="wd-description">
        The assignment is available online Submit a link to the landing page of
      </textarea>
      <br />
      <table>
        <tr>
          <td align="right" valign="top">
            <label htmlFor="wd-points">Points</label>                     {/* clicking this label */}
          </td>
          <td>
            <input id="wd-points" value={100} />                          {/* selects this field */}
```

```
          </td>
        </tr>
        {/* Complete on your own */}
      </table>
    </div>
);}
```

All form elements (***input*** and ***select***) must have an ***id*** attribute set to the following values, based on the labels associated with the element:

wd-points	***wd-group***	***wd-display-grade-as***	***wd-submission-type***	***wd-text-entry***
wd-website-url	***wd-media-recordings***	***wd-student-annotation***	***wd-file-upload***	***wd-assign-to***
wd-due-date	***wd-available-from***	***wd-available-until***	***wd-name***	

Make sure that clicking the labels above or next to a text fields, gives focus to the field. Clicking the labels next to a checkbox, toggles the checkbox. Clicking the label above a date input field, gives focus to the date input field. That is, clicking the label or text next to the corresponding field should act on that field. In the ***Courses*** component, add a ***Route*** to navigate to the new ***Assignment Editor*** screen when clicking the title of an assignment. For now all assignments navigate to the same ***AssignmentEditor*** component and display the same information for all assignments. Later chapters will discuss how to retrieve information for a specific assignment and display the corresponding information.

1.5 Committing Code to Source Control

So far the Web application is running on your local development environment, only accessible to you. To make it available to everyone on the Web, the application needs to be deployed to a remote public server that will host and run the application, serving the content to anyone who knows the URL of the servers where the application is running. To deploy the application, first the source code needs to be committed and pushed to a source repository such as GitHub. This section demonstrates how to share the source code to a public GitHub repository and then deploy it to ***Vercel***, a popular service for hosting ***Next.js React*** Web applications.

1.5.1 Install a Git Client

Git is a popular source control software developers use to share and collaborate on projects. On macOS you already get a command line git client. You can fully interact with ***github.com*** from the terminal or ***Visual Studio Code***. On Windows OS, you'll need to install a git client from where you will be able to issue the same commands from a console. Download git for windows from https://git-scm.com/download/win, run the installer and follow the instructions. At the end of the installation you should be able to execute git commands from new console instances. All examples in this course assumes you have git installed in your OS.

You can also install a graphical git client if you prefer. There are a lot of alternatives, but the author suggests ***Sourcetree*** since it works well and consistently in both macOS and Windows. To install Sourcetree download from https://www.sourcetreeapp.com/, install, and follow instructions. We will not be covering how to use Sourcetree, but you are free to use it if you wish. All examples in this course will be using the command line git client.

1.5.2 Ignoring Files and Directories

Git can keep track of all the files in your project, but there are some files or directories that you don't want to keep track of, for instance compiled classes, libraries, etc. You can configure which files and directories you want git to ignore by listing them in a file called ***.gitignore*** at the root of your project, which should already exist. Create the file if it does not already exist. With a text editor or from IntelliJ, edit the file ***.gitignore***. Towards the top of the file, in a blank line, type the following. ***Note*** the period at the beginning of the file!!

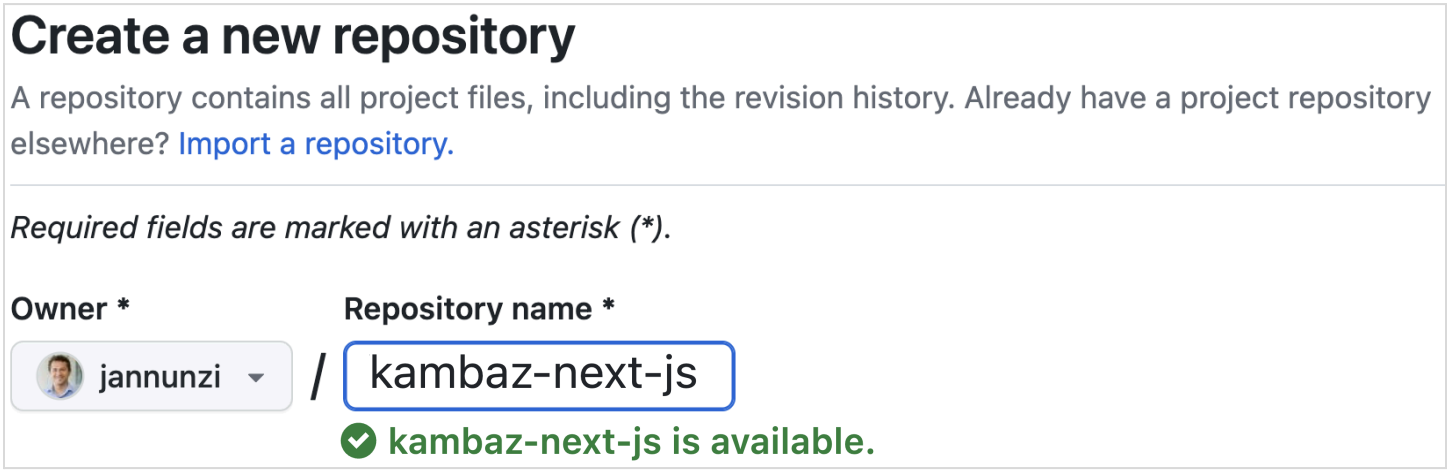
.gitignore

```
.idea
node_modules
```

This tells git that the ***.idea*** folder should be ignored because it is a directory specific to IntelliJ and not relevant to React project itself. If you are using other IDEs, then you might want to add other files or directories here relevant to your specific IDE. The ***node_modules*** folder should also be ignored since it containsmn library files that should not be added to source control. ***Note***: make sure that IDE specific folders and the ***node_modules*** folders don't make it into the repository!!

1.5.3 Creating a Remote Source Repository

If you don't already have an account on ***github.com***, create a new account or use an account you already have at ***github.com***. Do not use the university's github or your work's source control if you already have one. Login to your ***github.com*** account and click the ***New*** button to create a new ***public repository*** called ***kambaz-next-js***, just like the name of the React project you created earlier. Here's an example on how it looks on my laptop. Your username "***jannunzi***" will obviously be different.



Once you create the remote repository on github, it will display commands on how to commit and push your code from your computer up to the remote repository. The commands will be similar to the ones shown below. The username "***jannunzi***" will obviously be different.

1.5.4 Adding and Committing Code to a Remote Repository

Now that we have a remote source repository, let's add, commit and push our code to the repository. From the terminal or console, make sure you are in the ***kambaz-next-js*** directory and then type the following commands which are based on the commands git suggested. Ignore the commentary on the right. Also, your actual URL below will differ for you, so ***don't blindly copy the example below***. Use the commands git suggested for you.

```
git  init                                                    # initializes local repository
git  add  .                                                  # adds all files to repository
git  commit  -m   "first commit"                             # commits files with message
git  remote  add  origin https://github.com/jannunzi/kambaz-next-js.git   # adds remote repository
git  push    -u   origin main                                # copies local files to remote
```

Refresh the remote repository on github.com and confirm that the files are now available there

1.5.5 Using Personal Access Tokens

While pushing to GitHub you might encounter an error stating that ***password authentication was removed on August 13, 2021***. One way to fix this is to generate a ***personal access token*** and use that instead of your password. To generate a personal access token go to your GitHub ***Settings***, and then ***Developer Settings***. Click on ***Personal access tokens*** and then on ***Generate new token***. Enter a short description in the ***Note*** field, select ***No expiration*** for ***Expiration***, and grant all access privileges by selecting all the checkboxes under ***Select scopes***. You are welcome to be more restrictive it you want. Click on ***Generate token*** and copy the long unique access token to your clipboard. Note that this will be the only opportunity to copy the token and you fail to do so you'll have to delete this token, create a brand new one, and try again. With the token copied to the clipboard, try pushing again to GitHub, but this time paste the token when asked for a password.

If you are not being asked for a password then GitHub might be using a cached authentication. To clear cached GitHub authentications on Windows go to the ***Credential Manager***, click on ***Windows Credentials***, click the GitHub credentials, and click ***Remove***. This time github should ask for your username and password again when trying to push. Paste the access token when asked for the password. To clear cached authentications on macOS, go to your ***Key Chain*** and search for github. Remove the GitHub key chain.

1.6 Deploying Next.js Projects to the Web

Create an account at https://vercel.com if you don't already have one. Choose your preferred authentication method, e.g., Google, GitHub, or email verifying your identify via email, phone or text. Import the Git repository created earlier, authorizing Vercel to interact with your GitHub repositories. In the **Import Git Repository** dialog, search for the repository by name and click ***Import***. In the ***New Project*** dialog verify the ***Project Name*** is the same as the name in your repo, and the ***Framework Preset*** is ***Next.js***, then click ***Deploy***. In the ***Deployment*** dialog, open the ***Build Logs*** so you can follow along how Vercel attempts to deploy your application, paying attention to cloning the GitHub project, building, and deploying. If any errors are listed in the logs, fix them in your code, commit, push again, and a new Vercel deployment will kick off automatically. Fix all bugs in the log until there's a successful deployment and you see a ***Congratulations*** message. Click on the ***Continue to Dashboard*** to see the ***Production Deployment*** in the ***Overview*** tab. Under the ***Domains*** label, click the URL to view your application running on Vercel. To see all your failed and successful deployments, click the ***Deployments*** tab which lists your last deployment at the top. Navigate to your latest deployment by clicking it and notice the ***Status***, ***Domains***, ***Source***, ***Deployment Configuration***, and ***Build Logs***. The ***Domains*** section contains several URLs of which we are only interested in the first and it's this URL which you will submit for this assignment. All URLs navigate to the same deployed application, but the URLs are different capturing useful information regarding the ***author***, ***branch***, and ***commit***.

1.7 Conclusion

By the end of this chapter, you'll be expected to have completed the following tasks.

1. Downloaded and install the latest ***Node.js*** as described.
2. Created a React application called ***kambaz-next-js*** as described.
3. Completed all the lab exercises.
4. Prototyped the Kambaz React user interface with HTML.
5. Pushed the source code of the React application ***kambaz-next-js*** to a remote source repository in ***GitHub.com*** as described.
6. Confirm the ***Labs/page.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kambaz. Add a link to your repository in GitHub. The link should have an ***ID*** attribute with a value of ***wd-github***.
7. In ***Labs/page.tsx***, add your full name: first name first and last name second. Use the same name as in Canvas.
8. Deployed the ***kambaz-next-js*** React application to ***Vercel*** as described.
9. As a deliverable in ***Canvas***, submit the URL to your React application running on Vercel.

Chapter 2 - Styling Web Pages with CSS

Web pages are formatted with ***HTML***, declaring how browsers render text in various ***styles***, that is, the collection of visual attributes of text such as font, size, and color. For instance, ***heading tags*** format their text to be much larger than their surrounding plain text. ***Paragraph tags*** add vertical spacing. ***List tags*** enumerate or bullet the line items they surround. ***Table tags*** format data into rows and columns. The default styling of Web pages is determined by a browser's ***style sheet*** written in ***CSS*** (***Cascade Style Sheets***), a declarative language that ***declares*** how each ***HTML*** element styles the text they are applied to.

Developers use ***CSS*** to write custom ***stylesheet documents*** that override the default styling. ***CSS*** works by referring to different parts of the ***DOM*** and configuring various style attributes such as foreground and background color, font, alignment, spacing, borders, paddings, etc. This chapter demonstrates how to style and layout Web pages using CSS, a powerful declarative language.

Various patterns and best practices have evolved over time in the industry that have become popular. Some of these have been collected into commercial and open source libraries such as Bootstrap, Foundation, Tailwind, Material Design, and Bulma. All these libraries define a set of ***CSS rules*** that can be readily applied to achieve a professional look and feel, powerful layouts, and ***responsive designs***. Using a library consists of becoming familiar with the CSS rules and applying them appropriately to HTML to achieve a particular visual goal. For this course we are going to be using the ***Bootstrap*** CSS library, but feel free to explore and use other libraries for your ***final project***.

The next section gives an opportunity to practice various ***CSS*** concepts and the ***Bootstrap*** library. After completing the practice exercises, apply the skills in the ***Kambaz*** section to style the Web application started in the previous chapter. Create a new branch called ***a2*** and do all the work described in this chapter in that branch. When done, add, commit and push the branch to GitHub. Deploy the new branch to ***Vercel*** and confirm it's available in a new URL, based on the branch name ***a2***.

2.1 Styling React Components with CSS (Cascading Style Sheets)

This section introduces several CSS, Bootstrap, and React Icons exercises to practice and learn how to style HTML documents. Use the same project created the last chapter. Using ***VS Code***, open the project created in the previous chapter (***kambaz-next-js***), and complete all the exercises under the ***src*** directory. If not already done, under the ***app/Labs*** directory, create a new directory called ***Lab2***, create ***page.tsx*** under ***app/Labs/Lab2*** and do all work in a new ***page.tsx*** file. Make sure to add links in ***app/Labs/page.tsx*** and ***app/Labs/TOC.tsx*** to the new exercises in ***app/Labs/Lab2/page.tsx***. Confirm you can navigate to the new exercises from the table of content.

2.1.1 Styling HTML Tags with the Style Attribute

An HTML tag's ***style*** attribute can configure the look and feel of the tag by changing the values of its style ***properties*** as shown below. The value of the ***style*** attribute is an object in ***JSON*** format (***JavaScript Object Notation***).

```
<element style={{property1: "value1", property2: "value2"}}>
  element body
</element>
```

Examples of properties ***property1*** and ***property2*** are foreground color, background color, font size, etc. The value of the properties are primitive data types such as strings or numbers, or declared variable JSON objects. To practice using the ***style*** attribute, copy and paste the example below into ***app/Labs/Lab2/page.tsx***.

app/Labs/Lab2/page.tsx

```
export default function Lab2() {
  return (
    <div id="wd-lab2">
      <h2>Lab 2 - Cascading Style Sheets</h2>
      <h3>Styling with the STYLE attribute</h3>
      <p style={{ backgroundColor: "blue",
                  color: "white" }}>
        Style attribute allows configuring look and feel
        right on the element. Although it's very convenient
        it is considered bad practice and you should avoid
        using the style attribute
      </p>
    </div>);}
```

The exercise above styles the paragraph tag with its ***style*** attribute, changing the background color by setting the ***backgroundColor*** property to ***blue*** and also changing the foreground color to white by setting the ***color*** property to ***white***. There are 100s of style attributes of which we'll only cover the most common [REF].

2.1.2 Importing CSS Documents from React

Instead of changing styles within HTML, it is a ***best practice*** to do all styling configuration in separate CSS files and then import the files. To practice importing ***CSS*** files, create a brand new file called ***app/Labs/Lab2/index.css*** in the same directory of the ***app/Labs/Lab2/page.tsx*** document, and copy and paste the following content.

app/Labs/Lab2/index.css

```
p {
   background-color: green;
   color: white;
}
```

Then, remove the ***style*** attribute highlighted in red below and instead, import the ***index.css*** file highlighted in green. Confirm that the paragraph now has a green background.

app/Labs/Lab2/page.tsx

```
import "./index.css";
export default function Lab2() {
  return (
    <div id="wd-lab2">
      <h2>Lab 2 - Cascading Style Sheets</h2>
      <h3>Styling with the STYLE attribute</h3>
      <p style={{ backgroundColor: "blue", color: "white" }}>
        Style attribute allows configuring look and feel
        right on the element. Although it's very convenient
        it is considered bad practice and you should avoid
        using the style attribute</p>
    </div> );}
```

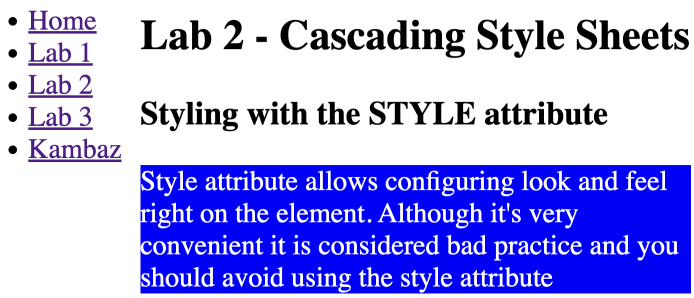


Figure 2.1.2.a - Styling with the STYLE attribute

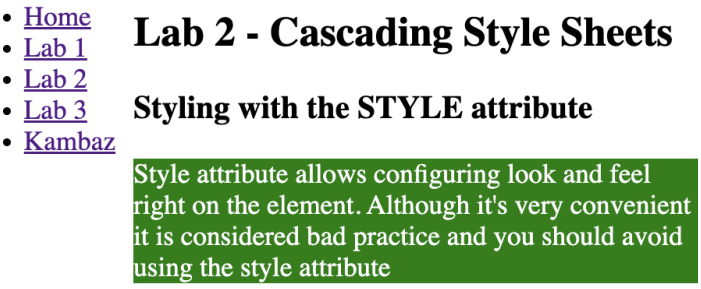


Figure 2.1.2.b - Styling with the CSS file

2.1.3 Selecting HTML Content with CSS ID Selectors

The CSS rules in previous exercises styled all paragraphs by using the name of the tag ***p*** and then specifying the style property values. Instead of changing the look and feel of all the tags of the same name, e.g., ***p***, we can refer to a specific tag by their ID, an attribute that uniquely identifies elements in a document. To practice using ***ID selectors***, in ***index.css***, comment out or delete the paragraph CSS rule highlighted in red as shown below, and add the two CSS rules referring to paragraphs with IDs ***id-selector-1*** and ***id-selector-2***, highlighted in green.

app/Labs/Lab2/index.css

```
/*p {
 background-color: green;
 color: white;}*/
p#wd-id-selector-1 {
 background-color: red;
 color: white;
}
p#wd-id-selector-2 {
 background-color: yellow;
 color: black;
}
```

app/Labs/Lab2/page.tsx

```
import "./index.css";
export default function Lab2() {
  return (
    <div id="wd-lab2">
      <h2>Lab 2 - Cascading Style Sheets</h2>
      <h3>Styling with the STYLE attribute</h3>
      …
      <div id="wd-css-id-selectors">
        <h3>ID selectors</h3>
        <p id="wd-id-selector-1">
Instead of changing the look and feel of all the
elements of the same name, e.g., P, we can refer to a specific element by its ID
        </p>
        <p id="wd-id-selector-2">
Here's another paragraph using a different ID and a different look and
          feel
        </p>
      </div>
    </div>
);}
```

Add the code above highlighted in green to ***page.tsx***, and confirm the page renders as shown in the figure below.

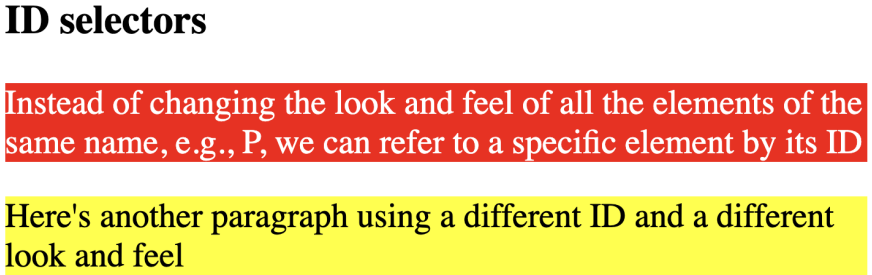


Figure 2.1.3 - Selecting with ID CSS selectors

2.1.4 Selecting HTML Content with CSS Class Selectors

Instead of using IDs to refer to a specific tag, the ***className*** attribute can select a group of tags, even if they are different

types of elements. To practice using ***class selectors***, copy the CSS rule below into ***index.css***, and the HTML at the end of ***page.tsx***. The ellipses below (...) means that there's code above and/or below that is not shown for brevity. Do not include the ellipses in actual code.

app/Labs/Lab2/index.css

```
...
.wd-class-selector {
 background-color: yellow;
 color: blue;
}
```

app/Labs/Lab2/page.tsx

```
...
<div id="wd-css-class-selectors">
  <h3>Class selectors</h3>

  <p className="wd-class-selector">
Instead of using IDs to refer to elements, you can use an element's CLASS attribute
  </p>

  <h4 className="wd-class-selector">
This heading has same style as paragraph above
  </h4>

</div>
...
```

The example above declares a selector that transforms the background and foreground color and then applies the transformation to several tags. The above example applies the style to two elements, the paragraph and the heading. Copy the code above into the stylesheet and page, and then confirm that the browser renders as shown in the figure below.

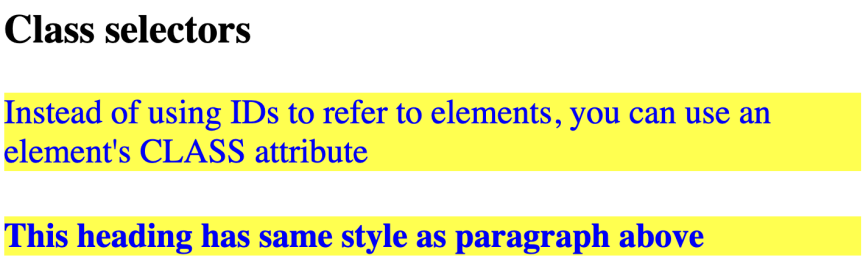


Figure 2.1.3 - Selecting with CSS class selectors

2.1.5 Selecting HTML Content Based on the Document Structure

Selectors can be combined to refer to tags in particular places in the document. A set of selectors separated by a ***space*** can refer to elements in a hierarchy. For instance: ***.selector1 .selector2 { … }*** refers to an element whose class is ***.selector2*** and is inside some ***ancestor*** whose class is ***.selector1***. Alternatively, separating class selectors with the ">" character establishes a direct parent/child relationship. To practice selecting elements using a set of selectors, copy the following content in ***page.tsx***. The code below does not show ellipses as previous examples, but the code is intended to be copied at the end of the file shown. Be sure to include the code within the function's return and within the wrapping *** ***.

app/Labs/Lab2/page.tsx

```
<div id="wd-css-document-structure">                          {/* this is parent element with selector
  <div className="wd-selector-1">                               .wd-selector-1
    <h3>Document structure selectors</h3>
    <div className="wd-selector-2">                           .wd-selector-2 is a direct child of
      Selectors can be combined to refer elements in particular  .wd-selector-1
      places in the document
      <p className="wd-selector-3">                           .wd-selector-3 is  a direct child of
        This paragraph's red background is referenced as      .wd-selector-2 and a descendant of
        <br />                                                .wd-selector-1
        .selector-2 .selector3<br />
        meaning the descendant of some ancestor.<br />
```

```
        <span className="wd-selector-4">                    .wd-selector-4 is a direct child of
          Whereas this span is a direct child of its parent  .wd-selector-3 and is a descendant of
        </span><br />                                       .wd-selector-2 and is a descendant of
          You can combine these relationships to create specific  .wd-selector-1
          styles depending on the document structure
      </p>                                                 */}
    </div>
  </div>
</div>
```

Now, in ***index.css***, style tags ***.wd-selector-3*** and ***.wd-selector-4*** as shown below.

app/Labs/Lab2/index.css

```
.wd-selector-1 .wd-selector-3 {                  /* refers to .wd-selector-3 as a descendant
   background-color: red;                           of .wd-selector-1   */
   color: white; }
.wd-selector-2 > .wd-selector-3 > .wd-selector-4 {   /* refers to .wd-selector-4 as a direct child
   background-color: yellow;                        of .wd-selector-3 which is a direct child
   color: blue;  }                                  of .wd-selector-2                */
```

Confirm that the page renders as shown below.

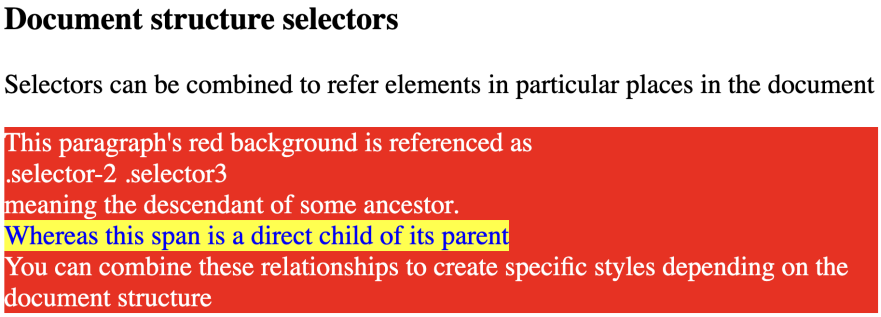


Figure 2.1.5 - Selecting HTML Content Based on the Document Structure

2.1.6 CSS Selection Rule Mechanism

CSS rules override the default styling of the tags the rules refer to. That means that there are at least two rules that apply to the same element, the default rule declared by the browser, and then the custom rule that overrides the default rule. The custom rules wins out over the default rule, but why? The ***Cascading*** in ***Cascading Style Sheets*** refer to the way CSS rules are applied to HTML elements based on a hierarchy of specificity and origin. This mechanism determines how styles are applied when there are multiple style rules that could affect an element. Here's how it works:

1. ***Specificity***: CSS rules are applied based on their specificity. This is a measure of how precise a selector is. For example, since ID selectors only refer to a single tag, whereas class selectors can refer to many tags, the ID selector is more specific than a class selector. Since a tag's name refers to all tags with that name regardless of ID or class, a class selector is more specific than a tag name selector. More specific selectors override more general ones.
2. ***Source Order***: If multiple rules have the same specificity, the last rule defined in the CSS will take precedence.
3. ***Inheritance***: Some styles are inherited by child elements from their parent elements, such as font styles. However, properties like width and margin are not inherited.

2.1.7 Styling an HTML Tag's Foreground Color with CSS

Foreground colors can be configured using the CSS ***color*** property as follows

```css
.some-css-selector {                        /* selects some DOM element    */
   color: blue;                             /* sets color property to blue */
}
```

Colors can be defined in various ways:

- As strings, e.g., ***white***, ***red***, ***blue***, etc. There are thousands of declared color names. [REF]
- As hexadecimals, e.g., #ABCDEF, where each group of 2 digits, from left to right specify the amound of red, green and blue.
- As RGB, e.g., rgb(12, 34, 56), where each digit, from left to right, specifies the amound of red, green and blue.

Here are a couple of examples:

`.the-sun { color: rgb(255,255,0); }`	`.the-sky { color: blue; }`	`.ketchup { color: #FF0000; }`

To practice working with foreground colors, copy the CSS rules below into ***index.css*** to declare several useful color classes. Copy the HTML code below into a new file called ***ForegroundColors.tsx***. Import the file into ***Lab2*** and confirm it renders as shown.

app/Labs/Lab2/index.css

```css
.wd-fg-color-black { color: black; }
.wd-fg-color-white { color: white; }
.wd-fg-color-blue { color: #7070ff; }
.wd-fg-color-red { color: #ff7070; }
.wd-fg-color-green { color: green; }
```

app/Labs/Lab2/ForegroundColors.tsx

```tsx
<div id="wd-css-colors">
  <h2>Colors</h2>
  <h3 className="wd-fg-color-blue">Foreground color</h3>
  <p className="wd-fg-color-red">
    The text in this paragraph is red but
    <span className="wd-fg-color-green">this text is green</span>
  </p>
</div>
```

Figure 2.1.7 - Foreground color

2.1.8 Styling an HTML Tag's Background Color with CSS

Similar to styling foreground color, the ***background-color*** CSS property is used to configure the background color of any tag. To practice working with background colors, copy the CSS rules shown below into ***index.css*** and the HTML code into a new file called ***BackgroundColors.tsx***. Import the file into ***Lab2*** and confirm the browser renders as shown.

app/Labs/Lab2/index.css *app/Labs/Lab2/BackgroundColors.tsx*

```css
.wd-bg-color-yellow {
   background-color: #ffff07; }
.wd-bg-color-blue {
   background-color: #7070ff; }
.wd-bg-color-red {
   background-color: #ff7070; }
.wd-bg-color-green {
   background-color: green; }
.wd-bg-color-gray {
   background-color: lightgray; }
```

```jsx
<div id="wd-css-background-colors">
  <h3 className="wd-bg-color-blue wd-fg-color-white">Background color</h3>
  <p className="wd-bg-color-red wd-fg-color-black">
    This background of this paragraph is red but
    <span className="wd-bg-color-green wd-fg-color-white">
      the background of this text is green and the foreground white
    </span>
  </p>
</div>
```

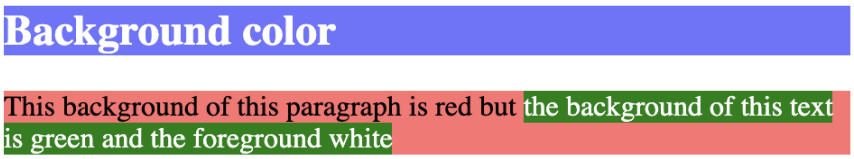


Figure 2.1.8 - Background color

2.1.9 Styling an HTML Tag's Borders with CSS

CSS has several properties that configure the look and feel of tag border around their content. Here's a sample of the properties that can be configured.

```css
.some-selector {                                /* configure border with several properties    */
  border-width: 10px;                           /* border's width. Can also provide per border */
  border-style: solid|dotted|dashed|double;     /* the style of the border                     */
  border-color: red | blue ...; }               /* the color of the border                     */
```

To practice styling borders, copy the CSS code below into ***index.css*** and the HTML code into a new file called ***Borders.tsx***. Import the new file into the ***Lab2*** component and confirm the browser renders as shown. Note how several classes are applied to the same tag to combine their cumulative styles to that tag.

app/Labs/Lab2/index.css

```css
.wd-border-fat  { border-width: 20px 30px 20px 30px; }
.wd-border-thin { border-width: 4px; }
.wd-border-solid { border-style: solid; }
.wd-border-dashed { border-style: dashed; }
.wd-border-yellow { border-color: #ffff07; }
.wd-border-red { border-color: #ff7070; }
.wd-border-blue { border-color: #7070ff; }
```

app/Labs/Lab2/Borders.tsx

```jsx
<div id="wd-css-borders">
  <h2>Borders</h2>
  <p className="wd-border-fat
     wd-border-red wd-border-solid">
    Solid fat red border</p>
  <p className="wd-border-thin
     wd-border-blue wd-border-dashed">
    Dashed thin blue border
  </p>
</div>
```

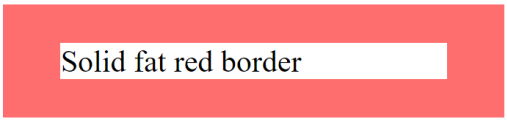


Figure 2.1.9.a - Solid fat red border

Figure 2.1.9.b - Dashed thin blue border

2.1.10 Styling an HTML Tag's Margins and Paddings with CSS

White space such as ***padding*** and ***margin*** can be styled with various CSS properties. Each side can be styled separately, e.g., top, bottom, left, right, or all the space around the content. To practice styling ***padding***, copy the CSS code below into ***index.css*** and the HTML into ***Padding.tsx***. Import and confirm the browser renders as shown.

app/Labs/Lab2/index.css

```
.wd-padded-top-left {
  padding-top: 50px;
  padding-left: 50px;
}
.wd-padded-bottom-right {
  padding-bottom: 50px;
  padding-right: 50px;
}
.wd-padding-fat {
  padding: 50px;
}
```

app/Labs/Lab2/Padding.tsx

```
<div id="wd-css-paddings">
  <h2>Padding</h2>
  <div className="wd-padded-top-left wd-border-fat
       wd-border-red wd-border-solid wd-bg-color-yellow">
    Padded top left </div>
  <div className="wd-padded-bottom-right wd-border-fat
       wd-border-blue wd-border-solid wd-bg-color-yellow">
    Padded bottom right </div>
  <div className="wd-padding-fat wd-border-fat
       wd-border-yellow wd-border-solid wd-bg-color-blue wd-fg-color-white">
    Padded all around  </div>
</div>
```

Confirm that ***Paddings.tsx*** renders as shown in the figures below.

Figure 2.1.10.a - Padding top left

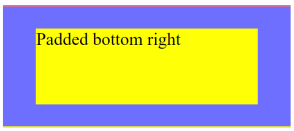


Figure 2.1.10.b - Padded bottom right

Figure 2.1.10.c - Padded all around

To practice styling ***margin***s, copy the CSS code below into ***index.css*** and the HTML into ***Margins.tsx***. Import and confirm the browser renders as shown.

app/Labs/Lab2/index.css

```
.wd-margin-bottom {
  margin-bottom: 50px;
}

.wd-margin-right-left {
  margin-left: 50px;
  margin-right: 50px;
}

.wd-margin-all-around {
  margin: 30px;
}
```

app/Labs/Lab2/Margins.tsx

```
<div id="wd-css-margins">
  <h2>Margins</h2>
  <div
    className="wd-margin-bottom wd-padded-top-left wd-border-fat
               wd-border-red wd-border-solid wd-bg-color-yellow">
    Margin bottom </div>
  <div
    className="wd-margin-right-left wd-padded-bottom-right wd-border-fat
               wd-border-blue wd-border-solid wd-bg-color-yellow">
    Margin left right </div>
  <div
    className="wd-margin-all-around wd-padding-fat wd-border-fat
               wd-border-yellow wd-border-solid wd-bg-color-blue wd-fg-color-white">
    Margin all around </div>
</div>
```

Confirm that ***Margins.tsx*** renders as shown in the figures below.

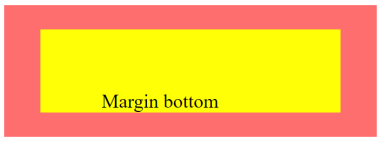


Figure 2.1.10.d - Margin bottom

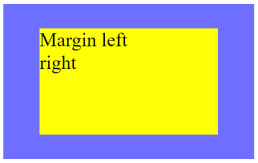


Figure 2.1.10.e - Margin left and right

Figure 2.1.10.f - Margin all around

2.1.11 Styling an HTML Tag's Corners with CSS

Tag border corners can be styled to be rounded to soften their look and feel. All corners can be styled together or individual corners can be styled separately. How rounded a corner is can be configured by setting the corner's border radius. To practice rounding some corners, copy the CSS and HTML below into ***index.css*** and ***Corners.tsx***. Import and confirm the browser renders as shown.

app/Labs/Lab2/index.css

```
.wd-rounded-corners-top {
  border-top-left-radius: 40px;
  border-top-right-radius: 40px;
}
.wd-rounded-corners-bottom {
 border-bottom-left-radius: 40px;
 border-bottom-right-radius: 40px;
}
.wd-rounded-corners-all-around {
 border-radius: 50px;
}
.wd-rounded-corners-inline {
 border-radius: 30px 0px 20px 50px;
}
```

app/Labs/Lab2/Corners.tsx

```
<div id="wd-css-borders">
  <h3>Rounded corners</h3>
  <p className="wd-rounded-corners-top wd-border-thin
     wd-border-blue wd-border-solid wd-padding-fat">
    Rounded corners on the top </p>
  <p className="wd-rounded-corners-bottom
     wd-border-thin wd-border-blue wd-border-solid wd-padding-fat">
    Rounded corners at the bottom </p>
  <p className="wd-rounded-corners-all-around
     wd-border-thin wd-border-blue wd-border-solid wd-padding-fat">
    Rounded corners all around </p>
  <p className="wd-rounded-corners-inline
     wd-border-thin wd-border-blue wd-border-solid wd-padding-fat">
    Different rounded corners </p>
</div>
```

Figure 2.1.11.a - Rounded corners on the top

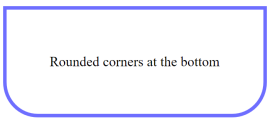


Figure 2.1.11.b - Rounded corners at the bottom

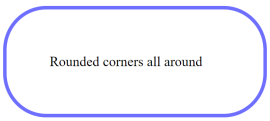


Figure 2.1.11.c - Rounded corners all around

Figure 2.1.11.d - Different rounded corners

2.1.12 Styling an HTML Tag's Dimensions with CSS

An element's dimensions can be styled with the ***width*** and ***height*** properties. To practice setting an element's dimensions, copy the CSS and HTML below into ***index.css*** and ***Dimensions.tsx***. Import and confirm the browser renders as shown.

app/Labs/Lab2/index.css *app/Labs/Lab2/Dimensions.tsx*

```css
.wd-dimension-portrait {
   width: 75px;
   height: 100px;
}
.wd-dimension-landscape {
   width: 100px;
   height: 75px;
}
.wd-dimension-square {
   width: 75px;
   height: 75px;
}
```

```
<div id="wd-css-dimensions">
  <h2>Dimension</h2>
  <div>
    <div className="wd-dimension-portrait wd-bg-color-yellow">
      Portrait
    </div>
    <div className="wd-dimension-landscape wd-bg-color-blue
                    wd-fg-color-white">
      Landscape
    </div>
    <div className="wd-dimension-square wd-bg-color-red">
      Square</div>
  </div>
</div>
```

By default the width of ***block*** elements such as div, headings, and paragraph, is the width of the parent container which often is the width of the entire window as shown below on the left. The height and width of elements can be overridden with the ***height*** and ***width*** CSS properties as shown below on the right. Notice that even though the widths of the DIVs below are now skinnier than their default widths, the taller heights still cause the divs below them to move further down.

By default, ***block*** elements like ***divs***, ***headings***, and ***paragraphs*** span the full width of their parent container, often matching the window's ***width***, as shown below on the left. You can override their height and width using CSS ***height*** and ***width*** properties, as seen below on the right. Even with narrower widths, the increased height of these DIVs pushes subsequent elements further down. This vertical spacing results from block elements stacking vertically, with their height, margins, or padding contributing to the separation.

Figure 2.1.12.a - Styling an HTML Tag's Dimensions with CSS

Figure 2.1.12.b - Styling an HTML Tag's Dimensions with CSS

2.1.13 Styling an HTML Tag's Relative Position with CSS

Browsers automatically determine the default position of all content displayed on the screen. You can override these default positions using the CSS ***position*** property. This property supports several values, with the most common ones being ***relative***, ***absolute***, ***static***, and ***fixed***. Setting the position property to ***relative*** allows you to shift an element from its default position without affecting the layout of surrounding elements. You can then use the ***top***, ***bottom***, ***left***, and ***right*** properties to specify the direction and distance of the shift. To practice positioning elements relatively, copy the CSS and HTML code provided below into ***index.css*** and ***Positions.tsx***, respectively. Verify that your browser renders the result as shown in Figure 2.1.13. Note that the background colors referenced in this exercise were defined in a previous exercise.

app/Labs/Lab2/index.css

```
.wd-pos-relative-nudge-up-right {
   position: relative;
   bottom: 30px;
   left: 30px;
}
.wd-pos-relative-nudge-down-right {
   position: relative;
   top: 20px;
   left: 20px;
}
.wd-pos-relative {
   position: relative;
}
```

app/Labs/Lab2/Positions.tsx

```
<div id="wd-css-position-relative">
  <h2>Relative</h2>
  <div className="wd-bg-color-gray">
    <div className="wd-bg-color-yellow
                    wd-dimension-portrait">
      <div className="wd-pos-relative-nudge-down-right">
        Portrait</div></div>
    <div className="wd-pos-relative-nudge-up-right
        wd-bg-color-blue wd-fg-color-white
        wd-dimension-landscape">
      Landscape</div>
    <div className="wd-bg-color-red wd-dimension-square">
      Square</div>
  </div>
</div>
```

2.1.14 Styling a Tag's Absolute Position with CSS

The CSS ***position*** property set to ***absolute*** positions an element relative to its nearest ancestor with a position value of ***relative***, ***absolute***, or ***fixed***. If no such ancestor exists, the element is positioned relative to the ***body*** tag, effectively aligning with the browser's ***viewport***. To practice using ***absolute*** positioning, copy the CSS and HTML code provided below into ***index.css*** and ***Positions.tsx***, respectively, and verify the browser renders the result as shown in the figure below. Note that several ***br*** elements are included at the end of the example to create space for the next exercise.

app/Labs/Lab2/index.css

```
.wd-pos-absolute-10-10 {
   position: absolute;
   top: 10px; left: 10px; }
.wd-pos-absolute-50-50 {
   position: absolute;
   top: 50px; left: 50px; }
.wd-pos-absolute-120-20 {
   position: absolute;
   top: 20px; left: 120px; }
```

app/Labs/Lab2/Positions.tsx

```
<div id="wd-css-position-absolute">
  <h2>Absolute position</h2>
  <div className="wd-pos-relative">
    <div className="wd-pos-absolute-10-10 wd-bg-color-yellow wd-dimension-portrait">
      Portrait</div>
    <div className="wd-pos-absolute-50-50 wd-bg-color-blue wd-fg-color-white
        wd-dimension-landscape">
      Landscape</div>
    <div className="wd-pos-absolute-120-20 wd-bg-color-red wd-dimension-square">
      Square</div>
  </div><br /><br /><br /><br /><br /><br />
</div>
```

The term "***absolute***" can be misleading, as it positions the element relative to a parent or the ***body*** tag rather than an absolute point on the screen. Additionally, an element with ***absolute*** positioning scrolls with the rest of the page content. In contrast, when precise positioning is needed regardless of other elements or scrolling, the ***fixed*** position property, covered in the next section, anchors an element relative to the browser's viewport.

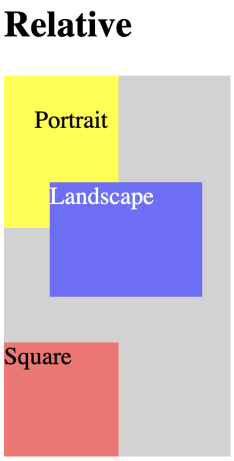


Figure 2.1.13 - Relative position

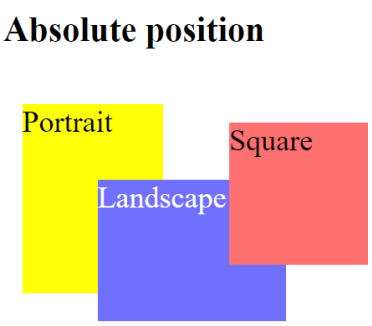


Figure 2.1.4 - Absolute position

2.1.15 Styling an HTML Tag's Fixed Position with CSS

Setting the CSS ***position*** property to ***fixed*** positions an element relative to the browser's ***viewport***. This ensures that the element remains in place, even when the page content is scrolled. To practice using ***fixed*** positioning, copy the CSS and HTML code provided below into ***index.css*** and ***Positions.tsx***, respectively, and verify that the browser renders the result as shown in the figure below. Note that the appearance may vary slightly depending on your screen size and scroll position.

app/Labs/Lab2/index.css

```
.wd-pos-fixed {
  position: fixed;
  right: 0px;
  bottom: 50%;
}
```

app/Labs/Lab2/Positions.tsx

```
<div id="wd-css-position-fixed">
  <h2>Fixed position</h2>
  Checkout the blue square that says "Fixed position" stuck all the way on the right and half
way down the page. It doesn't scroll with the rest of the page. Its position is "Fixed".
  <div className="wd-pos-fixed
    wd-dimension-square wd-bg-color-blue
    wd-fg-color-white">
    Fixed position
  </div>
</div>
```

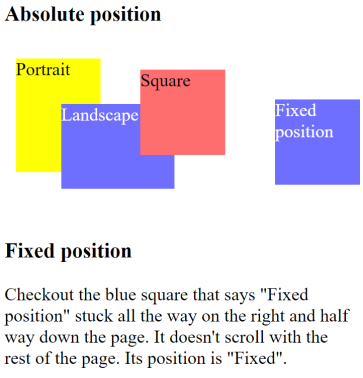


Figure 2.1.15 - Fixed position

2.1.16 Styling an HTML Tag's Z Index with CSS

When rendering HTML content, the browser automatically calculates the positions and dimensions of elements, assigning each a dedicated rectangular space on the screen to prevent overlap. However, using the CSS ***position*** property to override default positioning can lead to elements overlapping. By default, elements are rendered in the order they appear in the HTML, meaning later-declared elements appear above earlier ones when they overlap. The ***z-index*** CSS property allows you to control this stacking order: elements with a higher ***z-index*** value are displayed above those with a lower value. The default value of ***z-index*** is ***auto*** where elements are rendered in the order they are declared. To practice setting an element's ***z-index***, copy the CSS and HTML code provided below into ***index.css*** and ***Zindex.tsx***, respectively, and confirm that the browser renders the result as shown in the figure below.

app/Labs/Lab2/index.css

```
.wd-zindex-bring-to-front {
   z-index: 10;
}
```

app/Labs/Lab2/Zindex.tsx

```
<div id="wd-z-index">
  <h2>Z index</h2>
  <div className="wd-pos-relative">
    <div className="wd-pos-absolute-10-10 wd-bg-color-yellow wd-dimension-portrait">
      Portrait </div>
    <div className="wd-zindex-bring-to-front wd-pos-absolute-50-50
        wd-dimension-landscape wd-bg-color-blue wd-fg-color-white">
      Landscape </div>
    <div className="wd-pos-absolute-120-20 wd-bg-color-red wd-dimension-square">
      Square </div>
  </div><br /><br /><br /><br /><br /><br /><br />
</div>
```

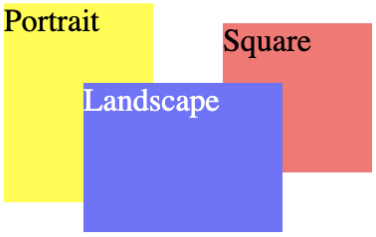


Figure 2.1.16 - Z-Index

2.1.17 Floating Images and Content with CSS

The CSS ***float*** property aligns elements to the left or right edges of the browser window while allowing adjacent content to wrap around the floated element. For instance consider the screen shot shown in Figure 2.1.17.a where long paragraphs flow around images aligned to the right and to the left. To accomplish this layout, images can be inserted in between several long portions of text. Normally images would flow inline with the rest of the adjacent text, but if the images are ***floated***, they align with the edges of the browser and the text would then flow around the images. The text in the screen shot is ***lorem ipsum***, a popular placeholder text often used when designing Web pages and other printed media. It can be found in https://www.lipsum.com.

app/Labs/Lab2/index.css

```
.wd-float-left {
   float: left;
   height: 100px;
}
```

app/Labs/Lab2/Float.tsx

```
<div id="wd-float-divs">
 <h2>Float</h2>
 <div>
   <img className="wd-float-right"
     src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg"/>
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
```

```css
.wd-float-right {
   float: right;
   height: 100px;
}

.wd-float-done {
   clear: both;
}
```

```html
   <img className="wd-float-left"
     src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg"/>
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   <img className="wd-float-right"
     src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg"/>
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   <img className="wd-float-left"
     src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg"/>
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   Lorem ipsum, dolor sit amet consectetur adipisicing elit. Eius hic ...
   <div className="wd-float-done"></div>
  </div>
</div>
```

Use the CSS and HTML code snippets above to practice using the ***float*** property to layout images as shown in the figure. Copy longer portions of the ***lorem ipsum*** text from the reference, or feel free to use your own text. Implement the example in a new ***Float.tsx*** file, import it and confirm it renders as shown. By default, HTML alone cannot arrange content horizontally, except through methods like ***table*** columns. The CSS ***float*** property can be applied to ***div*** elements to create horizontal layouts, as demonstrated in Figure 2.1.17.b. To practice this horizontal arrangement, copy the CSS and HTML code provided below into ***Float.tsx***, import it, and confirm that the browser renders the layout as shown in the figure below.

app/Labs/Lab2/index.css

```css
.wd-float-left {
   float: left;
}

.wd-float-right {
   float: right;
   height: 100px;
}

.wd-float-done {
   clear: both;
}
```

app/Labs/Lab2/Float.tsx

```html
<div id="wd-float-divs">
 <h2>Float</h2>
 <div>
   <div className="wd-float-left wd-dimension-portrait wd-bg-color-yellow">
     Yellow </div>
   <div className="wd-float-left wd-dimension-portrait wd-bg-color-blue wd-fg-color-white">
     Blue </div>
   <div className="wd-float-left wd-dimension-portrait wd-bg-color-red">
     Red </div>
   <img className="wd-float-right"
     src="https://www.staradvertiser.com/wp-content/uploads/2021/08/web1_Starship-gap2.jpg"/>
   <div className="wd-float-done"></div>
 </div>
</div>
```

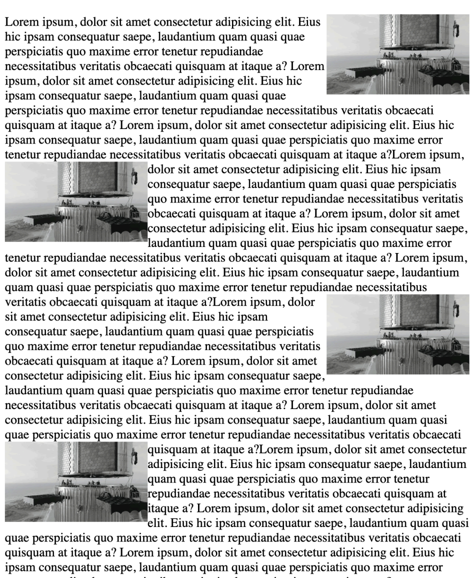


Figure 2.1.17.a - Floating Images and Content with CSS

Figure 2.1.17.b - Floating Images and Content with CSS

2.1.18 Laying Out Content in a Grid Using CSS

Building on the concept of horizontal content arrangement using the CSS ***float*** property, you can create sophisticated grid layouts with rows and columns, as illustrated in the figure below. To practice implementing a grid layout, copy the provided CSS and HTML code below into ***index.css*** and ***GridLayout.tsx***, respectively, and verify that the browser renders the layout as shown in the figure below.

app/Labs/Lab2/index.css

```css
.wd-grid-row {
   clear: both;
}
.wd-grid-col-half-page {
   width: 50%;
   float: left;
}
.wd-grid-col-third-page {
   width: 33%;
   float: left;
}
.wd-grid-col-two-thirds-page
{
   width: 67%;
   float: left;
}
.wd-grid-col-left-sidebar {
   width: 20%;
   float: left;
}
.wd-grid-col-main-content {
   width: 60%;
   float: left;
}
.wd-grid-col-right-sidebar {
   width: 20%;
   float: left;
}
```

app/Labs/Lab2/GridLayout.tsx

```tsx
<div id="wd-css-grid-layout">
  <div id="wd-css-left-right-layout">
    <h2>Grid layout</h2>
    <div className="wd-grid-row">
      <div className="wd-grid-col-half-page wd-bg-color-yellow">
        <h3>Left half</h3>      </div>
      <div className="wd-grid-col-half-page wd-bg-color-blue wd-fg-color-white">
        <h3>Right half</h3>      </div>
    </div>
  </div>
  <div id="wd-css-left-third-right-two-thirds" className="wd-grid-row">
    <div className="wd-grid-col-third-page wd-bg-color-green
        wd-fg-color-white">
      <h3>Left third</h3>    </div>
    <div className="wd-grid-col-two-thirds-page wd-bg-color-red wd-fg-color-white">
      <h3>Right two thirds</h3>    </div>
  </div>
  <div id="wd-css-side-bars" className="wd-grid-row">
    <div className="wd-grid-col-left-sidebar wd-bg-color-yellow">
      <h3>Side bar</h3>
      <p>This is the left sidebar</p>    </div>
    <div className="wd-grid-col-main-content wd-bg-color-blue wd-fg-color-white">
      <h3>Main content</h3>
      <p>
        This is the main content. This is the main content. This is the
        main content.
      </p>    </div>

    <div className="wd-grid-col-right-sidebar wd-bg-color-green wd-fg-color-white">
      <h3>Side bar</h3>
      <p>This is the right sidebar</p>
    </div>
  </div>
</div>
```

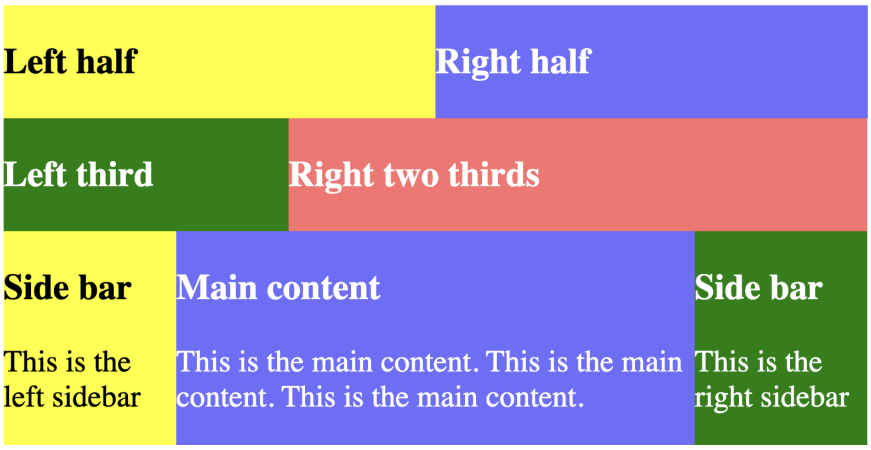


Figure 2.1.18 - Laying Out Content in a Grid Using CSS

2.1.19 Laying Out Content with CSS Flex

Flexbox Layout (***Flexible Box***, or just ***Flex***) provides a simpler way to layout and distribute content in an HTML document. It all starts with creating a container element that configures the layout and behavior of its child elements. To illustrate some of the features of flex, create a container with the ***display*** property set to ***flex*** as shown below.

app/Labs/Lab2/Flex.tsx

```
<div id="wd-css-flex">
  <h2>Flex</h2>
  <div className="wd-flex-row-container">
    <div className="wd-bg-color-yellow">Column 1</div>
    <div className="wd-bg-color-blue">Column 2</div>
    <div className="wd-bg-color-red">Column 3</div>
  </div>
</div>
```

app/Labs/Lab2/index.css

```
.wd-flex-row-container {
  display: flex;
  flex-direction: row;
}
```

Note how DIV elements inside the container render horizontally as a row of element instead of stacking the elements vertically. Flex simplifies laying out content horizontally. Flex child elements can also be configured to grow and expand to fill into empty spaces. The styling below illustrates how the last column can expand into the empty space to its right.

app/Labs/Lab2/Flex.tsx

```
<div id="wd-css-flex">
  <h2>Flex</h2>
  <div className="wd-flex-row-container">
    <div className="wd-bg-color-yellow">
      Column 1</div>
    <div className="wd-bg-color-blue">
      Column 2</div>
    <div className="wd-bg-color-red wd-flex-grow-1">
      Column 3</div>
  </div>
</div>
```

app/Labs/Lab2/index.css

```
.wd-flex-row-container {
  display: flex;
  flex-direction: row;
}
.wd-flex-grow-1 {
  flex-grow: 1;
}
```

The rest of the flex child elements can be configured independently to have specific widths to fit whatever content is needed as shown below.

app/Labs/Lab2/Flex.tsx

```
<div id="wd-css-flex">
  <h2>Flex</h2>
  <div className="wd-flex-row-container">
    <div className="wd-bg-color-yellow wd-width-75px">
      Column 1</div>
    <div className="wd-bg-color-blue">
      Column 2</div>
    <div className="wd-bg-color-red wd-flex-grow-1">
      Column 3</div>
  </div>
</div>
```

app/Labs/Lab2/index.css

```
.wd-flex-row-container {
  display: flex;
  flex-direction: row;
}
.wd-flex-grow-1 {
  flex-grow: 1;
}
.wd-width-75px {
  width: 75px;
}
```

Figure 2.1.19.a - Laying Out Content with CSS Flex

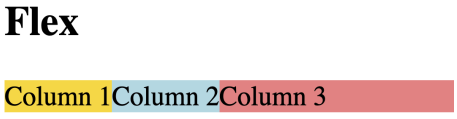


Figure 2.1.19.b - Laying Out Content with CSS Flex

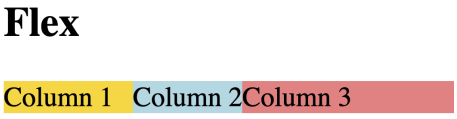


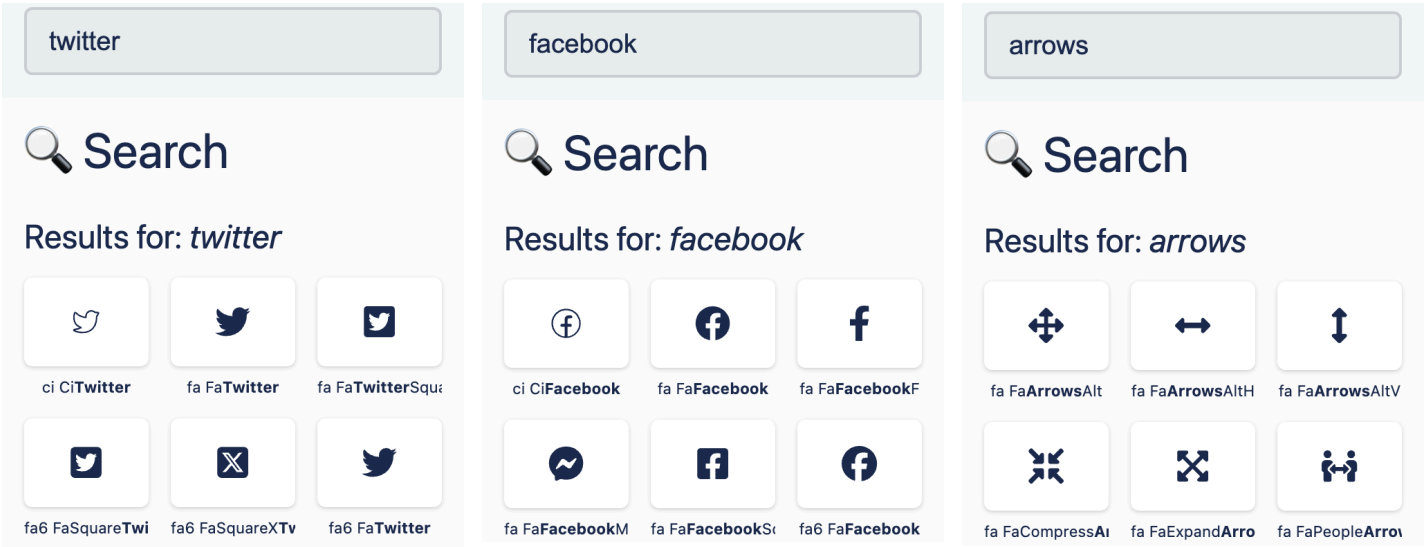
Figure 2.1.19.c - Laying Out Content with CSS Flex

2.2 Decorating Documents with React Icons

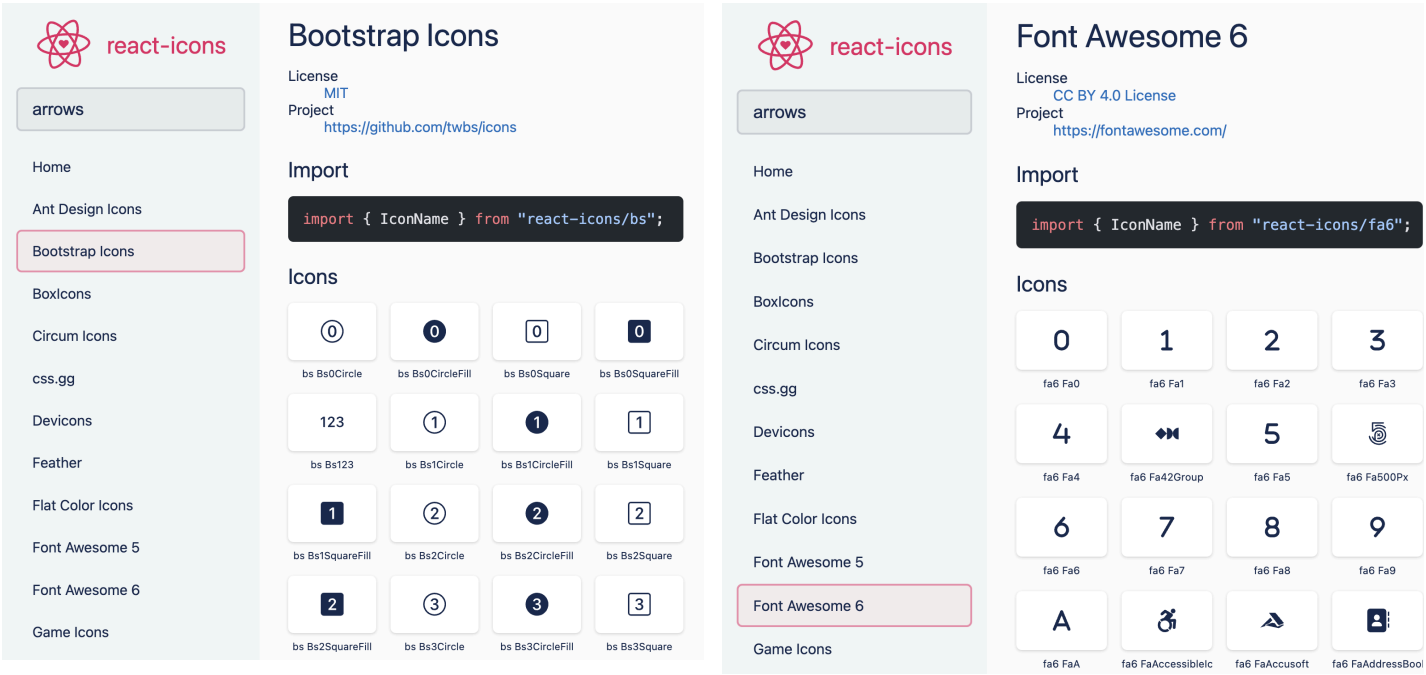
React icons is a CSS library that aggregates icons from various sources including ***Fontawesome***, ***Bootstrap***, and ***Heroicons***. Install React icons from the root of your project as shown below.

```
npm install react-icons
```

Head over to https://react-icons.github.io/react-icons to search for icons of interest. Search for icons by typing a topic in the search box. Here are a few example searches found from several icon sources.



Navigate through the icons by their vendors as shown below, e.g, ***Bootstrap*** and ***Fontawesome***.



The icons are made available as ***React*** components that can be imported and used as tags within HTML code. To practice using ***React Icons***, create the ***ReactIconSampler*** component as shown below. Import the component at the end of ***Lab2*** component and confirm it renders as shown below on the right.

app/Labs/Lab2/ReactIcons.tsx

```
import { FaCalendar, FaEnvelopeOpenText, FaRegClock } from "react-icons/fa";
import { AiOutlineDashboard } from "react-icons/ai";
import { FaBookBible } from "react-icons/fa6";
import { VscAccount } from "react-icons/vsc";
export default function ReactIconsSampler() {
  return (
    <div id="wd-react-icons-sampler" className="mb-4">
      <h3>React Icons Sampler</h3>
      <div className="d-flex">
        <VscAccount className="fs-3 text" />
        <AiOutlineDashboard className="fs-3 text" />
        <FaBookBible className="fs-3 text" />
        <FaCalendar className="fs-3 text" />
        <FaEnvelopeOpenText className="fs-3 text" />
        <FaRegClock className="fs-3 text" />
      </div>
    </div>
);}
```

Figure 2.2 - Decorating Documents with React Icons

2.3 Styling Webpages with the React Bootstrap CSS Library

React Bootstrap is a ***React*** library containing a plethora of useful CSS styles and React components that implement various widgets, layouts, and responsive design. This section contains exercises to practice using the ***React Bootstrap*** CSS styles and components to style and layout HTML content. Complete these exercises in the project created in the previous chapter (***kambaz-next-js***), and do all your work in ***app/Labs/Lab2/page.tsx***. When complete, deploy the project to ***Vercel*** and make sure the exercises are publicly available.

2.3.1 Installing React Bootstrap

Install ***React Bootstrap*** with ***npm*** from the root of the project as follows.

```
npm install react-bootstrap bootstrap
```

The ***React Bootstrap*** library will be downloaded from ***npm*** and installed in you ***node_modules*** directory located at the root of the project. Once installed, import the library from ***app/layout.tsx*** as shown below.

app/layout.tsx

```
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
// import "./globals.css";
import "bootstrap/dist/css/bootstrap.min.css";
...
```

2.3.2 Laying Out HTML Content with React Bootstrap Containers

React Bootstrap Containers center, pad content horizontally, and establish responsive layout. The padding adapts to the size of the screen for 6 different breakpoints corresponding to 6 different screen sizes: ***extra small*** (***xs***), ***small*** (***sm***), ***medium*** (***md***), ***large*** (***lg***), ***extra large*** (***xl***) and ***extra extra large*** (***xxl***). ***Fluid Containers*** define a constant thin margin all around the screen. To practice with containers, replace the root ***div*** element with ***Container*** in ***Lab2*** as shown below. Refresh the browser and confirm the ***Lab2*** screen has a margin around it. The heading is not flush with the left hand side and the font is not the default browser font.

app/Labs/Lab2/page.tsx

```
import "./index.css";
export default function Lab2() {
  return (
    <Container>
      <h2>Lab 2 - Cascading Style Sheets</h2>
      <h3>Styling with the STYLE attribute</h3>
      ...
    </Container>
```

2.3.3 Laying Out HTML Content with React Bootstrap Grids

It's easy to break a page vertical in HTML with the ***p*** and ***div*** tags. It's a little harder to layout content horizontally. Some resort to HTML tables to layout content horizontally using table ***rows*** and ***columns***, but this is generally considered a bad practice. HTML tables should be used for displaying tabular content only, not laying out HTML content. Nevertheless, laying out content like a table is convenient, so to achieve the same functionality as tables, but without tables, we can use CSS instead. React Bootstrap provides ***Row*** and ***Column*** components to layout content in a ***grid***. To practice with ***Bootstrap grids***, create a ***BootstrapGrids*** component using the code below, and import it to ***Lab2***.

app/Labs/Lab2/BootstrapGrids.tsx

```
<h2>Bootstrap</h2>
<div id="wd-bs-grid-system">
  <h2>Grid system</h2>
  <Row>
    <Col className="bg-danger text-white">           <h3>Left half</h3>    </Col>
    <Col className="bg-primary text-white">          <h3>Right half</h3>   </Col>
  </Row>
  <Row>
    <Col xs={4} className="bg-warning">              <h3>One third</h3>    </Col>
    <Col xs={8} className="bg-success text-white">   <h3>Two thirds</h3>   </Col>
  </Row>
  <Row>
    <Col xs={2} className="bg-black text-white">     <h3>Sidebar</h3>      </Col>
    <Col xs={8} className="bg-secondary text-white"> <h3>Main content</h3> </Col>
    <Col xs={2} className="bg-info">                 <h3>Sidebar</h3>      </Col>
  </Row>
</div>
```

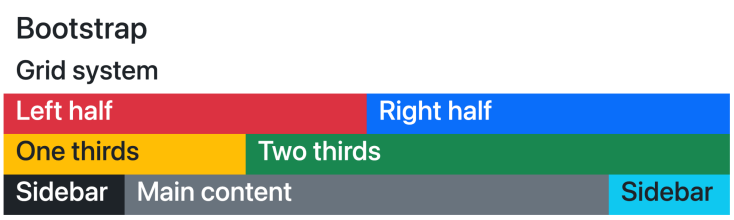


Figure 2.3.3 - Laying Out HTML Content with React Bootstrap Grids

2.3.4 Implementing Responsive HTML Content with React Bootstrap Grids

React Bootstrap Grids can adapt to the size of the screen, that is, they can be responsive. We can achieve this by applying more than one ***.col*** class which only applies for a given window size. To practice with ***Bootstrap responsive*** grids, create a ***BootstrapGrids*** component as shown below. Confirm it looks similar to image shown. Resize the browser and confirm that the grid shows 4 columns, then 2 and then just 1.

app/Labs/Lab2/BootstrapGrids.tsx

```
<div id="wd-bs-responsive-grids">
  <h2>Responsive grid system</h2>
  <Row>
    <Col xs={12} md={6} xl={3} className="bg-warning">
         <h3>Column A</h3>
    </Col>
    <Col xs={12} md={6} xl={3} className="bg-primary text-white">
         <h3>Column B</h3>
    </Col>
    <Col xs={12} md={6} xl={3} className="bg-danger text-white">
         <h3>Column C</h3>
    </Col>
    <Col xs={12} md={6} xl={3} className="bg-success text-white">
         <h3>Column D</h3>
    </Col>
  </Row>
</div>
```

Figure 2.3.4.a - Wide browser window shows 4 columns

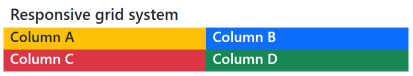


Figure 2.3.4.b - Moderate width browser window shows 2 columns

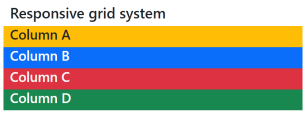


Figure 2.3.4.c - Thin browser window shows only 1 column

Let's try a more dramatic example by adding more content spread over more columns and rows. Copy the HTML code below to the end of ***BootstrapGrids.tsx***, and save. Refresh the browser and confirm that the column layout changes as you resize the browser window.

app/Labs/Lab2/BootstrapGrids.tsx

```
<div id="wd-bs-responsive-dramatic">
  <h2>Responsive grid system</h2>
  <Row>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-warning">
         <h4>1</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-primary text-white">
         <h4>2</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-danger text-white">
         <h4>3</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-success text-white">
         <h4>4</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-warning">
         <h4>5</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-primary text-white">
         <h4>6</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-danger text-white">
         <h4>7</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-success text-white">
         <h4>8</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-warning">
         <h4>9</h4></Col>
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-primary text-white">
         <h4>10</h4></Col>
```

```
    <Col xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-danger text-white">
         <h4>11</h4></Col>
   <Col  xs={12} sm={6} md={4} lg={3} xl={2} xxl={1} className="bg-success text-white">
         <h4>12</h4></Col>
   </Row>
</div>
```

Figure 2.3.4.d - Extra extra large (xxl) screen size renders 12 column grid

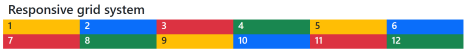


Figure 2.3.4.e - Extra large (xl) screen size renders 6 column grid

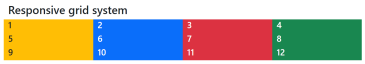


Figure 2.3.4.f - Large (lg) screen size renders 4 column grid

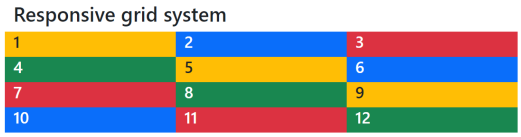


Figure 2.3.4.g - Medium screen (md) size renders 3 column grid

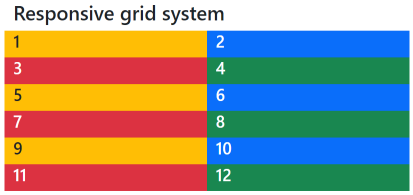


Figure 2.3.4.h - Small (sm) screen size renders 2 column grid

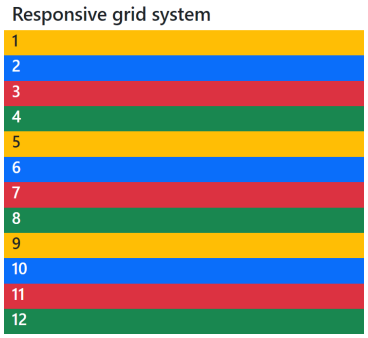


Figure 2.3.4.i - Extra small (xm) screen renders 1 column grid

2.3.5 Hiding and Showing Responsive HTML Content with React Bootstrap

As users shrink or widen the browser window, there may be more or less space to show some content. Bootstrap can configure content to show or hide depending on the width of the screen. As described earlier in Responsive grids, Bootstrap breaks up the screen into 5 sizes: ***extra small***, ***small***, ***medium***, ***large***, ***extra large***, and ***extra extra large***. Create a new component called ***ScreenSizeLabel*** as shown below which displays and hides different labels at different screen sizes. Add the styling to ***index.css*** to position the label at the top left corner. Create a component as shown below and confirm that the label displays the current screen size when you resize the window.

app/Labs/Lab2/ScreenSizeLabel.tsx

```
export default function ScreenSizeLabel() {
  return (
    <div id="wd-screen-size-label">
      <div className="d-block d-sm-none">
        XS - Extra Small (&lt;576px) </div>
      <div className="d-none d-sm-block d-md-none">
        S - Small (≥576px) </div>
      <div className="d-none d-md-block d-lg-none">
        M - Medium (≥768px) </div>
      <div className="d-none d-lg-block d-xl-none">
        L - Large (≥992px) </div>
      <div className="d-none d-xl-block d-xxl-none">
        XL - Extra Large (≥1200px) </div>
      <div className="d-none d-xxl-block">
        XXL - Extra Extra Large (≥1400px) </div>
    </div>
);}
```

app/Labs/Lab2/index.css

```
#wd-screen-size-label {
  position: fixed;
  top: 0;
  left: 0;
  background-color: black;
  color: white;
  padding: 5px;
  font-size: 12px;
  z-index: 1000;
  width: 220px;
  text-align: center;
}
```

Figure 2.3.5 - Hiding and Showing Responsive HTML Content with React Bootstrap

2.3.6 Styling HTML Tables with React Bootstrap

The ***React Bootstrap Table*** component and classes that enhance the look and feel of common HTML widgets such as tables, lists, and form elements. Let's start with tables. To practice with styling ***HTML tables***, create a component with the code shown below. Refresh the browser and confirm it looks similar to image shown.

app/Labs/Lab2/BootstrapTables.tsx

```
<div id="wd-css-styling-tables">
  <h2>Tables</h2>
  <Table>
    <thead>
      <tr className="table-dark"><th>Quiz</th><th>Topic</th><th>Date</th><th>Grade</th></tr>
    </thead>
    <tbody>
      <tr className="table-warning"><td>Q1</td><td>HTML</td><td>2/3/21</td><td>85</td></tr>
      <tr className="table-danger"><td>Q2</td><td>CSS</td><td>2/10/21</td><td>90</td></tr>
      <tr className="table-primary"><td>Q3</td><td>JavaScript</td><td>2/17/21</td><td>90</td></tr>
    </tbody>
    <tfoot>
      <tr className="table-success"><td colSpan={3}>Average</td><td>90</td></tr>
    </tfoot>
  </Table>
</div>
```

Quiz	Topic	Date	Grade
Q1	HTML	2/3/21	85
Q2	CSS	2/10/21	90
Q3	JavaScript	2/17/21	90
Average			90

Figure 2.3.6 - Styling HTML Tables with React Bootstrap

2.3.7 Implementing Responsive Tables with React Bootstrap

In general it is a good practice to minimize the number of scrollbars shown at any one time in a browser screen. In browsers, large amounts of content extends vertically beyond the height of the window, and then scrollbars allow you to access that extra content by scrolling vertically. Sometimes it is necessary to use additional scrollbars when appropriate such as tables or images that might be too wide to fit horizontally. Bootstrap provides tables that can show scrollbars when they don't fit in a screen. To practice with ***React Bootstrap Responsive Tables***, create a component with the code below and confirm it looks similar to image shown. Resize the window and confirm that the table shows a horizontal scroll bar when the screen is too small to fit the table comfortably.

app/Labs/Lab2/BootstrapTables.tsx

```
<div id="wd-css-responsive-tables">
  <h2>Responsive tables</h2>
  <Table responsive>
    <thead>
      <tr><th>Very</th><th>long</th><th>set</th><th>of</th><th>columns</th>
          <th>Very</th><th>long</th><th>set</th><th>of</th><th>columns</th>
          <th>Very</th><th>long</th><th>set</th><th>of</th><th>columns</th>
      </tr>
    </thead>
    <tbody>
      <tr>
         <td>Very</td><td>long</td><td>set</td><td>of</td><td>columns</td>
         <td>Very</td><td>long</td><td>set</td><td>of</td><td>columns</td>
         <td>Very</td><td>long</td><td>set</td><td>of</td><td>columns</td>
      </tr>
    </tbody>
  </Table>
</div>
```

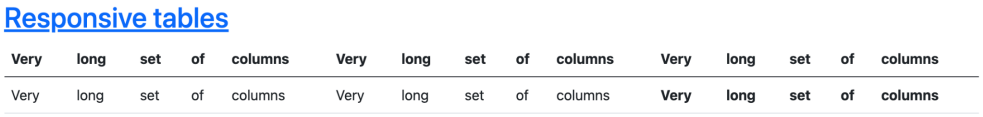


Figure 2.3.7.a - Responsive tables don't add scrollbars if they fit

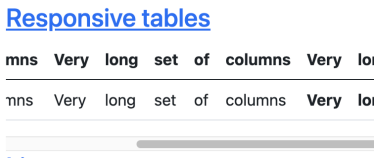


Figure 2.3.7.b - Responsive tables add scrollbars when they don't fit

2.3.8 Styling Lists with Rect Bootstrap

Another set of React Bootstrap components can make simple HTML lists look more presentable. The ***ListGroup*** and ***ListGroupItem*** components can be applied to ***ul*** and ***li*** tags correspondingly to make list stand out. The ***active*** attribute can be applied to an ***ListGroupItem*** components to highlight it. To practice with ***React Bootstrap List Group***, create a component with the code below and confirm it looks similar to image shown in Figure 2.3.8.

app/Labs/Lab2/BootstrapLists.tsx

```
<div id="wd-css-styling-lists">
  <h2>Favorite movies</h2>
  <ListGroup>
    <ListGroupItem active>Aliens</ListGroupItem>
    <ListGroupItem>Terminator</ListGroupItem>
    <ListGroupItem>Blade Runner</ListGroupItem>
    <ListGroupItem>Lord of the Ring</ListGroupItem>
    <ListGroupItem disabled>Star Wars</ListGroupItem>
  </ListGroup>
</div>
```

2.3.9 Styling Hyperlink Lists with React Bootstrap

The same ***ListGroup*** and ***ListGroupItem*** Bootstrap components can be configured as ***action***s and ***href*** to implement a list of anchor links. To practice with ***React Bootstrap hyperlink lists***, create a component with the code below and confirm it looks similar to image shown in Figure 2.3.9 and that the links work.

app/Labs/Lab2/BootstrapLists.tsx

```
<div id="wd-css-hyperlink-list">
  <h3>Favorite books</h3>
  <ListGroup>
    <ListGroupItem   action active href="https://en.wikipedia.org/wiki/Dune_(novel)">
      Dune
    </ListGroupItem>
    <ListGroupItem   action href="https://en.wikipedia.org/wiki/The_Lord_of_the_Rings">
      Lord of the Rings
    </ListGroupItem>
    <ListGroupItem   action href="https://en.wikipedia.org/wiki/The_Forever_War">
      The Forever War
    </ListGroupItem>
    <ListGroupItem   action href="https://en.wikipedia.org/wiki/2001:_A_Space_Odyssey_(novel)">
      2001 A Space Odyssey
    </ListGroupItem>
    <ListGroupItem   action   disabled href="https://en.wikipedia.org/wiki/Ender%27s_Game">
      Ender's Game
    </ListGroupItem>
    <ListGroupItem action onClick={() => alert("New book added")}>
      Add another book
    </ListGroupItem>
  </ListGroup>
</div>
```

Figure 2.3.8 - Favorite movies

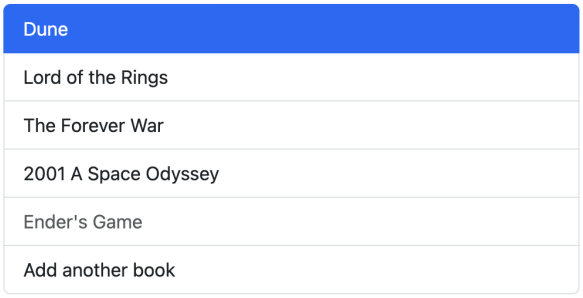


Figure 2.3.9 - Favorite books

2.3.10 Styling Forms with React Bootstrap

React Bootstrap has tons of components to style form elements especially to make them friendly for mobile Web applications. To practice with ***React Bootstrap form components***, create a component with the code below and confirm it renders similar to the image shown.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-styling-forms">
  <h2>Forms</h2>
  <FormGroup className="mb-3" controlId="wd-email">
    <FormLabel>Email address</FormLabel>
    <FormControl type="email" placeholder="name@example.com" />
  </FormGroup>
  <FormGroup className="mb-3" controlId="wd-textarea">
    <FormLabel>Example textarea</FormLabel>
    <FormControl as="textarea" rows={3} />
  </FormGroup>
</div>
```

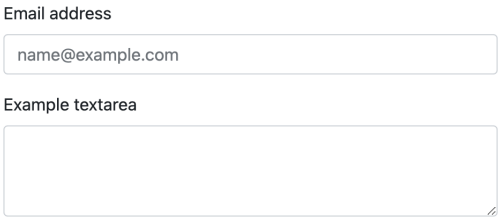


Figure 2.3.10 - Styling Forms with React Bootstrap

2.3.11 Styling Dropdowns with React Bootstrap

Dropdowns can also be styled professionally. To practice with ***React Bootstrap dropdown styling***, create a component with the code below and confirm it looks similar to image shown.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-styling-dropdowns">
  <h3>Dropdowns</h3>
  <FormSelect>
     <option selected>Open this select menu</option>
     <option value="1">One</option>
     <option value="2">Two</option>
     <option value="3">Three</option>
  </FormSelect>
</div>
```

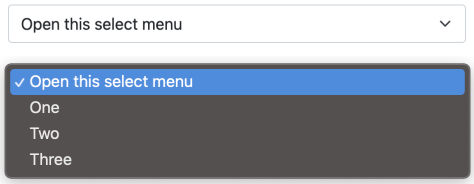


Figure 2.3.11 - Styling Dropdowns with React Bootstrap

2.3.12 Styling Switches with React Bootstrap

Checkboxes can be styled to look like switches with React Bootstrap components ***Form.Check*** with ***type="switch"***. To practice with ***Bootstrap form switches***, create a component with the code below.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-styling-switches">
  <h3>Switches</h3>
  <Form.Check type="switch" checked={false} label="Unchecked switch checkbox input"/>
  <Form.Check type="switch" checked={true}  label="Checked switch checkbox input"/>
  <Form.Check type="switch" checked={false} label="Unchecked disabled switch checkbox input" disabled/>
  <Form.Check type="switch" checked={true}  label="Checked disabled switch checkbox input"   disabled/>
</div>
```

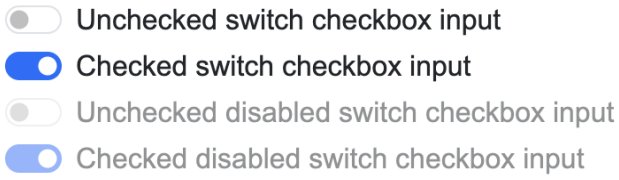


Figure 2.3.12 - Styling Switches with React Bootstrap

2.3.13 Styling Range and Sliders with React Bootstrap

Range input fields render as sliders. To practice with ***Bootstrap sliders***, create a component with the code below and confirm it renders similar to image shown.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-styling-range-and-sliders">
  <h3>Range</h3>
  <FormGroup controlId="wd-range1">
    <FormLabel>Example range</FormLabel>
    <FormRange min="0" max="5" step="0.5" />
  </FormGroup>
</div>
```

Figure 2.3.13 - Styling Range and Sliders with React Bootstrap

2.3.14 Styling Addons with React Bootstrap

Addons decorate input fields to give some context on the type and formate of the information to type in the input field. To practice with ***Bootstrap addons***, create a component with the code below and confirm it renders similar to image shown.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-styling-addons">
  <h3>Addons</h3>
  <InputGroup className="mb-3">
    <InputGroup.Text>$</InputGroup.Text>
    <InputGroup.Text>0.00</InputGroup.Text>
    <FormControl />
  </InputGroup>
  <InputGroup>
    <FormControl />
    <InputGroup.Text>$</InputGroup.Text>
    <InputGroup.Text>0.00</InputGroup.Text>
  </InputGroup>
</div>
```

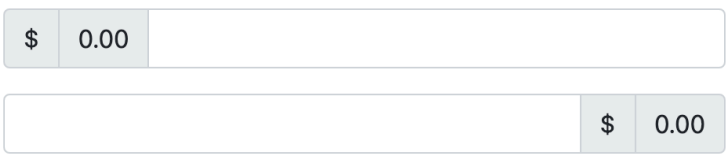


Figure 2.3.14 - Styling Addons with React Bootstrap

2.3.15 Styling Responsive Forms with React Bootstrap

Forms can be configured to display either horizontally or vertically depending on the size of the containing element. To practice with Bootstrap responsive forms, create a component with the code below and confirm it renders similar to image shown. Resize the window to show how the form changes layout as the window resizes.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-responsive-forms-1">
  <h3>Responsive forms</h3>
  <Form.Group as={Row} className="mb-3" controlId="email1">
    <Form.Label column sm={2}> Email </Form.Label>
    <Col sm={10}>
      <Form.Control type="email" value="email@example.com" />
    </Col>
  </Form.Group>
  <Form.Group as={Row} className="mb-3" controlId="password1">
    <Form.Label column sm={2}> Password </Form.Label>
    <Col sm={10}>
      <Form.Control type="password" />
    </Col>
  </Form.Group>
  <Form.Group as={Row} className="mb-3" controlId="textarea2">
    <Form.Label column sm={2}> Bio </Form.Label>
    <Col sm={10}>
      <Form.Control as="textarea" style={{height: "100px"}}/>
    </Col>
  </Form.Group>
</div>
```

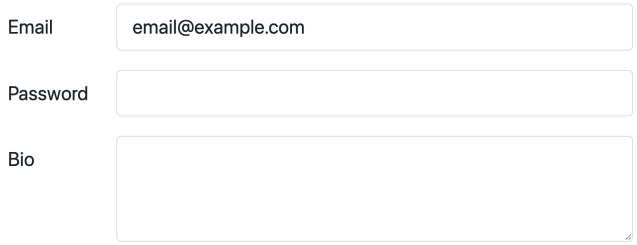


Figure 2.3.15.a - Styling Responsive Forms with React Bootstrap

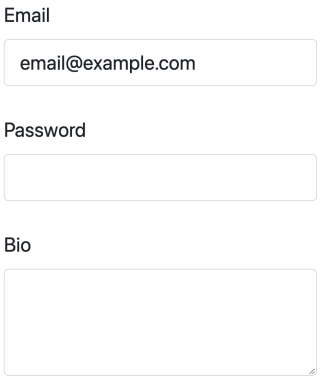


Figure 2.3.15.b - Styling Responsive Forms with React Bootstrap

Here's another example. Create a component with the code below and confirm it renders similar to image shown. Resize the window to show how the form changes layout as the window resizes.

app/Labs/Lab2/BootstrapForms.tsx

```
<div id="wd-css-responsive-forms-2">
  <h3>Responsive forms</h3>
  <Form>
   <Form.Group as={Row} className="mb-3">
     <Form.Label column sm={2}> Email </Form.Label>
     <Col sm={10}>
       <Form.Control type="email" placeholder="Email" />
     </Col>
```

```
  </Form.Group>
  <Form.Group as={Row} className="mb-3">
    <Form.Label column sm={2}> Password </Form.Label>
    <Col sm={10}>
      <Form.Control type="password" placeholder="Password" />
    </Col>
  </Form.Group>
  <fieldset>
    <Form.Group as={Row} className="mb-3">
      <Form.Label as="legend" column sm={2}> Radios </Form.Label>
      <Col sm={10}>
        <Form.Check type="radio" label="first radio"  name="formHorizontalRadios" checked/>
        <Form.Check type="radio" label="second radio" name="formHorizontalRadios"/>
        <Form.Check type="radio" label="third radio"  name="formHorizontalRadios"/>
      </Col>
    </Form.Group>
  </fieldset>
  <Form.Group as={Row} className="mb-3">
    <Col sm={{ span: 10, offset: 2 }}>
      <Form.Check label="Remember me" />
    </Col>
  </Form.Group>
  <Form.Group as={Row} className="mb-3">
    <Col>
      <Button type="submit">Sign in</Button>
    </Col>
  </Form.Group>
 </Form>
</div>
```

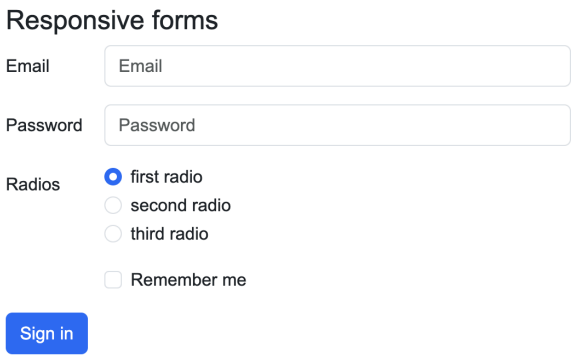


Figure 2.3.15.c - Styling Responsive Forms with React Bootstrap

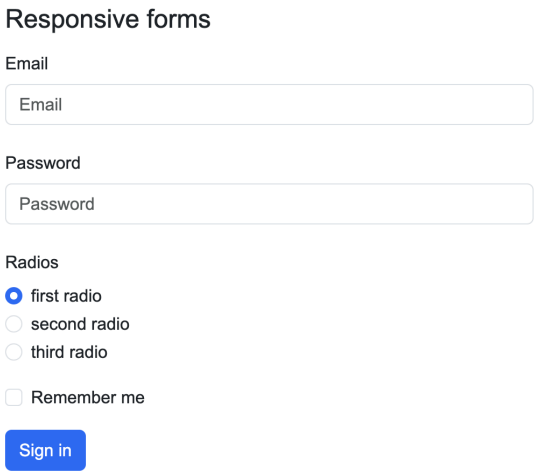


Figure 2.3.15.d - Styling Responsive Forms with React Bootstrap

2.3.16 Styling Navigation Tabs with React Bootstrap

Bootstrap provides several common navigation widgets such as tabs, menus, and pills. Let's take a look at tabs first. To practice with Bootstrap tabs, create a component with the code below and confirm it renders similar to image shown.

app/Labs/Lab2/BootstrapNavigation.tsx

```
<div id="wd-css-navigating-with-tabs">
  <h2>Tabs</h2>
  <Nav variant="tabs">
    <NavItem>
      <NavLink href="#/Labs/Lab2/Active">Active</NavLink>
    </NavItem>
```

```
    <NavItem>
      <NavLink href="#/Labs/Lab2/Link1">Link 1</NavLink>
    </NavItem>
    <NavItem>
      <NavLink href="#/Labs/Lab2/Link2">Link 2</NavLink>
    </NavItem>
    <NavItem>
      <NavLink href="#/Labs/Lab2/Disabled" disabled>Disabled</NavLink>
    </NavItem>
  </Nav>
</div>
```

Figure 2.3.16 - Styling Navigation Tabs with React Bootstrap

2.3.17 Styling Navigation Pills with React Bootstrap

Pills are another navigation widget listing several links. Here's an example of using the ***Bootstrap Pills*** to refactor the ***TOC*** component with Bootstrap's ***pills*** variant of the ***Nav*** component. Confirm the ***TOC*** highlights the corresponding link as you navigate between the labs.

app/Labs/TOC.tsx

```
import { Nav, NavItem, NavLink } from "react-bootstrap";
import Link from "next/link";
export default function TOC() {
 return (
   <Nav variant="pills">
     <NavItem>
       <NavLink href="/Labs" as={Link}>Labs</NavLink>
     </NavItem>
     <NavItem>
       <NavLink href="/Labs/Lab1" as={Link}>Lab 1</NavLink>
     </NavItem>
     <NavItem>
       <NavLink href="/Labs/Lab2" as={Link}>Lab 2</NavLink>
     </NavItem>
     <NavItem>
       <NavLink href="/Labs/Lab3" as={Link}>Lab 3</NavLink>
     </NavItem>
     <NavItem>
       <NavLink href="/" as={Link}>Kambaz</NavLink> jga
     </NavItem>
     <NavItem>
       <NavLink href="https://github.com/jannunzi">My GitHub</NavLink>
     </NavItem>
   </Nav>
);}
```

Figure 2.3.17 - Styling Navigation Pills with React Bootstrap

2.3.18 Styling Navigation with React Bootstrap Cards

Cards combine images, titles, paragraphs and buttons into a reusable component that can quickly summarize a topic. To practice with ***Bootstrap cards***, create a component with the code below and confirm it renders similar to image shown.

Use an image of your own, or download one from my Flickr account and save it to ***public/images/stacked.jpg***. Import all necessary Bootstrap components, e.g, ***Card***, and ***Button***.

app/Labs/Lab2/BootstrapNavigation.tsx ***Browser***

```
<div id="wd-css-navigating-with-cards">
  <h2>
    Cards
  </h2>
  <Card style={{ width: "18rem" }}>
    <CardImg variant="top" src="images/stacked.jpg" />
    <CardBody>
      <CardTitle>Stacking Starship</CardTitle>
      <CardText>
        Stacking the most powerful rocket in history. Mars or bust!
      </CardText>
      <Button variant="primary">Boldly Go</Button>
    </CardBody>
  </Card>
</div>
```

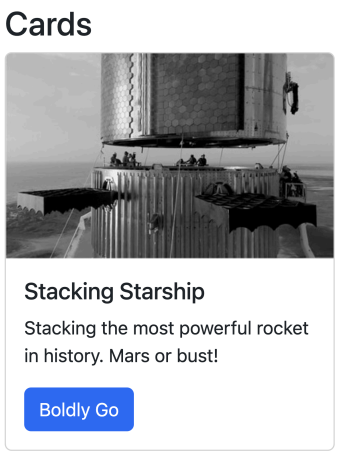


Figure 2.3.18 - Styling Navigation with React Bootstrap Cards

2.4 Styling Kambaz with CSS and Bootstrap

The previous chapter implemented a prototype of the ***Kambaz*** Web application using plain HTML without overriding the default browser style sheet. This section revisits the ***Kambaz*** screens using ***CSS*** and ***Bootstrap*** to layout and color the screens so they look more like the screenshots provided. Make an effort to style the HTML code to make the screens look as close as possible to their intended look and feel, but don't fret if the screens are not pixel perfect and instead, follow the guidelines and requirements to practice. Several code snippets are provided as a suggestion on how to achieve the layout and style. Feel free to use the code as provided or ignore it to implement as you will. If you do implement the styling on your own, make sure it matches the screenshots provided.

The previous chapter used ***table***, ***tr***, and ***td*** elements to layout screens horizontally, but in general this is considered a bad practice and CSS alternatives are preferred. Remove the ***table*** elements in preparation of using ***CSS*** instead as shown below.

app/(Kambaz)/Layout.tsx

```
<div id="wd-kambaz">
  ~~<table><tbody><tr><td valign="top" width="200">~~
  <div className="d-flex">
    <div>
```

```
      <KambazNavigation />
    </div>
      </td><td valign="top" width="100%">
    <div className="flex-fill">
      {children}
    </div>
  </div>
  </td></tr></tbody></table>
</div>
```

Do the same for the ***Courses*** and ***Home*** screens replacing the ***table*** elements with ***div*** elements styled with ***Bootstrap flex classes*** and ***display classes***. Style the ***Course*** header as shown below and confirm the ***CourseNavigation*** sidebar and the ***Course Status*** sidebar appears and hides as you resize the window. Add any missing imports as needed.

app/(Kambaz)/Courses/[cid]/layout.tsx

```
<div id="wd-courses">
  <h2 className="text-danger">
      <FaAlignJustify className="me-4 fs-4 mb-1" />
      Course {cid} </h2> <hr />
  <table><tbody><tr><td valign="top" width="200">
  <div className="d-flex">
    <div className="d-none d-md-block">
      <CourseNavigation />
    </td><td valign="top" width="100%">
    </div>
    <div className="flex-fill">
      {children}
    </div></div>
    </td></tr>
  </tbody></table>
</div>
```

app/(Kambaz)/Courses/[cid]/Home/page.tsx

```
<div id="wd-courses">
  <table><tbody><tr><td valign="top">
  <div className="d-flex" id="wd-home">
    <div className="flex-fill me-3">
      <Modules />
    </div>
  </td><td valign="top">
    <div className="d-none d-xl-block">
      <CourseStatus />
    </div>
  </div>
  </td></tr></tbody></table>
</div>
```

2.4.1 Styling the Kambaz Navigation Sidebar

The ***Kambaz*** Web application has several screens implementing different features. Figure 2.4.1.a bellow illustrates the ***Kambaz Navigation*** sidebar with the ***Dashboard*** link selected displaying a grid of courses that allow navigating to different courses. Without CSS the best the previous chapter could do was implement the list of links shown below in Figure 2.4.1.b. With CSS and Bootstrap, this section demonstrates how to style the ***Kambaz Navigation*** as shown below in Figure 2.4.1.c, as well as configure it to navigate to the ***Kambaz Dashboard*** and ***Courses Home*** screens.

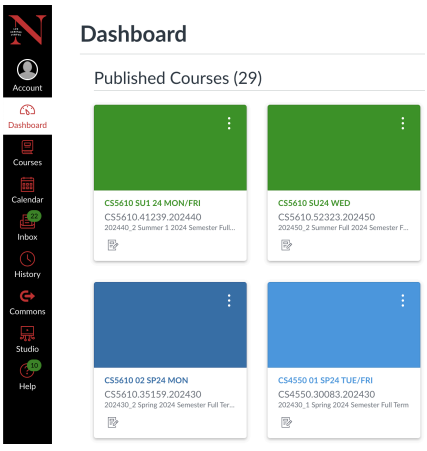


Figure 2.4.1.a - ***Kambaz Navigation*** and ***Dashboard***

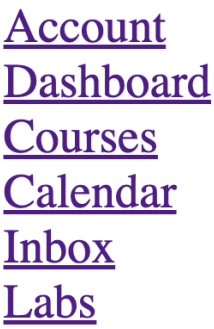


Figure 2.4.1.b - ***Kambaz Navigation*** so far

Figure 2.4.1.c - ***Kambaz Navigation*** this chapter

The previous chapter implemented the ***Kambaz Navigation*** sidebar in ***app/(Kambaz)/Navigation.tsx*** as rendered above. Using ***Bootstrap's Links and Buttons*** to ***Navigation.tsx***, style the sidebar with ***CSS*** so that it looks more like the sidebar in the screenshot above on the right. Use the code below as a guide to apply the ***Bootstrap*** classes and ***React Icons*** to render

the ***Dashboard*** and ***Calendar*** as icons. Explore other icons for the rest of the links. The icons don't have to match the ones showed here, but should reflect the meaning and intention of the link's text and target screen.

app/(Kambaz)/Navigation.tsx

```
import { AiOutlineDashboard } from "react-icons/ai";
import { IoCalendarOutline } from "react-icons/io5";
import { LiaBookSolid, LiaCogSolid } from "react-icons/lia";
import { FaInbox, FaRegCircleUser } from "react-icons/fa6";
import { ListGroup, ListGroupItem } from "react-bootstrap";
import Link from "next/link";
export default function KambazNavigation() {
 return (
   <ListGroup className="rounded-0 position-fixed bottom-0 top-0 d-none d-md-block bg-black z-2" style={{ width: 120 }}
              id="wd-kambaz-navigation">
     <ListGroupItem className="bg-black border-0 text-center" as="a"
              target="_blank" href="https://www.northeastern.edu/" id="wd-neu-link">
       <img src="/images/NEU.png" width="75px" alt="Northeastern University" />
     </ListGroupItem><br />
     <ListGroupItem className="border-0 bg-black text-center">
       <Link href="/Account" id="wd-account-link" className="text-white text-decoration-none">
         <FaRegCircleUser className="fs-1 text-white" />
         <br />
         Account
       </Link>
     </ListGroupItem><br />
     <ListGroupItem className="border-0 bg-white text-center">
       <Link href="/Dashboard" id="wd-dashboard-link" className="text-danger text-decoration-none">
         <AiOutlineDashboard className="fs-1 text-danger" />
         <br />
         Dashboard
       </Link>
     </ListGroupItem><br />
     {/* complete styling the rest of the links */}
   </ListGroup>
);}
```

The ***Northeastern*** logo can be found in the ***Brand Center*** website. Apply the following ***Bootstrap*** classes to fix the position of the ***Kambaz Navigation*** sidebar and stretch it vertically the whole height of the screen.

- ***position-fixed*** - applies ***display: fixed*** so that the sidebar stays fixed in one place and doesn't scroll wit the rest of the screen
- ***bottom-0 top-0*** - applies ***bottom: 0*** and ***top: 0*** so that the ***top*** edge is fixed at the ***top*** of the screen and the ***bottom*** edge is fixed at the ***bottom*** of the screen, effectively stretching the height of the element the whole height of the window.
- ***d-none d-md-block*** - applies ***display: none*** so that the element is initially hidden. And then applies ***display: block*** to display the element when the screen width reaches ***mid*** sized screens, e.g., ***768*** pixels
- ***z-2*** - applies ***z-index: 2*** which brings the element above other elements with a lower ***z-index***.
- ***bg-black*** - applies ***background-color: black***

Confirm that the the ***Kambaz Navigation*** sidebar stretches the whole height of the screen, does not scroll with the rest of the ***Dashboard*** and disappears when the screen is narrow, but appears again when the the screen widens.

app/(Kambaz)/Navigation.tsx

```
export default function KambazNavigation() {
  return (
    <ListGroup id="wd-kambaz-navigation" style={{ width: 120 }}
         className="rounded-0 position-fixed bottom-0 top-0 d-none d-md-block bg-black z-2">
         ...
    </ListGroup>
);}
```

The implementation does not need to be pixel perfect, but here are some rough requirements that should be consider when implementing the ***Kambaz Navigation***. Use your own access to ***Canvas*** to help guide your design

- The ***Kambaz Account Sign*** screen must be the default screen when navigating to ***Kambaz***
- Icons must be ***red***, except the ***Account*** icon which must be ***white***
- Icons don't have to match exactly, but must be similar. Use an equivalent React Icons
- The font size and style must be similar
- The whole sidebar must have a black background
- Selected (active) links must have a white background with red text
- Non selected links must have a black background with white text
- The width of the sidebar must be about 110 pixels wide, +/- 5 pixels.
- The icons and text must be centered in the sidebar

Once you are satisfied with the styling of the ***Kambaz Navigation*** sidebar, visit all the screens and make sure that the ***Kambaz Navigation*** styling is as shown above. The ***Kambaz Navigation*** sidebar is now stuck to the left of the screen and does not scroll with the main content on the right, but the ***Dashboard*** now renders under the sidebar which overlaps the main content. This is because ***position-fixed*** removes the ***DIV*** from the normal layout flow and no longer pushes other content to render below it. This can be fixed by offsetting the main content to the right to compensate for the width of the sidebar. Additionally, the sidebar appears and disappears at different screen sizes, so the offset needs to apply only when the sidebar is present and not apply when the sidebar is hidden. Create class ***wd-main-content-offset*** in a new ***styles.css*** file that offsets ***Kambaz***'s main content by ***140*** pixels, enough to move the Routes to the right off under the ***Kambaz Navigation*** sidebar. The class is declared inside a ***@media*** query that applies the ***140*** pixel offset only when the screen is wider than ***768*** pixels, exactly when the sidebar appears. Navigate to the ***Dashboard*** and confirm that it does not render under the sidebar for all screen sizes

app/(Kambaz)/layout.tsx

```
import { ReactNode } from "react";
import KambazNavigation from "./Navigation";
import "./styles.css";
export default function KambazLayout( ... ) {
 return (
   <div id="wd-kambaz">
     <div className="d-flex">
       <div>
         <KambazNavigation />
       </div>
       <div className="wd-main-content-offset p-3 flex-fill">
         {children}
       </div>
     </div>
   </div>
);}
```

app/(Kambaz)/styles.css

```
body   {
  margin: 0;
  padding: 0;
}

body * {
  font-family: Arial,
  Helvetica, sans-serif;
}

@media (min-width: 768px) {
  #wd-kambaz .wd-main-content-offset {
    padding-left: 140px !important;
  }
}
```

2.4.2 Styling the Kambaz Dashboard Screen

The previous chapter implemented the ***Kambaz Dashboard*** as the default screen when first login into ***Kambaz*** and it lists courses as shown below. Clicking on a course navigates to that specific course. Since JavaScript has not been officially covered yet, this chapter will just implement a single course and all courses in the ***Kambaz Dashboard*** will navigate to that same course. Later chapters will implement a data structure representing multiple courses. Courses are rendered in a responsive grid pattern that adapts (responds) to the width of the screen. When the screen is wide, each row of the grid displays several courses, as shown in Figure 2.4.2.a below. As the screen becomes narrower, the courses that don't fit, wrap to the next row as shown in Figures 2.4.2.b, and 2.4.2.c below. When the screen is too narrow, the ***Kambaz Navigation*** is hidden and courses are displayed in a single column as shown in Figure 2.4.2.d.

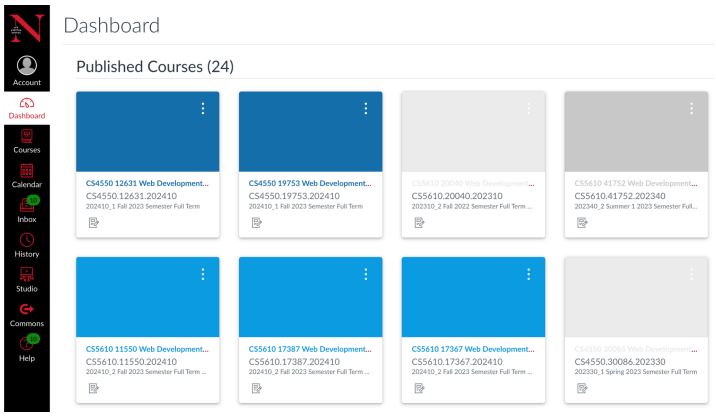


Figure 2.4.2.a - ***Dashboard*** at widest screen

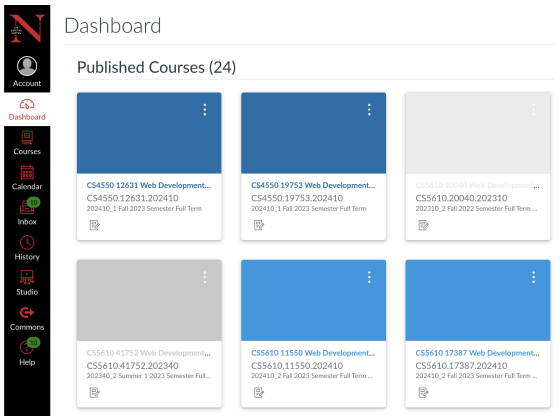
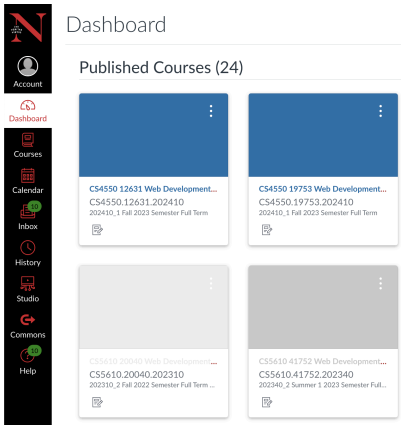


Figure 2.4.2.b - As screen narrows, courses wrap



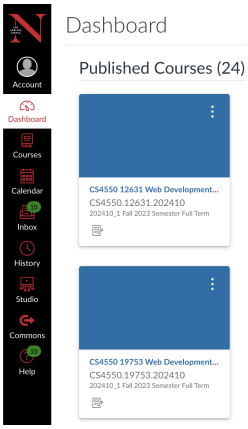


Figure 2.4.2.c - As screen narrows, courses wrap even further

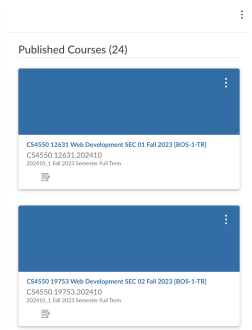


Figure 2.4.2.d - At narrowest screen, just one column of courses shows

The behavior of adapting the layout of the screen and hiding and showing content at various screen sizes is referred to as ***Responsive Design***, allowing developers to create content that can be viewed in multiple devices with varying screen sizes. Refactor the current ***Dashboard*** implementation so that it behaves as described above. Use ***React Bootstrap***'s responsive ***Grid Cards*** which render a row of ***Bootstrap Cards*** from left to right and wrap as the window becomes narrower. Use the ***page.tsx*** file shown below as an example. Feel free to come up with your own courses. Replace the course's image ***<img>*** with ***React Bootstrap***'s ***<CardImg>***, replace the course's title ***<h5>*** with ***React Bootstrap***'s ***<CardTitle>***, replace the course's description paragraph ***<p>*** with ***React Bootstrap***'s ***<CardText>***, replace ***<button>*** with ***React Bootstrap***'s ***<Button>***. Download images for each course and save them to ***public/images***. The code below uses an image of the React logo. Feel free to use your own implementation as long as it looks and behaves as described.

app/(Kambaz)/Dashboard/page.tsx

```
...
<div id="wd-dashboard">
 <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
 <h2 id="wd-dashboard-published">Published Courses (12)</h2> <hr />
 <div id="wd-dashboard-courses">
  <Row xs={1} md={5} className="g-4">
   <Col className="wd-dashboard-course" style={{ width: "300px" }}>
    <Card>
     <Link href="/Courses/1234/Home"
           className="wd-dashboard-course-link text-decoration-none text-dark">
      <CardImg variant="top" src="/images/reactjs.jpg" width="100%" height={160}/>
      <CardBody>
       <CardTitle className="wd-dashboard-course-title text-nowrap overflow-hidden">CS1234 React JS</CardTitle>
```

```
        <CardText  className="wd-dashboard-course-description overflow-hidden" style={{ height: "100px" }}>
         Full Stack software developer</CardText>
        <Button variant="primary">Go</Button>
       </CardBody>
      </Link>
     </Card>
    </Col>
    <Col  className="wd-dashboard-course" style={{ width: "300px" }}> ... Another course ... </Col>
    <Col  className="wd-dashboard-course" style={{ width: "300px" }}> ... Another course ... </Col>
   </Row>
</div></div>
...
```

Your implementation does not need to pixel perfect, but here are some rough requirements you should consider when implementing the ***Kambaz Dashboard***. Use your own access to ***Canvas*** to help guide your design

- The ***Dashboard*** link must be rendered as selected in the ***Kambaz Navigation*** sidebar with red text, red icon, and white background
- A ***Dashboard*** title must appear at the top of the screen as shown
- Clicking the ***Dashboard*** link must display the ***Kambaz Dashboard*** screen
- Clicking any course in ***Kambaz Dashboard***, reveals the ***Home*** screen implemented in previous chapters
- A ***horizontal rule*** (***<hr/>***) must appear below the title as shown
- A ***Published Courses*** sub title must appear as shown
- At least 7 courses must be rendered under the ***Published Courses*** as shown
- Courses must render in a grid of rows and columns as shown
- Use ***Grid Cards*** and ***Bootstrap Cards*** must to render each course as a card with an image or solid color at the top and the course's title and description, linked to the course ***Home*** screen
- The widths of the courses as shown are about 250 and 270 pixels, and don't change as the window narrows
- There must be white spacing between courses above and below of between 30 and 40 pixels
- There must be white spacing between the right most edge of the ***Kambaz Navigation*** sidebar and the left most edge of the left most course
- When the window is at its widest, rows should fit at least 4 courses per row
- When the window shrinks and courses don't fit, then courses should wrap to the next row

2.4.3 Styling the Course Navigation Sidebar

Clicking a course in the ***Kambaz Dashboard*** navigates to that course's ***Home*** screen. For now, implement the ***Kambaz Dashboard*** to navigate to the same course. Later chapters will introduce ***JavaScript*** and revisit this implementation to navigate to the corresponding course. The previous chapter implemented the ***Course Navigation*** sidebar in ***app/(Kambaz)/Courses/[cid]/Navigation.tsx***. Style the ***Course Navigation*** sidebar to look more like the one from ***Canvas***. The code below illustrates how to apply styles to the ***Courses Navigation*** sidebar so that it renders closer to the target look and feel shown here on the right. Refactor the list of links using ***Bootstrap List and buttons*** so that the clickable area is wider and easier to use in smaller screens. The ***styles.css*** file is implemented at the root of the ***Kambaz*** folder and is the same we used in the ***Kambaz Navigation*** sidebar and is documented earlier in this document. Feel free to use and modify the code below as necessary or implement your own version as long as it looks and behaves as shown.

app/(Kambaz)/Courses/[cid]/Navigation.tsx

```
import Link from "next/link";
export default function CourseNavigation() {
  return (
    <div id="wd-courses-navigation" className="wd list-group fs-5 rounded-0">
      <Link href="/Courses/1234/Home" id="wd-course-home-link"
        className="list-group-item active border-0"> Home </Link><br />
      <Link href="/Courses/1234/Modules" id="wd-course-modules-link"
        className="list-group-item text-danger border-0"> Modules </Link><br />
      <Link href="/Courses/1234/Piazza" id="wd-course-piazza-link"
```

```
        className="list-group-item text-danger border-0"> Piazza </Link><br />
      <Link href="/Courses/1234/Zoom" id="wd-course-zoom-link"
        className="list-group-item text-danger border-0"> Zoom </Link><br />
      <Link href="/Courses/1234/Assignments" id="wd-course-quizzes-link"
        className="list-group-item text-danger border-0"> Assignments </Link><br />
      <Link href="/Courses/1234/Quizzes" id="wd-course-assignments-link"
        className="list-group-item text-danger border-0"> Quizzes </Link><br />
      <Link href="/Courses/1234/People/Table" id="wd-course-people-link"
        className="list-group-item text-danger border-0" > People </Link><br />
    </div>
);}
```

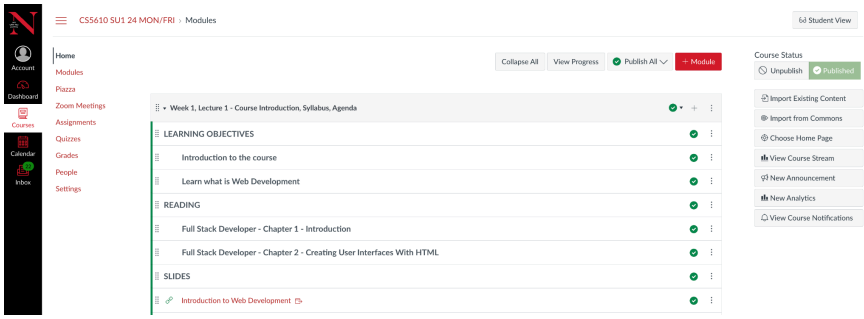
The CSS rule in ***styles.css*** shown below overrides ***Bootstrap*** classes ***list-group***, ***list-group-time***, and ***active*** so that the sidebar renders with a white background color, red text, and black border of the highlighted link.

app/(Kambaz)/styles.css

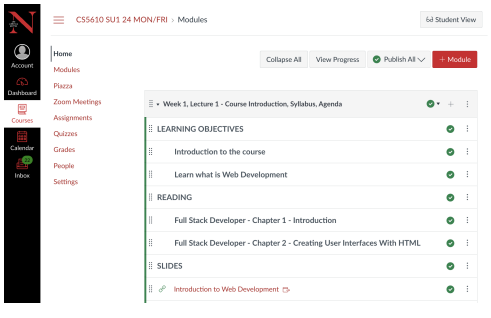
```
...
.list-group.wd > .list-group-item.active {
  color: black;
  background-color: white;
  border-left: 3px solid black !important;
}
```

2.4.4 Styling the Modules Screen

The ***Modules*** in the ***Modules Screen*** is also used in ***Home Screen*** so style the ***Modules*** first. Clicking on a course in the ***Kambaz Dashboard*** displays the course's ***Home*** screen as shown below. The ***Course Navigation*** sidebar provides various links to navigate to different screens related to the course such as ***Home***, ***Modules***, ***Assignments***, and ***Grades***. The ***Course Home*** screen is displayed by default and its link is rendered as selected when you first navigate to a course from the ***Dashboard***. In a wide screen the ***Course Home*** screen renders 4 columns as shown in the Figure 2.4.4.a shown below. The first column is the ***Kambaz Navigation*** sidebar, the second column is the ***Course Navigation*** sidebar, and the third column contains the ***Course Home*** showing a list of modules starting at ***Week 1***. The last column are various buttons and links we'll refer to as the ***Course Status***. If the window narrows below some threshold, the last column is hidden and the third column takes the remaining width as shown in Figure 2.4.4.b below.



2.4.4.a. Course Home screen renders 4 columns in a wide screen



2.4.4.b. Course Status hides as screen narrows

If the window narrows below some other threshold, the first and second column are hidden, and the third column takes the remaining width as shown in Figure 2.4.4.c shown below. Even though the ***Kambaz Navigation*** is hidden, it is still available by clicking the top left icon and displays as shown in screenshot Figure 2.4.4.d shown below. The ***Course Navigation*** is also hidden, but available by clicking the top right corner icon and displays as shown in Figure 2.4.4.e shown below.

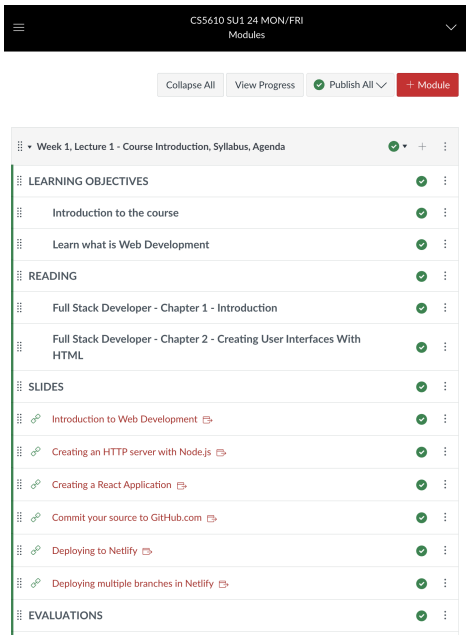


Figure 2.4.4.c - Hide ***Kambaz*** and ***Courses Navigation*** at narrowest screen

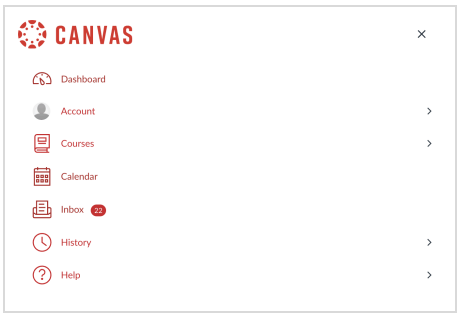


Figure 2.4.4.d - Click top left icon to show ***Kambaz Navigation***

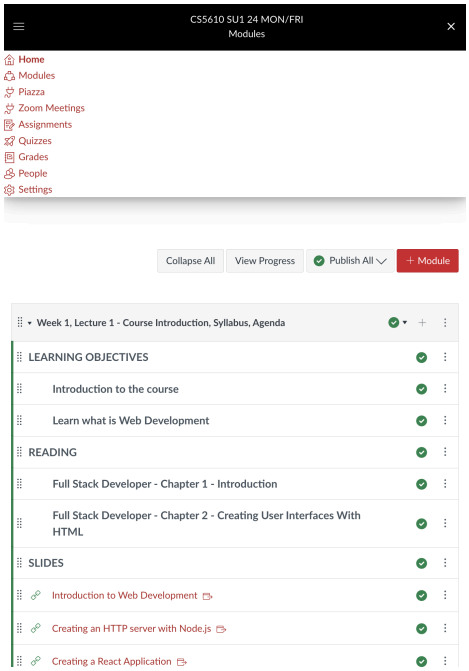
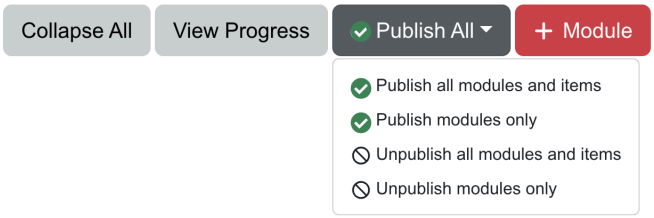


Figure 2.4.4.e - Click top right icon to show ***Courses Navigation***

Since ***Course Home*** screen's main content is the ***Modules***, style those first as shown in Figures 2.4.4 below. At the top of the ***Modules*** there's a collection of buttons and dropdowns including ***Collapse All***, ***View Progress***, ***Publish All***, ***Module*** as shown on the right. First, create the green circular checkmark in a ***GreenCheckmark*** component as shown below.



app/(Kambaz)/Courses/[cid]/Modules/GreenCheckmark.tsx

```
import { FaCheckCircle, FaCircle } from "react-icons/fa";
export default function GreenCheckmark() {
  return (
    <span className="me-1 position-relative">
      <FaCheckCircle style={{ top: "2px" }} className="text-success me-1 position-absolute fs-5" />
      <FaCircle className="text-white me-1 fs-6" />
    </span>);}
```

Style the ***Modules Screen*** screen and ***Home Screen*** to look and behave as shown in the screenshots below.

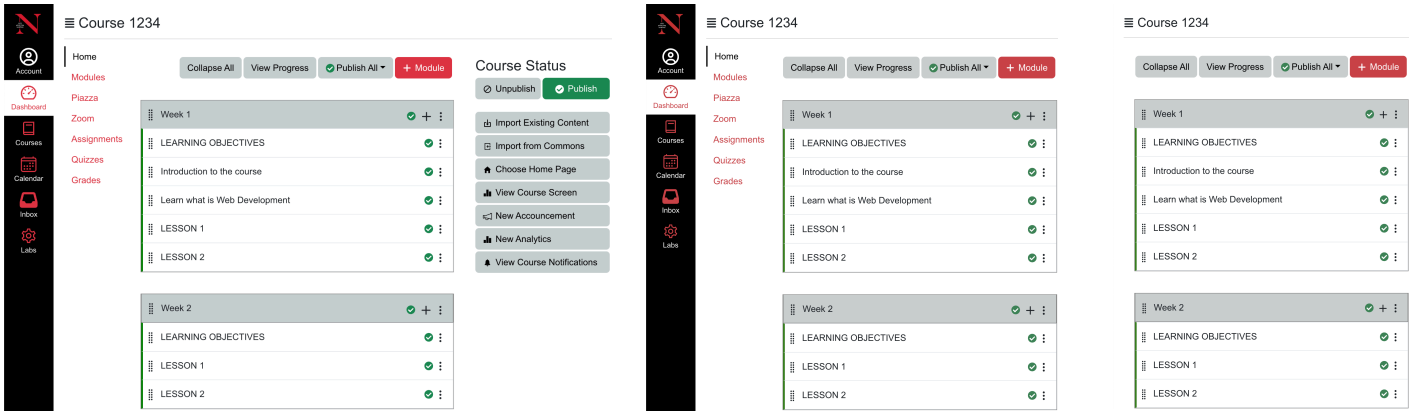


Figure 2.4.4.f - ***Course Home*** screen renders 4 columns in a wide screen

Figure 2.4.4.g - ***Course Status*** hides as screen narrows

Figure 2.4.4.h - Hide ***Kambaz*** and ***Courses Navigation*** at narrowest screen

Implement the controls at the top of the ***Module*** in a new ***ModulesControls*** component. Feel free to use the code below as an example or implement your own as long as it looks and behaves as described.

app/(Kambaz)/Courses/[cid]/Modules/ModulesControls.tsx

```
import { Button, Dropdown, DropdownItem, DropdownMenu, DropdownToggle } from "react-bootstrap";
import { FaPlus } from "react-icons/fa6";
import GreenCheckmark from "./GreenCheckmark";
export default function ModulesControls() {
 return (
   <div id="wd-modules-controls" className="text-nowrap">
     <Button variant="danger" size="lg" className="me-1 float-end" id="wd-add-module-btn">
       <FaPlus className="position-relative me-2" style={{ bottom: "1px" }} />
       Module
     </Button>
     <Dropdown className="float-end me-2">
       <DropdownToggle variant="secondary" size="lg" id="wd-publish-all-btn">
         <GreenCheckmark /> Publish All
       </DropdownToggle>
       <DropdownMenu>
         <DropdownItem id="wd-publish-all">
           <GreenCheckmark /> Publish All
         </DropdownItem>
         <DropdownItem id="wd-publish-all-modules-and-items">
           <GreenCheckmark /> Publish all modules and items
         </DropdownItem>
         <DropdownItem id="wd-publish-modules-only">
           <GreenCheckmark /> Publish modules only
         </DropdownItem>
         {/* Create two more items with IDs wd-unpublish-all-modules-and-items and wd-unpublish-modules-only with
             labels Unpublish all modules and items and Unpublish modules only */}
       </DropdownMenu>
     </Dropdown>
     {/* Implement the View Progress and Collapse All buttons with IDs wd-view-progress and wd-collapse-all */}
   </div>
);}
```

Import the component at the top of the ***Modules*** components and confirm they look as shown in Figures 2.4.4.f through 2.4.4.h. You might need to add several break elements to add some vertical spacing. Refactor the ***Modules*** component by replacing HTML tags ***ul*** and ***li*** with ***Bootstrap*** components ***ListGroup*** and ***ListGroupItem*** as shown below. The following code illustrates how you might implement and style modules ***Week 1*** and ***Week 2***. Feel free to use the code as a guide, or implement your own version as long as it looks and behaves as described. Override the ***btn-secondary*** and ***bg-secondary*** ***Bootstrap*** classes and implement a new border class as show in the ***style.css*** below.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
<div>
  <ModulesControls /><br /><br /><br /><br />
  <ListGroup className="rounded-0" id="wd-modules">
    <ListGroupItem className="wd-module p-0 mb-5 fs-5 border-gray">
      <div className="wd-title p-3 ps-2 bg-secondary"> Week 1 </div>
      <ListGroup className="wd-lessons rounded-0">
        <ListGroupItem className="wd-lesson p-3 ps-1">
          LEARNING OBJECTIVES </ListGroupItem>
        <ListGroupItem className="wd-lesson p-3 ps-1">
          Introduction to the course </ListGroupItem>
        <ListGroupItem className="wd-lesson p-3 ps-1">
          Learn what is Web Development </ListGroupItem>
      </ListGroup>
    </ListGroupItem>
    <ListGroupItem className="wd-module p-0 mb-5 fs-5 border-gray">
      <div className="wd-title p-3 ps-2 bg-secondary"> Week 2 </div>
```

app/(Kambaz)/styles.css

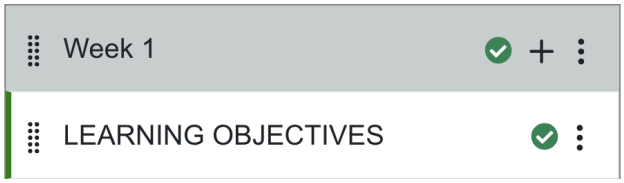
```
body { margin: 0;  padding: 0; }
body * { font-family:
         Arial, Helvetica,
         sans-serif; }
@media (min-width: 768px) {
  #wd-kambaz .wd-main-content-offset {
    padding-left: 140px !important;
  }
}

.btn-secondary {
  --bs-btn-color: #000 !important;
  --bs-btn-bg: #c7cdd1 !important;
  --bs-btn-border-color: #c7cdd1 !important;
  --bs-btn-hover-color: #fff !important;
}
```

```
      <ListGroup className="wd-lessons rounded-0">
        <ListGroupItem className="wd-lesson p-3 ps-1">
          LESSON 1 </ListGroupItem>
        <ListGroupItem className="wd-lesson p-3 ps-1">
          LESSON 2 </ListGroupItem>
      </ListGroup>
    </ListGroupItem>
  </ListGroup>
</div>
```

```
.bg-secondary {
  --bs-secondary-rgb: 199, 205, 209;
}
.list-group-item {
  --bs-list-group-border-color: gray;
}
.wd-lesson {
  border-left: 3px solid green !important;
}
```

Each module title and lesson has a set of control buttons at the right end of the title as shown here on the right. Implement each set of control buttons as separate component. Start with the ***LessonControlButtons*** with the ***GreenCheckmark*** plus a vertical ellipsis as shown below. Using the component below as an example, implement ***ModuleControlButtons*** with an additional ***BsPlus*** icon.



app/(Kambaz)/Courses/[cid]/Modules/LessonControlButtons.tsx

```
import { IoEllipsisVertical } from "react-icons/io5";
import GreenCheckmark from "./GreenCheckmark";
export default function LessonControlButtons() {
  return (
    <div className="float-end">
      <GreenCheckmark />
      <IoEllipsisVertical className="fs-4" />
    </div> );}
```

Add the new ***LessonControlButtons*** and ***ModuleControlButtons*** components to the ***Modules*** component as shown below.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
  <ModulesControls /><br /><br /><br />
  <ListGroup className="rounded-0" id="wd-modules">
    <ListGroupItem className="wd-module p-0 mb-5 fs-5 border-gray">
      <div className="wd-title p-3 ps-2 bg-secondary">
        <BsGripVertical className="me-2 fs-3" /> Week 1 <ModuleControlButtons />
      </div>
      <ListGroup className="wd-lessons rounded-0">
        <ListGroupItem className="wd-lesson p-3 ps-1">
          <BsGripVertical className="me-2 fs-3" /> LEARNING OBJECTIVES <LessonControlButtons />
        </ListGroupItem>
        <ListGroupItem className="wd-lesson p-3 ps-1">
          <BsGripVertical className="me-2 fs-3" /> Introduction to the course <LessonControlButtons />
        </ListGroupItem>
        ...
      </ListGroup>
    </ListGroupItem>
  </ListGroup>
```

2.4.5 Styling the Home Screen

Since the ***Modules*** in the ***Modules Screen*** are also used in the ***Home Screen***, we almost done with the ***Home Screen***. The only thing left to style is the the ***CourseStatus*** component at the right of the ***Home Screen*** contains several buttons. Here's an example how the first few buttons can styled. Implement and style the rest of the buttons so that the ***Home Screen*** looks like and behaves as documented in the screen shot in Figures 4.4. Your implementation does not need to pixel perfect. Use your own access to ***Canvas*** to help guide your design.

app/(Kambaz)/Courses/Home/Status.tsx

```
import { MdDoNotDisturbAlt } from "react-icons/md";
import { FaCheckCircle } from "react-icons/fa";
import { BiImport } from "react-icons/bi";
import { LiaFileImportSolid } from "react-icons/lia";
import { Button } from "react-bootstrap";
{/* Find more icons */}
export default function CourseStatus() {
 return (
   <div id="wd-course-status" style={{ width: "350px" }}>
     <h2>Course Status</h2>
     <div className="d-flex">
       <div className="w-50 pe-1">
         <Button variant="secondary" size="lg" className="w-100 text-nowrap ">
           <MdDoNotDisturbAlt className="me-2 fs-5" /> Unpublish </Button> </div>
       <div className="w-50">
         <Button variant="success" size="lg" className="w-100">
           <FaCheckCircle className="me-2 fs-5" /> Publish </Button> </div>
     </div>
     <br />
     <Button variant="secondary" size="lg" className="w-100 mt-1 text-start">
       <BiImport className="me-2 fs-5" /> Import Existing Content </Button>
     <Button variant="secondary" size="lg" className="w-100 mt-1 text-start">
       <LiaFileImportSolid className="me-2 fs-5" /> Import from Commons </Button>
     {/* Complete the rest of the buttons */}
   </div> );}
```

2.4.6 Implementing the People Screen

The ***People*** screen displays a list of students, teaching assistants, and faculty associated with the selected course. This section implements and styles a prototype of the screen.

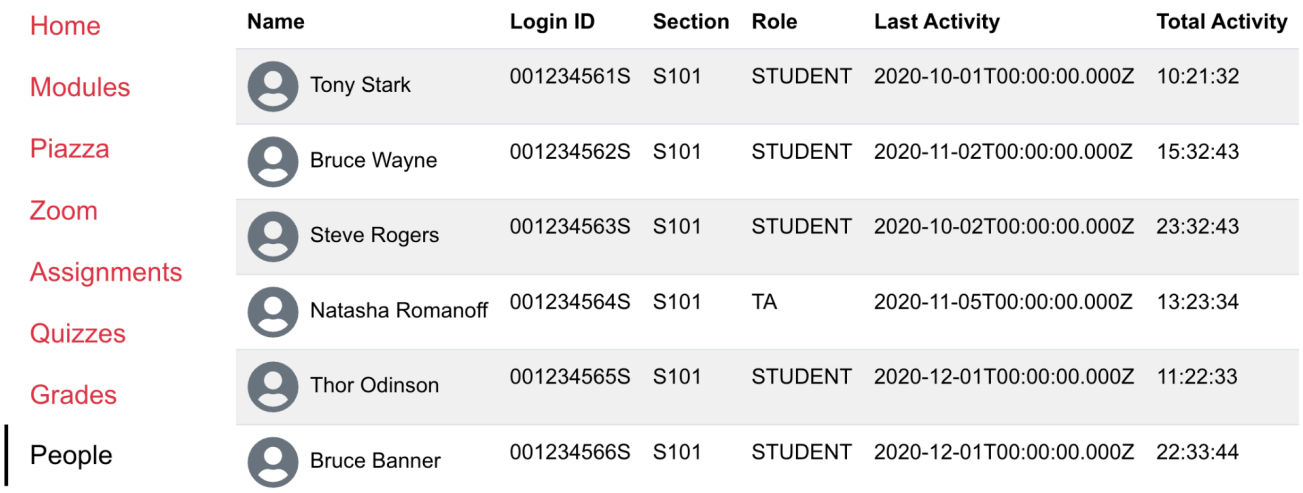


Figure 2.4.6 - The People Screen displaying a table of people enrolled in the course

The component below implements the ***People*** screen as a table that displays each user as a separate row. Practice by showing a total of three users. Navigate to a course from the ***Dashboard*** and then to the ***People*** screen to confirm that it renders as shown above.

app/(Kambaz)/Courses/[cid]/People/Table/page.tsx

```
import { Table } from "react-bootstrap";
import { FaUserCircle } from "react-icons/fa";
export default function PeopleTable() {
 return (
  <div id="wd-people-table">
   <Table striped>
```

```
   <thead>
    <tr><th>Name</th><th>Login ID</th><th>Section</th><th>Role</th><th>Last Activity</th><th>Total Activity</th></tr>
   </thead>
   <tbody>
    <tr><td className="wd-full-name text-nowrap">
         <FaUserCircle className="me-2 fs-1 text-secondary" />
         <span className="wd-first-name">Tony</span>{" "}
         <span className="wd-last-name">Stark</span></td>
     <td className="wd-login-id">001234561S</td>
     <td className="wd-section">S101</td>
     <td className="wd-role">STUDENT</td>
     <td className="wd-last-activity">2020-10-01</td>
     <td className="wd-total-activity">10:21:32</td></tr>
         {/* Add at least 3 more users such as Bruce Wayne, Steve Rogers, and Natasha Romanoff */}
   </tbody>
  </Table>
 </div> );}
```

2.4.7 Styling the Assignments Screen (On Your Own)

Clicking ***Assignments*** in the ***Course Navigation*** sidebar displays the ***Assignments*** screen as shown on the right.

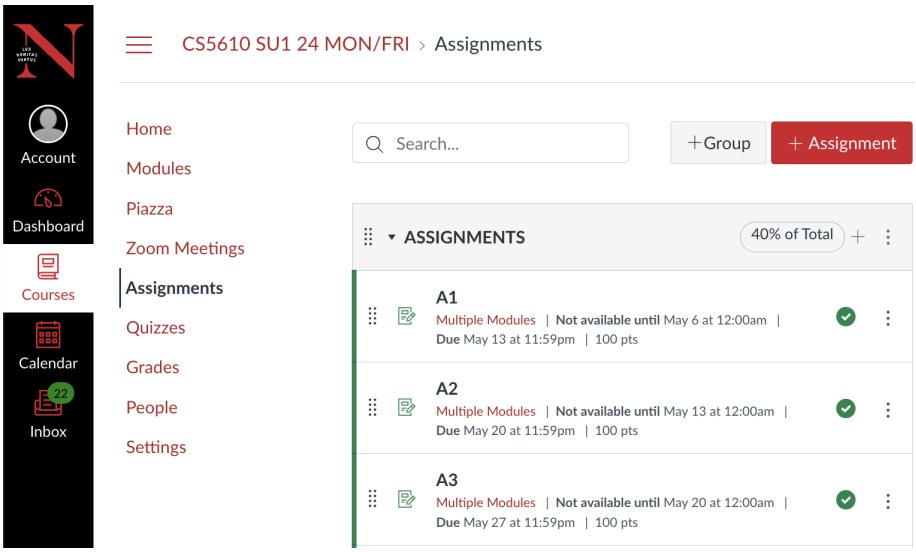


Figure 2.4.7 - The Assignments Screen Screenshot

On your own, use the implementation of the ***Home*** and ***Modules Screen***s as examples, and create and style the ***Assignments Screen*** as shown here on the right. The implementation does not have to be pixel perfect, but make an effort to use icons, colors, spacing, font size, as close as possible to the screen shot shown here on the right. Make an effort to style the ***Assignments Screen*** as shown above, but it doesn't need to be pixel perfect. Here are some requirements you need to follow

- Buttons must be floated to the right as shown
- The ***Search for Assignment*** text field must render as shown including the ***placeholder*** text, the ***magnifying glass***, and justified to the left.
- Use ***React Icons*** similar to the ones shown
- The ***Group*** and ***Assignment*** buttons must be justified to the right, in the color shown, and ***plus*** icons. Use the same colors used in the ***Modules*** and ***Home Screen***
- White spaces around and between content must be rendered with ***Bootstrap***'s margin and padding classes
- The border to the left of the line items must be rendered green as shown
- The titles of the assignments, ***A1***, ***A2***, etc, must be rendered as shown
- The subtext under the titles such as the due dates, start times, and points, must render as shown, but the dates and times can be different

2.4.8 Styling Edit Assignment Screen (On Your Own)

Clicking on the title of any assignment displays the ***Edit Assignment Screen*** as shown below in Figure 2.4.8. For now this screen displays the same content regardless which assignment you click. In later chapters the content will be different depending which assignment you click. The screen provides a form elements for faculty to edit the assignment including the ***Assignment Name***, ***Description***, ***Points***, and ***Due Date***. Use ***React Bootstrap Components*** such as ***Form***, ***Form.Group***, and ***Form.Label***, to style the ***Edit Assignment*** screen as close to the screen shot here on the right. On your own, make an effort to style the screen as close as possible to the wireframes provided, but it is not required to be pixel perfect.

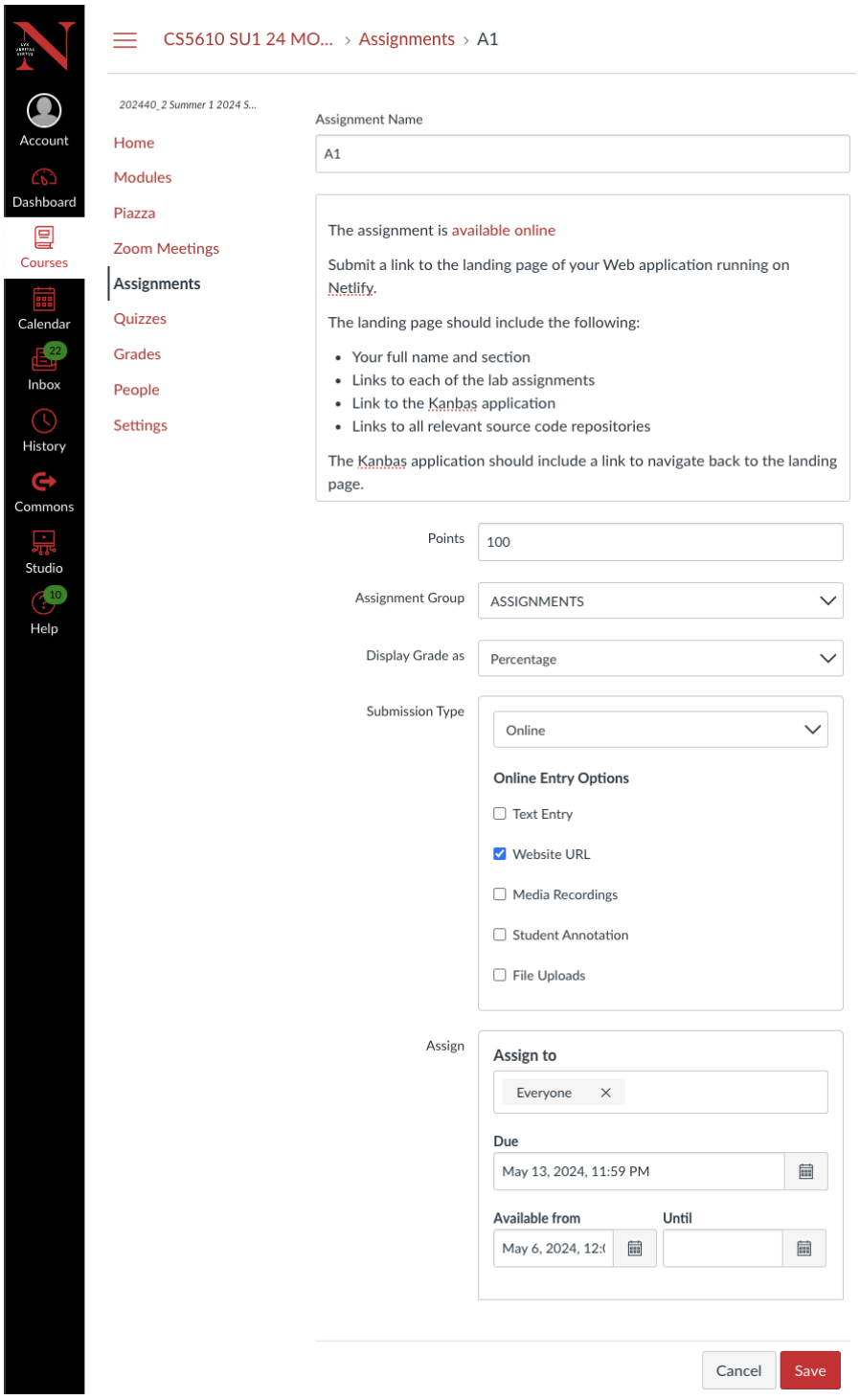


Figure 2.4.8 - The Assignment Editor Screen Screenshot

2.4.9 Styling the Account Screens (On Your Own)

Using Bootstrap classes and the styling of the ***Course Navigation*** sidebar as an example, style the ***Account Screens*** so that they look as shown in Figures 2.4.9. Using ***form*** and ***button*** Bootstrap classes, style the ***Sign In*** screen as shown below. Also style the ***Account Navigation*** sidebar based on the ***Course Navigation*** sidebar.

app/(Kambaz)/Account/Signin/page.tsx

```
import Link from "next/link";
export default function Signin() {
  return (
    <div id="wd-signin-screen">
      <h1>Sign in</h1>
      <Form.Control id="wd-username"
             placeholder="username"
             className="mb-2"/><br />
      <Form.Control id="wd-password"
             placeholder="password" type="password"
             className="mb-2"/><br />
      <Link id="wd-signin-btn"
            href="/Account/Profile"
            className="btn btn-primary w-100 mb-2">
            Sign in </Link><br />
      <Link id="wd-signup-link" href="/Account/Signup">Sign up</Link>
    </div> );}
```

Using the ***Sign In*** screen as an example, style the ***Sign Up*** and ***Profile*** screens so that they look as shown below.

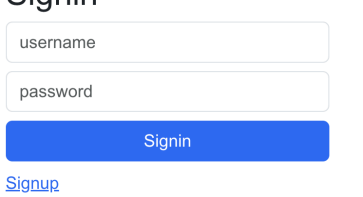


Figure 2.4.9.a
Styled Signin Screen

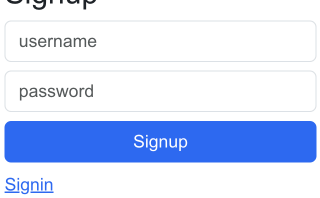


Figure 2.4.9.b
Styled Signup Screen

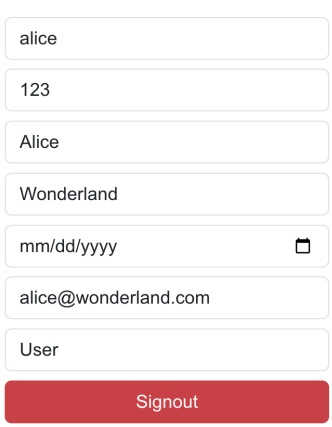


Figure 2.4.9.c
Styled Profile Screen

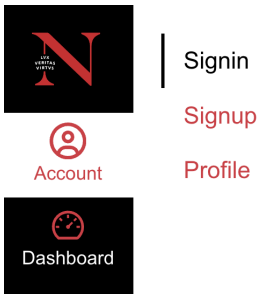


Figure 2.4.9.d
Styled Account Navigation

2.6 Delivery

1. In the same React application created in earlier chapters, ***kambaz-next-js***, complete all the exercises described in this document
2. In a branch called ***a2***, add, commit and push the source code of the React application ***kambaz-next-js*** to the same remote source repository in ***GitHub.com*** created in an earlier chapter. Here's an example of how to add, commit and push your code

```
$ git checkout -b a2
$ git add .
$ git commit -am "a2 CSS and Bootstrap"
```

```
$ git push
```

3. Deploy the ***a2*** branch to the same ***Vercel*** created in an earlier chapter. Configure Vercel to deploy all branches to separate URLs. From your Vercel's dashboard go to ***Site settings > Build & deploy > Branches > Branch deploys*** and select ***All***. Now each time you commit to a branch, the application will be available at a URL that contains the name of the branch
4. Push the source code of the React application ***kambaz-next-js*** to a remote source repository in ***GitHub.com*** as described
5. ***Labs/page.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kambaz. Add a link to your repository in GitHub. The link should have an ***ID*** attribute with a value of ***wd-github***.
6. In ***Labs/page.tsx***, add your full name: first name first and last name second. Use the same name as in Canvas.
7. Deploy the branch for this chapter to your ***kambaz-next-js*** React application to ***Vercel*** as described.
8. As a deliverable in ***Canvas***, submit the URL to the ***a2*** branch deployment of your React application running on Vercel.

Chapter 3 - Creating Single Page Applications with React

JavaScript, officially known as ***ECMAScript***, is a programming language most famous for writing scripts that run in and control web browsers. The name "***ECMAScript***" originates from the ***European Computer Manufacturers Association*** (***ECMA***), which standardized the language in 1997 to ensure consistency across implementations. While "***JavaScript***" is the popular name used by developers, ***ECMAScript*** is its formal designation, reflecting its roots in the standardization process. Over the years, ***ECMAScript*** has evolved significantly, with a major milestone in 2015 when ***ECMAScript 2015*** (commonly referred to as ***ES6***) was released. This version introduced transformative features like ***arrow functions***, ***classes***, ***let***/***const*** declarations, and ***modules***, making JavaScript more robust and suitable for modern application development. These advancements laid the groundwork for libraries like ***React***, which simplifies building ***Single Page Applications*** (***SPA***s). While ***JavaScript*** was historically used for creating static web pages, this chapter explores how to leverage it for handling user input, manipulating data structures, parameterizing components, and creating data-driven user interfaces with React.

3.1 Learning Objectives

By the end of this chapter, you will be able to:

- Understand the basics of JavaScript and its role in Web development
- Learn about **JavaScript variables, constants, data types** and their applications
- Work with **Boolean variables and conditionals**
- Use the **ternary conditional operator** to generate conditional output
- Define and utilize **JavaScript functions**, including arrow functions and implied returns
- Implement **template string literals** for efficient string handling
- Manipulate **JavaScript data structures**, including arrays and objects
- Utilize **array functions** such as `map()`, `find()`, `filter()`, and `findIndex()`
- Convert data using **JSON (JavaScript Object Notation)** and `JSON.stringify()`
- Learn and apply **the spread operator** and destructuring in JavaScript
- Use **dynamic styling** in React applications with HTML classes and style attributes
- Parameterize React components using **child components** and **location-based data**
- Implement a **data-driven Kambaz application**
- Work with **data-driven navigation, dashboards, and courses**
- Develop **data-driven modules and assignments screens**
- Understand the structure of a **single-page application (SPA)** using React.

3.2 Introduction to JavaScript

To achieve the learning objectives of understanding JavaScript fundamentals and their application in React for building dynamic SPAs, we'll begin with hands-on exercises that cover core concepts like variables, functions, data structures, and more.

In the following exercises, we'll learn about the ***JavaScript*** programming language. We'll create a component for each exercise and import them into the ***Lab3*** component. Create ***app/Labs/Lab3/page.tsx*** and import it from the ***Labs*** component as shown below. Make sure the ***Lab3*** route ends with an asterisk (*) since several exercises will rely on nested routes embedded in the ***Lab3*** screen.

app/Labs/Lab3/page.tsx

```
export default function Lab3() {
```

```
  return (
    <div id="wd-lab3">
      <h3>Lab 3</h3>
    </div>
);}
```

3.2.1 Variables and Constants

Variables can store state information about applications. To practice declaring variables and constants, create the ***VariablesAndConstants*** component below and import it from the ***Lab3*** component. Confirm the browser displays as shown on the right. We'll be creating several components to practice various features of the ***JavaScript*** language. Import them into the ***Lab3*** component and confirm the output is as described for each of the lab exercises.

app/Labs/Lab3/VariablesAndConstants.tsx

```
export default function VariablesAndConstants() {
 var functionScoped = 2;
 let blockScoped = 5;
 const constant1 = functionScoped - blockScoped;
 return(
   <div id="wd-variables-and-constants">
     <h4>Variables and Constants</h4>
     functionScoped = { functionScoped }<br/>
     blockScoped = { blockScoped }<br/>
     constant1 = { constant1 }<hr/>
   </div>
);}
```

app/Labs/Lab3/page.tsx

```
import VariablesAndConstants from "./VariablesAndConstants";

export default function Lab3() {
  return(
    <div id="wd-lab3">
      <h3>Lab 3</h3>
      <VariablesAndConstants/>
    </div>
  );
}
```

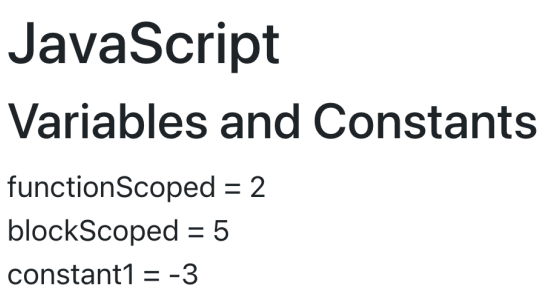


Figure 3.2.1 - Variables and Constants

3.2.2 Data Types

JavaScript declares several datatypes such as ***Number***, ***String***, ***Date***, and so on. To practice with variable types, create the ***VariableTypes*** component shown below and import it at the bottom of the ***Lab3*** component. Confirm that the browser renders as shown here on the right. Note that we had to convert the boolean variable into a string type before it could render in the browser.

app/Labs/Lab3/VariableTypes.tsx

```
export default function VariableTypes() {
  let numberVariable = 123;
  let floatingPointNumber = 234.345;
  let stringVariable = 'Hello World!';
  let booleanVariable = true;
  let isNumber = typeof numberVariable;
```

```
  let isString = typeof stringVariable;
  let isBoolean = typeof booleanVariable;
  return(
    <div id="wd-variable-types">
      <h4>Variables Types</h4>
      numberVariable = { numberVariable }<br/>
      floatingPointNumber = { floatingPointNumber }<br/>
      stringVariable = { stringVariable }<br/>
      booleanVariable = { booleanVariable + "" }<br/>
      isNumber = { isNumber }<br/>
      isString = { isString }<br/>
      isBoolean = { isBoolean }<hr/>
    </div>
);}
```

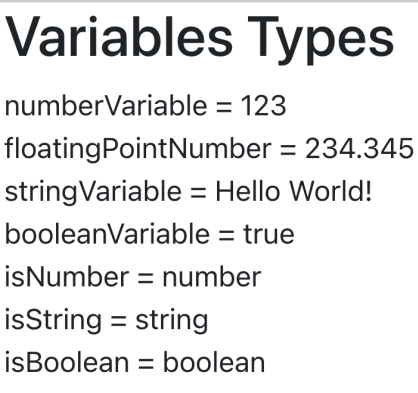


Figure 3.2.2 - Data Types

3.2.3 Boolean Variables

To practice with Boolean data types, create a component called ***BooleanVariables*** and import it in the ***Lab3*** component. Use the previous lab exercises as a guide of how to complete this exercise. The new component should add a new section called ***Boolean Variables*** that displays each of the new variables so that the browser renders as shown below on the right. You might need to cast the boolean values to string by concatenating an empty string to display the variables, e.g., ***false3 = {false3 + ""}
. Add ***function, ***export***, and other keywords as needed.

app/Labs/Lab3/BooleanVariables.tsx

```
let numberVariable = 123, floatingPointNumber = 234.345;
let true1 = true, false1 = false;
let false2 = true1 && false1;
let true2 = true1 || false1;
let true3 = !false2;
let true4 = numberVariable === 123; // always use === not ==
let true5 = floatingPointNumber !== 321.432;
let false3 = numberVariable < 100;
return (
  <div id="wd-boolean-variables">
    <h4>Boolean Variables</h4>
    true1     = {true1 + ""}     <br />
    false1    = {false1 + ""}    <br />
    false2    = {false2 + ""}    <br />
    true2     = {true2 + ""}     <br />
    true3     = {true3 + ""}     <br />
    true4     = {true4 + ""}     <br />
    true5     = {true5 + ""}     <br />
    false3    = {false3 + ""}    <hr />
  </div>
);
```

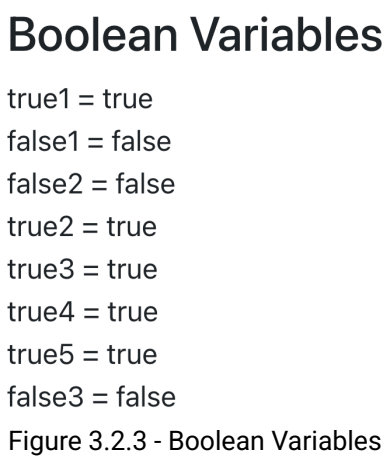


Figure 3.2.3 - Boolean Variables

3.2.4 Conditionals

Conditional expressions allow scripts to make decisions based on predicates that compare values and variables. Scripts can decide to execute different parts of the code based on the result of these predicates using ***if/else*** and other constructs. Create the following components and import them into the ***Lab3*** component. Confirm that the components render as shown.

The most common use of conditionals is ***if/else*** statements that evaluate a predicate and can decide to execute one of two different code blocks depending on whether the predicate evaluates to ***true*** or ***false***. To practice with ***if/else***, create a component called ***IfElse*** based on the code shown below. It should render a new section labeled ***If Else*** and render as shown below. The ***true1*** paragraph is only rendered if ***true1*** is true. The ***ternary operators*** ? and : can be used to render one of two options based on the value of a boolean expression.

app/Labs/Lab3/IfElse.tsx

```
let true1 = true, false1 = false;
...
return (
   <div id="wd-if-else">
      <h4>If Else</h4>
      { true1 && <p>true1</p> }
      { !false1 ? <p>!false1</p> : <p>false1</p> } <hr/>
   </div>
)
```

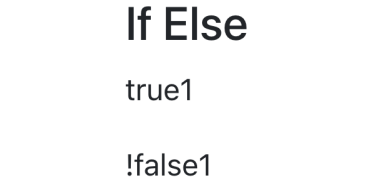


Figure 3.2.4 - Conditionals

3.2.5 Ternary Conditional Operator

Ternary conditional operators are concise alternative to ***if/else*** statements. It takes three arguments

1. A ***predicate*** expression that evaluates to true or false followed by a question mark (?)

2. An expression that evaluates if the ***predicate*** is ***true*** followed by a colon (:)
3. Followed by an expression that evaluates iff the ***predicate*** is ***false***

To practice the ternary operator, create a new component called ***TernaryOperator*** based on the code shown below which should render as shown below on the right.

app/Labs/Lab3/TernaryOperator.tsx

```
let loggedIn = true;
return(
   <div id="wd-ternary-operator">
      <h4>Logged In</h4>
      { loggedIn ? <p>Welcome</p> : <p>Please login</p> } <hr/>
   </div>
)
```

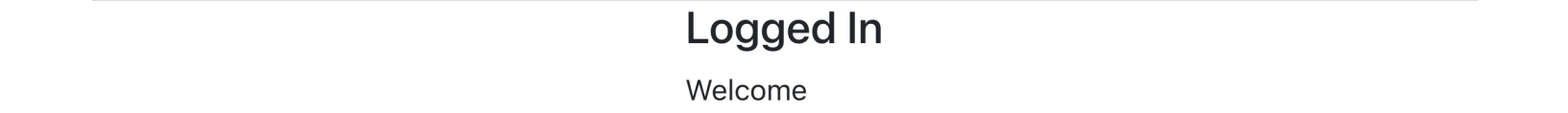


Figure 3.2.5 - Ternary Conditional Operator

3.2.6 Generating conditional output

With boolean expressions we can render content based on some logic. The following example decides rendering one content versus another based on a simple boolean constant ***loggedIn***. If a user is ***loggedIn***, then the component renders a greeting, otherwise suggests the user should login. Implement the following component to practice conditional rendering.

app/Labs/Lab3/ConditionalOutputIfElse.tsx

```
const ConditionalOutputIfElse = () => {
  const loggedIn = true;
  if(loggedIn) {
    return (<h2 id="wd-conditional-output-if-else-welcome">Welcome If Else</h2>);
  } else {
    return (<h2 id="wd-conditional-output-if-else-login">Please login If Else</h2>);
  }
};
export default ConditionalOutputIfElse;
```

A more compact way we can achieve the same thing is by including the conditional content in a boolean expression that short circuits the content if its false, or evaluates the expression if it's true. Implement the equivalent component shown.

app/Labs/Lab3/ConditionalOutputInline.tsx

```
const ConditionalOutputInline = () => {
 const loggedIn = false;
 return (
   <div id="wd-conditional-output-inline">
     { loggedIn && <h2>Welcome Inline</h2>      }
     {!loggedIn && <h2>Please login Inline</h2> }
   </div>
);};
export default ConditionalOutputInline;
```

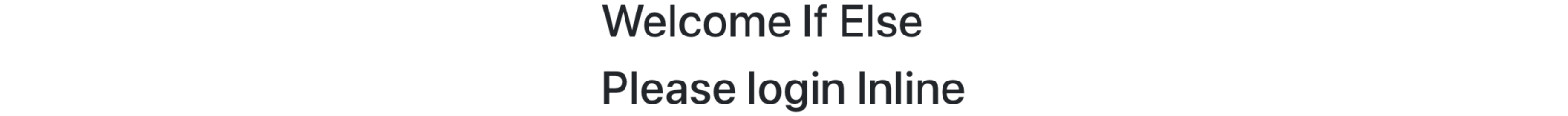


Figure 3.2.6 - Generating conditional output

3.3 JavaScript Functions

Functions allow reusing an algorithm by wrapping it in a named, parameterized block of code. ***JavaScript*** supports two styles of functions based on the history of language. Functions are declared using the following syntax.

```
function <functionName> (<parameterList>) { <functionBody> }
```

To practice using functions create a new component called ***LegacyFunctions*** based on the code below. Import this new component in the ***Lab3*** component and confirm the browser renders as shown.

app/Labs/Lab3/LegacyFunctions.tsx

```
function add(a: number, b: number) {
  return a + b;
}
export default function LegacyFunctions() {
  const twoPlusFour = add(2, 4);
  console.log(twoPlusFour);
  return (
    <div id="wd-legacy-functions">
      <h4>Functions</h4>
      <h5>Legacy ES5 functions</h5>
      twoPlusFour = {twoPlusFour}    <br />
      add(2, 4) = {add(2, 4)}        <hr />
    </div>
);}
```

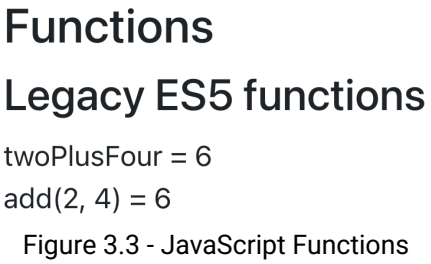


Figure 3.3 - JavaScript Functions

3.3.1 Arrow functions

A new version of ***JavaScript*** was introduced in 2015 and is officially referred to as ***ECMAScript 6*** or ***ES6***. A new syntax for declaring functions was introduced which is less verbose and provides tons of new features we'll explore throughout this course. This function syntax is often referred to as ***arrow functions***. To practice using ES6 functions, create a new component called ***ArrowFunctions*** based on the code below. Import this new component in the ***Lab3*** component and confirm the browser renders as shown.

app/Labs/Lab3/ArrowFunctions.tsx

```
const subtract = (a: number, b: number) => {
  return a - b;
};

export default function ArrowFunctions() {
  const threeMinusOne = subtract(3, 1);
  console.log(threeMinusOne);
  return (
    <div id="wd-arrow-functions">
      <h4>New ES6 arrow functions</h4>
      threeMinusOne = {threeMinusOne}   <br />
      subtract(3, 1) = {subtract(3, 1)} <hr />
```

```
    </div>
  );
}
```

NOTE: Throughout the last couple of exercises we've provided code in the return statement to render the variables in the browser and asked that you confirm the output matches. Going forward we'll omit the return statement, but continue to implement it and confirm the output matches the one provided.

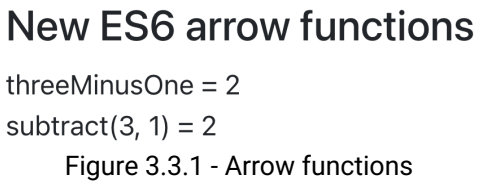


Figure 3.3.1 - Arrow functions

3.3.2 Implied Returns

One of the new features of the new ES6 functions is ***implied returns***, that is, if the body of the function consists of just returning some value or expression, then the return statement is optional and can be replaced with just the value or expression. To practice this feature create a new component called ***ImpliedReturn*** based on the code below. Import this new component in the ***Lab3*** component and confirm the browser renders as shown.

app/Labs/Lab3/ImpliedReturn.tsx

```
const multiply = (a: number, b: number) => a * b;
const fourTimesFive = multiply(4, 5);
console.log(fourTimesFive);

return (
  <div id="wd-implied-return">
    <h4>Implied return</h4>
    fourTimesFive = {fourTimesFive}   <br />
    multiply(4, 5) = {multiply(4, 5)} <hr />
  </div>
);
```

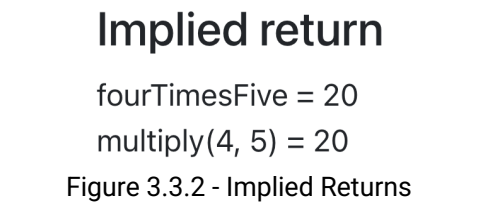


Figure 3.3.2 - Implied Returns

3.3.3 Template String Literals

Generating dynamic HTML consists of writing code that manipulates and concatenates strings to generate new HTML strings based on some program logic. Basically consists of one language writing code in another language, symilar to what a compiler does. Working with strings can be error prone especially if you have to use lots of extra operations and variables to concatenate the resulting string. JavaScript template strings provide a better approach by allowing embedding expressions and algorithms right within strings themselves. To practice, implement a new component called ***TemplateLiterals*** based on the code below. Import this new component in ***JavaScript*** and confirm the browser renders as shown. In your return statement, wrap the HTML output in a ***DIV*** whose ***ID*** is ***wd-template-literals***. Do not hard code the results ***5***, ***Welcome home alice***, etc. Instead use the values of variables ***result1***, ***result2***, etc., to render the the output shown above.

app/Labs/Lab3/TemplateLiterals.tsx

```
export default function TemplateLiterals() {
  const five = 2 + 3;
  const result1 = "2 + 3 = " + five;
  const result2 = `2 + 3 = ${2 + 3}`;
  const username = "alice";
  const greeting1 = `Welcome home ${username}`;
  const loggedIn = false;
  const greeting2 = `Logged in: ${loggedIn ? "Yes" : "No"}`;
  return (
    <div id="wd-template-literals">
      <h4>Template Literals</h4>
      result1 = {result1}     <br />
      result2 = {result2}     <br />
      greeting1 = {greeting1} <br />
      greeting2 = {greeting2} <hr />
    </div>
  );
}
```

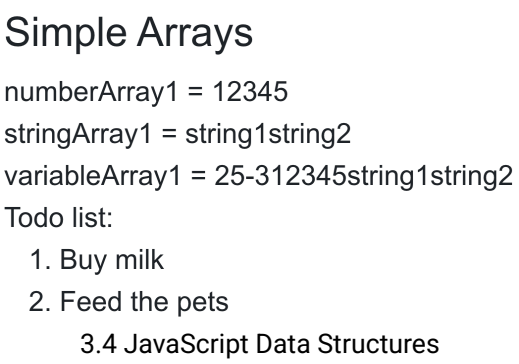
3.3.3 Template String Literals

3.4 JavaScript Data Structures

Up to this point we have been discussing ***primitive datatypes*** such as ***strings***, ***numbers***, and ***booleans***. These can be combined into ***complex datatypes*** such as ***arrays*** and ***objects***. Arrays can group together several values into a single variable. Arrays can group together values of different datatypes, e.g., number arrays, string arrays, and even a mix and match of datatypes in the same array. Not that you would ever want to actually do that. To practice with arrays create a component called ***SimpleArrays*** and use the code below as a guide to rendering the content on the right. Import the component to the ***Lab3*** component and confirm the browser renders as shown. Note that the arrays render without the commas. This feature will come in handy when the array items are ***HTML*** elements.

app/Labs/Lab3/SimpleArrays.tsx

```
export default function SimpleArrays() {
  var functionScoped = 2;  let blockScoped = 5;
  const constant1 = functionScoped - blockScoped;
  let numberArray1 = [1, 2, 3, 4, 5];
  let stringArray1 = ["string1", "string2"];
  let htmlArray1 = [<li>Buy milk</li>, <li>Feed the pets</li>];
  let variableArray1 = [ functionScoped, blockScoped, constant1,
                         numberArray1, stringArray1 ];
  return (
    <div id="wd-simple-arrays">
      <h4>Simple Arrays</h4>
      numberArray1 = {numberArray1}     <br />
      stringArray1 = {stringArray1}     <br />
      variableArray1 = {variableArray1} <br />
      Todo list:
      <ol>{htmlArray1}</ol>
      <hr />
    </div> );}
```

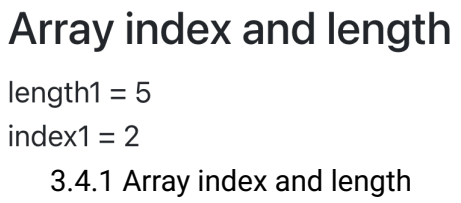


3.4.1 Array index and length

An array's length is available as property ***length***. The ***indexOf()*** function allows finding where a particular array member is found. To practice with array indices and length, implement a new component called ***ArrayIndexAndLength*** based on the code below. Import this new component in ***Lab3*** and confirm the browser renders as shown.

app/Labs/Lab3/ArrayIndexAndLength.tsx

```
export default function ArrayIndexAndLength() {
  let numberArray1 = [1, 2, 3, 4, 5];
  const length1 = numberArray1.length;
  const index1 = numberArray1.indexOf(3);
  return (
    <div id="wd-array-index-and-length">
      <h4>Array index and length</h4>
      length1 = {length1} <br />
      index1 = {index1}   <hr />
    </div>
);}
```



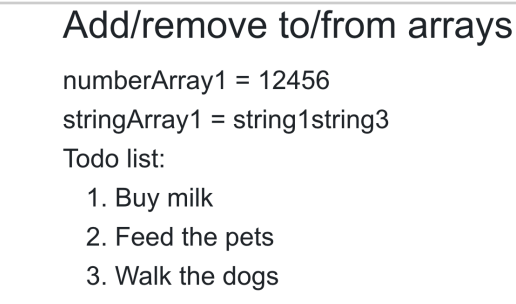
3.4.2 Adding and Removing Data to/from Arrays

In most languages arrays are immutable, whereas in JavaScript elements can easily be added and removed from arrays. The ***push()*** function appends elements at the end of an array. The ***splice()*** function removes/adds elements anywhere in the array. To practice adding/removing data from arrays, implement component ***AddingAndRemovingDataToFromArrays*** based on the code below. Import this new component in ***Lab3*** and confirm the browser renders as shown.

app/Labs/Lab3/AddingAndRemovingToFromArrays.tsx

```
export default function AddingAndRemovingToFromArrays() {
  let numberArray1 = [1, 2, 3, 4, 5];
  let stringArray1 = ["string1", "string2"];
  let todoArray = [<li>Buy milk</li>, <li>Feed the pets</li>];
  numberArray1.push(6); // adding new items
  stringArray1.push("string3");
  todoArray.push(<li>Walk the dogs</li>);
  numberArray1.splice(2, 1); // remove 1 item starting at 2
  stringArray1.splice(1, 1);
  return (
    <div id="wd-adding-removing-from-arrays">
      <h4>Add/remove to/from arrays</h4>
```

```
      numberArray1 = {numberArray1} <br />
      stringArray1 = {stringArray1} <br />
      Todo list:
      <ol>{todoArray}</ol><hr />
    </div> );}
```



3.4.2 Adding and Removing Data to/from Arrays

3.4.3 For Loops

We can operate on each array value by iterating over them in a ***for loop***. To practice with for loops, implement a new component called ***ForLoops*** based on the code below. Import this new component in ***Lab3*** and confirm the browser renders as shown.

app/Labs/Lab3/ForLoops.tsx

```
export default function ForLoops() {
  let stringArray1 = ["string1", "string3"];
  let stringArray2 = [];
  for (let i = 0; i < stringArray1.length; i++) {
    const string1 = stringArray1[i];
    stringArray2.push(string1.toUpperCase());
  }
  return (
    <div id="wd-for-loops">
      <h4>Looping through arrays</h4>
      stringArray2 = {stringArray2} <hr />
    </div> );}
```

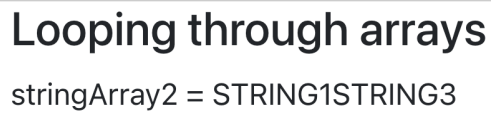


Figure 3.4.3 - For Loops

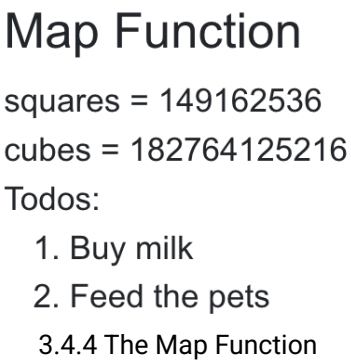
3.4.4 The Map Function

An array's ***map*** function can iterate over an array's values, apply a function to each value, and collate all the results in a new array. The first example below iterates over the ***numberArray1*** and calls the ***square*** function for each element. The ***square*** function was declared earlier in this document and it accepts a parameter and returns the square of the parameter. The ***map*** function collates all the squares into a new array called ***squares*** as shown below. The second example does the same thing, but uses a function that calculates the ***cubes*** of all numbers in the same ***numberArray1*** array. To practice with ***map***, implement a new component called ***MapFunction*** based on the code below. Import this new component in ***WorkingWithArrays*** and confirm the browser renders as shown.

app/Labs/Lab3/MapFunction.tsx

```
export default function MapFunction() {
  let numberArray1 = [1, 2, 3, 4, 5, 6];
```

```
  const square = (a: number) => a * a;
  const todos = ["Buy milk", "Feed the pets"];
  const squares = numberArray1.map(square);
  const cubes = numberArray1.map((a) => a * a * a);
  return (
    <div id="wd-map-function">
      <h4>Map Function</h4>
      squares = {squares} <br />
      cubes = {cubes} <br />
      Todos:
      <ol>
        {todos.map((todo) => (
          <li>{todo}</li>
        ))}
      </ol> <hr/>
    </div>
  );
}
```



3.4.4 The Map Function

3.4.5 The Find Function

An array's ***find*** function can search for an item in an array and return the element it finds. The find function takes a function as an argument that serves as a predicate. The predicate should return true if the element is the one you're looking for. The predicate function is invoked for each of the elements in the array and when the function returns true, the find function stops because it has found the element that it was looking for. To practice, implement a new component called ***FindFunction*** based on the code below. Import this new component in ***JavaScript*** and confirm the browser renders as shown.

app/Labs/Lab3/FindFunction.tsx

```
export default function FindFunction() {
  let numberArray1 = [1, 2, 3, 4, 5];
  let stringArray1 = ["string1", "string2", "string3"];
  const four = numberArray1.find((a) => a === 4);
  const string3 = stringArray1.find((a) => a === "string3");
  return (
    <div id="wd-find-function">
      <h4>Find Function</h4>
      four = {four} <br />
      string3 = {string3} <hr />
    </div>
);}
```

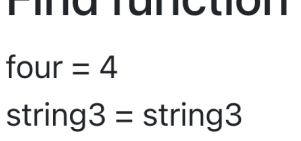


Figure 3.4.5 - The Find Function

3.4.6 The Find Index Function

Alternatively we can use ***findIndex*** function to determine the index where an element is located inside an array. Copy the code and display the content as shown below.

app/Labs/Lab3/FindIndex.tsx

```
let numberArray1 = [1, 2, 4, 5, 6];
let stringArray1 = ['string1', 'string3'];

const fourIndex = numberArray1.findIndex(a => a === 4);
const string3Index = stringArray1.findIndex(a => a === 'string3');
```

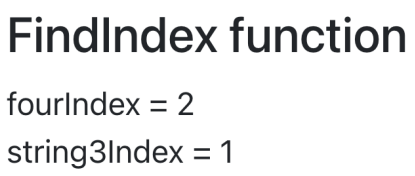


Figure 3.4.6 - The Find Index Function

3.4.7 The Filter Function

The ***filter*** function can look for elements that meet a criteria and collate them into a new array. For instance, the example below is looking through the ***numberArray1*** array for all values that are greater than 2. Then we look for all even numbers and then for all odd numbers. All the results are stored in corresponding arrays with appropriate names. To practice, implement a new component called ***FilterFunction*** based on the code below. Import this new component in ***Lab3*** and confirm the browser renders as shown.

app/Labs/Lab3/FilterFunction.tsx

```
export default function FilterFunction() {
  let numberArray1 = [1, 2, 4, 5, 6];
  const numbersGreaterThan2 = numberArray1.filter((a) => a > 2);
  const evenNumbers = numberArray1.filter((a) => a % 2 === 0);
  const oddNumbers = numberArray1.filter((a) => a % 2 !== 0);
  return (
    <div id="wd-filter-function">
      <h4>Filter Function</h4>
      numbersGreaterThan2 = {numbersGreaterThan2}  <br />
      evenNumbers = {evenNumbers}    <br />
      oddNumbers = {oddNumbers}      <hr />
    </div>
);}
```

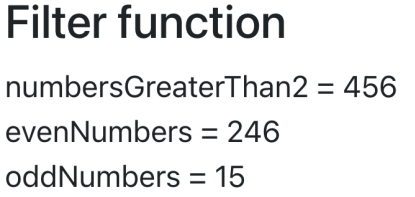


Figure 3.4.7 - The Filter Function

3.4.8 JSON Stringify

JavaScript has a global object called ***JSON*** which stands for ***JavaScript Object Notation***. The object provides several useful formatting functions such as ***stringify()*** and ***parse()***. Stringify converts ***JavaScript*** data structures to formatted strings. For instance let's format the following arrays and display them in the browser. Note how the array is rendered with square brackets and items are separated by commas.

app/Labs/Lab3/JsonStringify.tsx

```
export default function JsonStringify() {
  const squares = [1, 4, 16, 25, 36];
  return (
    <div className="wd-json-stringify">
      <h3>JSON Stringify</h3>
      squares = {JSON.stringify(squares)}
      <hr />
    </div>
);}
```

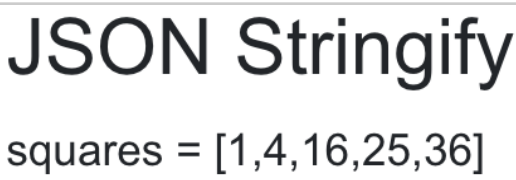


Figure 3.4.8 - JSON Stringify

3.4.9 JavaScript Object Notation (JSON)

Multiple values, of various datatypes can be combined together to create complex datatypes called ***objects***. For example the code below declares a ***house*** object collecting several numbers, strings, arrays, and other objects to represent a particular instance of a house. The ***house*** variable is assigned an ***object literal*** declared within opening and closing curly braces ***{*** and ***}***. Objects contain pairs of ***properties*** and values separated by commas. Values can be of any datatype including ***Number***, ***String***, ***Boolean***, arrays and other objects. In the example below we declared a ***house*** with 4 ***bedrooms***, 2.5 ***bathrooms*** and 2000 ***squareFeet***. The house has a nested object stored in property ***address*** which contains ***String*** properties such as ***street***, ***city*** and ***state***. The ***owners String*** array declares the names of the owners. To practice with JSON, create a ***House*** component as shown below, import it into the ***Lab3*** component, and confirm it renders as shown below.

app/Labs/Lab3/House.tsx

```
export default function House() {
  const house = {
    bedrooms: 4,      bathrooms: 2.5,
    squareFeet: 2000,
    address: {
      street: "Via Roma", city: "Roma", state: "RM", zip: "00100",  country: "Italy", },
    owners: ["Alice", "Bob"],
  };
  return (
    <div id="wd-house">
      <h4>House</h4>
      <h5>bedrooms</h5>      {house.bedrooms}
      <h5>bathrooms</h5>     {house.bathrooms}
      <h5>Data</h5>
      <pre>{JSON.stringify(house, null, 2)}</pre>
      <hr />
    </div>
);}
```

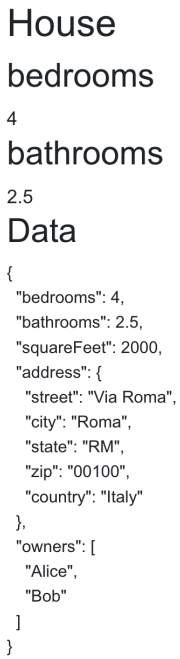


Figure 3.4.9 - JavaScript Object Notation (JSON)

3.4.10 Rendering a Data Structure

Let's bring together several of the concepts covered so far and implement a ***Todo*** list application that renders a list of todos dynamically using React. In a new directory ***app/Labs/Lab3/todo***, implement the ***TodoItem*** component in a ***TodoItem.tsx*** file as shown below. Import the component into the ***Lab3*** component and confirm that it renders as shown in the Figure 3.4.10a.

app/Labs/Lab3/todos/TodoItem.tsx

```
const TodoItem = ( { todo = { done: true, title: 'Buy milk',
                              status: 'COMPLETED' } }) => {
 return (
   <ListGroupItem>
     <input type="checkbox" className="me-2"
            defaultChecked={todo.done}/>
     {todo.title} ({todo.status})
   </ListGroupItem>
 );}
export default TodoItem;
```

Figure 3.4.10a - Rendering a Data Structure

Create a JSON file ***todos.json*** that contains an array of todos as shown below.

app/Labs/Lab3/todos/todos.json

```
[
  { "title": "Buy milk",          "status": "CANCELED",     "done": true   },
  { "title": "Pickup the kids",   "status": "IN PROGRESS",  "done": false  },
  { "title": "Walk the dog",      "status": "DEFERRED",     "done": false  }
]
```

Now let's implement a ***TodoList*** component that renders the array of todos as shown below. Import the component in ***Labs/Lab3/page.tsx***, refresh the browser, and confirm the ***TodoList*** renders a list of checkboxes and todo items.

/app/Labs/Lab3/todos/TodoList.tsx *Browser*

```
import TodoItem from "./TodoItem";
import todos from "./todos.json";
export default function TodoList() {
 return(
   <>
     <h3>Todo List</h3>
     <ListGroup>
       { todos.map(todo => {
           return(<TodoItem todo={todo}/>);   })}
     </ListGroup><hr/>
   </>
);}
```

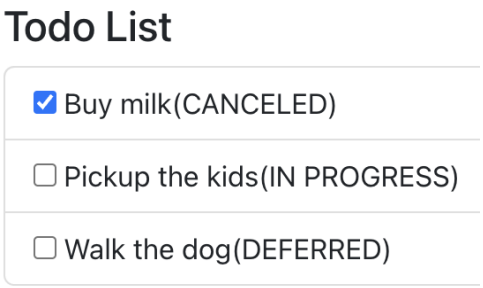


Figure 3.4.10b - Rendering a Data Structure

3.4.11 The Spread Operator

The spread operator (...) is used to expand, or copy an iterable object or array into another object or array. In the example below we declare array ***arr1*** and then copy its content (spread) into array ***arr2***. The resulting array ***arr2*** contains the contents of ***arr1***, followed by the rest of the items already declared in ***arr2***. The spread operator can also be applied to objects as illustrated in the following example. Below, ***obj1*** declares an object with three properties ***a***, ***b***, and ***c***. We then spread ***obj1*** onto ***obj2*** so that ***obj2*** ends up with the properties from both ***obj1*** and ***obj2***. When declaring ***obj3***, we first spread ***obj1*** and then declare ***b*** with a value of 4. Since ***obj1*** also has a property called ***b*** with a value of ***2***, there is a collision of properties in ***obj3***. The collision is resolved by keeping the last declaration overriding any previous values, so ***obj3.b*** ends up being ***4***. To practice the spread operator, create the ***Spreading*** component as shown below, import it in the ***Lab3*** component and confirm it renders as shown Figure 3.4.11.

app/Labs/Lab3/Spreading.tsx

```
export default function Spreading() {
 const arr1 = [ 1, 2, 3 ];
 const arr2 = [ ...arr1, 4, 5, 6 ];
 const obj1 = { a: 1, b: 2, c: 3 };
 const obj2 = { ...obj1, d: 4, e: 5, f: 6 };
 const obj3 = { ...obj1, b: 4 };
 return (
   <div id="wd-spreading">
     <h2>Spread Operator</h2>
     <h3>Array Spread</h3>
     arr1 = { JSON.stringify(arr1) }  <br />
     arr2 = { JSON.stringify(arr2) }  <br />
     <h3>Object Spread</h3>
     { JSON.stringify(obj1) }         <br />
     { JSON.stringify(obj2) }         <br />
     { JSON.stringify(obj3) }         <br />  <hr />
   </div>);}
```

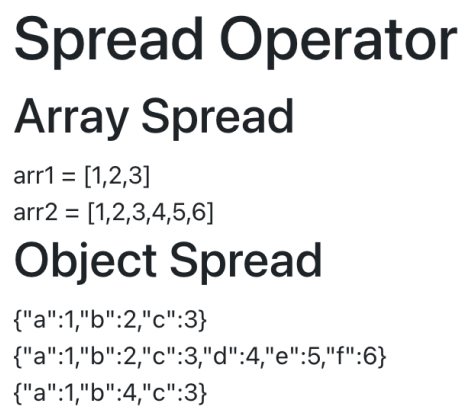


Figure 3.4.11 - The Spread Operator

3.4.12 Destructing

While the spreader operator is used to expand an iterable object into the list of arguments, the destructing operator is used to unpack values from arrays, or properties from objects, into distinct variables. In the example below we declare object ***person*** and array ***numbers***. These can be unpacked, or ***destructed***, into new variables or constants by an object's property name or an array's item position. The curly brackets around constants ***name*** and ***age***, destruct the object ***person*** on the right side of the assignment, and assigns the properties of the same name into the new constants. The constants ***name*** and ***age*** end up having the values of ***person.name*** and ***person.age*** respectively. Essentially it is the equivalent to

```
const name = person.name
const age = person.age
```

While object destructing is based on the names of the properties, destructing arrays is based on the positions of the items. In the example below, we declare the ***numbers*** array and then use the square brackets to destruct the array into new constants ***first***, ***second***, and ***third***. These new constants end up with the values of ***numbers[0]***, ***numbers[1]***, and ***numbers[2]***. Essentially it is equivalent to

```
const first = numbers[0]
const second = numbers[1]
const third = numbers[2]
```

To practice destructing objects and arrays, create component ***Destructing*** as shown below, import it in the ***Lab3*** component, and confirm it renders as shown below.

app/Labs/Lab3/Destructing.tsx ***Browser***

```
export default function Destructing() {
 const person = { name: "John", age: 25 };
 const { name, age } = person;
 // const name = person.name
 // const age = person.age
 const numbers = ["one", "two", "three"];
 const [ first, second, third ] = numbers;
 return (
   <div id="wd-destructing">
     <h2>Destructing</h2>
     <h3>Object Destructing</h3>
     const &#123; name, age &#125; =
           &#123; name: "John", age: 25 &#125;<br /><br />
     name = {name}<br />
     age = {age}
```

```
    <h3>Array Destructing</h3>
    const [first, second, third] = ["one","two","three"]<br/><br/>
    first = {first}<br />
    second = {second}<br />
    third = {third}<hr />
  </div>
);}
```

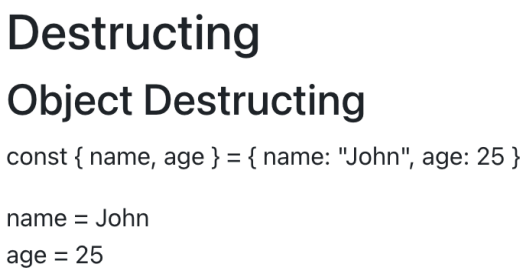


Figure 3.4.12a - Destructing objects

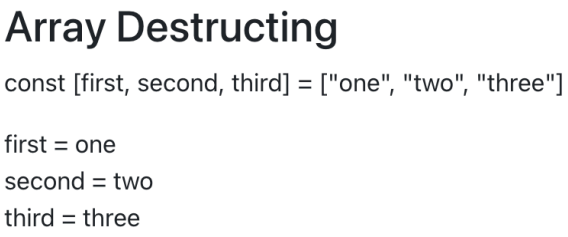


Figure 3.4.12b - Destructing arrays

3.4.13 Destructing Function Parameters

The destructing objects syntax is very popular in React, especially when passing parameters to functions. In the example below we declare two functions ***add*** and ***subtract*** using the new arrow function syntax. The ***add*** function takes two arguments ***a*** and ***b*** and returns the sum of the arguments. The ***subtract*** function takes a single object argument with properties ***a*** and ***b*** with values ***4*** and ***2***. In the argument list declaration, ***subtract*** uses object destructing to declare constants ***a*** and ***b*** which unpacks the values ***4*** and ***2*** from the object argument with properties of the same name. To practice ***function destructing***, copy the code below into a ***FunctionDestructing*** component, import it into the ***Lab3*** component, and confirm it renders as shown below on the right.

app/Labs/Lab3/FunctionDestructing.tsx

```
export default function FunctionDestructing() {
 const add = (a: number, b: number) => a + b;
 const sum = add(1, 2);
 const subtract = ({ a, b }: { a: number; b: number }) => a - b;
 const difference = subtract({ a: 4, b: 2 });
 return (
   <div id="wd-function-destructing">
     <h2>Function Destructing</h2>
     const add = (a, b) =&gt; a + b;<br />
     const sum = add(1, 2);<br />
     const subtract = (&#123; a, b &#125;) =&gt; a - b;<br />
     const difference = subtract(&#123; a: 4, b: 2 &#125;);<br/>
     sum = {sum}<br />
     difference = {difference} <hr />
   </div>
);}
```

Function Destructing

const add = (a, b) => a + b;
const sum = add(1, 2);
const subtract = ({ a, b }) => a - b;
const difference = subtract({ a: 4, b: 2 });
sum = 3
difference = 2

3.4.13 Destructing Function Parameters

3.4.14 Destructing Imports

Let's create a simple library to illustrate various ways of importing the functions and constants declared in the ***Math*** library below. The functions ***add***, ***subtract***, ***multiply***, and ***divide*** are all exported with the ***export*** keyword so that they can be imported individually. The ***Math*** constant declares an object containing references to the local functions. We export the ***Math*** object as the ***default export*** so that the functions can be imported as a single object map.

app/Labs/Lab3/Math.ts

```
export function add(a: number, b: number): number      { return a + b; }
export function subtract(a: number, b: number): number { return a - b; }
export function multiply(a: number, b: number): number { return a * b; }
export function divide(a: number, b: number): number   { return a / b; }
const Math = {
  add,
  subtract,
  multiply,
  divide,
};
export default Math;
```

To demonstrate how the functions can be imported in several ways, create the ***DestructingImports*** component below. The component implements a table we're going to fill out with three different ways we can import the functions from the ***Math*** library we implemented above. First, the functions can be imported as the single ***Math*** object that contains references to all the functions as ***import Math from "./Math"***. Then we can access each of the functions through the ***Math*** instance as ***Math.add()***, ***Math.subtract()***, etc. An alternative way to import the functions as a single object is using the ***import * as NAME from "./Math"***, where ***NAME*** is a custom local object name, e.g., ***Matematica***. We can then using the object's name to invoke the functions as ***Matematica.add()***, ***Matematica.subtract()***, etc. Finally, the functions can be imported individually by destructing the exported functions as ***import {add, subtract, multiply, divide} from "./Math"***.

app/Labs/Lab3/DestructingImports.tsx

```
import Math, { add, subtract, multiply, divide } from "./Math";
import * as Matematica from "./Math";
export default function DestructingImports() {
  return (
    <div id="wd-destructuring-imports">
      <h2>Destructing Imports</h2>
      <table className="table table-sm">
        <thead>
          <tr>
            <th>Math</th>
```

```
            <th>Matematica</th>
            <th>Functions</th>
          </tr>
        </thead>
        <tbody>
          {/* see next code block */}
        </tbody>
      </table>
      <hr />
    </div>
);}
```

Here are several examples demonstrating using the functions through the objects ***Math***, ***Matematica***, and then using the functions individually. Implement the ***Math.ts*** library and the ***DestructingImports*** component and confirm it renders as shown. Complete missing code.

app/Labs/Lab3/DestructingImports.tsx

```
<tr>
  <td>Math.add(2, 3) = {Math.add(2, 3)}</td>
  <td>Matematica.add(2, 3) =
     {Matematica.add(2, 3)}</td>
  <td>add(2, 3) = {add(2, 3)}</td>
</tr>
<tr>
  <td>Math.subtract(5, 1) = {Math.subtract(5, 1)}</td>
  <td>Matematica.subtract(5, 1) =
     {Matematica.subtract(5, 1)}</td>
  <td>subtract(5, 1) = {subtract(5, 1)}</td>
</tr>
```

Destructing Imports

Math	Matematica	Functions
Math.add(2, 3) = 5	Matematica.add(2, 3) = 5	add(2, 3) = 5
Math.subtract(5, 1) = 4	Matematica.subtract(5, 1) = 4	subtract(5, 1) = 4
Math.multiply(3, 4) = 12	Matematica.multiply(3, 4) = 12	multiply(3, 4) = 12
Math.divide(8, 2) = 4	Matematica.divide(8, 2) = 4	divide(8, 2) = 4

Figure 3.4.14 - Destructing Imports

3.5 Dynamic Styling

React can generate content dynamically based on algorithms written in JavaScript. We can also dynamically style the content by programmatically controlling the classes and styles applied to the content. In the next couple of exercises we first learn to work with classes and then with styles.

3.5.1 Working with HTML classes

Let's start practicing simple things, like classes and styles. Under the ***Labs/Lab3*** folder, create a new component ***Classes*** with a matching styling file.

app/Labs/Lab3/Classes.tsx

```
import './Classes.css';
export default function Classes() {
  return (
    <div>
      <h2>Classes</h2>
```

app/Labs/Lab3/Classes.css

```
.wd-bg-yellow    { background-color: lightyellow; }
.wd-bg-blue      { background-color: lightblue;   }
.wd-bg-red       { background-color: lightcoral;  }
.wd-bg-green     { background-color: lightgreen;  }
.wd-fg-black     { color: black;                  }
```

```
      <div className="wd-bg-yellow wd-fg-black wd-padding-10px">      .wd-padding-10px { padding: 10px;                }
        Yellow background  </div>
      <div className="wd-bg-blue wd-fg-black wd-padding-10px">
        Blue background    </div>
      <div className="wd-bg-red wd-fg-black wd-padding-10px">
        Red background     </div><hr/>
    </div>
  )
};
```

From the ***Lab3*** component, import the new ***Classes*** component and confirm the component renders as shown below.

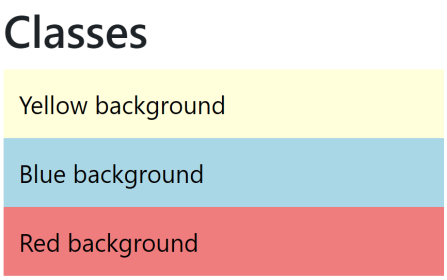


Figure 3.5.1a - Working with HTML classes

The previous example used static classes such as ***wd-bg-yellow***. Instead we could calculate the class we want to apply based on some logic. Here's an example of creating the classes dynamically by concatenating a ***color*** constant. Refresh the screen and confirm components render as expected.

app/Labs/Lab3/Classes.tsx

```
export default function Classes() {
  const color = 'blue';
  return (
    <div id="wd-classes">
      <h2>Classes</h2>
      <div className={`wd-bg-${color} wd-fg-black wd-padding-10px`}>
        Dynamic Blue background
      </div> ...
  );
}
```

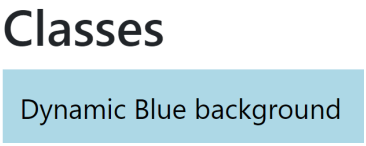


Figure 3.5.1b - Working with HTML classes

Even more interesting is using expressions to conditionally choose between a set of classes. The example below uses either a ***red*** or ***green*** background based on the ***dangerous*** constant. Try with ***dangerous true*** and ***false*** and confirm it renders red or green as expected.

app/Labs/Lab3/Classes.tsx

```
export default function Classes() {
 const color = 'blue';
 const dangerous = true;
 return (
   <div id="wd-classes">
     <h2>Classes</h2>
     <div className={`${dangerous ? 'wd-bg-red' : 'wd-bg-green'}
                                    wd-fg-black wd-padding-10px`}>
       Dangerous background
```

```
      </div> ...
  );
}
```

Figure 3.5.1c - Working with HTML classes

3.5.2 Working with the HTML Style attribute

In React, the ***styles*** attribute accepts a ***JSON*** object where the properties are ***CSS*** properties and the values are CSS values. To practice how this works, implement the ***Styles*** component below and then import it into the ***Lab3*** component. The ***Styles*** component declares constant JSON objects that can be applied to elements using the ***styles*** attribute. Alternatively, the styles attribute accepts a JSON literal object instance which results in a weird syntax of double curly brackets highlighted below. Refresh the browser and confirm the browser renders as expected.

Labs/Lab3/Styles.tsx ***Browser***

```
export default function Styles() {
  const colorBlack = { color: "black" };
  const padding10px = { padding: "10px" };
  const bgBlue = { "backgroundColor": "lightblue",
                   "color": "black",   ...padding10px
  };
  const bgRed = {
    "backgroundColor": "lightcoral",
    ...colorBlack,   ...padding10px
  };
  return(
    <div id="wd-styles">
      <h2>Styles</h2>
      <div style={{"backgroundColor": "lightyellow",
        "color": "black", padding: "10px" }}>
        Yellow background</div>
      <div style={ bgRed }> Red background </div>
      <div style={ bgBlue }>Blue background</div>
    </div>
);};
```

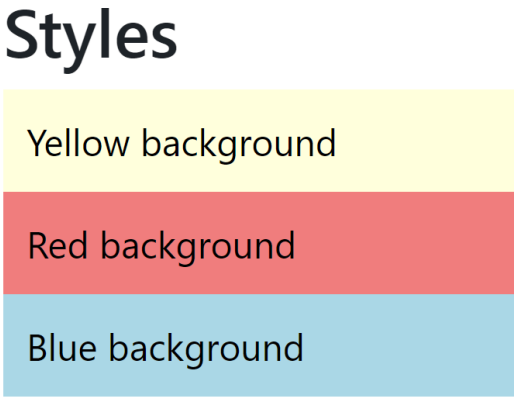


Figure 3.5.2 - Working with the HTML Style attribute

3.6 Parameterizing Components

React components can be parameterized by using the familiar HTML attribute syntax, which passes attribute values to the component's function as an object map parameter. The following ***Add*** component can receive properties ***a*** and ***b*** deconstructed from the attributes of its equivalent HTML attributes syntax. Implement the ***Add*** component below and confirm that passing it ***a=3*** and ***b=4*** results in ***a + b = 7***. Note that the values of ***a*** and ***b*** are descructed from the object parameter in the ***Add*** function parameter list.

app/Labs/Lab3/Add.tsx

```
export default function Add({ a, b }: { a: number; b: number }) {
  return (
    <div id="wd-add">
      <h4>Add</h4>a = {a}
      b = {b}         <br />
      a + b = {a + b} <hr />
    </div>
  );
}
```

app/Labs/Lab3/page.tsx

```
import Add from "./Add";
export default function Lab3() {
  return (
    <div id="wd-lab3" className="container">
      <h3>Lab 3</h3>
      ...
      <Add a={3} b={4} />
    </div>
  );
}
```

3.6.1 Child Components

In the previous section we discussed passing data to a component through attributes. Another way to pass data to a component is in its body, that is, between the opening and closing tag of the element. In HTML it's common to wrap content with specific tags to add certain formatting. For instance the tags ***h1*** and ***p*** shown below format the content in their bodies with specific font sizes and margins. Basically these elements take the content in the body and return a transformed version of the content.

```
<h1>This content is formatted as a heading of size 1</h1>
<p>This content is formatted as a paragraph</p>
```

We can implement ***React*** components to take content in their body and return a new version of that content. The content in the body of a ***React*** component is passed to the component function as parameter called ***children***. For instance, the component below takes a number in its body and returns the square of the number. Import the new component, use to compute the square of 4 and confirm it renders the correct result

app/Labs/Lab3/Square.tsx

```
import React, { ReactNode } from "react";
export default function Square({ children }: { children: ReactNode }) {
  const num = Number(children);
  return <span id="wd-square">{num * num}</span>;
}
```

app/Labs/Lab3/page.tsx

```
import Square from "./Square";
export default function Lab3() {
  return (
    <div id="wd-lab3">
      <h3>JavaScript</h3>
      ...
      <h4>Square of 4</h4>
      <Square>4</Square>
      <hr />
    </div>
  );
}
```

Here's another example that highlights the content in its body making the content red on yellow. The example below receives the ***children*** property as a property and then renders it in the return statement. The ***children*** property is a special property that contains the ***body*** of component when using its opening and closing tags as shown below.

app/Labs/Lab3/Highlight.tsx

```
import { ReactNode } from "react";
export default function Highlight({ children }: { children: ReactNode }) {
  return (
    <span id="wd-highlight" style={{ backgroundColor: "yellow", color: "red" }}>
      {children}
    </span>
  );
}
```

Implement the ***Highlight*** component as shown above, import it in ***Lab3***, and confirm it highlights the content shown.

app/Labs/Lab3/page.tsx

```
...
import Highlight from "./Highlight";

export default function Lab3() {
 return (
   <div id="wd-lab3">
     <h3>Lab 3</h3>
     ...
     <Highlight>
        Lorem ipsum dolor sit amet consectetur adipisicing elit. Suscipitratione eaque illo minus cum, saepe totam
        vel nihil repellat nemo explicabo excepturi consectetur. Modi omnis minus sequi maiores, provident voluptates.
     </Highlight>
   </div>
 );
}
```

3.6.2 Working with Location

The ***useLocation()*** hook returns several properties related to the current URL, in particular the ***pathname*** property containing the URL itself. We can use the ***pathname*** to drive UI logic such as highlighting, showing or hiding content based on the URL. The example below destructs the ***pathname*** from ***useLocation()*** and then checks to see if the URL contains either ***Lab1***, ***Lab2***, or ***Lab3*** to then add the ***active*** class to highlight the correct pill. Confirm that the correct tab highlights when you click each of the pills. jga

app/Labs/TOC.tsx

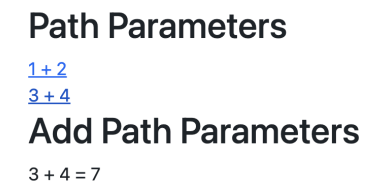
```
import { Nav, NavItem, NavLink } from "react-bootstrap";
import { useLocation } from "react-router";
import Link from "next/link";
export default function TOC() {
  const { pathname } = useLocation();
  return (
    <Nav variant="pills" id="wd-toc">
      <NavItem> <NavLink as={Link} href="/Labs/Lab1" id="wd-a1"
          active={pathname.includes("Lab1")}> Lab 1 </NavLink> </NavItem>
      <NavItem> <NavLink as={Link} href="/Labs/Lab2" id="wd-a2"
          active={pathname.includes("Lab2")}> Lab 2 </NavLink> </NavItem>
      <NavItem> <NavLink as={Link} href="/Labs/Lab3" id="wd-a3"
          active={pathname.includes("Lab3")}> Lab 3 </NavLink> </NavItem>
      <NavItem> <NavLink as={Link} href="/Kambaz" id="wd-a3"> Kambaz </NavLink> </NavItem>
      <NavItem> <NavLink href="https://github.com/jannunzi" target="_blank"> My GitHub </NavLink> </NavItem>
    </Nav>
  );
}
```

3.6.3 Encoding Path Parameters

The URL can also be used to encode parameters when navigating between screens. Components can parse parameters from the path using the ***useParams*** React hook. The ***Add*** component below is parsing parameters ***a*** and ***b*** from the path and calculating the arithmetic addition of the parameters.

app/Labs/Lab3/AddPathParameters.tsx

```
import { useParams } from "react-router-dom";
export default function AddPathParameters() {
  const { a, b } = useParams();
  return (
    <div id="wd-add"> <h4>Add Path Parameters</h4>
      {a} + {b} = {parseInt(a as string) + parseInt(b as string)}
    </div>
  );
}
```



3.6.3 Encoding Path Parameters

Path parameter names such as ***a*** and ***b***, are declared in the ***path*** attribute of the ***Route*** component for the target screen. For instance the ***Route*** component below uses the ***colon*** character to declare parameters ***a*** and ***b***. The first link encodes values 1 and 2 for parameters ***a*** and ***b***, wheres as the second link encodes values 3 and 4 for parameters ***a*** and ***b***. Import ***PathParameters*** component in your ***Lab3*** component and confirm clicking the first link, the URL matches the ***Route*** and renders the ***Add*** component with parameters ***a=1*** and ***b=2*** and so it renders ***1 + 2 = 3***. Confirm clicking the second link sets ***a=3*** and ***b=4*** and so it renders ***3 + 4 = 7***. Add '***/****' to the ***Lab3*** route in the ***Labs*** component as shown below on the right.

app/Labs/Lab3/PathParameters.tsx

```
import { Routes, Route, Link } from "react-router-dom";
import AddPathParameters from "./AddPathParameters";
export default function PathParameters() {
  return (
   <div id="wd-path-parameters">
    <h2>Path Parameters</h2>
    <Link href="/Labs/Lab3/add/1/2">1 + 2</Link> <br />
    <Link href="/Labs/Lab3/add/3/4">3 + 4</Link>
    <Routes>
      <Route path="add/:a/:b" element={<AddPathParameters />} />
    </Routes>
   </div>
  );
}
```

3.7 Debugging

The ***Web Dev Tools*** provide several tools to analyze various types of source code common in the Web development process. So far we've looked at the ***Elements*** and ***Styles*** tab where we can analyze the ***DOM*** data structure generated by the browser when it parsed the ***HTML*** or content dynamically generated by ***JavaScript***. Now that we have been implementing functions and algorithms, we should become familiar with the ***Console*** and ***Source*** tabs.

3.7.1 Writing to the Console from JavaScript

A useful feature of modern browsers is providing a development environment where developers can analyze the performance of their scripts. One way to analyze scripts are behaving correctly is to write output to the ***console*** from within scripts. To practice writing to the ***console***, bring up the console on the browser by right clicking on the page and selecting ***Inspect***. The page will split in half displaying useful developer tools similar as shown below.

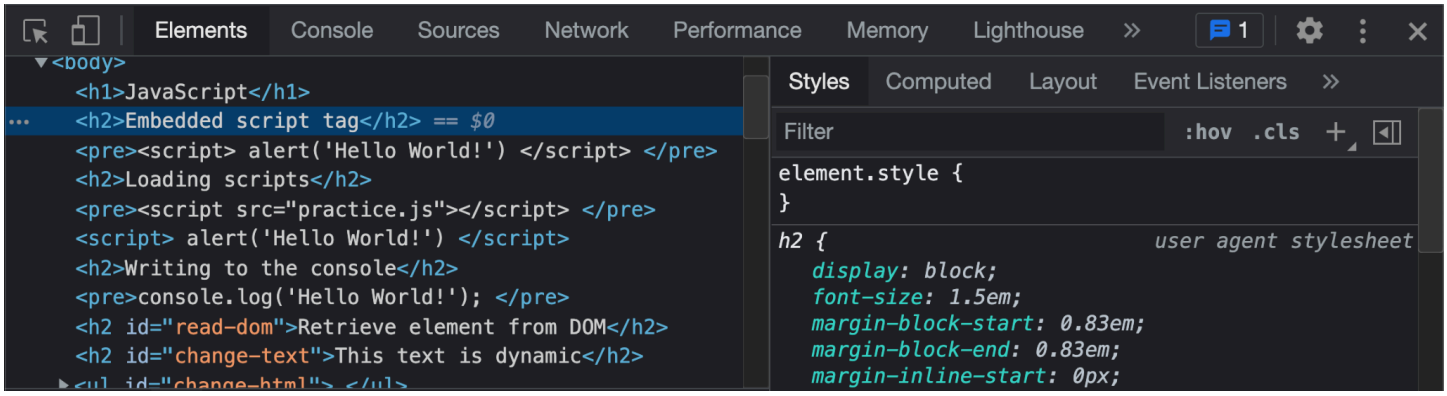


Figure 3.7.1a - Writing to the Console from JavaScript

Click on the ***Console*** tab where we will be logging to throughout this chapter.

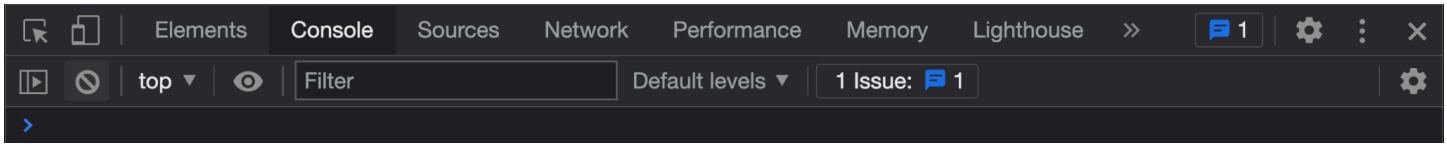


Figure 3.7.1b - Writing to the Console from JavaScript

Add a ***console.log()*** statement to the ***Lab3*** component, reload the screen and confirm that "***Hello World!***" is displayed in the console.

app/Labs/Lab3/page.tsx	*Console*

`export default function Lab3() {`
`  console.log('Hello World!');`
`  return(`
`  <div>`
`  <h3>JavaScript</h3>`
`  ...`
`  </div>`
`  );`
`}`
	`Hello World!`

We can also display ***JSON*** data to confirm it is what we expect. Add a ***console.log()*** statement to print the ***house*** JSON object in the ***House*** component as shown below. Confirm the ***house*** object prints to the console.

app/Labs/Lab3/House.tsx	*Console*

`export default function House() {`
`  const house = {`
`  ...`
`  };`
`  console.log(house);`
`  return (`
`  <div id="wd-house">`
`  ...`
`  </div>`
`  );`	

```
}
```

```
▼ {bedrooms: 4, bathrooms: 2.5, squareFeet: 2000, address: {…}, owners: Array(2)} i
  ▼ address:
      city: "Roma"
      country: "Italy"
      state: "RM"
      street: "Via Roma"
      zip: "00100"
    ▶ [[Prototype]]: Object
    bathrooms: 2.5
    bedrooms: 4
  ▶ owners: (2) ['Alice', 'Bob']
    squareFeet: 2000
```

Figure 3.7.1c - Writing to the Console from JavaScript

3.7.2 Breakpoints

Often analyzing code requires a more indepth effort of stepping through the code as its running. The ***Sources*** tab in the ***Web Dev Tools*** gives access to all the ***JavaScript*** code loaded by the browser so that developers can add breakpoints and step through the code. To practice debugging, open the ***Web Dev Tools*** and select the ***Sources*** tab as shown below. On the left sidebar, navigate through the folders looking for the ***ForLoops*** component we wrote earlier, and click on it to show its source code on the middle pane as shown below. Explore setting breakpoints on the line numbers and reload the page by pressing ***Ctrl+R*** on Windows or ⌘***+R*** on macOS. Confirm that the execution pauses on the breakpoint. In the example screenshot below we have added a breakpoint on line 6 of the ***ForLoops*** component and reloaded the page. Execution paused and line 6 is highlighted giving us developers an opportunity to analyze the current state of the application. On the right sidebar we can see the values of variables within the current scope such as ***string1***, ***stringArray1***, and ***stringArray2***.

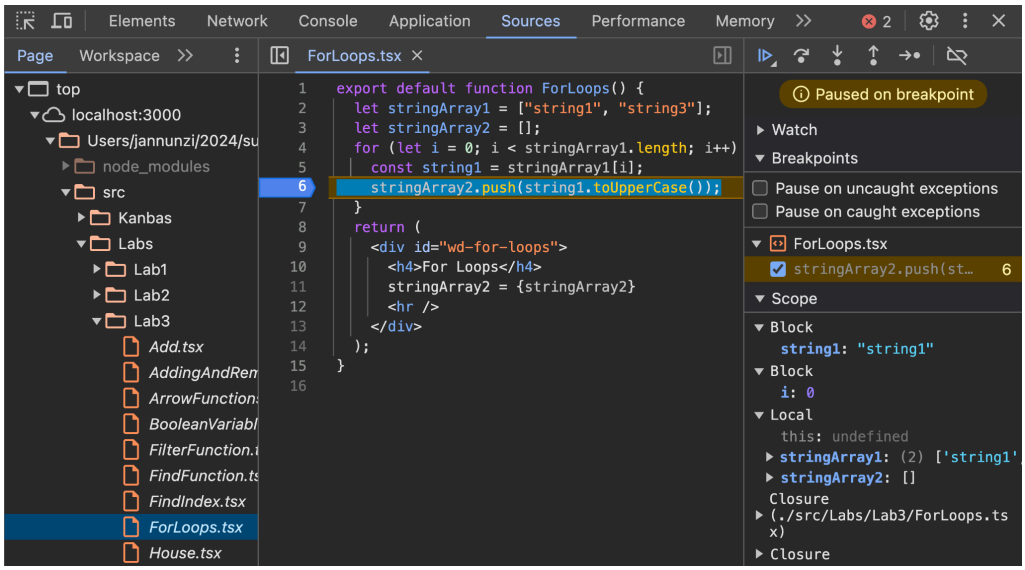


Figure 3.7.2a - Breakpoints

The right sidebar provides buttons at the top to control the execution shown here on the right. The first button is for pausing or resuming execution. The second button is for stepping over the next function. The second button is for stepping into a function. The third button is for stepping out of the current function. The fifth button is for stepping one line at a time. The last button is for enabling and disabling breakpoints.

Figure 3.7.2b - Breakpoints

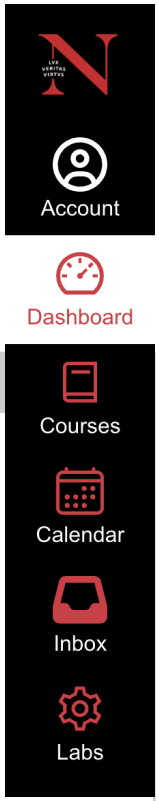
3.8 Implementing a Data Driven Kambaz Application

Previous chapters implemented the ***Kambaz*** application using ***React*** components and screens that aggregated their output into a single application. The ***Kambaz*** application is currently displaying static content that can not be modified and does not depend on the changing context. For instance, the ***Dashboard*** screen should be different based on who logs in, and ***Courses*** screen should display different ***Modules*** and ***Assignments***, depending on which course a user selects in the ***Dashboard*** screen. This chapter revisits the implementation by making it ***data driven***, that is the content will be dynamic based on ***JSON*** data files that determine the courses displayed in the ***Dashboard***, and the ***Modules*** and ***Assignments*** displayed as users select a particular course. Before beginning, confirm the ***Kambaz*** landing screen implementation is similar to the code shown below. In particular, make sure the ***Account*** and ***Courses*** paths are as shown.

app/(Kambaz)/page.tsx

```
export default function Kambaz() {
  return (
    <div id="wd-kambaz">
      <KambazNavigation />
      <div className="wd-main-content-offset p-3">
        <Routes>
          <Route path="/" element={<Navigate href="Account" />} />
          <Route path="/Account/*" element={<Account />} />
          <Route path="/Dashboard" element={<Dashboard />} />
          <Route path="/Courses/:cid/*" element={<Courses />} />
          <Route path="/Calendar" element={<h1>Calendar</h1>} />
          <Route path="/Inbox" element={<h1>Inbox</h1>} />
        </Routes>
      </div>
    </div>
);}
```

3.8.1 Data Driven Kambaz Navigation



The current implementation of the ***KambazNavigation*** component consists of a hardcoded list group of links to navigate to the ***Dashboard*** and other places in ***Kambaz***. Instead of hardcoding the links, create a data structure that configures the ***label***s, ***path***s, and ***icon***s as an array and then map over the data structure creating the links dynamically. Here's an example of how to implement the ***Kambaz Navigation*** component. Confirm the sidebar renders as shown here on the right and the navigation still works. Note that the ***Courses*** link has been intentionally configured to navigate to the ***Dashboard***, since it only makes sense to navigate to the ***Courses*** screen from the ***Dashboard***.

app/(Kambaz)/Navigation.tsx

```
import { AiOutlineDashboard } from "react-icons/ai";
import { IoCalendarOutline } from "react-icons/io5";
import { LiaBookSolid, LiaCogSolid } from "react-icons/lia";
import { FaInbox, FaRegCircleUser } from "react-icons/fa6";
import { Link, useLocation } from "react-router-dom";
export default function KambazNavigation() {
  const { pathname } = useLocation();
  const links = [
    { label: "Dashboard", path: "/Kambaz/Dashboard", icon: AiOutlineDashboard },
    { label: "Courses",   path: "/Kambaz/Dashboard", icon: LiaBookSolid },
    { label: "Calendar",  path: "/Kambaz/Calendar",  icon: IoCalendarOutline },
    { label: "Inbox",     path: "/Kambaz/Inbox",     icon: FaInbox },
    { label: "Labs",      path: "/Labs",             icon: LiaCogSolid },
  ];
  return (
    <ListGroup id="wd-kambaz-navigation" style={{width: 120}}
         className="rounded-0 position-fixed bottom-0 top-0 d-none d-md-block bg-black z-2">
      <ListGroupItem id="wd-neu-link" target="_blank" href="https://www.northeastern.edu/"
        action className="bg-black border-0 text-center">
        <img src="/images/NEU.png" width="75px" /></ListGroupItem>
```

```
      <ListGroupItem as={Link} href="/Kambaz/Account" className={`text-center border-0 bg-black
            ${pathname.includes("Account") ? "bg-white text-danger" : "bg-black text-white"}`}>
        <FaRegCircleUser className={`fs-1 ${pathname.includes("Account") ? "text-danger" : "text-white"}`} />
        <br />
        Account
      </ListGroupItem>
      {links.map((link) => (
        <ListGroupItem key={link.path} as={Link} to={link.path} className={`bg-black text-center border-0
              ${pathname.includes(link.label) ? "text-danger bg-white" : "text-white bg-black"}`}>
          {link.icon({ className: "fs-1 text-danger"})}
          <br />
          {link.label}
        </ListGroupItem>
      ))}
    </ListGroup>
);}
```

3.8.2 Implementing a Kambaz "Database"

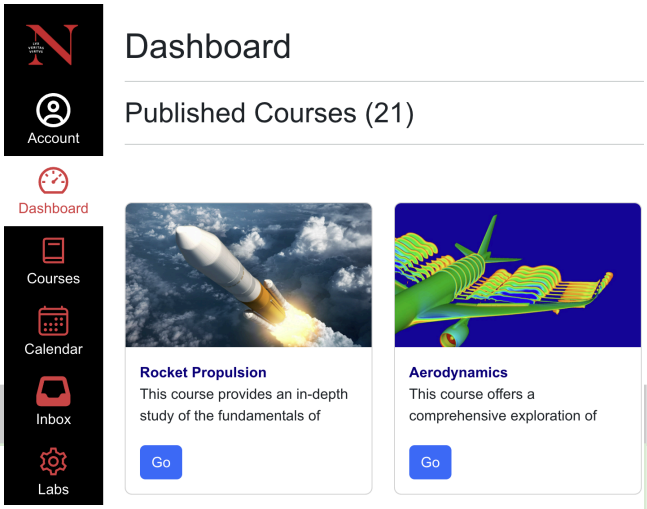
Implement a data file that will contain all the data needed in the ***Kambaz*** application. Start by adding courses in a ***courses.json*** file under a new ***Database*** folder. Download a sample data file containing the courses from my gist on GitHub. Save the file to ***app/(Kambaz)/Database/courses.json***. Create a database file that contains all the data needed for the ***Kambaz*** application. For now we just have the courses, but later sections will add users, assignments, modules, etc.

app/(Kambaz)/Database/index.ts

```
import courses from "./courses.json";
export {  courses  };
```

3.8.3 Data Driven Dashboard Screen

Based on your implementation of the ***Dashboard*** in previous chapters, refactor the component so that it dynamically renders the ***courses*** in the ***Database***. The following code is an example of how you can implement the ***Dashboard*** component dynamically mapping through an array of ***courses***. Confirm that the ***Dashboard*** component renders the courses as a grid and that clicking on a course navigates to the ***Home*** screen, but now the ***URL*** encodes the ***ID*** of the course.

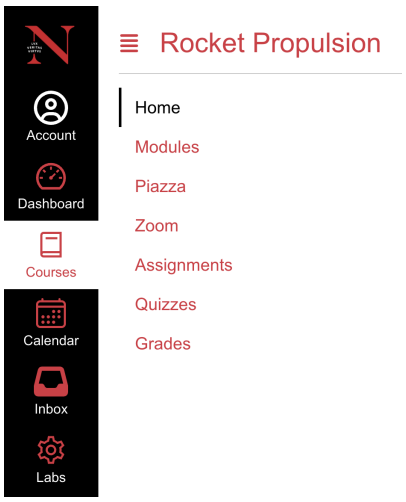


app/(Kambaz)/Dashboard/page.tsx

```
import Link from "next/link";
import * as db from "./Database";
export default function Dashboard() {
  const courses = db.courses;
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h2 id="wd-dashboard-published">Published Courses ({courses.length})</h2> <hr />
      <div id="wd-dashboard-courses">
        <Row xs={1} md={5} className="g-4">
          {courses.map((course) => (
            <Col className="wd-dashboard-course" style={{ width: "300px" }}>
              <Card>
                <Link href={`/Kambaz/Courses/${course._id}/Home`}
                      className="wd-dashboard-course-link text-decoration-none text-dark" >
                  <CardImg src="/images/reactjs.jpg" variant="top" width="100%" height={160} />
                  <CardBody className="card-body">
                    <CardTitle className="wd-dashboard-course-title text-nowrap overflow-hidden">
                      {course.name} </CardTitle>
```

```
                <CardText className="wd-dashboard-course-description overflow-hidden" style={{ height: "100px" }}>
                  {course.description} </CardText>
                <Button variant="primary"> Go </Button>
              </CardBody>
            </Link>
          </Card>
        </Col>
      ))}
    </Row>
  </div>
</div>);}
```

The ***Link*** component is used to create the hyperlinks and encode the course's ID as part of the path. This can then be used in other components to parse the ID from the path and display content specific to the selected course. Confirm that clicking on different courses encodes the corresponding course ID in the path.



3.8.4 Data Driven Courses Screen

Now that clicking on a course on the ***Dashboard*** encodes the course's ***ID*** on the path, the ***Courses*** screen can be refactored to parse the course's ***ID*** from the path and then render the corresponding course's name. The ***Route*** for the ***Courses*** component encodes the course's ***id*** in the ***path*** attribute and is available to the ***Courses*** component through the ***useParams()*** hook. With the ***cid*** parameter, the ***course*** can be looked up from the ***Database*** for the ***course*** with the same ***_id***. Refactor the ***Courses*** component to render the ***name*** of the course you click on in the ***Dashboard*** as shown here on the right.

app/(Kambaz)/Courses/[cid]/page.tsx

```
import { courses } from "../Database";
import { FaAlignJustify } from "react-icons/fa6";
import { Navigate, Route, Routes, useParams } from "react-router";
export default function Courses() {
  const { cid } = useParams();
  const course = courses.find((course) => course._id === cid);
  return (
    <div id="wd-courses">
      <h2 className="text-danger">
        <FaAlignJustify className="me-4 fs-4 mb-1" />
        {course && course.name}
      </h2>
      ...
    </div>
  );
}
```

3.8.5 Data Driven Course Navigation (On Your Own)

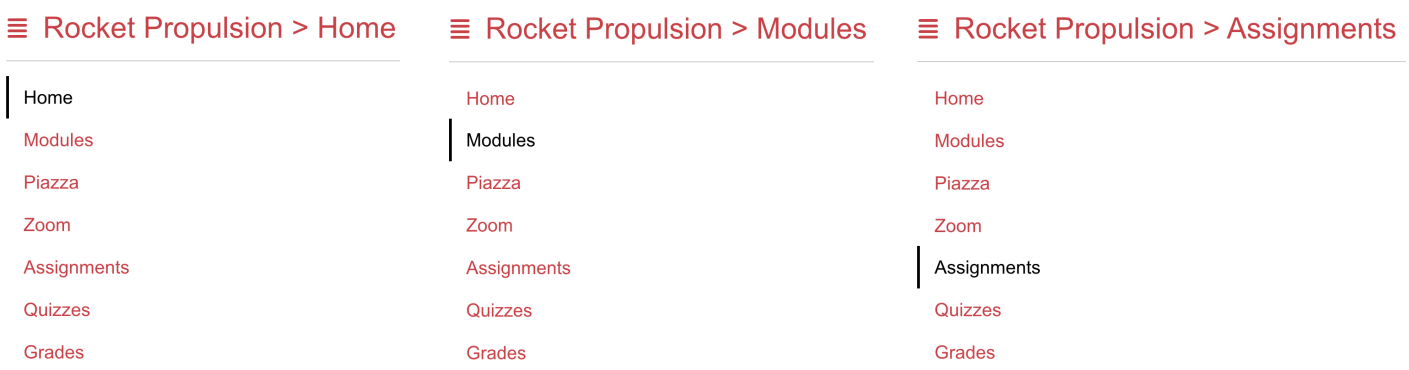
Based on the earlier ***Data Driven Kambaz Navigation*** example, refactor the ***Course Navigation*** sidebar so that it is not a list of hardcoded links and instead uses the following array.

```
const links = ["Home", "Modules", "Piazza", "Zoom", "Assignments", "Quizzes", "Grades", "People"];
```

Use ***useParams()*** to retrieve the current course's ***ID*** and use ***useLocation()*** to retrieve the current ***pathname***. The links should highlight when you click on them and the URL should encode the course's ***ID*** in the path. Navigation to the ***Home***, ***Modules***, ***Assignments***, and ***Grades*** screen should still work as before.

3.8.6 Implementing the Breadcrumb Component

A ***breadcrumb*** is graphical representation that helps users know where they are within a set of screens. For instance the course name at the top of the ***Courses*** screen lets users know they are in a particular course. As they click through the links in the ***Courses Navigation*** sidebar, we can append the name of the section to the name of the course to further indicate what section are we in. The screenshots below illustrate navigating to the ***Home***, ***Modules***, and ***Assignments*** screens. In the ***Courses*** component, implement the breadcrumbs as shown below.



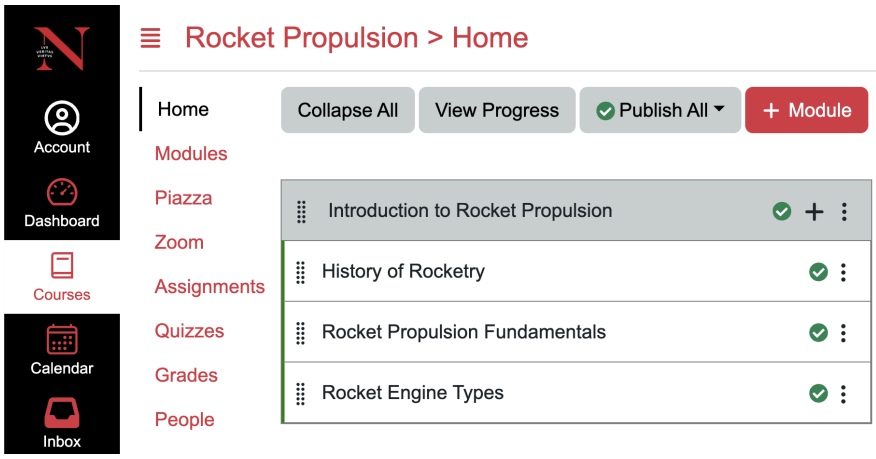
Below is a suggestion on how you could implement the breadcrumbs.

app/(Kambaz)/Courses/[cid]/page.tsx

```
import { Navigate, Route, Routes, useParams, useLocation } from "react-router";
export default function Courses() {
  const { cid } = useParams();
  const course = courses.find((course) => course._id === cid);
  const { pathname } = useLocation();
  return (
    <div id="wd-courses">
      <h2 className="text-danger">
        <FaAlignJustify className="me-3 fs-4 mb-1" />
        {course && course.name} &gt; {pathname.split("/")[4]}
      </h2> </div> );}
```

3.8.7 Data Driven Modules Screen

Currently the ***Modules*** and ***Home*** screen render the same course modules regardless which course you click on in the ***Dashboard***. In this section we're going to modify the ***Modules*** screen so that we display the corresponding modules for the selected course. Each course has a different set of modules and each module has its set of lessons. Use the data provided and save it to ***app/(Kambaz)/Database/modules.json*** file containing at least 3 modules for each course. Include the modules in the ***Database*** as shown below.



app/(Kambaz)/Database/index.ts

```
import courses from "./courses.json";
import modules from "./modules.json";
export { courses, modules };
```

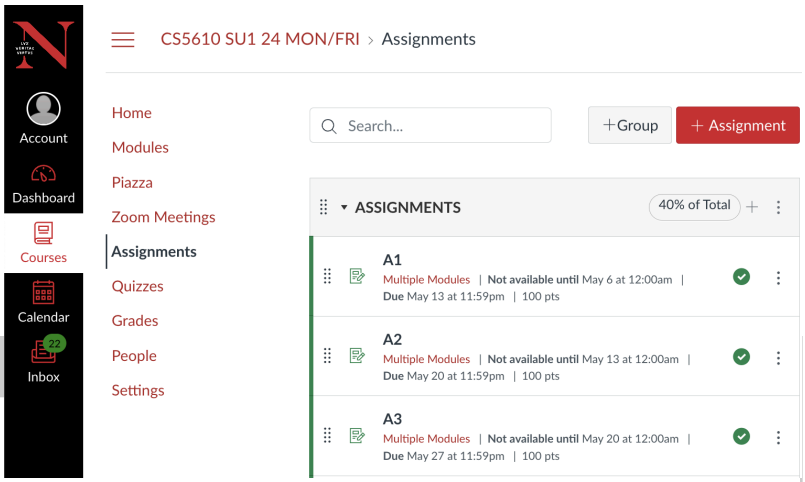
Below is an example of how you can use the ***modules*** in the ***Database*** to render a list of modules. Use ***useParams()*** to retrieve the course ID from the URL and then retrieve the corresponding modules for the course. Confirm that navigating to different courses, the corresponding modules and lessons are rendered as shown above.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { useParams } from "react-router";
import * as db from "../../Database";
export default function Modules() {
  const { cid } = useParams();
  const modules = db.modules;
  return (
      ...
      <ListGroup id="wd-modules" className="rounded-0">
        {modules
          .filter((module: any) => module.course === cid)
          .map((module: any) => (
          <ListGroupItem className="wd-module p-0 mb-5 fs-5 border-gray">
            <div className="wd-title p-3 ps-2 bg-secondary">
              <BsGripVertical className="me-2 fs-3" /> {module.name} <ModuleControlButtons />
            </div>
            {module.lessons && (
              <ListGroup className="wd-lessons rounded-0">
                {module.lessons.map((lesson: any) => (
                  <ListGroupItem className="wd-lesson p-3 ps-1">
                    <BsGripVertical className="me-2 fs-3" /> {lesson.name} <LessonControlButtons />
                  </ListGroupItem>
                ))}</ListGroup>)}</ListGroupItem>))}</ListGroup>...);}
```

3.8.8 Data Driven Assignments Screen (On Your Own)

Based on your implementation of the ***Assignments*** screen in previous chapters, refactor the component so that it renders the assignments for the currently selected course. Use the data provided as an example to create an ***assignments.json*** file which contains various assignments for the courses in the ***courses.json*** file provided. Include the ***assignments*** in the ***Database*** as shown below.



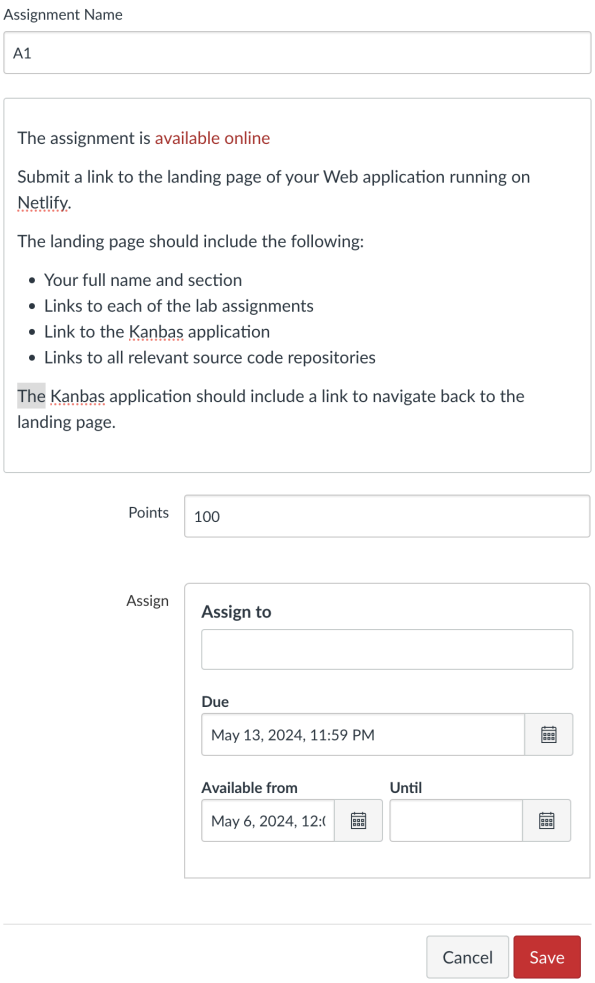
app/(Kambaz)/Database/page.tsx

```
import courses from "./courses.json";
import modules from "./modules.json";
import assignments from "./assignments.json";
export default {
  courses, modules, assignments, };
```

Use the ***useParams()*** hook to retrieve the ***course***'s ID and then find all the assignments for that course from the database's ***assignments*** array. Render the assignments as links that encode the ***course***'s ID and the ***assignment***'s ID in the URL's path. The ***assignment***'s ***ID*** will be used by a router to render the corresponding assignment in the ***AssignmentEditor*** screen. Modify ***assignments.json*** as needed.

3.8.8.1 Data Driven Assignment Editor Screen (On Your Own)

Based on the ***Assignment Editor*** screen implemented in earlier chapters, refactor the screen so that it renders the information for the assignment selected in the ***Assignments*** screen. Use ***useParams()*** to parse the ***assignment***'s ***ID***s from the URL and retrieve the assignment from the database's ***assignments*** object. Display the ***title*** of the selected assignment as well as the ***description***, ***points***, ***due date***, and ***available date***. Use ***useParams()*** to parse the ***course***'s ***ID*** from the URL and implement the ***Cancel*** and ***Save*** buttons as ***Link*** so that clicking either one navigates back to the ***Assignments*** screen for the current course. The ***Assignment Editor*** screen should look as shown here on the right. Other fields such as ***Submission Type*** and ***Assignment Group*** are not shown for brevity, but they should still appear as implemented in earlier chapters.



3.8.9 Data Driven People Screen

Previous chapters implemented and styled the ***People*** screen that displays a list of users as a table. The current implementation is static, always showing the same hard coded list of students. In this section, refactor the ***People*** screen to display students, teaching assistants, and faculty associated with the selected course. Use the data provided as an example list of users and save the file as ***users.json*** file in the ***Database*** folder. Import ***users.json*** into ***Database/index.ts*** so that it can be imported from the ***People/Table/page.tsx***. Also create an ***enrollments.json*** file that contains enrollment data establishing which students are enrolled in which course as shown below.

app/(Kambaz)/Database/enrollments.json

```
[
  { "_id": "1", "user": "123", "course": "RS101" },  { "_id": "2", "user": "234", "course": "RS101" },
  { "_id": "3", "user": "345", "course": "RS101" },  { "_id": "4", "user": "456", "course": "RS101" },
  { "_id": "5", "user": "567", "course": "RS101" },  { "_id": "6", "user": "234", "course": "RS102" },
  { "_id": "7", "user": "789", "course": "RS102" },  { "_id": "8", "user": "890", "course": "RS102" },
  { "_id": "9", "user": "123", "course": "RS102" } ]
```

In the ***PeopleTable*** screen, parse the current course ID from the path using ***useParams*** so that the users associated with the course can be filtered based on their enrollment.

app/(Kambaz)/Courses/[cid]/People/Table/page.tsx

```
import React from "react";
import { useParams } from "react-router-dom";
import * as db from "../../Database";
export default function PeopleTable() {
  const { cid } = useParams();
  const { users, enrollments } = db;
  return (
    <div id="wd-people-table"> ...  </div> );}
```

Replace the hardcoded rows displaying the users with an expression that displays only the users related to the currently selected course. The ***users.filter*** expression below filters the array of users in the database. The filter function searches the users in the ***enrollments*** array based on the user's ID and the user's enrollment in the selected course. Navigate to various courses and confirm that only the users enrolled in the courses are shown in the ***People*** screen.

app/(Kambaz)/Courses/[cid]/People/Table/page.tsx

```
<tbody>
  {users
    .filter((usr) =>
      enrollments.some((enrollment) => enrollment.user === usr._id && enrollment.course === cid)
    )
    .map((user: any) => (
      <tr key={user._id}>
        <td className="wd-full-name text-nowrap">
          <FaUserCircle className="me-2 fs-1 text-secondary" />
          <span className="wd-first-name">{user.firstName}</span>
          <span className="wd-last-name">{user.lastName}</span>
        </td>
        <td className="wd-login-id">{user.loginId}</td>
        <td className="wd-section">{user.section}</td>
        <td className="wd-role">{user.role}</td>
        <td className="wd-last-activity">{user.lastActivity}</td>
        <td className="wd-total-activity">{user.totalActivity}</td>
      </tr>
    ))}
</tbody>
```

3.9 Deliverables

1. In the same React application created in earlier chapters, ***kambaz-next-js***, complete all the exercises described in this document
2. In a branch called ***a3***, add, commit and push the source code of the React application ***kambaz-next-js*** to the same remote source repository in ***GitHub.com*** created in an earlier chapter. Here's an example of how to add, commit and push your code

   ```
   $ git checkout -b a3
   $ git add .
   $ git commit -am "a3 JavaScript"
   $ git push
   ```

3. Deploy the ***a3*** branch to the same ***Vercel*** project created in an earlier chapter. Configure Vercel to deploy all branches to separate URLs. From your Vercel's dashboard go to ***Site settings > Build & deploy > Branches > Branch deploys*** and select ***All***. Now each time you commit to a branch, the application will be available at a URL that contains the name of the branch
4. Make sure ***Labs/page.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kambaz. Add a link to your repository in GitHub. The link should have an ***ID*** attribute with a value of ***wd-github***.
5. In ***Labs/page.tsx***, add your full name: first name first and last name second. Use the same name as in Canvas.
6. Deploy the branch for this chapter to your ***kambaz-next-js*** React application to ***Vercel*** as described.
7. As a deliverable in ***Canvas***, submit the URL to the ***a3*** branch deployment of your React application running on Vercel.

Chapter 4 - Maintaining State in React Applications

In an application, ***state*** is the collection of data values stored in the various constants, variables and data structures in an application. ***Application state*** is data that is relevant across the entire application or a significant subset of related components. ***Component state*** is data that is only relevant to a specific component or a small set of related components. If information is relevant across several or most components, then it should live in the ***application state***. If information is relevant only in one component, or a small set of related components, then it should live in the ***component state***. For instance, the information about the currently logged in user could be stored in a profile, e.g., ***username***, ***first name***, ***last name***, ***role***, ***logged in***, etc., and it might be relevant across the entire application. On the other hand, filling out shipping information might only be relevant while checking out, but not relevant anywhere else, so shipping information might best be stored in the ***ShippingScreen*** or ***Checkout*** components in the component's state. This chapter introduces how to maintain state at the application level as well as at the component level. The ***Redux*** state management library is described to handle application state, and ***React*** state hooks are used to manage component state.

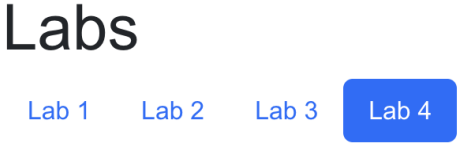
4.1 Learning Objectives

By the end of this chapter, you will be able to:

- Understand **state management** in React applications
- Learn how to **handle user input with controlled components**
- Use the **useState** hook to manage component-level state
- Explore **two-way data binding** for form elements
- Implement **React forms** with various input types (text fields, checkboxes, radio buttons, dropdowns)
- Manage **application-wide state** using Redux
- Understand the **Redux store, actions, and reducers**
- Use **Redux Toolkit** to simplify state management
- Connect React components to Redux using **useSelector** and **useDispatch**
- Handle **side effects** with the **useEffect** hook
- Add state management to the **Kambaz user interface**
- Implement **dynamic content rendering** based on application state

4.2 Managing State and User Input with Forms

This section presents ***React*** examples to program the browser, interact with the user, and generate dynamic HTML. Use the same project you worked on the last chapter. After you work through the examples you will apply the skills while creating a ***Kambaz*** on your own. Using ***IntelliJ***, ***VS Code***, or your favorite IDE, open the project you created in previous chapters. ***Include all the work in the Labs section as part of your final deliverable***. Do all your work in a new branch called ***a4*** and deploy it to ***Vercel*** to a branch deployment of the same name. To get started, create a ***Lab4*** screen that will host all the exercises in this chapter. Import the component into the ***Labs*** screen created in an earlier chapter and add a route to navigate to ***Lab4***. Style the links using Bootstrap pills as shown here on the right.



4.2.1 Handling User Events

Users interact with the Web application user interface by clicking their mouse, typing at their keyboards, and, on mobile devices, tapping, swiping, and pinching at the screen. As they interact with the graphical user interface, they generate a stream of events that need to be handled by interpreting the user intent, modifying the Web application state, and

rerendering the user interface to reflect the new state and give feedback to the user that their actions are having the intended effect. The next few sections consider the various types of events users generate and how they can be handled.

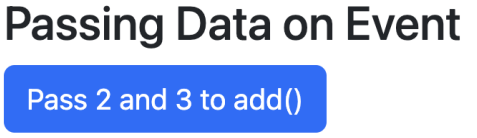
4.2.1.1 Handling Click Events

The ***onClick*** attribute can be used to declare a function that handles clicks. The example below calls function ***hello*** when you click the ***Click Hello*** button. Add the component to ***Lab4*** and confirm it behaves as expected.

app/Labs/Lab4/ClickEvent.tsx

```
const hello = () => {                                    // declare a function to handle the event
  alert("Hello World!");
};
const lifeIs = (good: string) => {
  alert(`Life is ${good}`);
};
export default function ClickEvent() {
  return (
    <div id="wd-click-event">
      <h2>Click Event</h2>
      <button onClick={hello} id="wd-hello-world-click">
        Hello World!</button>                            // configure the function call
      <button onClick={() => lifeIs("Good!")}
              id="wd-life-is-good-click">                // wrap in function if you need to pass parameters
        Life is Good!</button>
      <button onClick={() => {                           // wrap in {} if you need more than one line of code
                hello();                                 // calling hello()
                lifeIs("Great!");                        // calling lifeIs()
              }} id="wd-life-is-great-click">
        Life is Great!
      </button>
      <hr/>
    </div>
);}
```

4.2.1.2 Passing Data when Handling Events



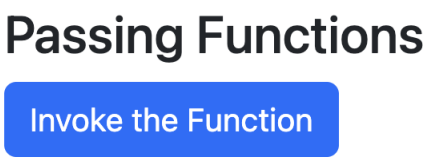
When handing an event, sometimes we need to pass parameters to the function handling the event. Make sure to wrap the function call in a ***closure*** as shown below. The example below calls ***add(2, 3)*** when the button is clicked, passing arguments ***a*** and ***b*** as ***2*** and ***3***. If you do not wrap the function call inside a closure, you risk creating an infinite loop. Add the component to ***Lab4*** and confirm it works as expected.

app/Labs/Lab4/PassingDataOnEvent.tsx

```
const add = (a: number, b: number) => {
  alert(`${a} + ${b} = ${a + b}`);                       // function expects a and b
};
export default function PassingDataOnEvent() {
  return (
    <div id="wd-passing-data-on-event">
      <h2>Passing Data on Event</h2>
      <button onClick={() => add(2, 3)}                  // use this syntax
              // onClick={add(2, 3)}                     // and not this syntax.
              className="btn btn-primary"                // Otherwise you risk
              id="wd-pass-data-click">                   // creating an infinite loop
        Pass 2 and 3 to add()
      </button>
      <hr/>
    </div>
);}
```

4.2.1.3 Passing Functions as Parameters

In JavaScript, functions can be treated as any other constant or variable, including passing them as parameters to other functions. The example below passes function ***sayHello*** to component ***PassingFunctions***. When the button is clicked, ***sayHello*** is invoked.



app/Labs/Lab4/PassingFunctions.tsx

```
export default function PassingFunctions(
 { theFunction }: { theFunction: () => void }) {                    // function passed in as a parameter
  return (
    <div>
      <h2>Passing Functions</h2>
      <button onClick={theFunction} className="btn btn-primary">     // invoking function with no arguments
        Invoke the Function
      </button>
      <hr/>
    </div>
);}
```

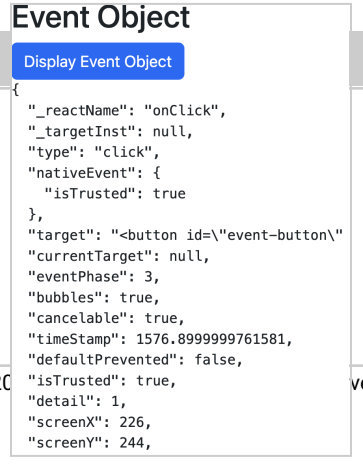
Include the component in ***Lab4***, declare a ***sayHello*** callback function, pass it to the ***PassingFunctions*** component, and confirm it works as expected.

app/Labs/Lab4/page.tsx

```
import PassingFunctions from "./PassingFunctions";              // import the component
export default function Lab4() {
  function sayHello() {                                         // implement callback function
    alert("Hello");
  }
  return (
    <div id="wd-passing-functions">
      <h2>Lab 4</h2>
      ...
      <PassingFunctions theFunction={sayHello} />               // pass callback function as a parameter
    </div>
);}
```

4.2.1.4 The Event Object

When an event occurs, JavaScript collects several pieces of information about when the event occurred, formats it in an ***event object*** and passes the object to the event handler function. The ***event object*** contains information such as a timestamp of when the event occurred, where the mouse was on the screen, and the DOM element responsible for generating the event. The example below declares event handler function ***handleClick*** that accepts an ***event object e*** parameter, removes the ***view*** property and replaces the ***target*** property to avoid circular references, and then stores the event object in variable ***event***. The component then renders the JSON representation of the event on the screen. Include the component in ***Lab4***, click the button and confirm the event object is rendered on the screen.



app/Labs/Lab4/EventObject.tsx

```
import { useState } from "react";                 // import useState
export default function EventObject() {           // (more on this later)
  const [event, setEvent] = useState(null);       // initialize event
  const handleClick = (e: any) => {               // on click receive event
    e.target = e.target.outerHTML;                // replace target with HTML
    delete e.view;                                // to avoid circular reference
    setEvent(e);                                  // set event object
  };                                              // so it can be displayed
  return (
    <div>
```

```
      <h2>Event Object</h2>
      <button onClick={(e) => handleClick(e)}        // button that triggers event
        className="btn btn-primary"                  // when clicked passes event
        id="wd-display-event-obj-click">             // to handler to update
        Display Event Object                         // variable
      </button>
      <pre>{JSON.stringify(event, null, 2)}</pre>    // convert event object into
      <hr/>                                          // string to display
    </div>
);}
```

4.2.2 Managing Component State

Web applications implemented with React can be considered as a set of functions that transform a set of data structures into an equivalent user interface. The collection of data structures and values are often referred to as an application ***state***. So far we have explored React applications that transform a static data set, or state, into a static user interface. We will now consider how the state can change over time as users interact with the user interface and how these state changes can be represented in a user interface.

Users interact with an application by clicking, dragging, and typing with their mouse and keyboard, filling out forms, clicking buttons, and scrolling through data. As users interact with an application they create a stream of events that can be handled by a set of event handling functions, often referred to as ***controllers***. Controllers handle user events and convert them into changes in the application's state. Applications render application state changes into corresponding changes in the user interface to give users feedback of their interactions. In Web applications, user interface changes consist of changes to the DOM.

4.2.2.1 Use State Hook

Updating the DOM with JavaScript is slow and can degrade the performance of Web applications. React optimizes the process by creating a ***virtual DOM***, a more compact and efficient version of the real DOM. When React renders something on the screen, it first updates the virtual DOM, and then converts these changes into updates to the actual DOM. To avoid unnecessary and slow updates to the DOM, React only updates the real DOM if there have been changes to the virtual DOM. We can participate in this process of state change and DOM updates by using the ***useState*** hook. The ***useState*** hook is used to declare ***state*** variables that we want to affect the DOM rendering. The syntax of the ***useState*** hook is shown below.

```
const [stateVariable, setStateVariable] = useState(initialStateValue);
```

The ***useState*** hook takes as argument the initial value of a ***state variable*** and returns an array whose first item consists of the initialized state variable, and the second item is a ***mutator*** function that allows updating the state variable. The array destructor syntax is commonly used to bind these items to local constants as shown above. The mutator function not only changes the value of the state variable, but it also notifies React that it should check if the state has caused changes to the virtual DOM and therefore make changes to the actual DOM. The following exercises introduce various use cases of the ***useState***.

4.2.2.2 Integer State Variables

To illustrate the point of the ***virtual DOM*** and how changes in state affect changes in the actual DOM, let's implement the simple ***Counter*** component as shown below. A ***count*** variable is initialized and then rendered successfully on the screen. Buttons ***Up*** and ***Down*** successfully update the ***count*** variable as evidenced in the console, but the changes fail to update the DOM as desired. This happens because as far as React is concerned, there has been no changes to the virtual DOM, and therefore no need to update the actual DOM.

app/Labs/Lab4/Counter.tsx

```
import { useState } from "react";
export default function Counter() {
  let count = 7;                                    // declare and initialize
  console.log(count);                               // a variable. print changes
  return (                                          // of the variable to the console
    <div id="wd-counter-use-state">
      <h2>Counter: {count}</h2>                     // render variable
      <button
        onClick={() => { count++; console.log(count); }}   // variable updates on console
        id="wd-counter-up-click">Up</button>        // but fails to update the DOM as desired
      <button
        onClick={() => { count--; console.log(count); }}
        id="wd-counter-down-click">Down</button>
<hr/></div>);}
```

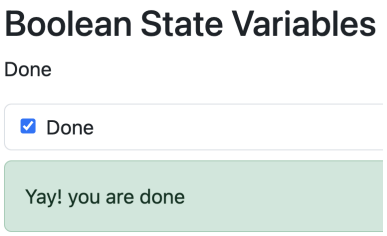
For the DOM to be updated as expected, we need to tell React that changes to a particular variable is indeed relevant to changes in the DOM. To do this, use the ***useState*** hook to declare the state variable, and update it using the mutator function as shown below. Now changes to the state variable are represented as changes in the DOM. Implement the ***Counter*** component, import it in ***Lab4*** and confirm it works as expected. Do the same with the rest of the exercises that follow.

app/Labs/Lab4/Counter.tsx

```
import { useState } from "react";                   // import useState
export default function Counter() {
  let count = 7;
  const [count, setCount] = useState(7);            // create and initialize
  console.log(count);                               // state variable
  return (
    <div>
      <h2>Counter: {count}</h2>                     // render state variable
      <button onClick={() => setCount(count + 1)}   // handle events and update
              id="wd-counter-up-click">Up</button>  // state variable with mutator
      <button onClick={() => setCount(count - 1)}   // now updates to the state
              id="wd-counter-down-click">Down</button>  // state variable do update the
<hr/></div>);}                                      // DOM as desired
```

4.2.2.3 Boolean State Variables

The ***useState*** hook works with all JavaScript data types and structures including ***booleans***, ***integers***, ***strings***, ***numbers***, ***arrays***, and ***objects***. The exercise below illustrates using the ***useState*** hook with ***boolean*** state variables. The variable is used to hide or show a DIV as well as render a checkbox as checked or not. Also note the use of ***onChange*** in the checkbox to set the value of state variable.



app/Labs/Lab4/BooleanStateVariables.tsx

```
import { useState } from "react";                   // import useState
export default function BooleanStateVariables() {
  const [done, setDone] = useState(true);           // declare and initialize
  return (                                          // boolean state variable
    <div id="wd-boolean-state-variables">
      <h2>Boolean State Variables</h2>
      <p>{done ? "Done" : "Not done"}</p>           // render content based on
      <label className="form-control">              // boolean state variable value
        <input type="checkbox" checked={done}       // change state variable value
               onChange={() => setDone(!done)} /> Done   // when handling events like
      </label>                                      // clicking a checkbox
      {done && <div className="alert alert-success">    // render content based on
               Yay! you are done</div>}             // boolean state variable value
<hr/></div>);}
```

4.2.2.4 String State Variables

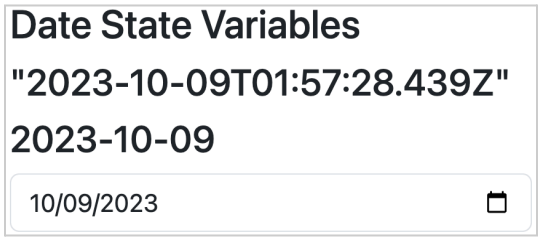
The ***StringStateVariables*** exercise below illustrates using ***useState*** with string state variables. The input field's ***value*** is initialized to the ***firstName*** state variable. The ***onChange*** attribute invokes the ***setFirstName*** mutator function to update the state variable. The ***e.target.value*** contains the value of the input field and is used to update the current value of the state variable.

app/Labs/Lab4/StringStateVariables.tsx

```
import { useState } from "react";                          // import useState
export default function StringStateVariables() {
  const [firstName, setFirstName] = useState("John");      // declare and
  return (                                                 // initialize
    <div>                                                  // state variable
      <h2>String State Variables</h2>
      <p>{firstName}</p>                                   // render string
      <FormControl                                         // state variable
        defaultValue={firstName}                           // initialize a
        onChange={(e) => setFirstName(e.target.value)}/>   // text input field with the state variable
<hr/></div>);}                                             // update the state variable at each key stroke
```

4.2.2.5 Date State Variables

The ***DateStateVariable*** component illustrates how to work with date state variables. The ***stateDate*** state variable is initialized to the current date using ***new Date()*** which has the string representation as shown here on the right. The ***dateObjectToHtmlDateString*** function can convert a ***Date*** object into the ***YYYY-MM-DD*** format expected by the HTML date input field. The function is used to initialize and set the date field's ***value*** attribute so it matches the expected format. Changes in date field are handled by the ***onChange*** attribute which updates the new date using the ***setStartDate*** mutator function.



app/Labs/Lab4/DateStateVariable.tsx

```
import { useState } from "react";                                  // import useState
export default function DateStateVariable() {
  const [startDate, setStartDate] = useState(new Date());          // declare and initialize with today's date
  const dateObjectToHtmlDateString = (date: Date) => {             // utility function to convert date object
    return `${date.getFullYear()}-${date.getMonth() + 1 < 10 ? 0 : ""}${    // to YYYY-MM-DD format for HTML date
      date.getMonth() + 1                                          // picker
    }-${date.getDate() + 1 < 10 ? 0 : ""}${date.getDate() + 1}`;
  };
  return (
    <div id="wd-date-state-variables">
      <h2>Date State Variables</h2>
      <h3>{JSON.stringify(startDate)}</h3>                         // display raw date object
      <h3>{dateObjectToHtmlDateString(startDate)}</h3>             // display in YYYY-MM-DD format for input
      <FormControl                                                 // of type date
        type="date"
        defaultValue={dateObjectToHtmlDateString(startDate)}       // set HTML input type to date
        onChange={(e) => setStartDate(new Date(e.target.value))}   // update when you change the date with
      />                                                           // the date picker
<hr/></div>);}
```

4.2.2.6 Object State Variables

The ***ObjectStateVariable*** component below demonstrates how to work with object state variables. We declare ***person*** object state variable with initial property values ***name*** and ***age***. The object is rendered on the screen using ***JSON.stringify*** to see the changes in real time. Two value of two input fields are initialized to the object's ***person.name*** string property and the object's ***person.age*** number property. As the user types in the input fields, the

onChange attribute passes the events to update the object's property using the ***setPerson*** mutator functions. The object is updated by creating new objects copied from the previous object value using the spreader operator (***...person***), and then overriding the ***name*** or ***age*** property with the ***target.value***.

app/Labs/Lab4/ObjectStateVariable.tsx

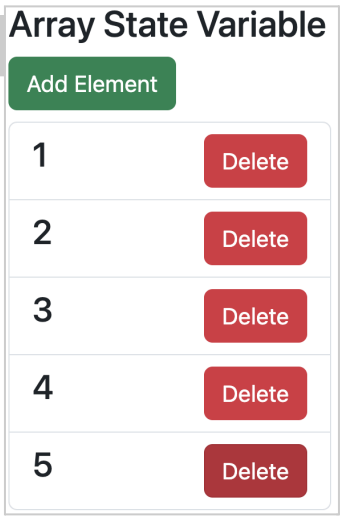
```
import { useState } from "react";
export default function ObjectStateVariable() {
  const [person, setPerson] = useState({ name: "Peter", age: 24 });        // declare and initialize object state
  return (                                                                 // variable with multiple fields
    <div>
      <h2>Object State Variables</h2>
      <pre>{JSON.stringify(person, null, 2)}</pre>                        // display raw JSON
      <FormControl                                                         // initialize input field with an object's
        defaultValue={person.name}                                         // field value
        onChange={(e) => setPerson({ ...person, name: e.target.value })}   // update field as user types. copy old
      />                                                                   // object, override specific field with new
      <FormControl                                                         // value
        defaultValue={person.age}                                          // update field as user types. copy old
        onChange={(e) => setPerson({ ...person,                            // object,
                                     age: parseInt(e.target.value) })}     // override specific field with new value
      />
      <hr/>
    </div>
  );
}
```

4.2.2.7 Array State Variables

The ***ArrayStateVariable*** component below demonstrates how to work with ***array*** state variables. An array of integers if declared as a state variable and function ***addElement*** and ***deleteElement*** are used to add and remove elements to and from the array. We render the array as a map of line items in an unordered list. We render the array's value and a ***Delete*** button for each element. Clicking the ***Delete*** button calls the ***deleteElement*** function which passes the ***index*** of the element we want to remove. The ***deleteElement*** function computes a new array filtering out the element by its position and updating the ***array*** state variable to contain a new array without the element we filtered out. Clicking the ***Add Element*** button invokes the ***addElement*** function which computes a new array with a copy of the previous ***array*** spread at the beginning of the new array, and adding a new random element at the end of the array. Add Bootstrap classes so the output renders as shown.

app/Labs/Lab4/ArrayStateVariable.tsx

```
import { useState } from "react";                          // import useState
export default function ArrayStateVariable() {
 const [array, setArray] = useState([1, 2, 3, 4, 5]);      // declare array state
 const addElement = () => {                                // event handler appends
   setArray([...array, Math.floor(Math.random() * 100)]);  // random number at end of
 };                                                        // array
 const deleteElement = (index: number) => {                // event handler removes
   setArray(array.filter((item, i) => i !== index));       // element by index
 };
 return (
  <div id="wd-array-state-variables">
   <h2>Array State Variable</h2>
   <button onClick={addElement}>Add Element</button>       // button calls addElement
   <ul>                                                    // to append to array
    {array.map((item, index) => (                          // iterate over array items
     <li key={index}> {item}
      <button onClick={() => deleteElement(index)}>        // render item's value
       Delete</button>                                     // button to delete element
     </li>))}                                              // by its index
   </ul><hr/></div>);}
```



4.2.2.8 Sharing State Between Components

State can be shared between components by passing references to state variables and/or functions that update them. The example below demonstrates a ***ParentStateComponent*** sharing ***counter*** state variable and ***setCounter*** mutator function with ***ChildStateComponent*** by passing it references to ***counter*** and ***setCounter*** as attributes.

app/Labs/Lab4/ParentStateComponent.tsx

```
import { useState } from "react";
import ChildStateComponent from "./ChildStateComponent";
export default function ParentStateComponent() {
  const [counter, setCounter] = useState(123);
  return (
    <div>
      <h2>Counter {counter}</h2>
      <ChildStateComponent
        counter={counter}
        setCounter={setCounter} />
      <hr/>
    </div>
);}
```

The ***ChildStateComponent*** can use references to ***counter*** and ***setCounter*** to render the state variable and manipulate it through the mutator function. Import ***ParentStateComponent*** into ***Lab4*** and confirm it works as expected.

app/Labs/Lab4/ChildStateComponent.tsx

```
export default function ChildStateComponent({
 counter,
 setCounter
} : {
 counter: number;
 setCounter: (counter: number) => void;
}) {
  return (
    <div id="wd-child-state">
      <h3>Counter {counter}</h3>
      <button onClick={() => setCounter(counter + 1)} id="wd-increment-child-state-click">
        Increment</button>
      <button onClick={() => setCounter(counter - 1)} id="wd-decrement-child-state-click">
        Decrement</button>
      <hr/>
    </div>
);}
```

4.3 Managing Application State with Redux

The ***useState*** hook is used to maintain the state within a component. State can be shared across components by passing references to state variables and mutators to other components. Although this approach is sufficient as a general approach to share state among multiple components, it is fraught with challenges when building larger, more complex applications. The downside of using ***useState*** across multiple components is that it creates an explicit dependency between these components, making it hard to refactor components adapting to changing requirements. The solution is to eliminate the dependency using libraries such as ***Redux***. This section explores the Redux library to manage state that is meant to be used across a large set of components, and even an entire application. We'll keep using ***useState*** to manage state within individual components, but use Redux to manage Application level state. To learn about redux, let's create a redux examples component that will contain several simple redux examples. Create an ***page.tsx*** file under ***app/Labs/Lab4/ReduxExamples/page.tsx*** as shown below. Import the new redux examples component into the Lab 4 component so we can see how it renders as we add new examples. Reload the browser and confirm the new component renders as expected.

app/Labs/Lab4/ReduxExamples/page.tsx

```
export default function ReduxExamples() {
  return(
    <div>
      <h2>Redux Examples</h2>
    </div>
  );
};
```

app/Labs/Lab4/page.tsx

```
import ReduxExamples from "./ReduxExamples";
export default const Lab4 = () => {
 return(
   <>
     <h2>Lab 4</h2>
     ...
     <ReduxExamples/>
   </>
 );};
```

4.3.1 Installing Redux

As mentioned earlier we will be using the ***Redux*** state management library to handle application state. To install ***Redux***, type the following at the command line from the root folder of your application.

```
npm  install  redux --save
```

After redux has installed, install ***react-redux*** and the redux ***toolkit***, the libraries that integrate ***redux*** with ***React***. At the command line, type the following commands.

```
npm  install  react-redux --save
npm  install  @reduxjs/toolkit --save
```

4.3.2 Create a Hello World Redux component

To learn about Redux, let's start with a simple Hello World example. Instead of maintaining state within any particular component, Redux declares and manages state in separate ***reducers*** which then ***provide*** the state to the entire application. Create ***helloReducer*** as shown below maintaining a state that consists of just a ***message*** state string initialized to ***Hello World***.

app/Labs/Lab4/ReduxExamples/HelloRedux/helloReducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  message: "Hello World",
};
const helloSlice = createSlice({
  name: "hello",
  initialState,
  reducers: {},
});
export default helloSlice.reducer;
```

Application state can maintain data from various components or screens across an entire application. Each would have a separate reducer that can be combined into a single ***store*** where reducers come together to create a complex, application wide state. The ***store.tsx*** below demonstrates adding the ***helloReducer*** to the store. Later exercises and the ***Kambaz*** section will add additional reducers to the store.

app/Labs/store/index.ts

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
const store = configureStore({
```

```
  reducer: { helloReducer }});
export default store;
```

The application state can then be shared with the entire Web application by wrapping it with a ***Provider*** component that makes the state data in the ***store*** available to all components within the ***Provider***'s body.

app/Labs/page.tsx

```
...
import store from "./store";
import { Provider } from "react-redux";
export default function Labs() {
  return (
    <Provider store={store}>
      <div className="container-fluid">
        <h1>Labs</h1>
        ...
      </div>
    </Provider>
  );
}
```

Components within the body of the ***Provider*** can then ***select*** the state data they want using the ***useSelector*** hook as shown below. Add the ***HelloRedux*** component to ***ReduxExamples*** and confirm it renders as shown below.

app/Labs/Lab4/ReduxExamples/HelloRedux/page.tsx

```
import { useSelector, useDispatch } from "react-redux";
export default function HelloRedux() {
  const { message } = useSelector((state: any) => state.helloReducer);
  return (
    <div id="wd-hello-redux">
      <h3>Hello Redux</h3>
      <h4>{message}</h4> <hr />
    </div>
  );
}
```

Redux Examples

Hello Redux

Hello World

4.3.3 Counter Redux - Dispatching Events to Reducers

To practice with Redux, let's reimplement the ***Counter*** component using Redux. First create ***counterReducer*** responsible for maintaining the counter's state. Initialize the state variable ***count*** to 0, and reducer function ***increment*** and ***decrement*** can update the state variable by manipulating their ***state*** parameter that contain state variables as shown below.

app/Labs/Lab4/ReduxExamples/CounterRedux/counterReducer.tsx

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  count: 0,
};
const counterSlice = createSlice({
  name: "counter",
  initialState,
  reducers: {
    increment: (state) => {
      state.count = state.count + 1;
    },
    decrement: (state) => {
      state.count = state.count - 1;
    },
  },
});
export const { increment, decrement } = counterSlice.actions;
```

```
export default counterSlice.reducer;
```

Add the ***counterReducer*** to the ***store*** as shown below to make the counter's state available to all components within the body of the ***Provider***.

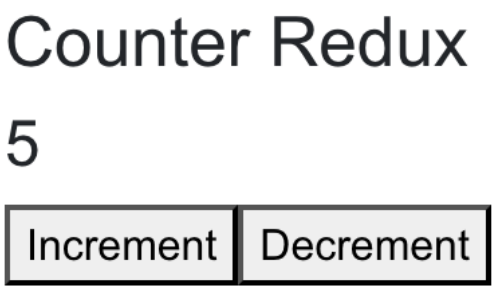
app/Labs/store/page.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
const store = configureStore({
  reducer: {
    helloReducer,
    counterReducer,
  },
});
export default store;
```

The ***CounterRedux*** component below can then ***select*** the ***count*** state from the store using the ***useSelector*** hook. To invoke the reducer function ***increment*** and ***decrement*** use a ***dispatch*** function obtained from a ***useDispatch*** function as shown below. Add ***CounterRedux*** to ***ReduxExamples*** and confirm it works as expected.

app/Labs/Lab4/ReduxExamples/CounterRedux/page.tsx

```
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./counterReducer";
export default function CounterRedux() {
  const { count } = useSelector((state: any) => state.counterReducer);
  const dispatch = useDispatch();
  return (
    <div id="wd-counter-redux">
      <h2>Counter Redux</h2>
      <h3>{count}</h3>
      <button onClick={() => dispatch(increment())}
              id="wd-counter-redux-increment-click"> Increment </button>
      <button onClick={() => dispatch(decrement())}
              id="wd-counter-redux-decrement-click"> Decrement </button>
      <hr/>
    </div>
);}
```



4.3.4 Passing Data to Reducers

Now let's explore how the user interface can pass data to reducer functions. Create a reducer that can keep track of the arithmetic addition of two parameters. When we call ***add*** reducer function below, the parameters are encoded as an object into a ***payload*** property found in the ***action*** parameter passed to the reducer function. Functions can extract parameters ***a*** and ***b*** as ***action.payload.a*** and ***action.payload.b*** and then use the parameters to update the ***sum*** state variable.

app/Labs/Lab4/ReduxExamples/AddRedux/addReducer.tsx

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  sum: 0,
};
const addSlice = createSlice({
  name: "add",
  initialState,
  reducers: {
    add: (state, action) => {
      state.sum = action.payload.a + action.payload.b;
    },
```

```
  },
});
export const { add } = addSlice.actions;
export default addSlice.reducer;
```

Add the new reducer to the store so it's available throughout the application as shown below.

app/Labs/store/page.tsx

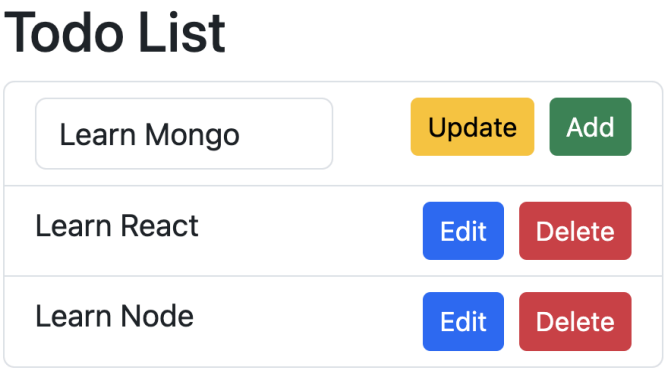
```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer   from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
import addReducer     from "../Lab4/ReduxExamples/AddRedux/addReducer";
const store = configureStore({
  reducer: {
    helloReducer,
    counterReducer,
    addReducer,
  },
});
export default store;
```

To tryout the new reducer, import the ***add*** reducer function as shown in the ***AddRedux*** component below. Maintain the values of ***a*** and ***b*** as local component state variables, and then pass them to ***add*** as a single object. Add ***AddRedux*** to ***ReduxExamples*** to confirm it works as expected.

app/Labs/Lab4/ReduxExamples/AddRedux/page.tsx

```
import { useSelector, useDispatch } from "react-redux";            // to read/write to reducer
import { useState } from "react";                                  // to maintain a and b parameters in UI
import { add } from "./addReducer";
export default function AddRedux() {
  const [a, setA] = useState(12);
  const [b, setB] = useState(23);                                  // a and b state variables to edit
  const { sum } = useSelector((state: any) => state.addReducer);   // parameters to add in the reducer
  const dispatch = useDispatch();                                  // read the sum state variable from the reducer
  return (                                                         // dispatch to call add redux function
    <div className="w-25" id="wd-add-redux">
      <h1>Add Redux</h1>                                           // render local state variables a and b, as well
      <h2>{a} + {b} = {sum}</h2>                                   // as application state variable sum
      <FormControl type="number" defaultValue={a}
        onChange={(e) => setA(parseInt(e.target.value))} />        // update the local component state variable a
      <FormControl type="number" defaultValue={b}
        onChange={(e) => setB(parseInt(e.target.value))} />        // update the local component state variable b
      <Button id="wd-add-redux-click"
              onClick={() => dispatch(add({ a, b }))}>
        Add Redux                                                  // on click, call add reducer function to
      </Button>                                                    // compute the arithmetic addition of a and b,
      <hr/>                                                        // and store it in application state
    </div>                                                         // variable sum
  );
}
```

4.3.5 Implementing a Todo List with Redux



Let's practice using local component state as well as application level state to implement a simple ***Todo List*** component. First we'll implement the component using only component state with ***useState*** which will limit the todos to only available within the ***Todo List***. We'll then add application state support to demonstrate how the todos can be shared with any component or screen in the application. Create the

TodoList component as shown below. Add ***Bootstrap*** classes to style the todos as shown here on the right.

app/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

```
import { useState } from "react";                                    // import useState
export default function TodoList() {
  const [todos, setTodos] = useState([                               // create todos array state variable
    { id: "1", title: "Learn React" },                               // initialize with 2 todo objects
    { id: "2", title: "Learn Node"  }]);
  const [todo, setTodo] = useState({ id: "-1", title: "Learn Mongo" }); // create todo state variable object
  const addTodo = (todo: any) => {                                   // event handler to add new todo
    const newTodos = [ ...todos, { ...todo,                          // spread existing todos, append new todo,
      id: new Date().getTime().toString() }];                        // override id
    setTodos(newTodos);                                              // update todos
    setTodo({id: "-1", title: ""});                                  // clear the todo
  };
  const deleteTodo = (id: string) => {                               // event handler to remove
    const newTodos = todos.filter((todo) => todo.id !== id);         // todo by their ID
    setTodos(newTodos);
  };
  const updateTodo = (todo: any) => {                                // event handler to
    const newTodos = todos.map((item) =>                             // update todo by
      (item.id === todo.id ? todo : item));                          // replacing todo
    setTodos(newTodos);                                              // by their ID
    setTodo({id: "-1", title: ""});
  };
  return (
    <div>
      <h2>Todo List</h2>
      <ListGroup
        <ListGroupItem>
          <Button onClick={() => addTodo(todo)}                      // add todo button
                  id="wd-add-todo-click"> Add </Button>              // update todo button
          <Button onClick={() => updateTodo(todo)}
                  id="wd-update-todo-click"> Update </Button>        // input field to update todo's title
          <FormControl value={todo.title}                            // for every keystroke
            onChange={(e) => setTodo({ ...todo, title: e.target.value })}/>  // update the todo's title,
        </ListGroupItem>                                             // but copy old values first
        {todos.map((todo) => (                                       // render all todos
          <ListGroupItem key={todo.id}>                              // as line items
            <Button onClick={() => deleteTodo(todo.id)}              // button to delete todo by their ID
                    id="wd-delete-todo-click"> Delete </Button>
            <Button onClick={() => setTodo(todo)}                    // button to select todo to edit
                    id="wd-set-todo-click"> Edit </Button>
            {todo.title}
          </ListGroupItem>
        ))}
      </ListGroup><hr/>
</div>);}
```

4.3.5.1 Breaking up Large Components

Let's break up the ***TodoList*** component into several smaller components: ***TodoItem*** and ***TodoForm***. ***TodoItem*** shown below breaks out the line items that render the todo's title, and ***Delete*** and ***Edit*** buttons. The component accepts references to the ***todo*** object, as well as ***deleteTodo*** and ***setTodo*** functions.

app/Labs/Lab4/ReduxExamples/todos/TodoItem.tsx

```
export default function TodoItem({ todo, deleteTodo, setTodo }: {   // breaks out todo item
  todo: { id: string; title: string };                              // todo to render
  deleteTodo: (id: string) => void;                                 // event handler to remove todo
  setTodo: (todo: { id: string; title: string }) => void;           // event handler to select todo
}) {
  return (
    <ListGroupItem key={todo.id}>
```

```
      <Button onClick={() => deleteTodo(todo.id)}                    // invoke delete todo with ID
              id="wd-delete-todo-click"> Delete </Button>
      <Button onClick={() => setTodo(todo)}                          // invoke select todo
              id="wd-set-todo-click"> Edit </Button>
      {todo.title}    </ListGroupItem>);}                            // render todo's title
```

Similarly we'll break out the form to ***Create*** and ***Update*** todos into component ***TodoForm*** shown below. Parameters ***todo***, ***setTodo***, ***addTodo***, and ***updateTodo***, to maintain dependencies between the ***TodoList*** and ***TodoForm*** component.

app/Labs/Lab4/ReduxExamples/todos/TodoForm.tsx

```
export default function TodoForm({ todo, setTodo, addTodo, updateTodo }: {    // breaks out todo form
  todo: { id: string; title: string };                                        // todo to be added or edited
  setTodo: (todo: { id: string; title: string }) => void;                     // event handler to update todo's title
  addTodo: (todo: { id: string; title: string }) => void;                     // event handler to add new todo
  updateTodo: (todo: { id: string; title: string }) => void;                  // event handler to update todo
}) {
  return (
    <ListGroupItem>
      <Button onClick={() => addTodo(todo)}                                   // invoke add new todo
              id="wd-add-todo-click"> Add </Button>                           // invoke update todo
      <Button onClick={() => updateTodo(todo)}                                // input field to update
              id="wd-update-todo-click"> Update </Button>                     // todo's title
      <FormControl value={todo.title}                                         // update title on each key stroke
        onChange={ (e) => setTodo({ ...todo, title: e.target.value }) }/>
    </ListGroupItem>
);}
```

Now we can replace the form and todo items in the ***TodoList*** component as shown below. Add the ***TodoList*** component to ***Lab4*** and confirm it works as expected.

app/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

```
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";                          // import TodoForm
export default function TodoList() {                        // import TotoItem
  ...
  return (
    <div id="wd-todo-list-redux">
      <h2>Todo List</h2>
      <ListGroup>
        <TodoForm
          todo={todo}                                       // TodoForm breaks out form to add or update todo
          setTodo={setTodo}                                 // pass state variables and
          addTodo={addTodo}                                 // event handlers
          updateTodo={updateTodo}/>                         // so component
        {todos.map((todo) => (                              // can communicate with TodoList's data and functions
          <TodoItem
            todo={todo}                                     // TodoItem breaks out todo item
            deleteTodo={deleteTodo}                         // pass state variables and
            setTodo={setTodo} />                            // event handlers to
        ))}                                                 // communicate with TodoList's data and functions
      </ListGroup>
<hr/></div>);}
```

4.3.5.2 Todos Reducer

Although the ***TodoList*** component might work as expected and it might be all we would need, it's implementation makes it difficult to share the local state data (the todos) outside its context with other components or screens. For instance, how would we go about accessing and displaying the todos, say, in the ***Lab3*** component or ***Kambaz***? We would have to move the todos state variable and mutator functions to a component that is parent to both the ***Lab3*** component and the ***TodoList*** component, e.g., ***Labs*** or even ***App***.

Instead, let's move the state and functions from the ***TodoList*** component to a reducer and store so that the todos can be accessed from anywhere within the ***Labs***. Create ***todosReducer*** as shown below, moving the ***todos*** and ***todo*** state variables to the reducer's ***initialState***. Also move the ***addTodo***, ***deleteTodo***, ***updateTodo***, and ***setTodo*** functions into the ***reducers*** property, reimplementing them to use the ***state*** and ***action*** parameters of the new reducer functions.

app/Labs/Lab4/ReduxExamples/todos/todosReducer.ts

```
import { createSlice } from "@reduxjs/toolkit";                          // import createSlice
const initialState = {                                                   // declare initial state of reducer
  todos: [                                                               // moved here from TodoList.tsx
    { id: "1", title: "Learn React" },                                   // todos has default todos
    { id: "2", title: "Learn Node" },
  ],
  todo: { title: "Learn Mongo" },                                        // todo has default todo
};
const todosSlice = createSlice({                                         // create slice
  name: "todos",                                                         // name slice
  initialState,                                                          // configure store's initial state
  reducers: {                                                            // declare reducer functions
    addTodo: (state, action) => {                                        // addTodo reducer function, action
      const newTodos = [                                                 // contains new todo. newTodos
          ...state.todos,                                                // copy old todos, append new todo
        { ...action.payload, id: new Date().getTime().toString() },      // in action.payload, override
      ];                                                                 // id as timestamp
      state.todos = newTodos;                                            // update todos
      state.todo = { title: "" };                                        // clear todo
    },
    deleteTodo: (state, action) => {                                     // deleteTodo reducer function,
      const newTodos = state.todos.filter((todo) => todo.id !== action.payload);  // action contains todo's ID to
      state.todos = newTodos;                                            // filter out of newTodos
    },
    updateTodo: (state, action) => {                                     // updateTodo reducer function
      const newTodos = state.todos.map((item) =>                         // rebuilding newTodos by replacing
        item.id === action.payload.id ? action.payload : item            // old todo with new todo in
      );                                                                 // action.payload
      state.todos = newTodos;                                            // update todos
      state.todo = { title: "" };                                        // clear todo
    },
    setTodo: (state, action) => {                                        // setTodo reducer function
      state.todo = action.payload;                                       // to update todo state variable
    },
  },
});
export const { addTodo, deleteTodo, updateTodo, setTodo } = todosSlice.actions;  // export reducer functions
export default todosSlice.reducer;                                       // export reducer for store
```

Add the new ***todosReducer*** to the ***store*** so that it can be provided to the rest of the ***Labs***.

app/Labs/store/page.tsx

```
import { configureStore } from "@reduxjs/toolkit";
import helloReducer from "../Lab4/ReduxExamples/HelloRedux/helloReducer";
import counterReducer from "../Lab4/ReduxExamples/CounterRedux/counterReducer";
import addReducer from "../Lab4/ReduxExamples/AddRedux/addReducer";
import todosReducer from "../Lab4/ReduxExamples/todos/todosReducer";
const store = configureStore({
  reducer: {
    helloReducer,
    counterReducer,
    addReducer,
    todosReducer,
  },
});
export default store;
```

Now that we've moved the state and mutator functions to the ***todosReducer***, refactor the ***TodoForm*** component to use the reducer functions instead of the parameters. Also select the ***todo*** from the reducer state, instead of the ***todo*** parameter.

app/Labs/Lab4/ReduxExamples/todos/TodoForm.tsx

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";                    // import useSelector, useDispatch
import { addTodo, updateTodo, setTodo } from "./todosReducer";             // to read/write to reducer
                                                                           // reducer functions
export default function TodoForm(                                          // remove dependency from
{ todo, setTodo, addTodo, updateTodo }                                     // parent component
) {
  const { todo } = useSelector((state: any) => state.todosReducer);        // retrieve todo from reducer
  const dispatch = useDispatch();                                          // create dispatch instance to
  return (                                                                 // invoke reducer functions
    <ListGroupItem>
      <Button onClick={() => dispatch(addTodo(todo))}                      // wrap reducer functions
              id="wd-add-todo-click"> Add </Button>                        // with dispatch
      <Button onClick={() => dispatch(updateTodo(todo))}
              id="wd-update-todo-click"> Update </Button>
      <FormControl                                                         // wrap reducer functions
        defaultValue={todo.title}                                          // with dispatch
        onChange={(e) => dispatch(setTodo({ ...todo, title: e.target.value }))}/>
    </ListGroupItem>
);}
```

Also reimplement the ***TodoItem*** component as shown below, using the reducer functions instead of the parameters.

app/Labs/Lab4/ReduxExamples/todos/TodoItem.tsx

```
import React from "react";
import { useDispatch } from "react-redux";                  // import useDispatch to invoke reducer
import { deleteTodo, setTodo } from "./todosReducer";       // functions deleteTodo and setTodo
export default function TodoItem({ todo,
  deleteTodo, setTodo                                       // remove dependency with
 }) {                                                       // parent component
  const dispatch = useDispatch();                           // create dispatch instance to invoke
  return (                                                  // reducer functions
    <ListGroupItem key={todo.id}>
      <Button onClick={() => dispatch(deleteTodo(todo.id))}  // wrap reducer functions with dispatch
              id="wd-delete-todo-click"> Delete </Button>
      <Button onClick={() => dispatch(setTodo(todo))}
              id="wd-set-todo-click"> Edit </Button>
      {todo.title}
    </ListGroupItem>
);}
```

Reimplement the ***TodoForm*** and ***TodoItem*** components as shown above and update the ***TodoList*** component as shown below. Remove unnecessary dependencies and confirm that it works as before.

app/Labs/Lab4/ReduxExamples/todos/TodoList.tsx

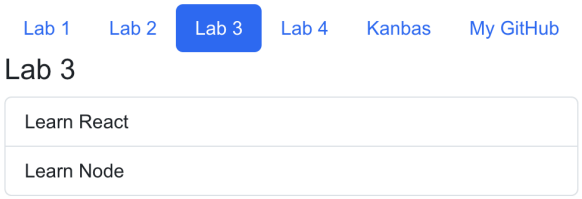
```
import React from "react";
import TodoForm from "./TodoForm";
import TodoItem from "./TodoItem";
import { useSelector } from "react-redux";                          // import useSelector to retrieve
export default function TodoList() {                                // data from reducer
  const { todos } = useSelector((state: any) => state.todosReducer);
  return (                                                          // extract todos from reducer and remove
    <div id="wd-todo-list-redux">                                   // all other event handlers
      <h2>Todo List</h2>
      <ListGroup>
        <TodoForm />                                                // remove unnecessary attributes
        {todos.map((todo: any) => (
          <TodoItem todo={todo} />                                  // remove unnecessary attributes,
        ))}                                                         // but still pass the todo
      </ListGroup>
      <hr/>
    </div>
);}
```

Now the todos are available to any component in the body of the ***Provider***. To illustrate this, select the todos from within the ***Lab3*** component as shown below and confirm the todos display in ***Lab3***.

app/Labs/Lab3/page.tsx

```
...
import { useSelector } from "react-redux";

export default function Lab3() {
  const { todos } = useSelector((state: any) => state.todosReducer);
  return (
    <div>
      <h2>Lab 3</h2>
      <ListGroup>
        {todos.map((todo: any) => (
          <ListGroupItem key={todo.id}>
            {todo.title}
          </ListGroupItem>
        ))}
      </ListGroup>
      <hr />
      ...
    </div>);}
```



4.4 Adding State to the Kambaz User Interface

The current ***Kambaz*** implementation reads data from a ***Database*** containing ***courses***, ***modules***, ***assignments***, and ***grades***, and dynamically renders screens ***Dashboard***, ***Home***, ***Module***, ***Assignments***, and ***Grades***. The data is currently static, and our ***Kambaz*** implementation is basically a set of functions that transform the data in the ***Database*** into an corresponding user interface. Since the data is static, the user interface is static as well. In this section we will use the component and application state skills we learned in the ***Labs*** section, to refactor the ***Kambaz*** application so we can create new ***courses***, ***modules*** and ***assignments***.

4.4.1 Adding State to the Kambaz Dashboard

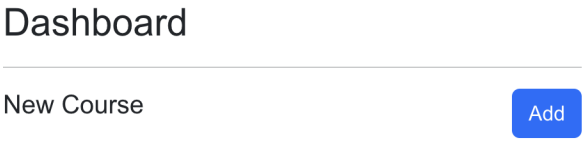
The current ***Dashboard*** implementation renders a static array of courses. This section illustrates how to refactor the ***Dashboard*** to implement ***CRUD*** operations such as ***create*** new courses, ***read*** courses, ***update*** existing course titles, and ***delete*** courses. Import the ***useState*** hook and convert the ***courses*** constant into a state variable as shown below. Make these changes in the current implementation using the code below as an example.

app/(Kambaz)/Dashboard/page.tsx

```
import React, { useState } from "react";                              // add useState hook
import Link from "next/link";
import * as db from "../Database";
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);          // create courses state
  return (                                                            // variable and initialize
    <div className="p-4" id="wd-dashboard">                           // with database's courses
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h2 id="wd-dashboard-published">Published Courses ({courses.length})</h2> <hr />
      <div className="row" id="wd-dashboard-courses">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            <div key={course._id} className="col" style={{ width: "300px" }}>
              <div className="card">
                 ...  {course.name}   ...
              </div>
            </div>
```

```
        ))}
      </div>
    </div>
  </div>
);}
```

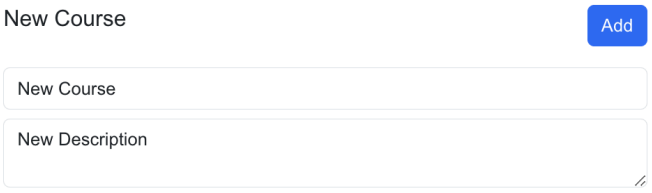
4.4.1.1 Creating New Courses



To create new courses, implement the ***addNewCourse*** function as shown below and add a new ***Add*** button that invokes ***addNewCourse*** function to append a new course at the end of the ***courses*** array. The ***addNewCourse*** function overrides the ***_id*** property with a unique timestamp.

app/(Kambaz)/Dashboard/page.tsx

```
import { v4 as uuidv4 } from "uuid";
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const course: any = {                                    // create a course object with default values
    _id: "0", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15",
    image: "/images/reactjs.jpg", description: "New Description"
  };
  const addNewCourse = () => {                             // create addNewCourse event handler that sets
    const newCourse = { ...course, _id: uuidv4() };        // courses as copy of current courses state array
    setCourses([...courses, newCourse ]);                  // add course at the end of the array
  };                                                       // overriding _id to current time stamp
  return (
    <div className="p-4" id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course                                       // add a title and button to invoke
          <button className="btn btn-primary float-end"    // addNewCourse. Note no argument syntax
                  id="wd-add-new-course-click"
                  onClick={addNewCourse} > Add </button>
      </h5><hr />
      ...
    </div>
);}
```



Confirm you can add new courses and the ***Published Courses*** counter increases. Modify the ***img*** tag so that it either renders a hardcoded image, e.g., ***"/images/react.jpg"***, or renders the course's ***image*** property, but then you'll need to add image properties to ***courses.json***. Convert the ***course*** constant into a state variable as shown below. Add a form to edit the ***course*** state variable's ***name***, and ***description*** properties. Confirm form shows values of the ***course*** state variable.

app/(Kambaz)/Dashboard/page.tsx

```
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({              // convert course into a state
    _id: "0", name: "New Course", number: "New Number",    // variable so we can change it
    startDate: "2023-09-10", endDate: "2023-12-15",        // and force a redraw of the UI
    image: "/images/reactjs.jpg", description: "New Description"
  });
  const addNewCourse = () => { ... };
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course ... </h5><br />
      <FormControl value={course.name} className="mb-2" />  // add input element for each of
```

```
      <FormControl value={course.description} rows={3}/>          // fields in course state
      <hr />      ...                                             // variable
    </div>
);}
```

Add ***onChange*** attributes to each of the input fields to update each of the fields using the ***setCourse*** mutator function, as shown below. Use your implementation of ***Dashboard*** and the code provided as an example. Confirm you can add new courses.

app/(Kambaz)/Dashboard/page.tsx

```
  return (
    <div>
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h5>New Course ... </h5> <br />
      <FormControl value={course.name} className="mb-2"
             onChange={(e) => setCourse({ ...course, name: e.target.value }) } />
      <FormControl value={course.description} rows={3}
             onChange={(e) => setCourse({ ...course, description: e.target.value }) } />
      ...
    </div>
  );
```

4.4.1.2 Deleting a Course

Now let's implement deleting courses by adding ***Delete*** buttons to each of the courses. The buttons invoke a new ***deleteCourse*** function that accepts the ID of the course to remove. The function filters out the course from the ***courses*** array. Use the code below as an example to refactor your ***Dashboard*** component. Confirm that you can remove courses.

app/(Kambaz)/Dashboard/page.tsx

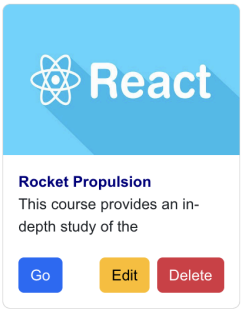
```
...
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({ ... });
  const addNewCourse = () => { ... };
  const deleteCourse = (courseId: string) => {                    // add deleteCourse event handler accepting
    setCourses(courses.filter((course) => course._id !== courseId));  // ID of course to remove by filtering out
  };                                                              // the course by its ID
  return (
    <div id="wd-dashboard">
      <h1>Dashboard</h1>
      ...
      <div className="row">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            ...
            <button className="btn btn-primary">
                    Go </button>
            <button onClick={(event) => {
                      event.preventDefault();                     // add Delete button next to the course's
                      deleteCourse(course._id);                   // name to invoke deleteCourse when clicked
                    }} className="btn btn-danger float-end"       // passing the course's ID and preventing
                    id="wd-delete-course-click">                  // the Link's default behavior to navigate
                    Delete                                        // to Course Screen
            </button>
            ...
          ))}
        </div>
      </div>
    </div>
);}
```

4.4.1.3 Editing a Course

Now let's implement editing an existing course by adding ***Edit*** buttons to each of the courses which invoke a new ***setCourse*** function that copies the current course into the ***course*** state variable, displaying the course in the form so you can edit it. Refactor your ***Dashboard*** component using the code below as an example. Confirm that clicking ***Edit*** on a course, copies the course into the form.

app/(Kambaz)/Dashboard/page.tsx

```
<button id="wd-edit-course-click"                  // next to the Delete button
  onClick={(event) => {                            // add an Edit button to copy the course
    event.preventDefault();                        // to be edited into the form so we can
    setCourse(course);                             // edit it. Prevent default to navigate
  }}                                               // to Course screen
  className="btn btn-warning me-2 float-end" >
  Edit
</button>
```



Add an ***Update*** button to the form so that the selected course can be updated with the values in the edited fields. Use the code below as an example. Confirm you can select, and then edit the selected course. Confirm that clicking ***Update*** actually updates the original course's ***name*** and ***description***.

app/(Kambaz)/Dashboard/page.tsx

```
...
export default function Dashboard() {
  const [courses, setCourses] = useState<any[]>(db.courses);
  const [course, setCourse] = useState<any>({ ... });
  const updateCourse = () => {
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        } else {
          return c;
        }
      })
    );
  };
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1>
      <hr />
      <h5>
        New Course
        <button className="btn btn-primary float-end"
                onClick={addNewCourse} id="wd-add-new-course-click">
          Add
        </button>
        <button className="btn btn-warning float-end me-2"
                onClick={updateCourse} id="wd-update-course-click">
          Update
        </button>
      </h5>
      ...
    </div>
  );
}
```

4.4.2 Courses Screen

The ***Dashboard*** component seems to be working fine, but the courses it is creating, deleting, and updating can not be used outside of the component. This is a problem because the ***Courses*** screen would want to be able to render the new courses, but it doesn't have access to the ***courses*** state variable in the ***Dashboard***. To fix this we need to either add redux so all courses are available everywhere, or move the ***courses*** state variable up to a component that is parent to both the

Dashboard and the ***Courses***. Let's take this last approach first, and then we'll explore adding ***Redux***. Let's move all the state variables and event handlers from the ***Dashboard***, and move them to the ***Kambaz*** component since it is parent to both the ***Dashboard*** and ***Courses*** component. Then add references to the state variables and event handlers as parameter dependencies in ***Dashboard*** as shown below. Refactor your ***Dashboard*** component based on the example code below.

app/(Kambaz)/Dashboard/page.tsx

```
...
export default function Dashboard(
{ courses, course, setCourse, addNewCourse,                          // move the state variables and
  deleteCourse, updateCourse }: {                                    // event handler functions
  courses: any[]; course: any; setCourse: (course: any) => void;     // to Kambaz and then accept
  addNewCourse: () => void; deleteCourse: (course: any) => void;     // them as parameters
  updateCourse: () => void; })
 {
  return (
    <div id="wd-dashboard">
      <h1>Dashboard</h1>
      ...
    </div>
  ); }
```

Refactor your ***Kambaz*** component moving the state variables and functions from the ***Dashboard*** component. Confirm the ***Dashboard*** still works the same, e.g., renders the courses, can add, updates, and remove courses

app/(Kambaz)/page.tsx

```
import KambazNavigation from "./KambazNavigation";
import { Routes, Route, Navigate } from "react-router-dom";
import Dashboard from "./Dashboard";
import Courses from "./Courses";
import * as db from "./Database";                                    // import the database
import { useState } from "react";                                    // import the useState hook
import { v4 as uuidv4 } from "uuid";

export default function Kambaz() {
  const [courses, setCourses] = useState<any[]>(db.courses);         // move the state variables here
  const [course, setCourse] = useState<any>({                        // from the Dashboard
    _id: "1234", name: "New Course", number: "New Number",
    startDate: "2023-09-10", endDate: "2023-12-15", description: "New Description",
  });
  const addNewCourse = () => {                                       // move the event handlers here
    setCourses([...courses, { ...course, _id: uuidv4() }]);          // from the Dashboard
  };
  const deleteCourse = (courseId: any) => {
    setCourses(courses.filter((course) => course._id !== courseId));
  };
  const updateCourse = () => {
    setCourses(
      courses.map((c) => {
        if (c._id === course._id) {
          return course;
        } else {
          return c;
        }
      })
    );
  };
  return (
    <div id="wd-kambaz">
      <KambazNavigation />
      <div className="wd-main-content-offset p-3">
        <Routes>
          <Route path="/" element={<Navigate href="Dashboard" />} />
          <Route path="Account" element={<h1>Account</h1>} />
          <Route path="Dashboard" element={
            <Dashboard                                               // pass a reference of the state
```

```
              courses={courses}                                          // variables and event handlers to
              course={course}                                            // the Dashboard so it can read
              setCourse={setCourse}                                      // the state variables and invoke
              addNewCourse={addNewCourse}                                // the event handlers from the
              deleteCourse={deleteCourse}                                // Dashboard
              updateCourse={updateCourse}/>
          } />
          <Route path="Courses/:cid/*" element={<Courses courses={courses} />} />     // also pass all the courses to
        </Routes>                                                        // the Courses screen since now
      </div>                                                             // it might contain new courses
    </div>);}                                                            // not initially in the database
```

Now that we have the ***courses*** declared in the ***Kambaz*** component, we can share them with the ***Courses*** screen component by passing them as an attribute. The ***Courses*** component destructs the courses from the parameter and then finds the course by the ***courseId*** path parameter searching through the ***courses*** parameter instead of the ***courses*** in the ***Database***. Refactor your ***Courses*** component as suggested below and confirm you can navigate to new courses created in the ***Dashboard***.

app/(Kambaz)/Courses/[cid]/page.tsx

```
// import { courses } from "../Database";
export default function Courses({ courses }: { courses: any[]; }) {    // don't load courses from Database
  const { cid } = useParams();                                          // accept courses from Kambaz
  const course = courses.find((course) => course._id === cid);
  return (...);                                                         // find the course by its ID
}
```

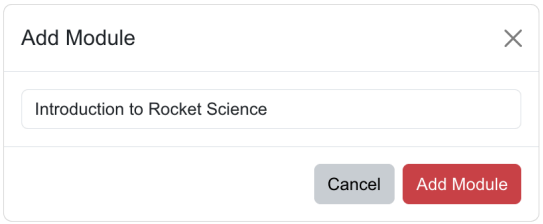
4.4.3 Modules

Now let's do the same with ***Modules*** refactoring the component adding state variables so that we can create, update, and remove modules. We'll discover the same limitation we had with ***courses***, i.e., we won't be able to share new modules outside the ***Modules*** screen. But instead of moving the modules state variable and functions to a shared common parent component, we'll instead use ***Redux*** to make the modules available throughout the application. The screenshot here on the right is for illustration purposes only. Reuse the HTML and CSS from previous chapters to style your modules. Refactor your ***Modules*** implementation by converting the ***modules*** array into a state variable as shown below. Confirm ***Modules*** renders as expected. Styling shown here is for illustration purposes. Use your HTML and CSS from previous chapters to style the modules.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import React, { useState } from "react";
import { useParams } from "react-router";
import * as db from "../../Database";
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  return ( ... );
}
```

4.4.3.1 Creating a Module



Let's create a modal dialog where users can edit the name of a new module based on ***Bootstrap's modal component***. The ***ModuleEditor*** component below pops up when you click the red ***+ Module*** button in the ***Modules*** and ***Home*** screens. You can type the name of the new module in an input field. As you type the name of the module, the ***setModuleName*** updates the module name, and clicking on the ***Add Module*** button calls the ***addModule*** function which actually adds the module.

app/(Kambaz)/Courses/[cid]/Modules/ModuleEditor.tsx

```
import { Modal, FormControl, Button } from "react-bootstrap";
export default function ModuleEditor({ show, handleClose, dialogTitle, moduleName, setModuleName, addModule,}: {
 show: boolean; handleClose: () => void; dialogTitle: string; moduleName: string; setModuleName: (name: string) => void;
 addModule: () => void; }) {
 return (
  <Modal show={show} onHide={handleClose}>
   <Modal.Header closeButton>
    <Modal.Title>{dialogTitle}</Modal.Title>
   </Modal.Header>
   <Modal.Body>
    <FormControl value={moduleName}
     onChange={(e) => { setModuleName(e.target.value); }} />
   </Modal.Body>
   <Modal.Footer>
    <Button variant="secondary" onClick={handleClose}> Cancel </Button>
    <Button variant="primary"
     onClick={() => {
      addModule();
      handleClose();
     }} > Add Module </Button>
   </Modal.Footer>
  </Modal>
);}
```

The ***+ Module*** button was implemented in the ***ModulesControls*** in a prior chapter. Let's refactor it so that it displays the ***ModuleEditor*** dialog when clicked.

app/(Kambaz)/Courses/[cid]/Modules/ModulesControls.tsx

```
import ModuleEditor from "./ModuleEditor";
export default function ModulesControls(
{ moduleName, setModuleName, addModule }:
{ moduleName: string; setModuleName: (title: string) => void; addModule: () => void; }) {
 const [show, setShow] = useState(false);
 const handleClose = () => setShow(false);
 const handleShow = () => setShow(true);
  return (
    <div id="wd-modules-controls" className="text-nowrap">
      <Button variant="danger" onClick={handleShow} >
        <FaPlus className="position-relative me-2" style={{ bottom: "1px" }} />
        Module
      </Button>
      ...
      <ModuleEditor show={show} handleClose={handleClose} dialogTitle="Add Module"
       moduleName={moduleName} setModuleName={setModuleName} addModule={addModule} />
    </div>
);}
```

In the ***Modules*** screen, declare a ***moduleName*** state variable that keeps track of the module name edited in the ***ModuleEditor*** dialog. The ***addModule*** function below creates a new module instance with the ***moduleName*** as the name and appends it to the end of the ***modules*** state variable. The ***setModuleName*** and ***addModule*** functions are passed to the ***ModulesControls*** component which will in turn pass them to the ***ModuleEditor*** dialog. The ***ModuleEditor*** dialog will invoke the ***setModuleName*** function when editing the module name in the dialog text field. The dialog will invoke the ***addModule*** function when the ***Add Module*** button is clicked. Confirm you can add modules.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { v4 as uuidv4 } from "uuid";
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  const [moduleName, setModuleName] = useState("");
  const addModule = () => {
```

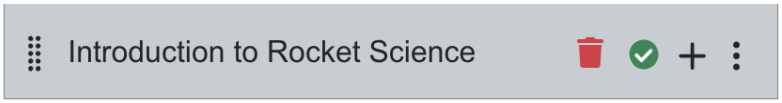
```
    setModules([ ...modules, { _id: uuidv4(), name: moduleName, course: cid, lessons: [] } ]);
    setModuleName("");
  };
  return (
    <div className="wd-modules">
      <ModulesControls setModuleName={setModuleName} moduleName={moduleName} addModule={addModule} />
      ...
    </div>
);}
```

4.4.3.2 Deleting a Module

To delete modules, let's add a ***trashcan*** icon to the ***ModuleControlButtons*** implement in an earlier chapter. We'll pass it a ***deleteModule*** we can call when clicking the ***trashcan*** and also pass the ID of the module to be deleted ***moduleId***.

app/(Kambaz)/Courses/[cid]/Modules/ModuleControlButtons.tsx

```
import { FaTrash } from "react-icons/fa";
export default function ModuleControlButtons(
{ moduleId, deleteModule }: { moduleId: string; deleteModule: (moduleId: string) => void; } ) {
  return (
    <div className="float-end">
      <FaTrash className="text-danger me-2 mb-1" onClick={() => deleteModule(moduleId)}/>
      <GreenCheckmark />
      <BsPlus className="fs-1" />
      <IoEllipsisVertical className="fs-4" />
    </div> );}
```



In the ***Modules*** screen, let's implement ***deleteModule*** function that removes a module by its ***ID*** and then pass the function to the ***ModuleControlButtons*** component along ***moduleId*** to be removed. Confirm you can remove modules.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
export default function Modules() {
  const addModule = () => { ... };
  const deleteModule = (moduleId: string) => {
    setModules(modules.filter((m) => m._id !== moduleId));
  };
  return (
    <div className="wd-modules">
      ...
      <ModuleControlButtons
        moduleId={module._id}
        deleteModule={deleteModule}/>
      ...
    </div>
  );
}
```

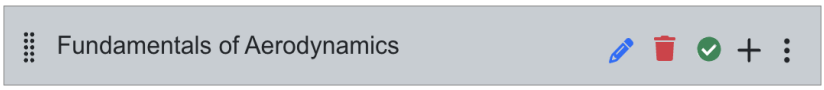
4.4.3.3 Editing a Module

In ***ModuleControlButtons***, add a pencil icon as shown below. Clicking the icon should call ***editModule*** with the ***moduleId*** of the module we want to edit.

app/(Kambaz)/Courses/[cid]/Modules/ModuleControlButtons.tsx

```
import { FaTrash } from "react-icons/fa";
import { FaPencil } from "react-icons/fa6";
export default function ModuleControlButtons({ moduleId, deleteModule, editModule }: {
  moduleId: string; deleteModule: (moduleId: string) => void;
```

```
  editModule: (moduleId: string) => void }) {
  return (
    <div className="float-end">
      <FaPencil onClick={() => editModule(moduleId)} className="text-primary me-3" />
      <FaTrash .../>
      <GreenCheckmark />
      <BsPlus className="fs-1" />
      <IoEllipsisVertical className="fs-4" />
    </div>
);}
```



In the ***Modules*** component, implement functions ***editModule*** and ***updateModule*** as shown below. Pass the ***editModule*** function to the ***ModuleControlsButtons*** component so that when the ***pencil*** icon is clicked, it will invoke the ***editModule*** to set the module's ***editing*** field to true. The ***updateModule*** accepts a ***module*** object and updates the corresponding ***module*** object in the ***modules*** array. If the ***module***'s ***editing*** field is not set (false), then the ***module***'s ***name*** is displayed. But if the ***pencil*** is clicked, the ***module***'s ***editing*** field is set to true and instead of the ***module***'s ***name***, an input field is displayed so the ***name*** can be edited using the ***updateModule*** function. If the ***Enter*** key is pressed, the ***module***'s ***editing*** field is set to false, and then the editing field is hidden and the ***module***'s name is shown again. Confirm you can edit the names of the modules.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
export default function Modules() {
  const { cid } = useParams();
  const [modules, setModules] = useState<any[]>(db.modules);
  const [moduleName, setModuleName] = useState("");
  const addModule = () => { ... }
  const deleteModule = (moduleId: string) => { ... }
  const editModule = (moduleId: string) => {                          // set the module's editing flag
    setModules(modules.map((m) => (m._id === moduleId ? { ...m, editing: true } : m)));  // to true so that we can display
  };                                                                  // the input field to edit name
  const updateModule = (module: any) => {                             // update any field(s) of a
    setModules(modules.map((m) => (m._id === module._id ? module : m)));  // module
  };
  return (
    <div className="wd-title p-3 ps-2 bg-secondary">
      <BsGripVertical className="me-2 fs-3" />
      {!module.editing && module.name}                                // show name if not editing
      { module.editing && (                                           // show input field if editing
        <FormControl className="w-50 d-inline-block"                  // when typing edit the module's
              onChange={(e) => updateModule({ ...module, name: e.target.value })}  // name
              onKeyDown={(e) => {                                     // if "Enter" key is
                if (e.key === "Enter") {                              // pressed then set editing
                  updateModule({ ...module, editing: false });        // field to false so we
                }                                                     // hide the text field
              }}
              defaultValue={module.name}/>
      )}
      <ModuleControlButtons
        moduleId={module._id}
        deleteModule={deleteModule}
        editModule={editModule}/>                                     // pass editModule function to
      </div>                                                          // so if pencil is clicked we can
    </div>                                                            // set editing to true
  );
}
```

4.4.3.4 Module Reducer

The ***Modules*** component seems to be working fine. We can create new modules, edit modules, and remove modules, BUT, it suffers a major flaw. Those new modules and edits can't be used outside the confines of the ***Modules*** component even though we would want to display the same list of modules elsewhere such as the ***Home*** screen. We could use the same

approach as we did for the ***Dashboard***, by moving the state variables and functions to a higher level component that could share the state with other components. Instead we're going to use ***Redux*** this time to practice application level state management. To start, create the ***reducer.tsx*** shown below containing the ***modules*** state variables as well as the ***addModule***, ***deleteModule***, ***updateModule***, and ***editModule*** functions reimplemented in the ***reducers*** property.

app/(Kambaz)/Courses/[cid]/Modules/reducer.ts

```
import { createSlice } from "@reduxjs/toolkit";                    // import createSlice
import { modules } from "../../Database";                          // import modules from database
import { v4 as uuidv4 } from "uuid";
const initialState = {                                             // create reducer's initial state with
  modules: modules,                                                // default modules copied from database
};
const modulesSlice = createSlice({                                 // create slice
  name: "modules",                                                 // name the slice
  initialState,                                                    // set initial state
  reducers: {                                                      // declare reducer functions
    addModule: (state, { payload: module }) => {                   // new module is in action.payload
      const newModule: any = {                                     // update modules in state adding new module
        _id: uuidv4(),                                             // at beginning of array. Override _id with
        lessons: [],                                               // timestamp
        name: module.name,
        course: module.course,
      };
      state.modules = [...state.modules, newModule] as any;
    },
    deleteModule: (state, { payload: moduleId }) => {              // module's ID to delete is in action.payload
      state.modules = state.modules.filter(                        // filter out module to delete
        (m: any) => m._id !== moduleId);
    },                                                             // module to update is in action.payload
    updateModule: (state, { payload: module }) => {                // replace module whose ID matches
      state.modules = state.modules.map((m: any) =>                // action.payload._id
        m._id === module._id ? module : m
      ) as any;
    },                                                             // select the module to edit
    editModule: (state, { payload: moduleId }) => {
      state.modules = state.modules.map((m: any) =>
        m._id === moduleId ? { ...m, editing: true } : m
      ) as any;
    },
  },
});                                                                // export all reducer functions
export const { addModule, deleteModule, updateModule, editModule } =
  modulesSlice.actions;                                            // export reducer
export default modulesSlice.reducer;
```

The reducers, ***store***, and ***Provider*** we worked on for the ***Labs*** only wrapped the lab exercises, so those won't be available here in ***Kambaz***. Instead, let's create a new ***store*** and ***Provider*** specific for the ***Kambaz*** application. Create a new store as shown below.

app/(Kambaz)/store.ts

```
import { configureStore } from "@reduxjs/toolkit";
import modulesReducer from "./Courses/Modules/reducer";
const store = configureStore({
  reducer: {
    modulesReducer,
  },
});
export default store;
```

Then provide the store to the whole React application as shown below.

src/App.tsx

```
import store from "./Kambaz/store";          // import the redux store
import { Provider } from "react-redux";      // import the redux store Provider
```

```
...
export default function App() {
  return (                                  // wrap your application with the Provider so all
    <HashRouter>                            // child elements can read and write to the store
      <Provider store={store}>
        <div>
          ...
        </div>
      </Provider>
    </HashRouter>
);}
```

Reimplement the ***Modules*** by removing the state variables and functions, and replacing them with selectors, dispatchers, and reducer functions as shown below. Confirm you can still add, remove, and edit modules as before. Also confirm the modules still work in the ***Home*** screen.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { addModule, editModule, updateModule, deleteModule }                  // import reducer functions to add,
  from "./reducer";                                                            // delete, and update modules
import { useSelector, useDispatch } from "react-redux";                        // import useSelector and useDispatch
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);       // retrieve modules state variables
  const dispatch = useDispatch();                                              // get dispatch to call reducer
  return (                                                                     // functions
    <div className="wd-modules">
      <ModulesControls moduleName={moduleName} setModuleName={setModuleName}
        addModule={() => {
          dispatch(addModule({ name: moduleName, course: cid }));              // wrap reducer functions with
          setModuleName("");                                                   // dispatch clear module name
        }} />
        <ListGroup id="wd-modules" className="rounded-0">
          {modules
            .filter((module: any) => module.course === cid)
            .map((module: any) => (
                  {!module.editing && module.name}
                  { module.editing && (
                    <FormControl className="w-50 d-inline-block"
                      onChange={(e) =>
                        dispatch(
                          updateModule({ ...module, name: e.target.value })    // wrap reducer functions with
                        )                                                      // dispatch
                      }
                      onKeyDown={(e) => {
                        if (e.key === "Enter") {
                          dispatch(updateModule({ ...module, editing: false }));
                        }                                                      // wrap reducer functions with
                      }}                                                       // dispatch
                      defaultValue={module.name} />
                  )}
                  <ModuleControlButtons moduleId={module._id}
                    deleteModule={(moduleId) => {
                      dispatch(deleteModule(moduleId));
                    }}
                    editModule={(moduleId) => dispatch(editModule(moduleId))} />    // wrap reducer
        ...                                                                        // functions with
      </ListGroup>                                                                 // dispatch
    </ListGroup>
  );
}
```

4.4.4 Account Screens

The ***Account Screens*** provide users access to their personal information and all related data such as courses they are enrolled in and courses they might be teaching. Users use the ***Sign In*** screen to identify themselves and access their

Profile screen to view their personal information. This section describes refactoring the ***Signin*** and ***Profile*** screens to confirm a user's identity and display their personal information.

4.4.4.1 Account Reducer

Implement an ***account reducer*** to keep track of the currently signed in user and share it across the entire application. Implement the ***account reducer*** as shown below and then add it to the ***Kambaz*** store.

app/(Kambaz)/Account/reducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
const initialState = {
  currentUser: null,
};
const accountSlice = createSlice({
  name: "account",
  initialState,
  reducers: {
    setCurrentUser: (state, action) => {
      state.currentUser = action.payload;
    },
  },
});
export const { setCurrentUser } = accountSlice.actions;
export default accountSlice.reducer;
```

app/(Kambaz)/store.ts

```
import { configureStore } from "@reduxjs/toolkit";
import modulesReducer from "./Courses/Modules/reducer";
import accountReducer from "./Account/reducer";
const store = configureStore({
  reducer: {
    modulesReducer,
    accountReducer,
  },
});
export default store;
```

4.4.4.2 Signin

Refactor the ***Signin*** screen by adding a ***credentials*** state variable for users to enter their credentials. When users click the ***Sign In*** button, search for a user with the credentials. If there's a user that matches, store it in the reducer by ***dispatch***ing it to the ***Account reducer*** using the ***setCurrentUser*** reducer function. Ignore the ***Sign In*** attempt if there's no match. After signing in, navigate to the ***Dashboard*** as shown below. Confirm that signing in navigates to the ***Dashboard*** only if valid credentials are used.

app/(Kambaz)/Account/Signin/page.tsx

```
import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { setCurrentUser } from "./reducer";
import { useDispatch } from "react-redux";
import * as db from "../Database";

export default function Signin() {
  const [credentials, setCredentials] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const signin = () => {
    const user = db.users.find(
      (u: any) => u.username === credentials.username && u.password === credentials.password);
    if (!user) return;
    dispatch(setCurrentUser(user));
    navigate("/Kambaz/Dashboard");
  };
  return (
    <div id="wd-signin-screen">
      <h1>Sign in</h1>
      <FormControl defaultValue={credentials.username}
             onChange={(e) => setCredentials({ ...credentials, username: e.target.value })}
             className="mb-2" placeholder="username" id="wd-username" />
      <FormControl defaultValue={credentials.password}
             onChange={(e) => setCredentials({ ...credentials, password: e.target.value })}
             className="mb-2" placeholder="password" type="password" id="wd-password" />
      <Button onClick={signin} id="wd-signin-btn" className="w-100" > Sign in </Button>
      <Link id="wd-signup-link" href="/Kambaz/Account/Signup"> Sign up </Link>
    </div>
```

```
);}
```

4.4.4.3 Dashboard

Now that the current user is stored in the ***Account reducer***, the ***Dashboard*** can filter the courses and only display the courses in which the current user is enrolled in. Refactor the ***Dashboard*** so that it only shows the courses the current user is enrolled in as shown below. Sign in as different users and confirm that the ***Dashboard*** only displays the courses a user is enrolled in. Note that new courses added will not render now since enrollments would also need to be modified. This will be addressed in later chapters.

app/(Kambaz)/Dashboard/page.tsx

```
import { useSelector } from "react-redux";
import * as db from "./Database";
...
export default function Dashboard(...) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const { enrollments } = db;
  return(
    ...
  {courses
    .filter((course) =>
      enrollments.some(
        (enrollment) =>
          enrollment.user === currentUser._id &&
          enrollment.course === course._id
        ))
    .map((course) => (
      <div className="wd-dashboard-course col" style={{ width: "300px" }} >
        ...
      </div>
  ))}
   ...
);}
```

4.4.4.4 Protecting Routes and Content

Now the ***Dashboard*** depends on a user being signed, the screen should only be accessible if the a user has signed in. Navigation to the ***Dashboard*** needs to be ***protected*** from users that are not signed in. Often applications have screens that are only accessible if users are logged in, usually because the information is sensitive and/or the information they are accessing is based on the identify of the user. We can protect navigating to certain routes in the user interface by checking if a user is signed in already or not and then either allowing access to the route, or navigating users to the sign in screen instead. The ***ProtectedRoute*** component below uses the ***currentUser*** in the store to determine whether there's someone signed in or not. The ***children*** parameter is a reference to the protected screen or component and if there's someone signed in, the ***ProtectedRoute*** returns the ***children*** reference allowing the signed in user to access the route. If no one is signed in, ***ProtedRoute*** navigates the user to the ***Sign in*** screen.

app/(Kambaz)/Account/ProtectedRoute.tsx

```
import { useSelector } from "react-redux";
import { Navigate } from "react-router-dom";
export default function ProtectedRoute({ children }: { children: any }) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  if (currentUser) {
    return children;
  } else {
    return <Navigate href="/Kambaz/Account/Signin" />;
}}
```

Use ***ProtectedRoute*** to protect the ***Dashboard*** and ***Courses*** routes so that users will only be able to navigate there if they are signed in. Confirm that the ***Dashboard*** and ***Courses*** screens are only accessible if the users are signed in.

app/(Kambaz)/page.tsx

```
<Routes>
  <Route path="/" element={<Navigate href="Dashboard" />} />
  <Route path="Account/*"       element={<Account />} />
  <Route path="Dashboard"       element={<ProtectedRoute><Dashboard ...             /></ProtectedRoute> } />
  <Route path="Courses/:cid/*" element={<ProtectedRoute><Courses courses={courses} /></ProtectedRoute> } />
  <Route path="Calendar"        element={<h1>Calendar</h1>} />
  <Route path="Inbox"           element={<h1>Inbox</h1>} />
</Routes>
```

On your own, use the the current user's role to only allow ***FACULTY*** to edit any content such as courses, modules, and assignments. If a user does not have the ***FACULTY*** role, hide all forms and buttons that would allow editing any content, e.g., ***New Course*** form, ***Add***, ***Delete***, ***Edit*** and ***Update*** buttons for ***Courses***, ***Modules***, and ***Assignments***, etc.

4.4.4.5 Account Navigation

Users can use the ***Account Navigation*** sidebar to navigate between the ***Account Screen***s ***Signin***, ***Signup***, and ***Profile***, but not all screens should be available if there's a current user or not. Reimplement the ***Account Navigation*** sidebar so that it hides the ***Signin*** and ***Signup*** navigation links if a user is already signed in, and hides the ***Profile*** link if a user is not yet signed in.

app/(Kambaz)/Account/Navigation.tsx

```
import { Link, useLocation } from "react-router-dom";
import { useSelector } from "react-redux";
export default function AccountNavigation() {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const links = currentUser ? ["Profile"] : ["Signin", "Signup"];
  const { pathname } = useLocation();
  ...
}
```

Also refactor the ***Account*** screen so that the default screen is ***Signin*** if no one is signed in yet, and ***Profile*** if someone is already signed in.

app/(Kambaz)/Account/page.tsx

```
import { useSelector } from "react-redux";
export default function Account() {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  ...
    <Routes>
      <Route path="/" element={<Navigate to={ currentUser ? "/Kambaz/Account/Profile" : "/Kambaz/Account/Signin" }/>}/>
      <Route path="/Signin" element={<Signin />} />
      <Route path="/Signup" element={<Signup />} />
      <Route path="/Profile" element={<Profile />} />
    </Routes>
  ...
);}
```

Confirm that the ***Account Navigation*** links are ***Sign In*** and ***Sign Up*** if no one is signed in yet, and ***Profile*** if someone is already signed in. Also confirm that clicking the ***Account*** link in the ***Kambaz Navigation*** sidebar displays the ***Signin*** screen if no one is signed in yet, and displays the ***Profile*** screen if someone is already signed in.

4.4.4.6 Profile

The ***Profile*** screen displays the current user's personal information. Refactor the ***Profile*** screen to retrieve the current user from the ***Account reducer***. If there's no ***currentUser***, then the screen should redirect to the ***Signin*** screen. If there's a

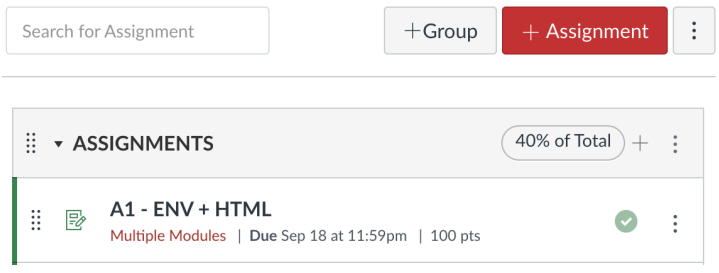
currentUser then the ***Profile*** screen should populate a form with the user's information. If the current user clicks a ***Sign Out*** button, then the current user should be nulled and navigate to the ***Signin*** screen. Refactor the ***Profile*** screen as shown below.

app/(Kambaz)/Account/Profile/page.tsx

```
import { Link, useNavigate } from "react-router-dom";
import { useState, useEffect } from "react";
import { useSelector, useDispatch } from "react-redux";
import { setCurrentUser } from "./reducer";
export default function Profile() {
  const [profile, setProfile] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const fetchProfile = () => {
    if (!currentUser) return navigate("/Kambaz/Account/Signin");
    setProfile(currentUser);
  };
  const signout = () => {
    dispatch(setCurrentUser(null));
    navigate("/Kambaz/Account/Signin");
  };
  useEffect(() => { fetchProfile(); }, []);
  return (
    <div className="wd-profile-screen">
      <h3>Profile</h3>
      {profile && (
        <div>
          <FormControl defaultValue={profile.username} id="wd-username" className="mb-2"
                       onChange={(e) => setProfile({ ...profile, username:  e.target.value })}/>
          <FormControl defaultValue={profile.password} id="wd-password" className="mb-2"
                       onChange={(e) => setProfile({ ...profile, password:  e.target.value })}/>
          <FormControl defaultValue={profile.firstName} id="wd-firstname" className="mb-2"
                       onChange={(e) => setProfile({ ...profile, firstName: e.target.value })}/>
          <FormControl defaultValue={profile.lastName} id="wd-lastname" className="mb-2"
                       onChange={(e) => setProfile({ ...profile, lastName:  e.target.value })}/>
          <FormControl defaultValue={profile.dob} id="wd-dob" className="mb-2"
                       onChange={(e) => setProfile({ ...profile, dob: e.target.value })} type="date"/>
          <FormControl defaultValue={profile.email} id="wd-email" className="mb-2"
                       onChange={ (e) => setProfile({ ...profile, email: e.target.value })}/>
          <select onChange={(e) => setProfile({ ...profile, role:  e.target.value })}
                  className="form-control mb-2" id="wd-role">
            <option value="USER">User</option>            <option value="ADMIN">Admin</option>
            <option value="FACULTY">Faculty</option>      <option value="STUDENT">Student</option>
          </select>
          <Button onClick={signout} className="w-100 mb-2" id="wd-signout-btn">
            Sign out
          </Button>
        </div>
      )}
</div>);}
```

4.4.5 Assignments (On Your Own)

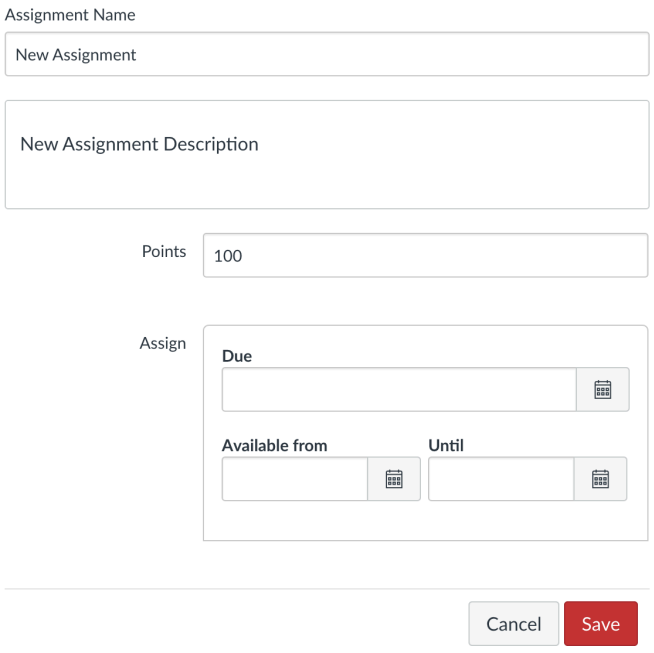
After completing the ***Dashboard***, ***Courses***, and ***Modules***, refactor the ***Assignments*** and ***AssignmentEditor*** screens so that faculty can create, update, and remove assignments as described in this section. Students can only view assignments.



4.4.5.1 Assignments Reducer

Following ***Modules/reducer.ts*** as an example, create an ***Assignments/reducer.ts*** in ***app/(Kambaz)/Courses/Assignments/*** initialized with ***db.assignments***. Implement reducer functions such as ***addAssignment***, ***deleteAssignment***, ***updateAssignment***, and any other functions as needed. Add the new reducer to the store in ***Kambaz/store/index.ts*** to add the assignments to the ***Kambaz*** application state.

4.4.5.2 Creating an Assignment



Refactor your ***Assignments*** screen as follows

- Clicking the ***+ Assignment*** button navigates to the ***AssignmentEditor*** screen
- The ***AssignmentEditor*** should allow editing at least the following fields: ***name***, ***description***, ***points***, ***due date***, ***available from date***, and ***available until date***.
- Clicking ***Save*** creates the new assignment and adds it to the ***assignments*** array state variable, navigates to the ***Assignments*** screen, which must now contain the newly created assignment.
- Clicking ***Cancel*** does not create the new assignment, and navigates back to the ***Assignments*** screen, without the new assignment.

4.4.5.3 Editing an Assignment

Refactor the ***AssignmentsEditor*** component as follows

- Clicking on an assignment in the ***Assignments*** screen navigates to the ***AssignmentsEditor*** screen, displaying the assignment's ***name***, ***description***, ***points***, ***due date***, ***available from date***, and ***available until date*** of the corresponding assignment.
- The ***AssignmentsEditor*** screen should allow editing the same fields listed earlier for corresponding assignment.
- Clicking ***Save*** updates the assignment's fields and navigates back to the ***Assignments*** screen with the updated assignment values.
- Clicking ***Cancel*** does not update the assignment, and navigates back to the ***Assignments*** screen which shows the assignments unchanged.

4.4.5.4 Deleting an Assignment

Refactor the ***Assignments*** component as follows

- Using the example of deleting modules, add a ***Delete*** button or ***trash*** icon to the right of each assignment.
- Clicking ***Delete*** on an assignment pops up a dialog asking if you are sure you want to remove the assignment.
- Clicking ***Yes*** or ***Ok***, dismisses the dialog, removes the assignment, and updates the ***Assignments*** screen without the deleted assignment.
- Clicking ***No*** or ***Cancel***, dismisses the dialog without removing the assignment

4.4.6 Refactor Dashboard and Courses (On Your Own)

The current ***Dashboard*** and ***Courses*** screens are implemented using ***useState*** to support various ***CRUD*** operations such as creating new courses, editing existing courses, and deleting courses. Using the implementation of the modules reducer as an example, reimplement the courses state management using ***Redux*** instead. Implement a ***courseReducer*** in ***app/(Kambaz)/Courses/reducer.ts*** which implements all needed methods to retrieve, add, edit, and delete courses in reducer functions such as ***addCourse***, ***deleteCourse***, and ***updateCourse***. Replace all uses of the state functions in

app/(Kambaz)/page.tsx, with the new reducer functions in ***courseReducer***. ***Kambaz/page.tsx***, ***Dashboard/page.tsx*** and ***Courses/page.tsx*** should not need to import ***Database/index.ts*** anymore. Reducer functions should only be imported and used within ***Dashboard/page.tsx***, and ***Courses/page.tsx***. Implement all necessary reducer functions needed to support all current course operations. Decide what state should be maintained in the reducer and what might make sense to be maintained as local state with ***useState***.

4.4.7 Enrollments (On Your Own)

Currently the ***Dashboard*** screen allows ***Faculty*** to ***Add***, ***Delete***, ***Edit***, and ***Update*** courses, as well as navigate to the course's content. Other types of the users can only view the list of the courses they are enrolled in. Refactor the ***Dashboard*** screen so that there's a new blue ***Enrollments*** button at the top right of the screen. Clicking the ***Enrollments*** button displays all the the courses. Clicking it again only shows the courses a user is enrolled in. Courses that the user is enrolled in should provide a red ***Unenroll*** button and courses that the user is not enrolled in should provide a green ***Enroll*** button. When a user clicks the ***Unenroll*** or ***Enroll*** button the enrollment status must actually change and the buttons should toggle to reflect the new state. If a user signs out, and then signs in again, the enrollment choices should still persist. If a user refreshes or reloads the page, the new enrollments are lost. Protect the route to a course so that only users enrolled in that course can navigate to the course, and stay in the ***Dashboard*** screen otherwise. Create new or modify existing reducers and store as needed.

4.5 Deliverables

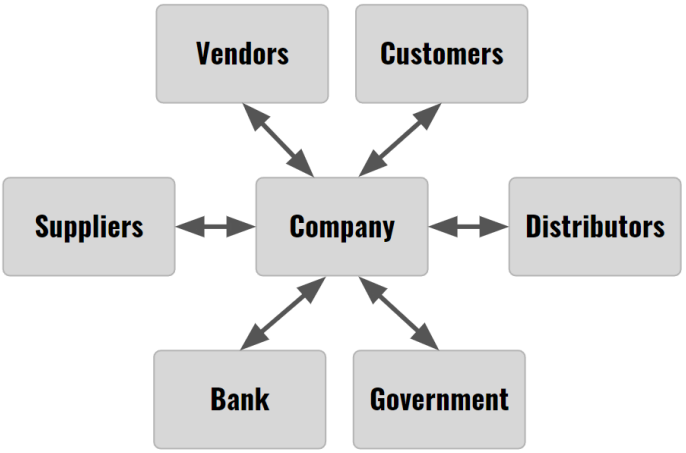
1. In the same React application created in earlier chapters, ***kambaz-next-js***, complete all the exercises described in this document.
2. In a branch called ***a4***, add, commit and push the source code of the React application ***kambaz-next-js*** to the same remote source repository in ***GitHub.com*** created in an earlier chapter. Here's an example of how to add, commit and push your code

```
$ git checkout -b a4
$ git add .
$ git commit -am "a4 Redux"
$ git push
```

3. Deploy the ***a4*** branch to the same ***Vercel*** project created in an earlier chapter. Configure Vercel to deploy all branches to separate URLs. From your Vercel's dashboard go to ***Site settings > Build & deploy > Branches > Branch deploys*** and select ***All***. Now each time you commit to a branch, the application will be available at a URL that contains the name of the branch
4. Make sure ***Labs/page.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kambaz. Add a link to your repository in GitHub.
5. In ***Labs/page.tsx***, add your full name: first name first, and last name second. Use the same name as in Canvas.
6. As a deliverable in ***Canvas***, submit the URL to the ***a4*** branch deployment of your React application running on Vercel.

Chapter 5 - Implementing RESTful Web APIs with Express.js

During the 90s, the adoption of the World Wide Web grew exponentially. A variety of commercial ventures explored numerous use cases, revolutionizing interactions between companies, their customers, and other businesses. The figure on the right illustrates several integration points between businesses, often referred to as ***business-to-business*** (***B2B***) interactions. Interactions between businesses and customers are commonly known as ***business-to-consumer*** (***B2C***) interactions. Many companies have largely automated customer interactions by implementing online storefronts where customers can browse products, place orders, submit reviews, and process returns.

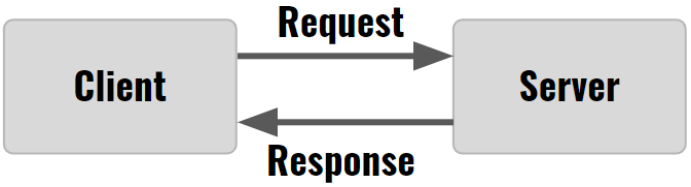


Creating visually appealing ***user interfaces*** is essential to capture customer attention, encourage purchases through marketing ads, and build long-term relationships through incentives like discounts and loyalty programs. User interfaces, as the name suggests, focus on application aspects that interact with users through visually engaging representations of data. Up to this point, these interfaces have used hard-coded JSON files, such as ***courses.json*** and ***modules.json***, to render data. Interfaces have been built to render and manipulate this data, updating the screen to reflect changes.

However, these updates have not been permanent; refreshing the browser results in lost changes and a reset application state. JavaScript applications running on clients like browsers, game consoles, or TV boxes have limited options for retrieving and storing data permanently. The next chapters address the challenges of retrieving, storing, updating, and deleting data permanently on remote servers and databases from React applications.

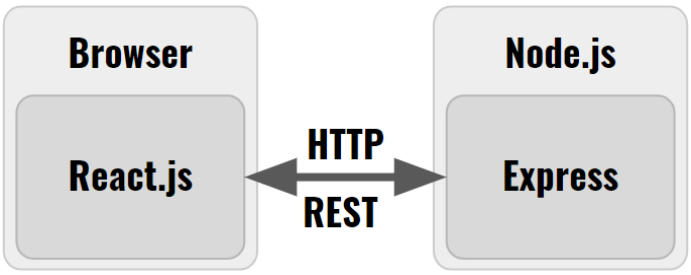
5.1 Installing and Configuring an HTTP Web Server

The Kambaz React Web application built so far is the ***client*** in a ***client/server architecture***. Users interact with client applications that implement user interfaces relying on servers to store data and execute complex logic that would be impractical on the client. Clients and servers interact through a sequence of requests and responses. Clients send ***requests*** to servers, servers execute some logic, fulfill the request, and then ***respond*** back to the client with results. This section discusses implementing HTTP servers using Node.js.



5.1.1 Introduction to Node.js

JavaScript is generally recognized as a programming language designed to execute in browsers, but it has been rescued from its browser confines by Node.js. Node.js is a JavaScript runtime to interpret and execute applications written in JavaScript outside of browsers, such as from a desktop console or terminal. This allows JavaScript applications written for the desktop to overcome many limitations faced by those in the browser. JavaScript running in a browser is restricted, with no access to the filesystem or databases, and limited network capabilities. In contrast, JavaScript running on a desktop has full access to the filesystem, databases, and unrestricted network access.



Conversely, desktop JavaScript applications generally lack a user interface and offer limited user interaction, while browser-based JavaScript applications provide rich and sophisticated interfaces for user interaction.

5.1.2 Installing Node.js

Node.js is a JavaScript runtime that can execute JavaScript on a desktop, allowing JavaScript programs to breakout from the confines and limitations of a browser. Node.js was installed during previous chapters while implementing the React Web application. Confirm the installation and check the version by typing the following in your computer terminal or console application.

```
$ node  -v
v22.11.0
```

If a Node installation is present, its version will be displayed on the console; otherwise, an error message will be shown, indicating that Node.js needs to be downloaded and installed from the URL below. As of this writing, Node.js 22.11.0 was the latest version, but any version recommended on Node's website can be installed.

```
https://nodejs.org/en
```

Once downloaded, double click on the downloaded file to execute the installer, give the operating system all the permissions it requests, accept all the defaults, let the installer complete, and restart the computer. Once the computer is up and running again, confirm Node.js installed properly by running the command ***node -v*** again from the command line.

5.1.3 Creating a Node.js Project

Another tool installed along with Node.js is ***npm*** or ***Node Package Manager*** which has been used thus far to run React applications in previous chapters. The ***npm*** command can also be used to install and execute Node.js ***packages*** on the local computer as well as creating brand new Node.js projects. To create a Node.js project create a directory with the name of the desired project and then change into that directory as shown below. Choose a directory name that does not contain any spaces, is all lowercase, and uses dashes between words.

```
$ mkdir  kambaz-node-server-app
$ cd     kambaz-node-server-app
```

NOTE: ***DO NOT*** create the Node.js project directory inside the existing React project directory. The React project should be in a directory called ***kambaz-next-js*** and the new ***kambaz-node-server-app*** directory should not be inside the React project directory. Instead, the two directories should be siblings, e.g., they should have the same parent directory.

Once in the Node.js project directory, use ***npm init*** to create a new Node.js project as shown below. This will kickoff an interactive session asking details about the project such as the name of the project and the author. The following is a sample interaction with sample answers. Each question provides a default answer which can be accepted or skipped by just pressing enter. It is fine to initially keep all the default values since they can be configured later.

```
$ npm  init
package name: (kambaz-node-server-app)
version: (1.0.0)
description: Node.js HTTP Web server for the Kambaz application
entry point: (index.js)
test command:
```

```
git repository: https://github.com/jannunzi/kambaz-node-server-app
keywords: Node, REST, Server, HTTP, Web Development, CS5610, CS4550
author: Jose Annunziato
license: (ISC)
```

The configuration will be written into a new file called ***package.json*** in a JSON format and it's distinctive of Node.js projects, like ***pom.xml*** files might be distinctive for Java projects.

5.1.4 Creating a Simple Hello World Node.js Program

Open the Node.js project directory created earlier with an IDE such as Visual Studio Code or IntelliJ, and at the root of the project, create a JavaScript file called ***Hello.js*** with the content shown below. The script uses the ***console.log()*** function to print the string ***'Hello World!'*** to the console and it is a common first program to write when learning a new language or infrastructure.

Hello.js

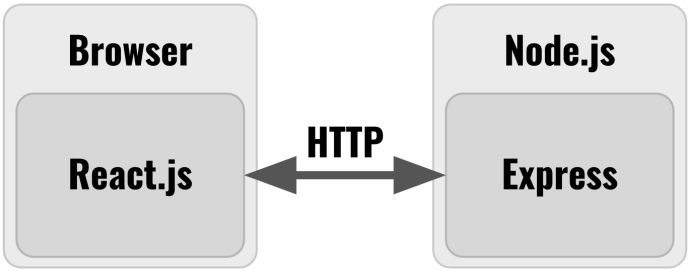
```
console.log("Hello World!");
```

At the command line, run the ***Hello.js*** application by using the ***node*** command and confirm the application prints ***Hello World!*** to the console as shown below.

```
$ node  Hello.js
Hello World!
```

Node.js programs consist of JavaScript files that are executed with the ***node*** command line interpreter. The following sections describe writing JavaScript applications that implement ***HTTP Web servers*** and ***RESTful Web APIs*** to integrate with React user interfaces. Upcoming chapters describe writing JavaScript applications that store and retrieve data from databases such as ***MongoDB***.

5.1.5 Creating a Node.js HTTP Web Server



Express is a very popular Node.js library that simplifies creating HTTP servers and Web APIs. Express HTTP servers can respond to HTTP requests from HTTP clients such as the React user interface implemented in earlier chapters. From the root directory of the Node.js project, install the ***express*** library from the terminal as shown below.

```
npm  install  express
```

Confirm that an ***express*** entry appears in ***package.json*** in the ***dependencies*** property. It is important these dependencies are listed in ***package.json*** so that they can be re-installed by other colleagues or when deploying to remote servers and cloud platforms such as ***AWS***, ***Heroku***, or ***Render.js***. New libraries are installed in a new folder called ***node_modules***. More Node.js packages can be found at npmjs.com. The following ***index.js*** implements an HTTP server that responds ***Hello World!*** when the server receives an HTTP request at the URL ***http://localhost:4000/hello***. Copy and paste the URL in a browser to send the HTTP request and the browser will render the response from the server. The ***require*** function is equivalent to the ***import*** keyword and loads a library into the local source. The ***express()*** function call creates an instance of the express library and assigns it to local constant ***app***. The ***app*** instance is used to configure the server on what to do when various types of requests are received. For instance the example below uses the ***app.get()*** function to configure an ***HTTP GET Request handler*** by mapping the URL pattern ***'/hello'*** to a function that handles the HTTP request.

index.js

```
const express = require('express')                          // equivalent to import
const app = express()                                       // create new express instance
app.get('/hello', (req, res) => {res.send('Hello World!')}) // create a route that responds 'hello'
app.listen(4000)                                            // listen to http://localhost:4000
```

A request to URL ***http://localhost:4000/hello*** triggers the function implemented in the second argument of ***app.get()***. The handler function receives parameters ***req*** and ***res*** which allows the function to participate in the ***request***/***response*** interaction, common in ***client***/***server*** applications. The ***res.send()*** function responds to the request with the text ***Hello World!*** Use ***node*** to run the server from the root of the project as shown below.

```
$ node  index.js
```

The application will run, start the server, and wait at port ***4000*** for incoming HTTP requests. Point your browser to ***http://localhost:4000/hello*** and confirm the server responds with ***Hello World!*** Stop the server by pressing ***Ctrl+C***. The string ***http://localhost:4000/hello*** is referred to as a ***URL*** (***Uniform Resource Locator***) and is used to .

5.1.6 Configuring Nodemon to Automatically Restart Node.js Server

React Web applications automatically transpile and restart every time code changes. Node.js can be configured to behave the same way by installing a tool called ***nodemon*** which monitors file changes and automatically restarts the Node application. Install nodemon globally (***-g***) as follows.

```
$ npm  install  nodemon  -g
```

On ***macOS*** you might need to run the command as a ***super user*** as follows. Type your password when prompted.

```
$ sudo  npm  install  nodemon  -g
```

Now instead of using the node command to start the server, use ***nodemon*** as follows:

```
$ nodemon  index.js
```

Confirm the server is still responding ***Hello World!***. Change the response string to ***Life is good!*** and without stopping and restarting the server, refresh the browser and confirm that the server now responds with the new string. To practice, create another endpoint mapped to the root of the application, e.g., ***"/"***. Navigate to ***http://localhost:4000*** with your browser and confirm the server responds with the message below.

index.js

```
const express = require('express')
const app = express()
app.get('/hello', (req, res) => {res.send('Life is good!')})   // http://Localhost:4000/hello responds "Life is good"
app.get('/', (req, res) => {                                   // http://Localhost:4000 responds "Welcome to Full ..."
  res.send('Welcome to Full Stack Development!')})
app.listen(4000)
```

5.1.7 Configuring Node.js to Use ES6

The React Web application created in earlier chapters has used the ***import*** keyword to load ES6 modules, but note that the ***index.js*** is using the ***require*** keyword instead to accomplish the same thing. Since Node version 12, ES6 syntax is

supported by configuring the ***package.json*** file and adding a new ***"type"*** property with value ***"module"*** as shown below in the highlighted text.

package.json

```
{
 "type": "module",                                          // type module turns on ES6
 "name": "kambaz-node-server-app",
 ...
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js"                                  // use npm start to start server
 },
```

Now, instead of using ***require()*** to load libraries, the familiar ***import*** statement can be used instead. Here's the ***index.js*** refactored to use ***import*** instead of ***require***. Restart the server, refresh the browser and confirm that the server responds as expected.

index.js S

```
import express from 'express';
const app = express();
app.get('/hello', (req, res) => {res.send('Life is good!')})
app.get('/', (req, res) => {res.send('Welcome to Full Stack Development!')})
app.listen(4000);
```

5.1.8 Creating HTTP Routes

The ***index.js*** file creates and configures an HTTP server listening for incoming HTTP requests. So far we've created a simple ***hello*** HTTP ***route*** that responds with a simple string. Throughout this and later chapters, we're going to create quite a few other HTTP routes, too many to define them all in ***index.js***. Instead, it is best practice to group routes into dedicated ***routing files*** that collect related HTTP routes. To illustrate this principle, move the routes defined in ***index.js*** to the ***Hello.js*** file created earlier as shown bellow.

index.js S

```
import express from 'express';
const app = express();
app.get('/hello', (req, res) => {                           // move this to Hello.js
  res.send('Life is good!')
});
app.get('/', (req, res) => {
  res.send('Welcome to Full Stack Development!')
});
app.listen(4000);
```

Copy the routes to ***Hello.js*** as shown below.

Hello.js S

```
console.log('Hello World!');                                // don't need this anymore
app.get('/hello', (req, res) => {                           // moved here from index.js
   res.send('Life is good!')
});
app.get('/', (req, res) => {
   res.send('Welcome to Full Stack Development!')
});
```

In our case ***Hello.js*** handles HTTP requests for a ***hello*** greeting and responds with a friendly reply. We're not done though. Notice that ***Hello.js*** references ***app*** which is undefined in the file. The ***app*** variable represents an express instance which

we would be tempted to create an instance in the file, but instead only one instance should be shared across all routes implemented per server instance. Instead of creating a new express instance, pass ***app*** as a parameter in a function we can import and invoke from ***index.js*** as shown below.

Hello.js

```
export default function Hello(app) {                    // function accepts app reference to express module
  app.get('/hello', (req, res) => {                     // to create routes here. We could have used the new
    res.send('Life is good!')                           // arrow function syntax instead
  })
  app.get('/', (req, res) => {
    res.send('Welcome to Full Stack Development!')
  })
}
```

Import ***Hello.js*** and pass app to the function as shown below. Note the ***.js*** extension in the import statement. Test ***http://localhost:4000/hello*** from the browser and confirm the reply is still friendly.

index.js

```
import express from 'express'
import Hello from "./Hello.js"                          // import Hello from Hello.js
const app = express()
Hello(app)                                              // pass app reference to Hello
app.listen(4000)
```

Although the ***Hello.js*** route implementation is perfectly fine, embedding the callback function declaration within the route definition can be a challenging syntax for some. An alternative (arguably better) syntax is to declare the callback functions on their own and then reference them in the route declarations as shown below.

Hello.js

```
export default function Hello(app) {
  const sayHello = (req, res) => {
    res.send("Life is good!");
  };
  const sayWelcome = (req, res) => {
    res.send("Welcome to Full Stack Development!");
  };
  app.get("/hello", sayHello);
  app.get("/", sayWelcome);
}
```

5.2 Lab Exercises

The following are a set of exercises to practice creating and integrating with an HTTP server from a React Web application. In your server application, create and import file ***Lab5/index.js*** where we will be implementing several server side exercises. Create a route that welcomes users to Lab 5.

Lab5/index.js

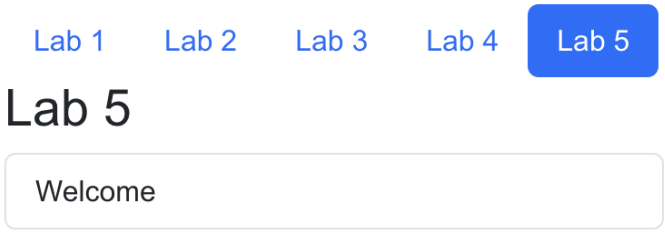
```
export default function Lab5(app) {                     // accept app reference to express module
  app.get("/lab5/welcome", (req, res) => {              // create route to welcome users to Lab 5.
    res.send("Welcome to Lab 5");                       // Here we are using the new arrow function syntax
  });
};
```

Import ***Lab5/index.js*** into the server ***index.js*** and pass it a reference to ***express*** as shown below. Restart the server and confirm that http://localhost:4000/lab5/welcome responds with the expected response.

index.js **S**

```
import Hello from "./Hello.js";
import Lab5 from "./Lab5/index.js";                    // import Lab5
const app = express();
Lab5(app);                                             // pass reference to express module
Hello(app);
```

In the React Web app project, create a ***Lab5*** React component to test the Node HTTP server. Import the new component into the existing set of labs and add a new link in ***TOC.tsx*** to navigate to ***Lab5*** by selecting the corresponding tab or link as shown here on the right. The example below creates a hyperlink that navigates to the http://localhost:4000/lab5/ welcome URL. Confirm the link navigates to the expected response.



app/Labs/Lab5/page.tsx **C**

```
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">                     // hyperlink navigates to
        <a href="http://localhost:4000/lab5/welcome"   // http://localhost:4000/lab5/welcome
           className="list-group-item">
           Welcome
        </a>
      </div><hr/>
    </div>
);}
```

5.2.1 Environment Variables

Currently we are integrating the React user interface with a Node server that are both running locally on our development computers, but ultimately these will both be running on remote servers. Let's configure the local environment so that it will be easy later on to configure our source code to run in any environment. There are two environments to consider: the ***local development environment*** and the ***remote production environment***. The ***local development environment*** consists of our computer where we do our development running two Node servers, one hosting the React user interface Web application, and the other hosting the Express HTTP server. The ***remote production environments*** will consist of the React user interface running on Vercel, and the Express HTTP server running on ***Render.com***, ***Heroku***, or ***AWS*** (your choice). Environments can be configured with ***environment variables*** declared in your ***Operating System*** or as ***environment files*** in your project. Environment files are named ***.env*** (with a leading period) and can be defined for each environment by appending ***.development*** for the local development environment, ***.test*** for the test environment, and ***.production*** for the production enviromenr. Instead of hardcoding ***http://localhost:4000*** everywhere in our source code, instead, declare an environment variable in the local environment file as shown below. Create the ***.env.development*** file at the root of your React project and copy the following content.

.env.development **C**

```
VITE_HTTP_SERVER=http://localhost:4000
```

In a React project created with the ***Vite*** tool, all environment variables must start with ***VITE_***. Each line declares a variable, followed by an equal sign, followed by the value of the variable. Make sure not to use extra spaces or unnecessary extra characters such as quotes, commas, or colons. Everytime a new variable is added, removed, or changes are made to an environment file, the React user interface Web application needs to be restarted. Environment variables can be accessed from the React source code through the global ***process*** object, in its ***env*** property. To instance, to access the value of the

VITE_HTTP_SERVER declared above, use ***process.env.VITE_HTTP_SERVER***. To practice declaring and using environment variables, create component ***EnvironmentVariables*** as shown below.

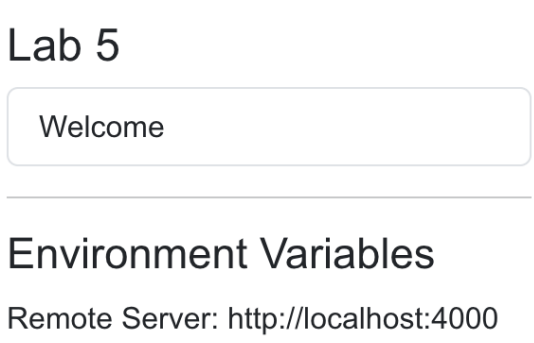
app/Labs/Lab5/EnvironmentVariables.tsx C

```
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function EnvironmentVariables() {
  return (
    <div id="wd-environment-variables">
      <h3>Environment Variables</h3>
      <p>Remote Server: {HTTP_SERVER}</p><hr/>
    </div>
);}
```

Import the component in component ***Lab5***, restart the React application and confirm the URL of the remove server displays as shown below.

app/Labs/Lab5/page.tsx C

```
import EnvironmentVariables from "./EnvironmentVariables";
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">
        <a href="http://localhost:4000/lab5/welcome"
           className="list-group-item">
           Welcome
        </a>
      </div><hr/>
      <EnvironmentVariables />
    </div>
);}
```



Make sure the URL to the remote server is never used in the React source code, instead prefer using the environment variable. Replace the ***http://localhost:4000*** in the previous exercise with the ***HTTP_SERVER*** constant as shown below. Confirm that the ***Welcome*** hyperlink still works.

app/Labs/Lab5/page.tsx C

```
import EnvironmentVariables from "./EnvironmentVariables";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function Lab5() {
  return (
    <div id="wd-lab5">
      <h2>Lab 5</h2>
      <div className="list-group">
        <a href={`${HTTP_SERVER}/lab5/welcome`} className="list-group-item">
          Welcome
        </a>
      </div><hr />
      <EnvironmentVariables />
    </div>
);}
```

Similarly, the Node Express server needs to be configure to run locally on your computer as well as when it is deployed in the remote environment. To do this, refactor ***index.js*** so that it uses the remote ***PORT*** environment variable if available, or port 4000 when running locally.

index.js S

```
import express from 'express'
```

```
import Hello from "./Hello.js"
import Lab5 from "./Lab5/index.js";
const app = express()
Lab5(app)
Hello(app)
app.listen(process.env.PORT || 4000)
```

5.2.2 Sending Data to a Server via HTTP Requests

Let's explore how we can integrate the React user interface with the Node server by sending information to the server from the browser. There are three ways to send information to the server:

1. ***Path parameters*** - parameters are encoded as segments of the path itself, e.g., ***http://localhost:4000/lab5/add/2/5***.
2. ***Query parameters*** - parameters are encoded as name value pairs in the query string after the ? character at the end of a URL, e.g., ***http://localhost:4000/lab5/add?a=2&b=5***.
3. ***Request body*** - data is sent as a string representation of data encoded in some format such as XML or JSON containing properties and their values, e.g., ***{a:2, b: 5}*** or ***<params a=2 b=5/>***.

We'll explore the first two in this section, and address the last one towards the end of the labs.

5.2.2.1 Sending Data to a Server with Path Parameters

React applications can pass data to servers by embedding it in a URL path as ***path parameters*** part of a URL. For instance the last two integers -- 2 and 4 -- at the end of the following URL can be parsed by a corresponding matching route on the server, add the two integers, and respond with the result of 6.

```
http://localhost:4000/lab5/add/2/4
```

The following route declarations can parse path parameters a and b encoded in paths ***/lab5/add/:a/:b*** and ***/lab5/subtract/:a/:b***. In ***PathParameters.js***, implement the routes below and import it in ***Lab5/index.js***. On your own create routes ***/lab5/multiply/:a/:b*** and ***lab5/divide/:a/:b*** that calculate the arithmetic multiplication and division.

Lab5/PathParameters.js ***S***

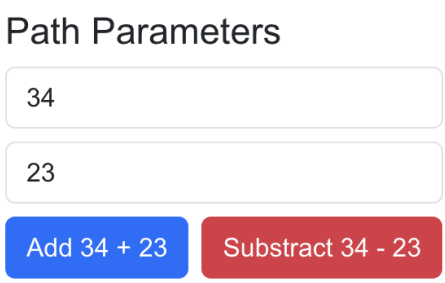
```
export default function PathParameters(app) {
 const add = (req, res) => {                    // route expects 2 path parameters after /lab5/add
   const { a, b } = req.params;                 // retrieve path parameters as strings
   const sum = parseInt(a) + parseInt(b);       // parse as integers and adds
   res.send(sum.toString());                    // sum as string sent back as response
 };                                             // don't send integers since can be interpreted as status
 const substract = (req, res) => {              // route expects 2 path parameters after /lab5/subtract
   const { a, b } = req.params;                 // retrieve path parameters as strings
   const sum = parseInt(a) - parseInt(b);       // parse as integers and subtracts
   res.send(sum.toString());                    // subtraction as string sent back as response
 };                                             // response is converted to string otherwise browser
 app.get("/lab5/add/:a/:b", add);               // would interpret integer response as a status code
 app.get("/lab5/subtract/:a/:b", substract);
};
```

Note when you import, make sure to include the extension ***.js*** as shown below. Pass a reference of ***app*** to the ***PathParameters*** function. Confirm that http://localhost:4000/lab5/add/6/4 responds with 10 and http://localhost:4000/lab5/subtract/6/4 responds with 2. Also confirm the routes you implemented on your own.

Lab5/index.js ***S***

```
import PathParameters from "./PathParameters.js";
```

```
export default function Lab5(app) {
  app.get("/lab5/welcome", (req, res) => {
    res.send("Welcome to Lab 5");
  });
  PathParameters(app);
}
```

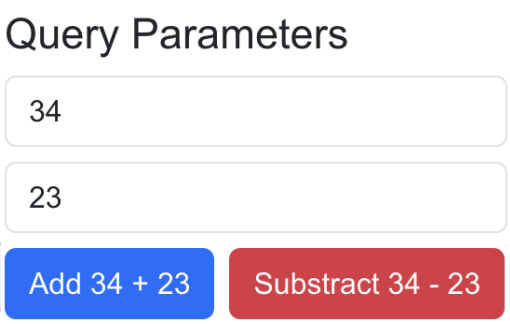


Meanwhile, in the React Web application, let's create a React component to test the new routes from our Web application. Web applications that interact with server applications are often referred to as ***client applications*** since they are the ***client*** in an application built using a ***client server architecture***. Create the component below that declares state variables ***a*** and ***b***, encodes the values in hyperlinks, and when you click them, server responds with the addition or subtraction of the parameters. Note that the name of the component is arbitrary. The fact that it is called the same as the routes in the server is a coincidence. Also helps us keep track of which UI components on the client are related to the server resources. On your own, create links that invoke the multiply and divide routes you implemented earlier. Import the new component in your ***Lab5*** component and confirm that clicking the links generates the expected response. Confirm all the routes work when you click the links in the browser.

app/Labs/Lab5/PathParameters.tsx *C*

```
import React, { useState } from "react";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function PathParameters() {
  const [a, setA] = useState("34");
  const [b, setB] = useState("23");
  return (
    <div>
      <h3>Path Parameters</h3>
      <FormControl className="mb-2" id="wd-path-parameter-a" type="number" defaultValue={a}
             onChange={(e) => setA(e.target.value)}/>
      <FormControl className="mb-2" id="wd-path-parameter-b" type="number" defaultValue={b}
             onChange={(e) => setB(e.target.value)}/>
      <a className="btn btn-primary me-2" id="wd-path-parameter-add"
         href={`${HTTP_SERVER}/lab5/add/${a}/${b}`}>
         Add {a} + {b}
      </a>
      <a className="btn btn-danger" id="wd-path-parameter-subtract"
         href={`${HTTP_SERVER}/lab5/subtract/${a}/${b}`}>
         Substract {a} - {b}
      </a>
      <hr />
    </div>
  );
}
```

5.2.2.2 Sending Data to a Server with Query Parameters



React applications can also send data to servers by encoding it as a ***query string*** after the ***question mark*** character (?) at the end of a URL. A ***query string*** consists of a list of name value pairs separated by the ***ampersand*** character (&) as shown below.

http://localhost:4000/lab5/calculator?operation=add&a=2&b=4

In ***Lab5/QueryParameters.js***, in your server application, create a route that can parse an ***operation*** and its parameters ***a*** and ***b*** as shown below. If the ***operation*** is ***add*** the route responds with the addition of the parameters. If the ***operation*** is ***subtract***, the route responds with the subtraction of the parameters. On your

own, also handle operations ***multiply*** and ***divide***. Import ***QueryParameters.js*** in ***Lab5/index.js*** and pass it a reference of ***app*** (not shown).

Lab5/QueryParameters.js **S**

```
export default function QueryParameters(app) {
 const calculator = (req, res) => {
   const { a, b, operation } = req.query;                    // retrieve a, b, and operation parameters in query
   let result = 0;
   switch (operation) {
     case "add":
       result = parseInt(a) + parseInt(b);                   // parse as integers since parameters are strings
       break;
     case "subtract":
       result = parseInt(a) - parseInt(b);
       break;
     // implement multiply and divide on your own
     default:
       result = "Invalid operation";
   }
   res.send(result.toString());                              // convert to string otherwise browser interprets
 };                                                          // as a status code
 app.get("/lab5/calculator", calculator);                    // e.g., lab5/calculator?a=5&b=2&operation=add
}
```

In a new component ***QueryParameters.tsx***, in your React Web application, create hyperlinks to test the new route as shown below. You'll need to declare ***const removeServer*** initialized to the environment variable ***VITE_HTTP_SERVER***. Confirm the following hyperlinks work as expected.

Test Hyperlinks	*Confirm Response*
http://localhost:4000/lab5/calculator?operation=add&a=34&b=23	57
http://localhost:4000/lab5/calculator?operation=subtract&a=34&b=23	11

app/Labs/Lab5/QueryParameters.tsx **C**

```
<div id="wd-query-parameters">
  <h3>Query Parameters</h3>
  <FormControl id="wd-query-parameter-a"
         className="mb-2"
         defaultValue={a} type="number"
         onChange={(e) => setA(e.target.value)} />
  <FormControl id="wd-query-parameter-b"
         className="mb-2"
         defaultValue={b} type="number"
         onChange={(e) => setB(e.target.value)} />
  <a id="wd-query-parameter-add"
     href={`${HTTP_SERVER}/lab5/calculator?operation=add&a=${a}&b=${b}`}>
    Add {a} + {b}
  </a>
  <a id="wd-query-parameter-subtract"
     href={`${HTTP_SERVER}/lab5/calculator?operation=subtract&a=${a}&b=${b}`}>
    Substract {a} - {b}
  </a>
  {/* create additional links to test multiply and divide. use IDs starting with wd-query-parameter- */}
  <hr />
</div>
```

5.2.2.3 On Your Own

On your own, remember to implement ***multiply*** and ***divide*** requests on the client and server that demonstrate multiplying and dividing numbers ***encoded in the request's path***. Now implement the same operations again, ***multiply*** and ***divide*** on the server and client that demonstrate, but multiplying and dividing parameters ***encoded in the query string***.

5.2.3 Working with Remote Objects on a Server

The examples so far have demonstrated working with integers and strings, but all primitive datatypes work as well, including objects and arrays. The example below declares an ***assignment*** object accessible at the route ***/lab5/assignment***. Import it to your ***Lab5/index.js*** in your server.

Lab5/WorkingWithObjects.js S

```
const assignment = {                                    // object state persists as long
  id: 1, title: "NodeJS Assignment",                    // as server is running
  description: "Create a NodeJS server with ExpressJS", // changes to the object persist
  due: "2021-10-10", completed: false, score: 0,        // rebooting server
};                                                      // resets the object
export default function WorkingWithObjects(app) {
  const getAssignment = (req, res) => {

    res.json(assignment);                               // use .json() instead of .send() if you know
  };                                                    // the response is formatted as JSON
  app.get("/lab5/assignment", getAssignment);
};
```

5.2.3.1 Retrieving Objects from a Server

In your React project, create a ***WorkingWithObjects*** component to test the new route as shown below. Import it in ***Lab5*** and confirm that http://localhost:4000/lab5/assignment responds with ***assignment*** object.

app/Labs/Lab5/WorkingWithObjects.tsx C

```
import React, { useState } from "react";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function WorkingWithObjects() {
  return (
    <div id="wd-working-with-objects">
      <h3>Working With Objects</h3>
      <h4>Retrieving Objects</h4>
      <a id="wd-retrieve-assignments" className="btn btn-primary"
         href={`${HTTP_SERVER}/lab5/assignment`}>
        Get Assignment
      </a><hr/>
    </div>
);}
```

Working With Objects

Retrieving Objects

Get Assignment

5.2.3.2 Retrieving Object Properties from a Server

We can retrieve individual properties in an object such as the ***title*** shown below.

Lab5/WorkingWithObjects.js S

```
export default function WorkingWithObjects(app) {
  const getAssignment = (req, res) => {
    res.json(assignment);
  };
  const getAssignmentTitle = (req, res) => {
    res.json(assignment.title);
  };
  app.get("/lab5/assignment/title", getAssignmentTitle);
  app.get("/lab5/assignment", getAssignment);
}
```

Retrieving Properties

Get Title

Confirm that the http://localhost:4000/lab5/assignment/title hyperlink below retrieves the assignment's title. In ***WorkingWithObjects***, add a link that retrieves the title as shown below. Confirm that clicking the link retrieves the assignment's title.

app/Labs/Lab5/WorkingWithObjects.tsx *C*

```
      ...
      <h4>Retrieving Objects</h4>
      <a id="wd-retrieve-assignments" className="btn btn-primary"
         href={`${HTTP_SERVER}/lab5/assignment`}>
        Get Assignment
      </a><hr/>
      <h4>Retrieving Properties</h4>
      <a id="wd-retrieve-assignment-title" className="btn btn-primary"
         href={`${HTTP_SERVER}/lab5/assignment/title`}>
        Get Title
      </a><hr/>
      ...
```

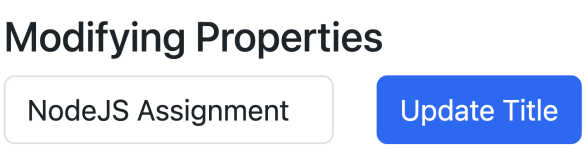
5.2.3.3 Modifying Objects in a Server

We can also use routes to modify objects or individual properties as shown below. The route retrieves the new title from the path and updates the ***assignment***'s object's ***title*** property.

Lab5/WorkingWithObjects.js *S*

```
const setAssignmentTitle = (req, res) => {
  const { newTitle } = req.params;                    // changes to objects in the server
  assignment.title = newTitle;                        // persist as long as the server is running
  res.json(assignment);                               // rebooting the server resets the object state
};
app.get("/lab5/assignment/title/:newTitle", setAssignmentTitle);
```

In ***WorkingWithObjects*** component in your React project, create an ***assignment*** state variable to test editing the ***assignment*** object on the server. Create an input field where we can type the new assignment title, and a link that invokes the route that updates the title. Confirm that you can change the assignment's title.



app/Labs/Lab5/WorkingWithObjects.tsx *C*

```
import React, { useState } from "react";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function WorkingWithObjects() {                  // create a state variable that holds
  const [assignment, setAssignment] = useState({                // default values for the form below.
    id: 1, title: "NodeJS Assignment",                          // eventually we'll fetch this initial
    description: "Create a NodeJS server with ExpressJS",       // data from the server and populate
    due: "2021-10-10", completed: false, score: 0,              // the form with the remote data so
  });                                                           // we can modify it here in the UI
  const ASSIGNMENT_API_URL = `${HTTP_SERVER}/lab5/assignment`
  return (
    <div>
      <h3 id="wd-working-with-objects">Working With Objects</h3>
      <h4>Modifying Properties</h4>                             // encode the title in the URL that
      <a id="wd-update-assignment-title"                        // updates the title
         className="btn btn-primary float-end"
         href={`${ASSIGNMENT_API_URL}/title/${assignment.title}`}>
        Update Title </a>
      <FormControl className="w-75" id="wd-assignment-title"    // form element to edit local state variable
        defaultValue={assignment.title} onChange={(e) =>        // used to encode in URL that updates
          setAssignment({ ...assignment, title: e.target.value })}/>  // property in remote object
      <hr />
      ...
    </div> );}
```

5.2.3.4 On Your Own

Now, on your own, create a ***module*** object with ***string*** properties ***id***, ***name***, ***description***, and ***course***. Feel free to use values of your choice.

- Create a route that responds with the ***module*** object, mapped to ***/lab5/module***
- In the UI, create a link labeled ***Get Module*** that retrieves the ***module*** object from the server mapped at ***/lab5/module***
- Confirm that clicking the link retrieves the module.
- Create another route mapped to ***/lab5/module/name*** that retrieves the ***name*** of the ***module*** created earlier
- In the UI, create a hyperlink labeled ***Get Module Name*** that retrieves the ***name*** of the ***module*** object
- Confirm that clicking the link retrieves the module's name.

On your own, in ***WorkingWithObjects.tsx***, create a ***module*** state variable to test editing the ***module*** object on the server. Create an input field where we can type the new module name, and a link that invokes the route that updates the name. Confirm that you can change the module's name. Create routes and a corresponding UI that can modify the ***score*** and ***completed*** properties of the ***assignment*** object. In the React application, create an input field of type ***number*** where you can type the new ***score*** and an input field of type ***checkbox*** where you can select the ***completed*** property. Create a link that updates the ***score*** and another link that updates the ***completed*** property. For the module, create routes and UI to edit the module's description.

5.2.4 Working with Remote Arrays on a Server

Now let's work with something a little more challenging. Create an array of objects and explore how to retrieve, add, remove, and update the array. Working with a collection of objects requires a general set of operations often referred to as ***CRUD*** or ***create***, ***read***, ***update***, and ***delete***. These operations capture common interactions with any collection of data such as ***creating*** and adding new instances to the collection, ***reading*** or retrieving items in a collection, ***updating*** or modifying items in a collection, and ***deleting*** or removing items from a collection.

5.2.4.1 Retrieving Arrays from a Server

Let's first create the array datastructure containing several todo objects. The most common integration between a client and server application is for a client application to retrieve all the instances of some collection. To illustrate this, create a route that retrieves the array of todo objects as shown below. Import the new route to ***Lab5/index.js***. Point the browser to http://localhost:4000/lab5/todos and confirm the server responds with the array of todos.

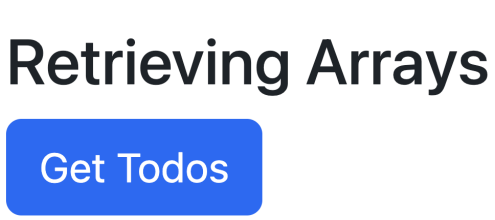
Lab5/WorkingWithArrays.js S

```
let todos = [
  { id: 1, title: "Task 1", completed: false },
  { id: 2, title: "Task 2", completed: true },
  { id: 3, title: "Task 3", completed: false },
  { id: 4, title: "Task 4", completed: true },
];
export default function WorkingWithArrays(app) {
  const getTodos = (req, res) => {
    res.json(todos);
  };
  app.get("/lab5/todos", getTodos);
};
```

In a new React component, create a hyperlink to test retrieving all todo objects in the array from the client. Add the new component to the ***Lab5*** complement and confirm that clicking the link retrieves the array.

app/Labs/Lab5/WorkingWithArrays.tsx C

```
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function WorkingWithArrays() {
  const API = `${HTTP_SERVER}/lab5/todos`;
  return (
    <div id="wd-working-with-arrays">
      <h3>Working with Arrays</h3>
      <h4>Retrieving Arrays</h4>
      <a id="wd-retrieve-todos" className="btn btn-primary" href={API}>
        Get Todos </a><hr/>
    </div>
);}
```



5.2.4.2 Retrieving Data From a Server by Primary Key

Another common operation is to retrieve a particular item from an array by its primary key, e.g., its ID property. The convention is to encode the ID of the item of interest as a path parameter. Note that we could have chosen to encode the ID as query parameter instead, but it is best practice to encode identifiers in the path instead. The example below parses the ID as a path parameter, finds the corresponding item, and responds with the item.

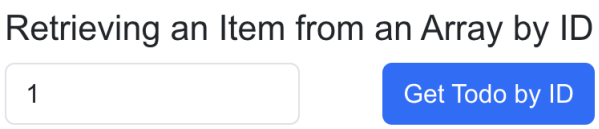
Lab5/WorkingWithArrays.js

```
  ...
  const getTodos = (req, res) => { ... }
  const getTodoById = (req, res) => {
    const { id } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    res.json(todo);
  };
  app.get("/lab5/todos", getTodos);
  app.get("/lab5/todos/:id", getTodoById);
  ...
```

Add a hyperlink to the React component to test retrieving an item from the array by its primary key. Confirm that you can type the ID in the UI and clicking the hyperlink retrieves the corresponding item.

app/Labs/Lab5/WorkingWithArrays.tsx

```
import React, { useState } from "react";
export default function WorkingWithArrays() {
      ...
      const [todo, setTodo] = useState({id: "1"});
      ...
      <h4>Retrieving Arrays</h4>
      <a id="wd-retrieve-todos" className="btn btn-primary" href={API}>
        Get Todos
      </a><hr/>
      <h4>Retrieving an Item from an Array by ID</h4>
      <a id="wd-retrieve-todo-by-id" className="btn btn-primary float-end" href={`${API}/${todo.id}`}>
        Get Todo by ID
      </a>
      <FormControl id="wd-todo-id" defaultValue={todo.id} className="w-50"
        onChange={(e) => setTodo({ ...todo, id: e.target.value })} />
      <hr />
      ...
```



5.2.4.3 Filtering Data From a Server With a Query String

The convention for retrieving a particular item from a collection is to encode the item's ID as a path parameter, e.g., ***/todos/123***. Another convention is that if the primary key is not provided, then the interpretation is that we want the entire collection of items, e.g., ***/todos***. We can also want to retrieve items by some other criteria other than the item's ID such as the item's ***title*** or ***completed*** properties. The best practice in this case is to use query strings instead of path parameters

when filtering items by properties other than the primary key, e.g., ***/todos?completed=true***. The example below refactors the ***/lab5/todos*** to handle the case when we want to filter the array by the ***completed*** query parameter.

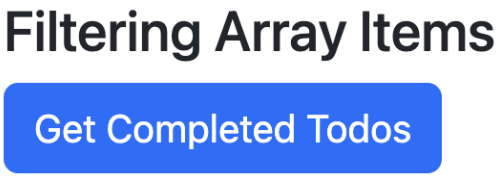
Lab5/WorkingWithArrays.js

```
const todos = [  { id: 1, title: "Task 1", completed: false },  { id: 2, title: "Task 2", completed: true },
                 { id: 3, title: "Task 3", completed: false },  { id: 4, title: "Task 4", completed: true },  ];
export default function WorkingWithArrays(app) {
  const getTodos = (req, res) => {
    const { completed } = req.query;
    if (completed !== undefined) {
      const completedBool = completed === "true";
      const completedTodos = todos.filter((t) => t.completed === completedBool);
      res.json(completedTodos);
      return;
    }
    res.json(todos);
  };
  const getTodoById = (req, res) => { ... }
  app.get("/lab5/todos", getTodos);
  app.get("/lab5/todos/:id", getTodoById);
};
```

Add a hyperlink to the React component to test retrieving all completed todos.

app/Labs/Lab5/WorkingWithArrays.tsx

```
  ...
  <h3>Retrieving Arrays</h3>
  ...
  <h3>Filtering Array Items</h3>
  <a id="wd-retrieve-completed-todos" className="btn btn-primary"
     href={`${API}?completed=true`}>
    Get Completed Todos
  </a><hr/>
  ...
```



5.2.4.4 Creating New Data in a Server

The examples we've seen so far have illustrated various ***read*** operations in our exploration of possible ***CRUD*** operations. Let's now take a look at the ***create*** operation. The example below demonstrates a route that creates a new item in the array and responds with the array now containing the new item. Note that it is implemented before the ***/lab5/todos/:id*** route, otherwise the ***:id*** path parameter would interpret the "***create***" in ***/lab5/todos/create*** as an ID, which would certainly create an error trying to parse it as an integer. Also note that the ***newTodo*** creates default values including a unique identifier field ***id*** based on a timestamp. Eventually primary keys will be handled by a database later in the course. Finally note that the response consists of the entire ***todos*** array, which is convenient for us for now, but a more common implementation would be to respond with ***newTodo***.

Lab5/WorkingWithArrays.js

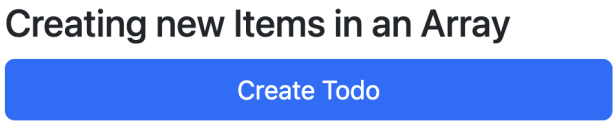
```
export default function WorkingWithArrays(app) {
  ...
  const createNewTodo = (req, res) => {
    const newTodo = {
      id: new Date().getTime(),
      title: "New Task",
      completed: false,
    };
    todos.push(newTodo);
    res.json(todos);
  };
  const getTodoById = (req, res) => { ... }
```

```
  app.get("/lab5/todos/create", createNewTodo);          // make sure to implement this BEFORE the /lab5/todos/:id
  app.get("/lab5/todos/:id", getTodoById);               // make sure to implement this route AFTER the
  ...                                                    // /lab5/todos/create route implemented above
```

Add a ***Create Todo*** hyperlink to the ***WorkingWithArrays*** component to test the new route. Confirm that clicking the link creates the new item in the array.

app/Labs/Lab5/WorkingWithArrays.tsx

```
...
<h3>Creating new Items in an Array</h3>
<a id="wd-retrieve-completed-todos" className="btn btn-primary"
   href={`${API}/create`}>
  Create Todo
</a><hr/>
...
```

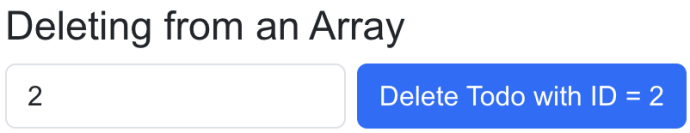


5.2.4.5 Deleting Data from a Server

Next let's consider the ***delete*** operation in the ***CRUD*** family of operations. The convention is to encode the ID of the item to delete as a path parameter as shown below. We search for the item in the set of items and remove it. Typically we would respond with a status of success or failure, but for now we're responding with all the todos for now.

Lab5/WorkingWithArrays.js

```
...
const getTodos = (req, res) => { ... }
const createNewTodo = (req, res) => { ... }
const getTodoById = (req, res) => { ... }
const removeTodo = (req, res) => {
  const { id } = req.params;
  const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
  todos.splice(todoIndex, 1);
  res.json(todos);
};
app.get("/lab5/todos/:id/delete", removeTodo);
app.get("/lab5/todos", getTodos);
app.get("/lab5/todos/create", createNewTodo);
app.get("/lab5/todos/:id", getTodoById);
...
```



To test the new route, create a link that encodes the todo's ID in a hyperlink to delete the corresponding item. We'll use the ***todo*** state variable created earlier to type the ID of the item we want to remove. Confirm that you can type the ID of an item, click the hyperlink and that the corresponding item is removed from the array.

app/Labs/Lab5/WorkingWithArrays.tsx

```
...
<h3>Removing from an Array</h3>
<a id="wd-remove-todo" className="btn btn-primary float-end" href={`${API}/${todo.id}/delete`}>
   Remove Todo with ID = {todo.id} </a>
<FormControl defaultValue={todo.id} className="w-50" onChange={(e) => setTodo({ ...todo, id: e.target.value })}/><hr/>
...
```

5.2.4.6 Updating Data on a Server

Finally let's consider the ***update*** operation in the ***CRUD*** family of operations. The convention is to encode the ID of the item to update as a path parameter as shown below. We search for the item in the set of items and update it. Typically we would respond with a status of success or failure, but for now we're responding with all the todos. Group the callback functions together towards the top of the routing file and the route declarations grouped towards the bottom of the file.

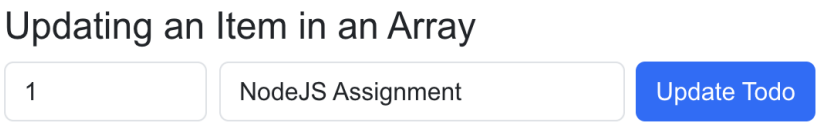
Lab5/WorkingWithArrays.js

```
  ...
  const updateTodoTitle = (req, res) => {
    const { id, title } = req.params;
    const todo = todos.find((t) => t.id === parseInt(id));
    todo.title = title;
    res.json(todos);
  };
  app.get("/lab5/todos/:id/title/:title", updateTodoTitle);
...
```

To test the new route, add an input field to edit the ***title*** property and a link that encodes both the ID of the item and the new value of the ***title*** property as shown below

app/Labs/Lab5/WorkingWithArrays.tsx

```
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export default function WorkingWithArrays() {
  const [todo, setTodo] = useState({
    id: "1",
    title: "NodeJS Assignment",
    description: "Create a NodeJS server with ExpressJS",
    due: "2021-09-09",
    completed: false,
  });
  return (
    <div>
      <h2>Working with Arrays</h2>
      ...
      <h3>Updating an Item in an Array</h3>
      <a href={`${API}/${todo.id}/title/${todo.title}`} className="btn btn-primary float-end">
        Update Todo</a>
      <FormControl defaultValue={todo.id} className="w-25 float-start me-2"
        onChange={(e) => setTodo({ ...todo, id: e.target.value })}/>
      <FormControl defaultValue={todo.title} className="w-50 float-start"
            onChange={(e) => setTodo({ ...todo, title: e.target.value }) }/>
      <br /><br /><hr />
      ...
```



5.2.4.7 On Your Own

Using the exercises so far as examples, implement routes and corresponding UI that allows editing a ***completed*** and ***description*** properties of ***todo*** items identified by their ID. Create the routes below in ***Lab5/index.js*** in the Node.js HTTP server project. In ***WorkingWithArrays*** component, add a text input field to edit the ***description*** and a checkbox input field to edit the ***completed*** property. Create a link that updates the ***description*** of the todo item whose ***id*** is encoded in the URL and another link that updates the ***completed*** property of the todo item whose ***id*** is encoded in the URL.

Property	*Routes*	*Response*	*Test*
completed	***/lab5/todos/:id/completed/:completed***	todos	Complete Todo ID = 1
description	***/lab5/todos/:id/description/:description***	todos	Describe Todo ID = 1

5.2.5 Asynchronous Communication with HTTP Servers

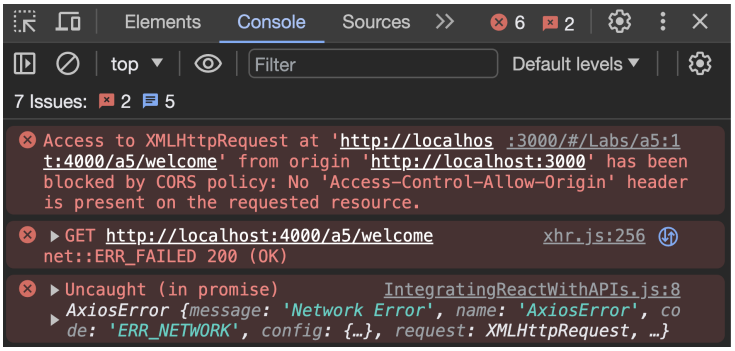
The exercises explored so far sent data encoded in the URL of hyperlinks. The links navigated to a separate browser window that displayed the server response. Even though we were able to send data to the server and affect changes to the server data, we have not considered how those changes can affect the user interface. Let's explore how to fully integrate the user interface with the server by sending and receiving HTTP requests and responses asynchronously.

5.2.5.1 Asynchronous JavaScript and XML

JavaScript Web applications, such as React applications, can communicate with server applications using a technology called ***AJAX*** or ***Asynchronous JavaScript and XML***. Using ***AJAX*** JavaScript applications can send and retrieve HTTP requests and response asynchronously from client JavaScript applications to remote HTTP server. Although ***XML*** was the original data format in ***AJAX***, ***JSON*** has overtaken as the dominant data format in modern Web applications, but the ***AJAX*** label still applies nevertheless. ***Axios*** is a popular JavaScript library that React user interface applications can use to communicate with servers using ***AJAX***. Install the library at the root of the React Web application project as shown below.

```
$ npm install axios
```

Let's use the same server routes implemented in earlier exercises, but instead of clicking on hyperlinks in the React client, we'll use ***axios*** to programmatically invoke the URLs giving us a chance to capture and handle the responses from the server and render the response in the user interface. The code below illustrates how to use the ***axios*** library to send an asynchronous request to the server and then capture the response in the user interface, without navigating to the URL, away from the current window. The ***fetchWelcomeOnClick*** function is tagged as ***async*** since it uses ***axios.get()*** to asynchronously send a request to the server, and returns the response from the server. Create the component below and import it in the ***Lab5*** component. Open the ***Web Dev Tools*** and confirm that clicking the ***Fetch Welcome*** button causes the ***CORS*** error shown here on the right.



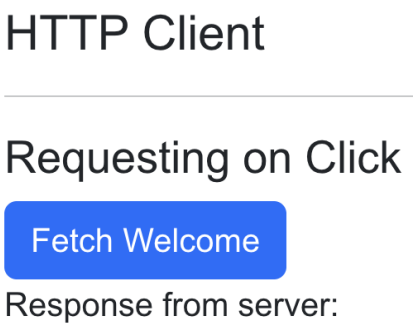
app/Labs/Lab5/HttpClient.tsx

```
import React, { useEffect, useState } from "react";
import axios from "axios";

const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;

export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");

  const fetchWelcomeOnClick = async () => {
    const response = await axios.get(`${HTTP_SERVER}/lab5/welcome`);
    setWelcomeOnClick(response.data);
  };
  return (
    <div>
      <h3>HTTP Client</h3> <hr />
      <h4>Requesting on Click</h4>
      <button className="btn btn-primary me-2" onClick={fetchWelcomeOnClick}>
        Fetch Welcome
      </button> <br />
      Response from server: <b>{welcomeOnClick}</b>
    </div>
  );
}
```



5.2.5.2 Configuring Cross Origin Request Sharing (CORS)

Servers and browsers limit ***JavaScript*** programs to only be able to communicate with the servers from where they are downloaded from. Since our ***React*** application is running locally from ***localhost:5173***, then they would only be able to communicate back to a server running on ***localhost:5173***, but our server is running on ***localhost:4000***, so when our ***JavaScript*** components try to communicate with ***localhost:4000***, the browser considers a different domain as a security

risk, stops the communication and throws a ***CORS*** exception. ***CORS*** stands for ***Cross Origin Request Sharing***, which governs the policies and mechanisms of how various resources can be shared across different domains or ***origins***. Browsers enforce ***CORS*** policies by first checking with the server if they are ok with receiving requests from different domains. If the server responds affirmatively, then browsers let the requests go through, otherwise they'll consider the attempt as a violation of ***CORS*** security policy, abort the request, and throw the exception. We can configure the ***CORS*** security policies by installing the ***cors*** Node.js library as shown below.

```
$ npm  install  cors
```

In ***index.js***, import the ***cors*** library and configure it as shown below to allow all requests from any origin. We'll narrow down this policy in a later chapter. Restart the server and refresh the React application. Confirm that the user interface is able to retrieve the ***Welcome to Lab 5*** message from the server without errors.

index.js

```
import express from "express";
import Lab5 from "./Lab5/index.js";
import cors from "cors";
const app = express();
app.use(cors());                    // make sure cors is used right after creating the app
Lab5(app);                          // express instance
app.listen(4000);
```

5.2.5.3 Creating a Client Library

The current ***HttpClient.tsx*** implementation makes a request to the server from the component itself using the ***axios*** library. In addition to ***retrieving*** (or ***reading***) data from the server, there will be other ***CRUD*** operations to ***create***, ***update***, and ***delete*** needed to interact with the server. Instead of implementing these in a React component, it's better to implement these in a reusable ***client library*** that can be shared across several user interface components. Move the ***axios.get()*** in ***HttpClient.tsx*** to a separate file called ***client.ts*** as shown below.

app/Labs/Lab5/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const fetchWelcomeMessage = async () => {
  const response = await axios.get(`${HTTP_SERVER}/lab5/welcome`);
  return response.data;
};
```

Now refactor ***HelloClient.ts*** to use the ***client.ts*** as shown below. Confirm that clicking on ***Fetch Welcome*** still works.

app/Labs/Lab5/HttpClient.tsx

```
import * as client from "./client";
export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");
  const fetchWelcomeOnClick = async () => {
    const message = await client.fetchWelcomeMessage();
    setWelcomeOnClick(message);
  };
  return ( ... );
}
```

5.2.5.4 Retrieving Data from a Server on Component Load

The previous exercise fetched data from the server when the user requested it. Often times we need to retrieve data from the server when you first navigate to a screen or a component is first loaded and displayed. Use React's ***useEffect*** hook function as shown below to invoke the ***fetchWelcomeOnLoad*** when a component or screen first loads. Now, when the ***HttpClient*** loads, the ***useEffect*** invokes ***fetchWelcomeOnLoad*** which retrieves the message from the server and sets the new ***welcomeOnLoad*** state variable. Confirm that if you refresh the screen, the ***welcome*** message appears without having to click on the ***Fetch Welcome*** button.

app/Labs/Lab5/HttpClient.tsx

```
import React, { useEffect, useState } from "react";
import * as client from "./client";
export default function HttpClient() {
  const [welcomeOnClick, setWelcomeOnClick] = useState("");
  const [welcomeOnLoad, setWelcomeOnLoad] = useState("");
  const fetchWelcomeOnClick = async () => {
    const message = await client.fetchWelcomeMessage();
    setWelcomeOnClick(message);
  };
  const fetchWelcomeOnLoad = async () => {
    const welcome = await client.fetchWelcomeMessage();
    setWelcomeOnLoad(welcome);
  };
  useEffect(() => {
    fetchWelcomeOnLoad();
  }, []);
  return (
    <div>
      <h3>HTTP Client</h3> <hr />
      <h4>Requesting on Click</h4>
      ...
      <hr />
      <h4>Requesting on Load</h4>
      Response from server: <b>{welcomeOnLoad}</b>
      <hr />
    </div>
);}
```

Requesting on Load

Response from server: **Welcome to Lab 5**

5.2.5.5 Working with Remote Objects on a Server Asynchronously

Let's now revisit the APIs that worked with the ***assignment*** object in ***WorkingWithObjects*** and create an asynchronous version. Let's add client functions to ***client.ts*** to fetch the assignment object from the server and update its title as shown below.

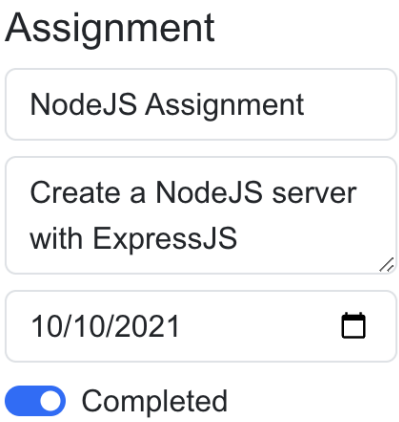
app/Labs/Lab5/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const fetchWelcomeMessage = async () => {
  const response = await axios.get(`${HTTP_SERVER}/lab5/welcome`);
  return response.data;
};
const ASSIGNMENT_API = `${HTTP_SERVER}/lab5/assignment`;
export const fetchAssignment = async () => {
  const response = await axios.get(`${ASSIGNMENT_API}`);
  return response.data;
};
export const updateTitle = async (title: string) => {
  const response = await axios.get(`${ASSIGNMENT_API}/title/${title}`);
  return response.data;
};
```

Then, in a new ***WorkingWithObjectsAsynchronously*** component, create a UI that fetches the assignment on load and then allows you to edit its title. Import the component in ***Lab5*** and confirm that the assignment is displayed on load.

app/Labs/Lab5/WorkingWithObjectsAsynchronously.tsx

```
import React, { useEffect, useState } from "react";
import * as client from "./client";
export default function WorkingWithObjectsAsynchronously() {
  const [assignment, setAssignment] = useState<any>({});
  const fetchAssignment = async () => {
    const assignment = await client.fetchAssignment();
    setAssignment(assignment);
  };
  useEffect(() => {
    fetchAssignment();
  }, []);
  return (
    <div id="wd-asynchronous-objects">
      <h3>Working with Objects Asynchronously</h3>
      <h4>Assignment</h4>
      <FormControl defaultValue={assignment.title} className="mb-2"
        onChange={(e) => setAssignment({ ...assignment, title: e.target.value }) } />
      <FormControl rows={3} defaultValue={assignment.description} className="mb-2"
        onChange={(e) => setAssignment({ ...assignment, description: e.target.value }) }/>
      <FormControl type="date" className="mb-2" defaultValue={assignment.due}
        onChange={(e) => setAssignment({ ...assignment, due: e.target.value })} />
      <div className="form-check form-switch">
        <input className="form-check-input" type="checkbox" id="wd-completed"
               defaultChecked={assignment.completed}
          onChange={(e) => setAssignment({ ...assignment, completed: e.target.checked }) } />
        <label className="form-check-label" htmlFor="wd-completed"> Completed </label>
      </div>
      <pre>{JSON.stringify(assignment, null, 2)}</pre>
      <hr />
    </div>
);}
```



Now let's add a button to update the assignment's title. Change the assignment's title, refresh the screen and confirm that the title has changed.

app/Labs/Lab5/WorkingWithObjectsAsynchronously.tsx

```
export default function WorkingWithObjectsAsynchronously() {
  const [assignment, setAssignment] = useState<any>({});
  const fetchAssignment = async () => {
    const assignment = await client.fetchAssignment();
    setAssignment(assignment);
  };
  const updateTitle = async (title: string) => {
    const updatedAssignment = await client.updateTitle(title);
    setAssignment(updatedAssignment);
  };
  ...
  return (
    <div id="wd-asynchronous-objects">
      <h3>Working with Objects Asynchronously</h3>
      <h4>Assignment</h4>
      ...
      <button className="btn btn-primary me-2" onClick={() => updateTitle(assignment.title)} >
        Update Title
      </button>
      <pre>{JSON.stringify(assignment, null, 2)}</pre>
      <hr />
    </div>
);}
```

5.2.5.6 Working with Remote Arrays on a Server Asynchronously

Now let's do the same thing to the arrays. Let's use ***axios*** so that we can manipulate remote arrays on the server from the user interface and update a user interface to reflect the changes in the remote array. We'll implement several client functions in ***client.ts*** that use ***axios*** to communicate with the server and we'll use them from a new component that will render the remote array in the use interface.

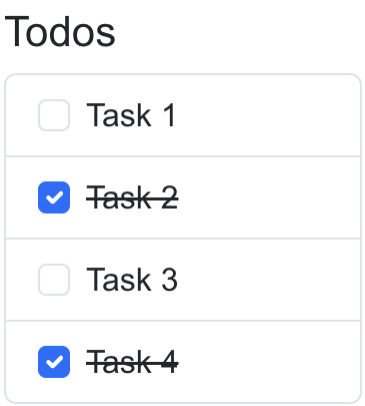
app/Labs/Lab5/client.ts

```
const TODOS_API = `${HTTP_SERVER}/lab5/todos`;
export const fetchTodos = async () => {
  const response = await axios.get(TODOS_API);
  return response.data;
};
```

The exercise below fetches the ***todos*** from the server and populate ***todos*** state variable which we can then render as a list of todos when the component loads. Confirm that the todos render when the component first loads.

app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import React, { useState, useEffect } from "react";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const fetchTodos = async () => {
    const todos = await client.fetchTodos();
    setTodos(todos);
  };
  useEffect(() => {
    fetchTodos();
  }, []);
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4>Todos</h4>
      <ListGroup>
        {todos.map((todo) => (
          <ListGroupItem key={todo.id}>
            <input type="checkbox" className="form-check-input me-2"
                   defaultChecked={todo.completed}/>
            <span style={{ textDecoration: todo.completed ? "line-through" : "none" }}>
              {todo.title} </span>
          </ListGroupItem>
        ))}
      </ListGroup> <hr />
    </div>
);}
```



5.2.5.7 Deleting Data from a Server Asynchronously

In ***client.ts***, add a ***removeTodo*** client function that sends a ***delete*** request to the server. The server will respond with an array with the surviving todos.

app/Labs/Lab5/client.ts

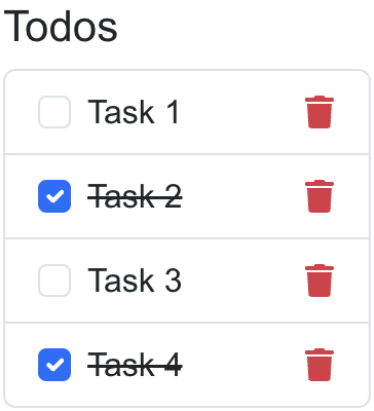
```
export const removeTodo = async (todo: any) => {
  const response = await axios.get(`${TODOS_API}/${todo.id}/delete`);
  return response.data;
};
```

In the ***WorkingWithArraysAsynchronously*** component, add ***remove*** buttons to each of the todos that invoke a new ***removeTodo*** function that uses the client to send asynchronous delete request to the server and updates the ***todos*** state

variable with the surviving todos. Use a ***trashcan*** icon to represent the ***remove*** button as shown below. Confirm that clicking on the new ***remove*** buttons actually removes the corresponding todo.

app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const fetchTodos = async () => { ... };
  const removeTodo = async (todo: any) => {
    const updatedTodos = await client.removeTodo(todo);
    setTodos(updatedTodos);
  };
  ...
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4>Todos</h4>
      <ListGroup>
        {todos.map((todo) => (
          <ListGroupItem key={todo.id}>
            <FaTrash onClick={() => removeTodo(todo)}
                     className="text-danger float-end mt-1" id="wd-remove-todo"/>
            ...
          </ListGroupItem>
        ))}
      </ListGroup><hr />
    </div>
);}
```



5.2.5.8 Creating New Data in a Server Asynchronously

A previous exercise implemented server route to create new todo items. In the React's project ***client.ts*** implement a ***createNewTodo*** client function that requests creating a new todo item from the server as shown below.

app/Labs/Lab5/client.ts

```
export const createNewTodo = async () => {
  const response = await axios.get(`${TODOS_API}/create`);
  return response.data;
};
```

In the ***WorkingWithArraysAsynchronously*** component, add a ***+ button icon*** to invoke the ***createNewTodo*** client function and update the ***todos*** state variables with the ***todos*** from the server. Confirm that clicking the ***+ button icon*** actually creates a new todo.

app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { FaPlusCircle } from "react-icons/fa";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const createNewTodo = async () => {
    const todos = await client.createNewTodo();
    setTodos(todos);
  };
  ...
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4> Todos <FaPlusCircle onClick={createNewTodo} className="text-success float-end fs-3" /> </h4>
      ...
    </div>
);}
```

5.2.6 Passing JSON Data to a Server in an HTTP Body

The exercises so far have sent data to the server as path and query parameters. This approach is limited to the maximum length of the URL string, and only string data types. Another concern is that data in the URL is sent over a network in clear text, so anyone snooping around between the client and server can see the data as plain text which is not a good option for exchanging sensitive information such as passwords and other personal data. A better approach is to encode the data as ***JSON*** in the ***HTTP request body*** which allows for arbitrarily large amounts of data as well as secure data encryption. To enable the server to parse ***JSON*** data from the ***request body***, add the following ***app.use()*** statement in ***index.js***. Make sure that it's implemented right after the ***CORS*** configuration statement. Now ***JSON*** data coming from the client is available in the ***request body*** in the ***request.body*** property in the server routes.

index.js

```
import Lab5 from "./Lab5/index.js";
import cors from "cors";
const app = express();
app.use(cors());
app.use(express.json());                                    // make sure this statement occurs AFTER
Lab5(app);                                                  // setting up CORS but BEFORE all the routes
app.listen(4000);
```

The hyperlinks and ***axios.get()*** in the exercises so far have sent data to the server using the ***HTTP GET method*** or ***verb***. HTTP defines several other ***HTTP methods*** or ***verbs*** including:

- ***GET*** - for retrieving data, but we've been also misusing it for creating, modifying and deleting data on the server. We'll start using it properly only for retrieving data
- ***POST*** - for creating new data typically embedded in the HTTP body
- ***PUT*** - for modifying existing data where updates are typically embedded in the HTTP body
- ***DELETE*** - for removing existing data
- ***OPTIONS*** - for retrieving allowed operations. Used to figure out if CORS policy allows communication with the other methods such as GET, POST, PUT and DELETE

The ***GET*** method, as the name suggests, is meant for only getting data from the server. We've been misusing it to implement routes that also creates, updates, and deletes data on the server. We did this mostly for academic purposes since it's the easiest HTTP method to work with. From now on we'll use the proper HTTP method for the right purpose.

5.2.6.1 Posting Data to Servers with HTTP POST Requests

To illustrate using the HTTP POST method, let's re-implement the route that creates new todos as shown below. The ***HTTP POST method*** takes the role of the verb meaning ***create***. Add the new ***app.post*** implementation highlighted in green below. Don't remove the old version highlighted in yellow so we don't break the other lab exercises. Note how the new implementation grabs the posted ***JSON*** data from ***req.body*** and uses it to define ***newTodo***. Also note that this version does not respond with the entire ***todos*** array and instead only responds with the newly created todo object instance. This is more reasonable since arrays can potentially be large and it would be expensive to transfer such large data structures over a network, especially if the client UI already has most of this data already displayed.

Lab5/WorkingWithArrays.js

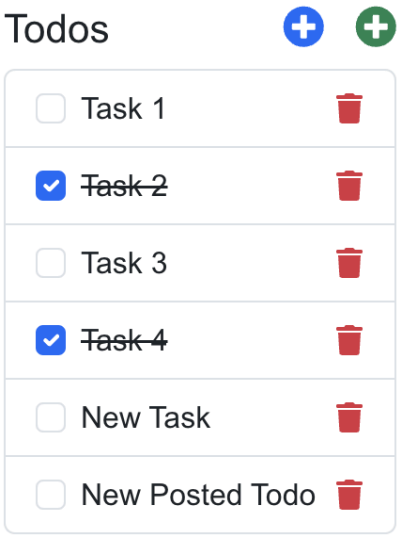
```
  ...
  const createNewTodo = (req, res) => {...}
  const postNewTodo = (req, res) => {
    const newTodo = { ...req.body, id: new Date().getTime() };
    todos.push(newTodo);
    res.json(newTodo);
  };
  ...
```

```
  app.get("/lab5/todos/create", createNewTodo);
  app.post("/lab5/todos", postNewTodo);
  ...
```

Back in the user interface, add a new ***postNewTodo*** client function in ***client.ts*** that posts new ***todo*** objects to the server as shown below. Note the second argument in the ***axios.post()*** method containing the new ***todo*** object instance sent to the server. The response this time contains the ***todo*** instance added to the ***todos*** array in the server instead of all the todos on the server.

app/Labs/Lab5/client.ts

```
export const createNewTodo = async () => {
  const response = await axios.get(`${TODOS_API}/create`);
  return response.data;
};
export const postNewTodo = async (todo: any) => {
  const response = await axios.post(`${TODOS_API}`, todo);
  return response.data;
};
```



In the ***WorkingWithArraysAsynchronously*** component, create a new ***+ button icon*** to invoke the new ***postNewTodo*** client function to send a new ***todo*** object to the server containing a default ***title*** and ***completed*** properties. Append the new ***todo*** object created on the server, to the local ***todos*** state variable to update the user interface with the new todo as shown below. Color the new ***+ button icon*** a different color so it's distinguishable from the ***createNewTodo*** button. Confirm that you can add new items to the array when you click on the new ***+ button icon***.

app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { FaPlusCircle } from "react-icons/fa";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const createNewTodo = async () => {
    const todos = await client.createNewTodo();
    setTodos(todos);
  };
  const postNewTodo = async () => {
    const newTodo = await client.postNewTodo({ title: "New Posted Todo", completed: false, });
    setTodos([...todos, newTodo]);
  };
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <h4>
        Todos
        <FaPlusCircle onClick={createNewTodo} className="text-success float-end fs-3"      id="wd-create-todo" />
        <FaPlusCircle onClick={postNewTodo}   className="text-primary float-end fs-3 me-3" id="wd-post-todo"   />
      </h4>
    </div>
);}
```

5.2.6.2 Deleting Data from Servers with HTTP DELETE Requests

Now that we have ***axios*** we can implement a better version of the remove operation. The current ***removeTodo*** implementation uses the ***HTTP GET*** method to request the server to remove data. The ***HTTP DELETE*** method is specifically suited for removing data from remote servers. In the server project, implement a better version of the delete operation as shown below. The new implementation uses the ***HTTP DELETE*** method declared in ***app.delete()*** which is distinct from ***app.get()*** for which we don't need the trailing ***/delete*** at the end of the ***URL***. Removing the element from the

array is the same either way. Although we could again respond with the entire array of surviving todos, it is better to just respond with a success status and let the user interface update its state variable. This reduces unnecessary data communication between the client and server.

Lab5/WorkingWithArrays.js

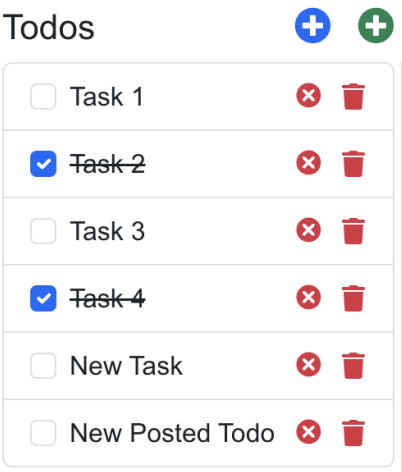
```
  ...
  const removeTodo = (req, res) => {
  const deleteTodo = (req, res) => {
    const { id } = req.params;
    const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
    todos.splice(todoIndex, 1);
    res.sendStatus(200);
  };
  app.delete("/lab5/todos/:id", deleteTodo);
  app.get("/lab5/todos/:id/delete", removeTodo);
  ...
```

In the React project create a new client function called ***deleteTodo*** as shown below. Note how it's implemented using ***axios.delete*** instead of ***axios.get*** so that it matches the server's ***app.delete*** as well as the ***URL*** format without the trailing ***/delete***.

app/Labs/Lab5/client.ts

```
export const removeTodo = async (todo: any) => {
  const response = await axios.get(`${TODOS_API}/${todo.id}/delete`);
  return response.data;
};
export const deleteTodo = async (todo: any) => {
  const response = await axios.delete(`${TODOS_API}/${todo.id}`);
  return response.data;
};
```

In the user interface component ***WorkingWithArraysAsynchronously***, add another ***delete*** button to try this new ***deleteTodo*** client function, but use a different icon, say an ***X*** so as to not confuse it with the ***trash***. Note that the new implementation ignores the response from the server and instead filters the removed todo from the local state variable. This is fine for now, but the operation is too optimistic assuming the server successfully deleted the item from the array and updating the user interface without confirmation. Later we'll deal with errors from the server to make sure the local state variable in the user interface is in synch with the remote array on the server.



app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { TiDelete } from "react-icons/ti";
import * as client from "./client";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  ...
  const deleteTodo = async (todo: any) => {
    await client.deleteTodo(todo);
    const newTodos = todos.filter((t) => t.id !== todo.id);
    setTodos(newTodos);
  };
  ...
  return (
    ...
    <ListGroupItem key={todo.id}>
      <FaTrash onClick={() => removeTodo(todo)}  className="text-danger float-end mt-1"      id="wd-remove-todo" />
      <TiDelete onClick={() => deleteTodo(todo)} className="text-danger float-end me-2 fs-3" id="wd-delete-todo" />
    </ListGroupItem>
    ...
);}
```

5.2.6.3 Updating Data on Servers with HTTP PUT Requests

Use ***HTTP PUT*** to reimplement the route that updates an item in an array as shown below. The route replaces the todo item whose ID matches the ***id*** path parameter with a combination of the original todo object and properties in the ***req.body***. This overrides any properties in the original todo object with matching properties in the ***req.body***. Note that this new implementation does not respond with the ***todos*** array, but instead responds with a simple ***OK*** status code of ***200***. This is more reasonable since there is no need to respond with an entire array since the user interface already has the array cached in the browser and it can just update the item in the ***todos*** state variable.

Lab5/WorkingWithArrays.js

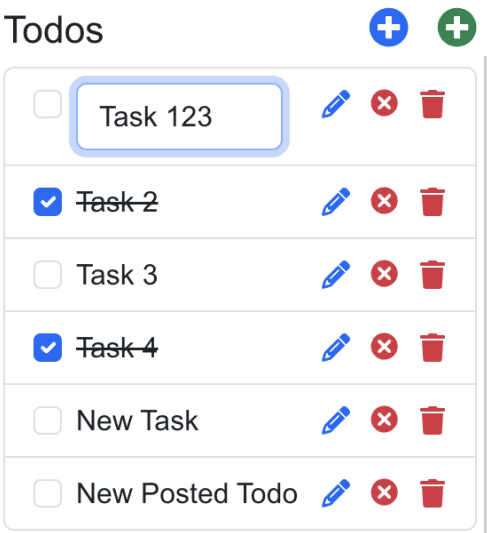
```
...
const updateTodo = (req, res) => {
  const { id } = req.params;
  todos = todos.map((t) => {
    if (t.id === parseInt(id)) {
      return { ...t, ...req.body };
    }
    return t;
  });
  res.sendStatus(200);
};
app.put("/lab5/todos/:id", updateTodo);
...
```

Back in the user interface, in ***clients.ts*** add a new ***updateTodo*** function that ***puts*** updates to the server as shown below. Note the second argument in the ***axios.put()*** method containing the updated ***todo*** object instance sent to the server. The response contains a status.

app/Labs/Lab5/client.ts

```
export const updateTodo = async (todo: any) => {
  const response = await axios.put(`${TODOS_API}/${todo.id}`, todo);
  return response.data;
};
```

In the ***WorkingWithArraysAsynchronously*** component, add an input field that shows up when you click a new ***pencil icon*** by setting the ***todo***'s ***editing*** property to true. Pressing the ***Enter*** key sets the ***todo***'s ***editing*** property to false, and shows the updated ***title*** again. Add an ***onChange*** attribute to the ***completed*** checkbox so that it updates the corresponding property of the ***todo*** object. Confirm you can edit the ***title*** and ***completed*** properties of the ***todos***, and that the changes persist after refreshing the page.



app/Labs/Lab5/WorkingWithArraysAsynchronously.tsx

```
import { FaPencil } from "react-icons/fa6";
export default function WorkingWithArraysAsynchronously() {
  const [todos, setTodos] = useState<any[]>([]);
  const editTodo = (todo: any) => {
    const updatedTodos = todos.map(
      (t) => t.id === todo.id ? { ...todo, editing: true } : t );
    setTodos(updatedTodos);
  };
  const updateTodo = async (todo: any) => {
    await client.updateTodo(todo);
    setTodos(todos.map((t) => (t.id === todo.id ? todo : t)));
  };
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      <ListGroup>
        {todos.map((todo) => (
```

```
          <ListGroupItem key={todo.id}>
            <FaPencil onClick={() => editTodo(todo)} className="text-primary float-end me-2 mt-1" />
            <input type="checkbox" defaultChecked={todo.completed} className="form-check-input me-2 float-start"
              onChange={(e) => updateTodo({ ...todo, completed: e.target.checked }) } />
              {!todo.editing ? ( todo.title ) : (
                <FormControl className="w-50 float-start" defaultValue={todo.title}
                  onKeyDown={(e) => {
                    if (e.key === "Enter") {
                      updateTodo({ ...todo, editing: false });
                    }
                  }}
                  onChange={(e) =>
                    updateTodo({ ...todo, title: e.target.value })
                  }
                />
              )}
    ...
);}
```

5.2.6.4 Handling Errors

The exercises so far have been very optimistic when interacting with the server, but it is good practice to handle edge cases and the unforeseen. In this section we're going to add error handling to some of the routes and user interface. For instance, the exercise below throws exceptions if the items being deleted or updated don't actually exist. Errors are reported by the server as status codes, where 404 is the infamous NOT FOUND error. Additionally a JSON object can be sent back as part of the response that can be used by user interfaces to better inform the user of what went wrong.

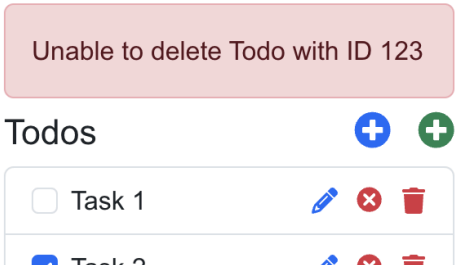
Lab5/WorkingWithArrays.js

```
  const deleteTodo = (req, res) => {
    const { id } = req.params;
    const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
    if (todoIndex === -1) {
      res.status(404).json({ message: `Unable to delete Todo with ID ${id}` });
      return;
    }
    todos.splice(todoIndex, 1);
    res.sendStatus(200);
  });
  const updateTodo = (req, res) => {
    const { id } = req.params;
    const todoIndex = todos.findIndex((t) => t.id === parseInt(id));
    if (todoIndex === -1) {
      res.status(404).json({ message: `Unable to update Todo with ID ${id}` });
      return;
    }
    todos = todos.map((t) => { ... });
    res.sendStatus(200);
  }
```

In the user interface we can ***catch*** the errors by wrapping the request in a ***try***/***catch*** clause as shown below. If the request fails with a HTTP error response, then the body of the ***try*** block is aborted and the body of the ***catch*** clause executes instead. The exercise below declares a ***errorMessage*** state variable that we populate with the error from the server if an error occurs. The error is rendered as an alert box as shown here on the right. To test, remove an item using the http://localhost:4000/lab5/todos/:id/delete route and then try to update or delete the same item using the user interface. Confirm you get an error if you try to delete or update a ***todo*** that does not exist.

app/Labs/Lab5/WorkingWithArrays.tsx

```
export default function WorkingWithArraysAsynchronously() {
  const [errorMessage, setErrorMessage] = useState(null);
  const updateTodo = async (todo: any) => {
    try {
      await client.updateTodo(todo);
```



```
      setTodos(todos.map((t) => (t.id === todo.id ? todo : t)));
    } catch (error: any) {
      setErrorMessage(error.response.data.message);
    }
  };
  const deleteTodo = async (todo: any) => {
    try {
      await client.deleteTodo(todo);
      const newTodos = todos.filter((t) => t.id !== todo.id);
      setTodos(newTodos);
    } catch (error: any) {
      console.log(error);
      setErrorMessage(error.response.data.message);
    }  };  ...
  return (
    <div id="wd-asynchronous-arrays">
      <h3>Working with Arrays Asynchronously</h3>
      {errorMessage && (<div id="wd-todo-error-message" className="alert alert-danger mb-2 mt-2">{errorMessage}</div>)}
      ...
    </div>
);}
```

5.3 Implementing the Kambaz Node.js HTTP Server

Kambaz is currently implemented entirely as a React application. Although various CRUD operations have been implemented to create, read, update, and delete courses and modules, these changes are not permanent and are lost when the browser is refreshed. To make the changes permanent, it is necessary to integrate the React user interface with a server that can access resources such as the file system, network, and database. In this section, server routes will be implemented to integrate the user interface with the server. In the next chapter, changes will be stored permanently in a MongoDB non-relational database.

5.3.1 Migrating the "Database" to the Server

Previous chapters, declared a ***Database*** component to consolidate all data files into a single access point. Ideally, the data should reside on the server side or within a dedicated database. In this chapter, the data will first be moved to the server, with a transition to a database in the subsequent chapter. Begin by creating a folder named ***Kambaz*** at the root of the Node.js project. Inside the ***Kambaz*** folder, create a ***Database*** directory and copy all JSON files from the React project. Then, change the file extensions of the JSON files to JavaScript, for example, rename ***users.json*** to ***users.js*** and ***courses.json*** to ***courses.js***. At the top of each newly converted JavaScript file, include an export default statement, as shown below.

Kambaz/Database/courses.js

```
export default [
  { _id: "RS101", name: "Rocket Propulsion", number: "RS4550", startDate: "2023-01-10",
    endDate: "2023-05-15", department: "D123", credits: 4, description: "..." }, ...
];
```

Do the same for all the JSON files and update the import statements in the ***Database*** component as shown below. Use the same data files from previous chapters. Feel free to modify the data in the files to customize the content or meet requirements in this chapter. Ignore unnecessary data files.

Kambaz/Database/index.js

```
import courses from "./courses.js";
import modules from "./modules.js";
import assignments from "./assignments.js";
import users from "./users.js";
```

```
import grades from "./grades.js";
import enrollments from "./enrollments.js";
export default { courses, modules, assignments, users, grades, enrollments };
```

5.3.2 Integrating the Account Screens with the Server with RESTful Web APIs

The ***Data Access Object*** (***DAO***) design pattern organizes data access by grouping it based on data types or collections. The following ***Users/dao.js*** file implements various CRUD operations for handling the ***users*** array in the ***Database***. Later sections in the chapter will create additional ***DAO***s for each of the data arrays: courses, modules, etc.

Kambaz/Users/dao.js

```
import db from "../Database/index.js";
import { v4 as uuidv4 } from "uuid";
let { users } = db;
export const createUser = (user) => {
 const newUser = { ...user, _id: uuidv4() };
 users = [...users, newUser];
 return newUser;
};
export const findAllUsers = () => users;
export const findUserById = (userId) => users.find((user) => user._id === userId);
export const findUserByUsername = (username) => users.find((user) => user.username === username);
export const findUserByCredentials = (username, password) =>
  users.find( (user) => user.username === username && user.password === password );
export const updateUser = (userId, user) => (users = users.map((u) => (u._id === userId ? user : u)));
export const deleteUser = (userId) => (users = users.filter((u) => u._id !== userId));
```

Like in the React project, install the ***uuid*** library in the Node.js project as shown below to generate unique identifiers when creating new instances of courses, modules, and other object instances.

```
npm install uuid
```

5.3.2.1 Integrating the React Sign In Screen with a RESTful Web API

DAOs provide an interface between an application and low-level database access, offering a high-level API to the rest of the application while abstracting the details and idiosyncrasies of using a particular database vendor. Similarly, routes create an interface between the HTTP network layer and the JavaScript object and function layer by transforming a stream of bits from a network connection request into a set of objects, maps, and function event handlers that are part of the client/server architecture in a multi-tiered application.

The Node.js server implements uses routes to integrate with the user interface and implements DAOs to communicate with the ***Database***. The server functions between these two layers, which is why it is often called the ***middle tier*** in a ***multi-tiered*** application. The following routes expose the database operations through a RESTful API, and the implementation of each function will be covered in the following sections. This chapter uses the ***Database*** component implemented in ***Database/index.ts***. Later chapters will refactor this by using an actual database.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  const createUser = (req, res) => { };
  const deleteUser = (req, res) => { };
  const findAllUsers = (req, res) => { };
  const findUserById = (req, res) => { };
  const updateUser = (req, res) => { };
  const signup = (req, res) => { };
  const signin = (req, res) => { };
  const signout = (req, res) => { };
```

```
  const profile = (req, res) => { };
  app.post("/api/users", createUser);
  app.get("/api/users", findAllUsers);
  app.get("/api/users/:userId", findUserById);
  app.put("/api/users/:userId", updateUser);
  app.delete("/api/users/:userId", deleteUser);
  app.post("/api/users/signup", signup);
  app.post("/api/users/signin", signin);
  app.post("/api/users/signout", signout);
  app.post("/api/users/profile", profile);
}
```

Import and configure the routes in ***index.js*** as shown below.

index.js

```
import express from "express";
...
import UserRoutes from "./Kambaz/Users/routes.js";

const app = express();
UserRoutes(app);
...
app.listen(process.env.PORT || 4000);
```

Routes implement ***RESTful Web API***s that clients can use to integrate with server functionality. The route implemented below extracts properties ***username*** and ***password*** from the request's body and passess them to the ***findUserByCredentials*** function implemented by the DAO. The resulting user is stored in the server variable ***currentUser*** to remember the logged in user. The user is then sent to the client in the response. Later sections will add error handling.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  ...
  const signin = (req, res) => {
    const { username, password } = req.body;
    currentUser = dao.findUserByCredentials(username, password);
    res.json(currentUser);
  };
  app.post("/api/users/signin", signin);
  ...
}
```

In the React user interface, under ***Kambaz/Account***, implement the ***client*** shown below to integrate with the user routes implemented in the server. The client function ***signin*** shown below posts a ***credentials*** object containing the ***username*** and ***password*** expected by the server. If the credentials are found, the response should contain the logged in user.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;

export const signin = async (credentials: any) => {
  const response = await axios.post( `${USERS_API}/signin`, credentials );
  return response.data;
};
```

Implement a ***Sign in*** screen users can use to authenticate with the application. The following component declares state variable ***credentials*** to edit the ***username*** and ***password***. Clicking the ***Sign in*** button posts the ***credentials*** to the server

using the ***client.signin*** function. When the server responds successfully, the currently logged in user is stored in the user reducer and navigate to the ***Profile*** screen implemented in a later section.

app/(Kambaz)/Account/Signin/page.tsx

```
import * as client from "./client";
export default function Signin() {
  const [credentials, setCredentials] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const signin = async () => {
    const user =  await client.signin(credentials);
    if (!user) return;
    dispatch(setCurrentUser(user));
    navigate("/Kambaz/Dashboard");
  };
  return ( ... );
}
```

5.3.2.2 Integrating the React Sign Up Screen with a RESTful Web API

The DAO implements functions ***createUser*** and ***findUserByUsername*** as shown below. The ***createUser*** DAO function accepts a user object from the user interface and then inserts the user into the ***Database***. The ***findUserByUsername*** accepts a ***username*** from the user interface and finds the user with the matching ***username***.

Users/dao.js

```
import { v4 as uuidv4 } from "uuid";
export const createUser = (user) => (users = [...users, { ...user, _id: uuidv4() }]);
export const findUserByUsername = (username) => users.find((user) => user.username === username);
```

The DAO functions are used to implement the ***sign up*** operation for users to sign up to the application. The ***signup*** route expects a user object with at least the properties ***username*** and ***password***. The DAO's ***findUserByUsername*** is called to check if a user with that username already exists. If such a user is found a 400 error status is returned along with an error message for display in the user interface. If the username is not already taken the user is inserted into the database and stored in the ***currentUser*** server variable. The response includes the newly created user. The ***signup*** route is mapped to the ***api/users/signup*** path.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
...
export default function UserRoutes(app) {
  ...
  const signup = (req, res) => {
    const user = dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json(
        { message: "Username already in use" });
      return;
    }
    currentUser = dao.createUser(req.body);
    res.json(currentUser);
  };
  app.post("/api/users/signup", signup);
  ...
}
```

Meanwhile in the React user interface Web app, implement a ***signup*** client that posts the new user to the Web API as shown below.

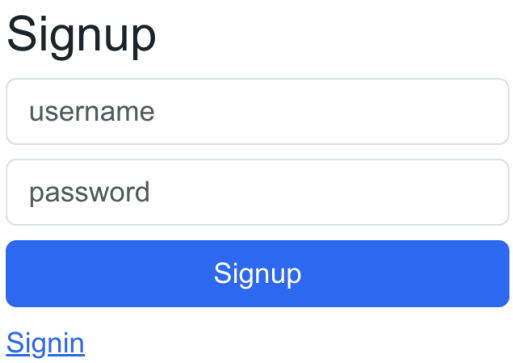
app/(Kambaz)/Account/client.ts

```
...
export const signup = async (user: any) => {
  const response = await axios.post(`${USERS_API}/signup`, user);
  return response.data;
};
...
```

If not already done so, implement a ***Sign up*** screen component that users can use to type their username and password, and post the credentials to the server for signing up. If the sign up is successful, navigate to the ***Profile*** screen. In the ***Account*** component, update the ***signup*** route to display the new ***Sign up*** screen. In the ***Sign in*** screen, create a ***Link*** to navigate to the ***Sign up*** screen. Confirm that you can signup with a new username and password. Style the ***Sign up*** screen so it looks as shown below on the right. Confirm navigates to profile and shows new user. Confirm you can navigate

app/(Kambaz)/Account/Signup/page.tsx

```
import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import * as client from "./client";
import { useDispatch } from "react-redux";
import { setCurrentUser } from "./reducer";
export default function Signup() {
  const [user, setUser] = useState<any>({});
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const signup = async () => {
    const currentUser = await client.signup(user);
    dispatch(setCurrentUser(currentUser));
    navigate("/Kambaz/Account/Profile");
  };
  return (
    <div className="wd-signup-screen">
      <h1>Sign up</h1>
      <FormControl value={user.username} onChange={(e) => setUser({ ...user, username: e.target.value })}
             className="wd-username b-2" placeholder="username" />
      <FormControl value={user.password} onChange={(e) => setUser({ ...user, password: e.target.value })}
             className="wd-password mb-2" placeholder="password" type="password"/>
      <button onClick={signup} className="wd-signup-btn btn btn-primary mb-2 w-100"> Sign up </button><br />
      <Link href="/Kambaz/Account/Signin" className="wd-signin-link">Sign in</Link>
    </div>
);}
```



5.3.2.3 Integrating the React Profile Screen with a RESTful Web API

In the ***User***'s ***DAO***, implement ***updateUser*** as shown below to update a single document by first identifying it by its primary key, and then updating the matching fields in the ***user*** parameter.

Users/dao.js

```
export const updateUser = (userId, user) => (users = users.map((u) => (u._id === userId ? user : u)));
```

In the ***User***'s ***routes***, make the ***DAO*** function available as a RESTful Web API as shown below. Map a route that accepts a user's primary key as a path parameter, passes the ID and request body to the DAO function and responds with the status.

Users/routes.js

```
export default function UserRoutes(app) {
  ...
  const updateUser = (req, res) => {
    const userId = req.params.userId;
    const userUpdates = req.body;
```

```
    dao.updateUser(userId, userUpdates);
    currentUser = dao.findUserById(userId);
    res.json(currentUser);
  };
  ...
  app.put("/api/users/:userId", updateUser);
}
```

In the React client application, add client function ***updateUser*** to send user updates to the server to be saved to the database.

app/(Kambaz)/Account/client.ts

```
export const updateUser = async (user: any) => {
  const response = await axios.put(`${USERS_API}/${user._id}`, user);
  return response.data;
};
```

In the ***Profile*** screen implement the ***updateProfile*** event handler as shown below to update the profile on the server. Add an ***Update*** button that invokes the new handler. Confirm that the profile changed by logging out and then logging back in.

app/(Kambaz)/Account/Profile/page.tsx

```
...
import * as client from "./client";
export default function Profile() {
  const [profile, setProfile] = useState<any>({});
  const dispatch = useDispatch();
  const navigate = useNavigate();
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const updateProfile = async () => {
    const updatedProfile = await client.updateUser(profile);
    dispatch(setCurrentUser(updatedProfile));
  };
  ...
  return (
    <div id="wd-profile-screen">
      <h3>Profile</h3>
      {profile && (
        ...
        <div>
          <button onClick={updateProfile} className="btn btn-primary w-100 mb-2"> Update </button>
          <button ...> Sign out </button>
        </div>
      )}
    </div>
);}
```

5.3.2.4 Retrieving the Profile from the Server

When a successful sign in occurs, the account information is stored in a server variable called ***currentUser***. The variable retains the signed-in user information as long as the server is running. The ***Sign in*** screen copies currentUser from the server into the ***currentUser*** state variable in the reducer and then navigates to the Profile screen. If the browser reloads, the ***currentUser*** state variable is cleared and the user is logged out. To address this, the browser must check whether someone is already logged in from the server and, if so, update the copy in the reducer. Create a route on the server to provide access to ***currentUser*** as shown below.

Users/routes.js

```
let currentUser = null;
export default function UserRoutes(app) {
  ...
```

```
  const signin = async (, res) => { ... };
  const profile = async (req, res) => {
    res.json(currentUser);
  };
  ...
  app.post("/api/users/signin",  signin);
  app.post("/api/users/profile", profile);
}
```

Then in the React Web app, implement a function to retrieve the ***account*** information from the server route implemented above as shown below.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
export const USERS_API = import.meta.env.VITE_HTTP_SERVER;
export const signin = async (user) => { ... };
export const profile = async () => {
  const response = await axios.post(`${USERS_API}/profile`);
  return response.data;
};
```

Create a new ***Session*** component that fetches the ***current user*** from the server and stores it in the store so that the rest of the application can have access to the ***current user***.

src/Account/Session.tsx

```
import * as client from "./client";
import { useEffect, useState } from "react";
import { setCurrentUser } from "./reducer";
import { useDispatch } from "react-redux";
export default function Session({ children }: { children: any }) {
  const [pending, setPending] = useState(true);
  const dispatch = useDispatch();
  const fetchProfile = async () => {
    try {
      const currentUser = await client.profile();
      dispatch(setCurrentUser(currentUser));
    } catch (err: any) {
      console.error(err);
    }
    setPending(false);
  };
  useEffect(() => {
    fetchProfile();
  }, []);
  if (!pending) {
    return children;
  }
}
```

Wrap the Kambaz application with ***Session*** component so that it renders before all other components to check if anyone is signed in. Once it figures out either way, it'll store the result in the store and let the rest of the components render. Confirm that it works by signing in and then from the Profile screen, reload the browser. Make sure the user information still renders correctly.

app/(Kambaz)/index.ts

```
...
import Session from "./Account/Session";
export default function Kambaz() {
  ...
  return (
    <Session>
```

```
      <div id="wd-kambaz">
      ...
      </div>
    </Session>
);}
```

5.3.2.5 Integrating Signout with a RESTful Web API

Implement a route for users to signout that resets the ***currentUser*** to null in the server as shown below.

Users/routes.js

```
let currentUser = null;
function UserRoutes(app) {
  ...
  const signout = (req, res) => {
    currentUser = null;
    res.sendStatus(200);
  };
  app.post("/api/users/signout", signout);
  ...
}
export default UserRoutes;
```

In the React user interface Web application, add a client function that can post to the ***signout*** route.

app/(Kambaz)/Account/client.ts

```
export const signout = async () => {
  const response = await axios.post(`${USERS_API}/signout`);
  return response.data;
};
```

In the ***Profile*** screen refactor the ***signout*** function to invoke the ***signout*** client function and then navigates to the ***Sign in*** screen. Confirm that you can signout and navigate to the ***Sign in*** screen.

app/(Kambaz)/Account/Profile/page.tsx

```
  ...
  const signout = async () => {
    await client.signout();
    dispatch(setCurrentUser(null));
    navigate("/Kambaz/Account/Signin");
  };
  ...
  <button onClick={signout} className="wd-signout-btn btn btn-danger w-100">
    Sign out
  </button>
  ...
```

5.3.3 Supporting Multiple User Sessions

The user authentication implemented so far is simple but supports only one signed-in user at a time. Web applications typically support multiple users signed in simultaneously. This section describes how to add session handling to the Node.js server to allow multiple users to be signed in at the same time.

5.3.3.1 Installing and Configuring Server Sessions

First, it is necessary to narrow down who is allowed to authenticate. Configure ***CORS*** to support cookies and restrict network access to come only from the React application as shown below.

index.js

```
const app = express();
app.use(
 cors({
   credentials: true,                                          // support cookies
   origin: process.env.CLIENT_URL || "http://localhost:3000",  // restrict cross origin resource
 })                                                            // sharing to the react application
);
app.use(express.json());
const port = process.env.PORT || 4000;
```

In the Node.js project, install the ***express-session*** library as shown below.

```
$ npm  install  express-session
```

Then in the server implementation file, import and configure the session library as shown below. Make sure to configure sessions ***after*** configuring cors.

index.js

```
import session from "express-session";      // import new server session library
const app = express();
app.use(cors({ ...  }));                    // configure cors first
const sessionOptions = {                    // configure server sessions after cors
  secret: "any string",                     // this is a default session configuration that works fine
  resave: false,                            // locally, but needs to be tweaked further to work in a
  saveUninitialized: false,                 // remote server such as AWS, Render, or Heroku. See later
};
app.use(
  session(sessionOptions)
);
```

Install the ***dotenv*** library to read configurations from environment variables on the server.

```
$ npm  install  dotenv
```

In a new ***.env*** file at the root of the project, declare the following environment variables.

.env

```
SERVER_ENV=development
CLIENT_URL=http://localhost:3000
SERVER_URL=http://localhost:4000
SESSION_SECRET=super secret session phrase
```

In ***index.js***, import the ***dotenv*** library to determine whether the application is running in the development environment, and configure the session accordingly. Note: the following configuration has been tested on ***Google's Chrome*** and ***Apple's Safari*** browsers.

index.js

```
import "dotenv/config";                                        // import the new dotenv library
import session from "express-session";                         // to read .env file
const app = express();
app.use(
  cors({
    credentials: true,
    origin: process.env.CLIENT_URL || "http://localhost:3000",  // use different front end URL in dev
  })                                                           // and in production
```

```
);
const sessionOptions = {                                    // default session options
  secret: process.env.SESSION_SECRET || "kambaz",
  resave: false,
  saveUninitialized: false,
};
if (process.env.SERVER_ENV !== "development") {             // in production
  sessionOptions.proxy = true;                              // turn on proxy support
  sessionOptions.cookie = {                                 // configure cookies for remote server
    sameSite: "none",
    secure: true,
    domain: process.env.SERVER_URL,
  };
}
app.use(session(sessionOptions));
app.use(express.json());                                    // make sure this comes AFTER configuring cors
...                                                         // and session, but BEFORE all the routes
```

The ***signup*** route retrieves the ***username*** from the request body. If a user with that username already exists, an error is returned. Otherwise, create the new user and store it in the session's ***currentUser*** property to remember that this new user is now the currently logged-in user.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
...
export default function UserRoutes(app) {
  ...
  const signup = (req, res) => {
    const user = dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json({ message: "Username already taken" });
      return;
    }
    const currentUser = dao.createUser(req.body);
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  };
```

An existing user can identify themselves by providing credentials. The ***signin*** route below looks up the user by their credentials, stores it in ***currentUser*** session, and responds with the user if they exist. Otherwise responds with an error.

Kambaz/Users/routes.js

```
  const signin = (req, res) => {
    const { username, password } = req.body;
    const currentUser = dao.findUserByCredentials(username, password);
    if (currentUser) {
      req.session["currentUser"] = currentUser;
      res.json(currentUser);
    } else {
      res.status(401).json({ message: "Unable to login. Try again later." });
    }
  };
```

If a user has already signed in, the ***currentUser*** can be retrieved from the session by using the ***profile*** route as shown below. If there is no ***currentUser***, an error is returned.

Kambaz/Users/routes.js

```
const profile = (req, res) => {
    const currentUser = req.session["currentUser"];
    if (!currentUser) {
      res.sendStatus(401);
```

```
      return;
    }
    res.json(currentUser);
  };
```

Users can be signed out by destroying the session.

Kambaz/Users/routes.js

```
  const signout = (req, res) => {
    req.session.destroy();
    res.sendStatus(200);
  };
```

If a user updates their profile, then the session must be kept in synch.

Kambaz/Users/routes.js

```
  const updateUser = (req, res) => {
    const userId = req.params.userId;
    const userUpdates = req.body;
    dao.updateUser(userId, userUpdates);
    const currentUser = dao.findUserById(userId);
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  };
```

5.3.3.2 Configuring Axios to Support Server Sessions

By default ***axios*** does not support cookies. To configure ***axios*** to include cookies in requests, use the ***axios.create()*** to create an instance of the library that includes cookies for credentials as shown below. Then replace all occurrences of the ***axios*** library with this new version ***axiosWithCredentials***.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
export const signin = async (credentials: any) => {
  const response = await axiosWithCredentials.post(`${USERS_API}/signin`, credentials);
  return response.data;
};
export const profile = async () => {
  const response = await axiosWithCredentials.post(`${USERS_API}/profile`);
  return response.data;
};
export const signup = async (user: any) => {
  const response = await axiosWithCredentials.post(`${USERS_API}/signup`, user);
  return response.data;
};
export const signout = async () => {
  const response = await axiosWithCredentials.post(`${USERS_API}/signout`);
  return response.data;
};
export const updateUser = async (user: any) => {
  const response = await axiosWithCredentials.put(`${USERS_API}/${user._id}`, user);
  return response.data;
};
```

5.3.4 Creating a RESTful Web API for Courses

Previous chapters implemented CRUD operations to create, read, update and delete courses in the Kambaz Dashboard. These changes were transient and were lost when users refreshed the browser. This section demonstrates implementing a RESTful Web API to integrate the ***Dashboard*** and ***Courses*** screen with the server. The API will migrate the CRUD operations from the user interface to the server where they belong.

5.3.4.1 Retrieving Courses

Now that the ***Database*** has been moved to the server, it must be made available to the React client application through a Web ***API*** (***Application Programming Interface***). The exercises below make the courses accessible at http://localhost:4000/api/courses for the React user interface to integrate. First implement a DAO to retrieve all courses from the ***Database***.

Kambaz/Courses/dao.js

```
import Database from "../Database/index.js";
export function findAllCourses() {
  return Database.courses;
}
```

Use the ***DAO*** to implement a route that retrieves all the courses.

Kambaz/Courses/routes.js

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  const findAllCourses = (req, res) => {
    const courses = dao.findAllCourses();
    res.send(courses);
  }
  app.get("/api/courses", findAllCourses);
}
```

In ***index.js*** import the new routes and pass a reference to the express module to the routes. Make sure to work ***AFTER*** the ***cors***, ***session***, and ***json use*** statements. Point your browser to http://localhost:4000/api/courses and confirm the server responds with an array of courses.

index.js

```
import express from "express";
import Lab5 from "./Lab5/index.js";
import UserRoutes from "./Kambaz/Users/routes.js";
import CourseRoutes from "./Kambaz/Courses/routes.js";
import cors from "cors";
const app = express();
app.use(cors(...));      // make sure cors is configured BEFORE session
app.use(session(...));   // make sure session is configure BEFORE express.json
app.use(express.json()); // make sure express.json is configure BEFORE all routes
UserRoutes(app);
CourseRoutes(app);
Lab5(app);
app.listen(4000);
```

Since the Dashboard displays courses a user is enrolled in, implement ***findCoursesForEnrolledUser*** as shown below to retrieve courses the current user is enrolled in.

Kambaz/Courses/dao.js

```
...
```

```
export function findCoursesForEnrolledUser(userId) {
  const { courses, enrollments } = Database;
  const enrolledCourses = courses.filter((course) =>
    enrollments.some((enrollment) => enrollment.user === userId && enrollment.course === course._id));
  return enrolledCourses;
}
```

Since enrolled courses are retrieved within the context of the currently logged in user, implement the following route to retrieve the courses in the user routes.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
import * as courseDao from "../Courses/dao.js";
export default function UserRoutes(app) {
  ...
  const findCoursesForEnrolledUser = (req, res) => {
    let { userId } = req.params;
    if (userId === "current") {
      const currentUser = req.session["currentUser"];
      if (!currentUser) {
        res.sendStatus(401);
        return;
      }
      userId = currentUser._id;
    }
    const courses = courseDao.findCoursesForEnrolledUser(userId);
    res.json(courses);
  };
  app.get("/api/users/:userId/courses", findCoursesForEnrolledUser);
  ...
}
```

Back in the user interface, create a ***client.ts*** file under the ***Kambaz/Courses*** that implements all the course related communication between the user interface and the server. Start by implementing the ***fetchAllCourses*** client function as shown below.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const fetchAllCourses = async () => {
  const { data } = await axios.get(COURSES_API);
  return data;
};
```

But the Dashboard only displays the course the current logged in user is enrolled in, so in the ***Users***'s client file, implement ***findMyCourses*** that retrieves the current user's courses using the new ***findCoursesForEnrolledUser*** end point.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
const axiosWithCredentials = axios.create({ withCredentials: true });
export const findMyCourses = async () => {
  const { data } = await axiosWithCredentials.get(`${USERS_API}/current/courses`);
  return data;
};
...
```

In the ***Kambaz*** component use ***useEffect*** to fetch the courses from the server on component load and update the ***courses*** state variable that populates the ***Dashboard***. Use the ***currentUser*** in the ***accountReducer*** as a dependency so that if a different user logs in, the courses will be reloaded from the server. Remove ***Database*** references from the user interface since we don't need it anymore. Also initialize the ***courses*** state variable as empty since we won't have the database anymore.

app/(Kambaz)/page.tsx

```
import * as db from "./Database";
import * as client from "./Courses/client";
import * as userClient from "./Account/client";
export default function Kambaz() {
  const [courses, setCourses] = useState<any[]>([]);
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const fetchCourses = async () => {
    try {
      const courses = await userClient.findMyCourses();
      setCourses(courses);
    } catch (error) {
      console.error(error);
    }
  };
  useEffect(() => {
    fetchCourses();
  }, [currentUser]);
  ...
}
```

In the ***Dashboard*** screen, remove the filtering of courses by enrollments since the server is already doing that. Restart the server and user interface to confirm that the ***Dashboard*** renders the courses the current user is enrolled in. Sign in as different users and confirm only the courses the user is enrolled in display in the ***Dashboard***.

app/(Kambaz)/Dashboard/page.tsx

```
import * as db from "./Database";
...
export default function Dashboard({ ... }) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const { enrollments } = db;
  ...
  return (
    <div id="wd-dashboard">
      ...
      {courses
        .filter((course) => enrollments.some((enrollment) =>
          enrollment.user === currentUser._id &&
          enrollment.course === course._id))
        .map((course) => ( ... ))}
      ...
    </div>
);}
```

5.3.4.2 Creating New Courses

Implement a route that creates a new course and adds it to the ***Database***. The new course is passed in the HTTP body from the client and is appended to the end of the courses array in the ***Database***. The new course is given a new unique identifier and sent back to the client in the response.

Kambaz/Courses/dao.js

```
import { v4 as uuidv4 } from "uuid";
...
export function createCourse(course) {
  const newCourse = { ...course, _id: uuidv4() };
```

```
  Database.courses = [...Database.courses, newCourse];
  return newCourse;
}
```

When a course is created, it needs to be associated with the creator. In a new ***Enrollments/dao.js*** file, implement ***enrollUserInCourse*** to enroll, or associate, a user to a course.

Kambaz/Enrollments/dao.js

```
import Database from "../Database/index.js";
import { v4 as uuidv4 } from "uuid";

export function enrollUserInCourse(userId, courseId) {
  const { enrollments } = Database;
  enrollments.push({ _id: uuidv4(), user: userId, course: courseId });
}
```

In the ***Users***'s routes, implement ***createCourse*** as shown below to create a new course and then enroll the ***currentUser*** in the new course. Respond with the ***newCourse*** so it can be rendered in the user interface.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
import * as courseDao from "../Courses/dao.js";
import * as enrollmentsDao from "../Enrollments/dao.js";
export default function UserRoutes(app) {
  const createCourse = (req, res) => {
    const currentUser = req.session["currentUser"];
    const newCourse = courseDao.createCourse(req.body);
    enrollmentsDao.enrollUserInCourse(currentUser._id, newCourse._id);
    res.json(newCourse);
  };
  app.post("/api/users/current/courses", createCourse);
  ...
}
```

In the account's ***client.ts***, add a ***createCourse*** client function that ***posts*** a new course to the server and returns the response's data which should be the brand new course created in the server.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
const axiosWithCredentials = axios.create({ withCredentials: true });
export const createCourse = async (course: any) => {
  const { data } = await axiosWithCredentials.post(`${USERS_API}/current/courses`, course);
  return data;
};
...
```

In the ***Kambaz*** component, refactor the ***addCourse*** function so that it ***posts*** the new course to the server and the new course in the response is appended to the end of the ***courses*** state variable. Confirm that creating a new course updates the user interface with the added course.

app/(Kambaz)/page.tsx

```
...
import * as userClient from "./Account/client";
export default function Kambaz() {
  const addNewCourse = async () => {
```

```
    const newCourse = await userClient.createCourse(course);
    setCourses([ ...courses, newCourse ]);
  };
  const findAllCourses = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

5.3.4.3 Deleting a Course

Implement a route that removes a course and all enrollments associated with the course. First implement a ***deleteCourse*** DAO function that filters the course by its ID and then filters out all enrollments by the course's ID as shown below.

Kambaz/Courses/dao.js

```
import Database from "../Database/index.js";
export function deleteCourse(courseId) {
  const { courses, enrollments } = Database;
  Database.courses = courses.filter((course) => course._id !== courseId);
  Database.enrollments = enrollments.filter(
    (enrollment) => enrollment.course !== courseId
);}
...
```

In the Course's routes, implement a ***delete*** route that parses the course's ID from the URL and uses the ***deleteCourse*** DAO function as shown below. Remember to group callback functions at the top and the route declarations at the bottom.

Kambaz/Courses/routes.js

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  const findAllCourses = (req, res) => { ... }
  const deleteCourse = (req, res) => {
    const { courseId } = req.params;
    const status = dao.deleteCourse(courseId);
    res.send(status);
  }
  app.delete("/api/courses/:courseId", deleteCourse);
  app.get("/api/courses", findAllCourses);
}
```

In the course's ***client.ts***, add a ***deleteCourse*** client function that ***deletes*** an existing course from the server and returns the status response from the server.

app/(Kambaz)/Courses/client.ts

```
export const deleteCourse = async (id: string) => {
  const { data } = await axios.delete(`${COURSES_API}/${id}`);
  return data;
};
```

In the ***Kambaz*** component, refactor ***deleteCourse*** to use ***courseClient*** to ***delete*** courses from the server and then filter out the course from the local ***courses*** state variable. Confirm clicking ***Delete*** actually removes the course from the ***Dashboard***.

app/(Kambaz)/page.tsx

```
import * as userClient from "./Account/client";
import * as courseClient from "./Courses/client";
export default function Kambaz() {
  const [courses, setCourses] = useState<any[]>([]);
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  const deleteCourse = async (courseId: string) => {
```

```
    const status = await courseClient.deleteCourse(courseId);
    setCourses(courses.filter((course) => course._id !== courseId));
};}
```

5.3.4.4 Updating a Course

In the Course's DAO, implement ***updateCourse*** function to update a course in the ***Database***. First lookup the course by its ID and then apply the updates to the course as shown below.

Kambaz/Courses/dao.js

```
export function updateCourse(courseId, courseUpdates) {
  const { courses } = Database;
  const course = courses.find((course) => course._id === courseId);
  Object.assign(course, courseUpdates);
  return course;
}
```

In the Course's routes, implement a ***put*** route that parses the ***id*** of course as a path parameter and updates uses the ***updateCourse*** DAO function to update the corresponding course with the updates in HTTP request body. If the update is successful, respond with status 204.

Kambaz/Courses/routes.js

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const updateCourse = (req, res) => {
    const { courseId } = req.params;
    const courseUpdates = req.body;
    const status = dao.updateCourse(courseId, courseUpdates);
    res.send(status);
  }
  app.put("/api/courses/:courseId", updateCourse);
  ...
}
```

In the course's ***client.ts***, add a ***updateCourse*** client function that ***updates*** an existing course in the server and returns the status response from the server.

app/(Kambaz)/Courses/client.ts

```
export const updateCourse = async (course: any) => {
  const { data } = await axios.put(`${COURSES_API}/${course._id}`, course);
  return data;
};
```

In the ***Kambaz*** component, refactor the ***updateCourse*** function so that it ***put***s the updated course to the server and then swaps out the old corresponding course with the new version in the ***course*** state variable. Confirm that clicking the ***Update*** button actually updates the course in the ***Dashboard***.

app/(Kambaz)/page.tsx

```
import * as courseClient from "./Courses/client";
export default function Kambaz() {
  const updateCourse = async () => {
    await courseClient.updateCourse(course);
    setCourses(courses.map((c) => {
        if (c._id === course._id) { return course; }
        else { return c; }
    });
```

```
  );};
  const addCourse = async () => {...};
  useEffect(() => {...}, []);
  return ( ... );
}
```

5.3.5 Creating a RESTful Web API for Modules

Now let's do the same thing we did for the courses, but for the modules. We'll need routes that deal with the modules similar to the operations we implemented for the courses. We'll need to implement all the basic ***CRUD*** operations: ***create*** modules, ***read***/retrieve modules, ***update*** modules and ***delete*** modules. The main difference will be that modules exist with the context of a particular course. Each course has a different set of modules, so the routes will need to take into account the course ID for which the modules we are operating on.

5.3.5.1 Retrieving a Course's Modules

Create ***DAO*** for the ***Modules*** to implement module data access from the ***Database***. Start by implementing ***findModulesForCourse*** to retrieve a course's modules by its ID as shown below.

Kambaz/Modules/dao.js

```
import Database from "../Database/index.js";
export function findModulesForCourse(courseId) {
  const { modules } = Database;
  return modules.filter((module) => module.course === courseId);
}
```

In the routes file for the ***Course*** add a route to retrieve the modules for a course by its ID encoded in the path. Parse the course ID from the path and then use the module's DAO ***findModulesForCourse*** function to retrieve the modules for that course.

Kambaz/Courses/routes.js

```
import * as dao from "./dao.js";
import * as modulesDao from "../Modules/dao.js";
export default function CourseRoutes(app) {
  ...
  const findModulesForCourse = (req, res) => {
    const { courseId } = req.params;
    const modules = modulesDao.findModulesForCourse(courseId);
    res.json(modules);
  }
  app.get("/api/courses/:courseId/modules", findModulesForCourse);
  ...
}
```

In the Course's ***client.js*** file in the React project, create findModulesForCourse to integrate the user interface with the server as shown below. Implement ***findModulesForCourse*** function shown below which retrieves the modules for a given course.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const findModulesForCourse = async (courseId: string) => {
  const response = await axios
    .get(`${COURSES_API}/${courseId}/modules`);
  return response.data;
};
```

```
...
```

Update the modules reducer implemented in earlier chapters as shown below. Remove dependencies from the ***Database*** since we've moved it to the server. Empty the ***modules*** state variable since we'll be populating it with the modules we retrieve from the server using the ***findModulesForCourse*** function. Add a ***setModules*** reducer function so we can populate the ***modules*** state variable when we retrieve the modules from the server.

app/(Kambaz)/Courses/[cid]/Modules/reducer.tsx

```
import db from "../Database";
const initialState = {
  modules: [],
};
const modulesSlice = createSlice({
  name: "modules",
  initialState,
  reducers: {
    setModules: (state, action) => {
      state.modules = action.payload;
    },
    addModule: (...) => {...},
    deleteModule: (...) => {...},
    updateModule: (...) => {...},
    editModule: (...) => {...},
  },
});
export const { addModule, deleteModule, updateModule, editModule, setModules } = modulesSlice.actions;
export default modulesSlice.reducer;
```

In the ***Modules*** component, import the new ***setModules*** reducer function and the new ***findModulesForCourse*** client function. Using a ***useEffect*** function, invoke the ***findModulesForCourse*** client function and dispatch the modules from the server to the reducer with the ***setModules*** function as shown below. Remove the filter since modules are already filtered on the server. Confirm that navigating to a course populates the corresponding modules.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { setModules, addModule, editModule, updateModule, deleteModule } from "./reducer";
import { useState, useEffect } from "react";
import * as coursesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const dispatch = useDispatch();
  const fetchModules = async () => {
    const modules = await coursesClient.findModulesForCourse(cid as string);
    dispatch(setModules(modules));
  };
  useEffect(() => {
    fetchModules();
  }, []);

 return (
   <div>
     ...
     <ListGroup id="wd-modules" className="rounded-0">
       {modules
         .filter((module: any) => module.course === cid)
         .map((module: any) => ( ... ))}
     </ListGroup>
   </div>
);}
```

5.3.5.2 Creating Modules for a Course

To create a new module in the ***Database***, implement ***createModule*** in the Module's DAO as shown below. The function accepts the new module as a parameter, set its primary key and then appends the new module to the ***Database's*** module array.

Kambaz/Modules/dao.js

```
import Database from "../Database/index.js";
import { v4 as uuidv4 } from "uuid";
export function createModule(module) {
  const newModule = { ...module, _id: uuidv4() };
  Database.modules = [...Database.modules, newModule];
  return newModule;
}
export function findModulesForCourse(courseId) { ... }
```

In the ***Course***'s routes, implement a ***post*** Web API for the user interface to integrate to when creating a new module for a given course. Parse the course's ID from the path and the new module from the request's body. Set the new module's course's property to the course's ID so that the module know what course it belongs to. Use the module's DAO's createModule function to create the new module and then respond with the new module.

Kambaz/Courses/routes.js

```
import Database from "../Database/index.js";
import * as dao from "./dao.js";
import * as modulesDao from "../Modules/dao.js";
export default function CourseRoutes(app) {
  ...
  const findModuleForCourse = async (req, res) => { ... }
  const createModuleForCourse = (req, res) => {
    const { courseId } = req.params;
    const module = {
      ...req.body,
      course: courseId,
    };
    const newModule = modulesDao.createModule(module);
    res.send(newModule);
  }
  app.post("/api/courses/:courseId/modules", createModuleForCourse);
  app.get("/api/courses/:courseId/modules", findModuleForCourse);
  ...
}
```

In the user interface, implement a ***createModuleForCourse*** client function that posts new modules from the user interface to the server as shown below. Encode the course's ID in the URL so the server knows what course the module belongs to.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const createModuleForCourse = async (courseId: string, module: any) => {
  const response = await axios.post(
    `${COURSES_API}/${courseId}/modules`,
    module
  );
  return response.data;
};
export const findModulesForCourse = async (courseId: string) => { ... };
...
```

In the ***Modules*** screen, implement a ***createModuleForCourse*** event handler that uses the new ***createModuleForCourse*** client function to send the module to the server and then dispatches the created module to the reducer so it's added to the

reducer's ***modules*** state variable. Update the ***ModulesControls addModule*** attribute so that it uses the new ***createModuleForCourse*** event handler. Confirm that new modules are created for the current course.

app/(Kambaz)/Courses/[cid]/Modules/index.ts

```
import { addModule, editModule, updateModule, deleteModule, setModules } from "./reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const createModuleForCourse = async () => {
    if (!cid) return;
    const newModule = { name: moduleName, course: cid };
    const module = await coursesClient.createModuleForCourse(cid, newModule);
    dispatch(addModule(module));
  };
  return (
    <div>
      <ModulesControls setModuleName={setModuleName} moduleName={moduleName} addModule={createModuleForCourse} />
      ...
    </div> );}
```

5.3.5.3 Deleting a Module

In the ***Modules DAO***, implement ***deleteModule*** to remove a module from the ***Database*** by its ***ID*** as shown below.

Kambaz/Modules/dao.js

```
import Database from "../Database/index.js";
export function deleteModule(moduleId) {
  const { modules } = Database;
  Database.modules = modules.filter((module) => module._id !== moduleId);
}
export function createModule(module) { ... }
export function findModulesForCourse(courseId) { ... }
```

Create a ***router*** file for the ***Modules*** and implement a route that handles an ***HTTP DELETE*** to remove a module by its ID. Parse the module's ID from the path and use the ***DAO***'s ***deleteModule*** function to remove the module from the ***Database***.

Kambaz/Modules/routes.js

```
import * as modulesDao from "./dao.js";
export default function ModuleRoutes(app) {
  const deleteModule = async (req, res) => {
    const { moduleId } = req.params;
    const status = await modulesDao.deleteModule(moduleId);
    res.send(status);
  }
  app.delete("/api/modules/:moduleId", deleteModule);
}
```

In the ***index.js*** server file, import the new ***ModuleRoutes*** and pass it a reference to the ***express*** library.

index.js

```
...
import ModuleRoutes from "./Kambaz/Modules/routes.js";
...
app.use(express.json()); // make sure this is configure BEFORE all the routes below
Lab5(app);
```

```
UserRoutes(app);
CourseRoutes(app);
EnrollmentRoutes(app);
ModuleRoutes(app);
app.listen(4000);
```

In the React user interface, create a new ***client*** file for ***Modules***, and implement the ***deleteModule*** function as shown below. Pass it the ***ID*** of the module to be removed, encode it in a URL, and sent it as an ***HTTP DELETE*** to the server.

app/(Kambaz)/Courses/[cid]/Modules/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const MODULES_API = `${HTTP_SERVER}/api/modules`;
export const deleteModule = async (moduleId: string) => {
 const response = await axios.delete(`${MODULES_API}/${moduleId}`);
 return response.data; };
```

In the ***Modules*** screen, import the new ***client*** file and use it to create the ***removeModule*** event handler as shown below. Use the new ***deleteModule*** client function to remove the module from the server. If successful, dispatch the deleted module's ID to the reducer to remove the module from the ***modules*** state variable as well. In the ***ModuleControlsButtons*** component, update the ***deleteModule*** attribute to use the new ***removeModule*** event handler. Confirm that clicking the trashcan of modules removes the modules. Refresh the screen to make sure that the modules is permanently deleted.

app/(Kambaz)/Courses/[cid]/Modules/index.js

```
import { addModule, editModule, updateModule, deleteModule, setModules } from "./reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
import * as modulesClient from "./client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const removeModule = async (moduleId: string) => {
    await modulesClient.deleteModule(moduleId);
    dispatch(deleteModule(moduleId));
  };
  ...
  return (
    <div>
      ...
      <ListGroup id="wd-modules" className="rounded-0">
        {modules.map((module: any) => (
              ...
              <ModuleControlButtons moduleId={module._id}
               deleteModule={(moduleId) => removeModule(moduleId)}
               editModule={(moduleId) => dispatch(editModule(moduleId))} />
        ))}
      </ListGroup>
    </div>);}
```

5.3.5.4 Update Module

In the ***Module***'s ***DAO***, implement ***updateModule*** to update a module in the ***Database*** by its ***ID***. First lookup the module by its ***ID*** and then apply the updates to the module as shown below.

Kambaz/Modules/dao.js

```
import Database from "../Database/index.js";
export function updateModule(moduleId, moduleUpdates) {
  const { modules } = Database;
```

```
  const module = modules.find((module) => module._id === moduleId);
  Object.assign(module, moduleUpdates);
  return module;
}
export function deleteModule(moduleId) { ... }
export function createModule(module) { ... }
export function findModulesForCourse(courseId) { ... }
```

In the ***Module***'s ***routes*** file, implement an ***HTTP PUT*** request handler that parses the ***ID*** of the course from the ***URL*** and the module updates from the ***HTTP*** request body. Use the ***DAO***'s ***updateModule*** function to apply the updates to the module.

Kambaz/Modules/routes.js

```
import * as modulesDao from "./dao.js";
export default function ModuleRoutes(app) {
  ...
  const updateModule = async (req, res) => {
    const { moduleId } = req.params;
    const moduleUpdates = req.body;
    const status = await modulesDao.updateModule(moduleId, moduleUpdates);
    res.send(status);
  }
  app.put("/api/modules/:moduleId", updateModule);
  ...
}
```

In the ***client*** file for ***Modules*** in the React user interface, implement the ***updateModule*** function as shown below. Pass it the module to be update. Encode the ***ID*** of the module in a URL, and sent the module updates in the body of an ***HTTP PUT*** request to the server.

app/(Kambaz)/Courses/[cid]/Modules/client.ts

```
import axios from "axios";
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const MODULES_API = `${HTTP_SERVER}/api/modules`;
export const updateModule = async (module: any) => {
  const { data } = await axios.put(`${MODULES_API}/${module._id}`, module);
  return data;
};
export const deleteModule = async (moduleId: string) => { ... };
```

In the ***Modules*** screen, implement a ***saveModule*** event handler as shown below. Use the new ***updateModule*** client function to update the module in the server. If successful, dispatch the updated module to the reducer to update the module in the ***modules*** state variable as well. In the ***onKeyDown*** event handler, invoke the ***saveModule*** event handler. Confirm that updating the module in the user interface actually modifies the module on the server. Refresh the screen to make sure that the modules has been modified.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
...
import { addModule, editModule, updateModule, deleteModule, setModules } from "./reducer";
import { useSelector, useDispatch } from "react-redux";
import * as coursesClient from "../client";
import * as modulesClient from "./client";
export default function Modules() {
  const { cid } = useParams();
  const [moduleName, setModuleName] = useState("");
  const { modules } = useSelector((state: any) => state.modulesReducer);
  const dispatch = useDispatch();
  const saveModule = async (module: any) => {
    await modulesClient.updateModule(module);
    dispatch(updateModule(module));
```

```
  };
  ...
  return (
    <div>
      ...
      <ListGroup id="wd-modules" className="rounded-0">
        {modules.map((module: any) => (
          ...
          {!module.editing && module.name}
          { module.editing && (
            <FormControl className="w-50 d-inline-block" value={module.name}/>
              onChange={(e) => dispatch(updateModule({ ...module, name: e.target.value })) }
              onKeyDown={(e) => {
                if (e.key === "Enter") {
                  saveModule({ ...module, editing: false });
                }
              }}
            )}
      ))}
    </ListGroup>
  </div> );}
```

5.3.6 Assignments and Assignments Editor (On Your Own)

In your Node.js server application, implement routes for ***creating***, ***retrieving***, ***updating***, and ***deleting*** assignments. In the React Web application, create an ***assignment client*** file that uses ***axios*** to send ***POST***, ***GET***, ***PUT***, and ***DELETE HTTP*** requests to integrate the React application with the server application. In the React user interface, refactor the ***Assignments*** and ***AssignmentEditor*** screens implemented in earlier chapters to use the new ***client*** file to ***CRUD*** assignments. New assignments, updates to assignments, and deleted assignments should persist if the screens are refreshed as long as the server is running.

5.3.7 Enrollments (On Your Own)

In your Node.js server application, implement routes to support the ***Enrollments*** screen. Users should be able to enroll and unenroll from courses. In the React application, implement an enrollments client that uses ***axios*** to integrate with the routes in the server. Enrollments should persist as long as the server is running.

5.3.8 People Table (Optional)

In your Node.js server application, implement routes to support the People screen. Users should be able to see all users enrolled in the course. Faculty should be able to create, update, and delete users. In the React application, implement an users client that uses ***axios*** to integrate with the routes in the server. User changes should persist as long as the server is running.

5.4 Deploying RESTful Web Service APIs to a Public Remote Server

Up to this point you should have a working two tiered application with the first tier consisting of a front end React user interface application and the second tier consisting of a Node Express HTTP server application. In this section we're going to learn how to replicate this setup so that it can execute on remote servers. All development should be done in the local development environment on your personal development computer, and only when we're satisfied that all works fine locally, then we can make an effort at deploying the application on remote servers. The React Web application is already configured to deploy and run remotely on Vercel when you commit and push to the GitHub repository containing the source for the project. This section demonstrates how to configure the Node Express HTTP server project to deploy to a remote server hosted by ***Render*** or ***Heroku*** and then integrate the remote React Web application on Vercel to the Node Express server deployed and running on ***Render*** (or ***Heroku***).

5.4.1 Committing and Pushing the Node Server Source to Github

First create a local Git repository in the Node Express project by typing ***git init*** at the command line at the root of the project. It's ok if the repository was already initialized.

```
$ git  init
```

Configure the Git repository to disregard unnecessary files in the repository by listing them in ***.gitignore***. Create a new file called ***.gitignore*** if it does not already exist. Note the leading period (.) in front of the file name. The file should contain at least ***node_modules***, but should also contain any IDE specific files or directories. If using IntelliJ, include the ***.idea*** folder as shown below. Environment files such as ***.env*** and ***.env.local*** should also be included in ***.gitignore***.

.gitignore

```
node_modules
.env.local
.env
```

Use ***git add*** to add all the source code into the repository and commit with a simple comment.

```
$ git  add  .
$ git  commit  -m  "first commit"
```

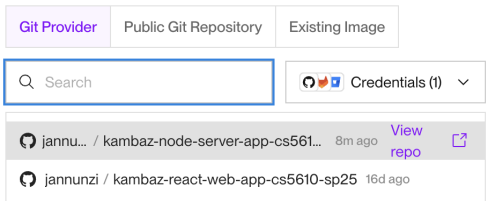
Head over to ***github.com*** and create a repository named ***kambaz-node-server-app***. Add this repository as the origin target using the git remote command. ***NOTE:*** make sure to use your github username instead of mine.

```
$ git remote add origin https://github.com/jannunzi/kambaz-node-server-app.git
```

Push the code in your local repository to the remote origin repository. Note that your branch might be called something else. Refresh the remote github repository and confirm the code is now available online.

```
$ git push -u origin main
```

5.4.2 Deploying a Node Server Application to Render.com from Github

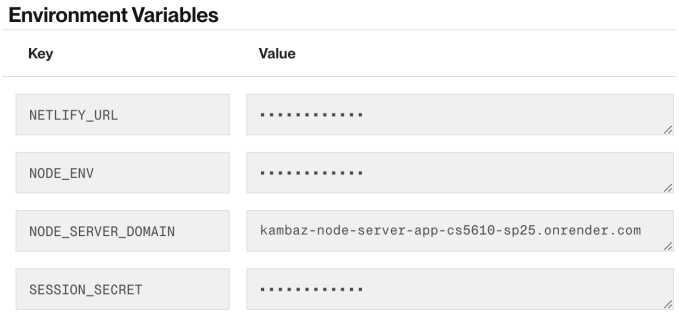


If you don't already have an account at ***render.com***, create a new account to deploy the Node server remotely. From the dashboard on the top right, select ***Add New*** and then ***Web Service***. In the ***You are deploying a web service*** screen, in the ***Source Code*** option, select the ***Git Provider*** tab, search for the git repository created earlier and select it from the dropdown list. In the ***Name*** field, type the name of the application, e.g., use the same name as the github repository ***kambaz-node-server-app***, or something similar. In the ***Build Command*** field type ***npm install***. In the ***Start Command*** field type ***npm start*** or ***node index.js***. Under the ***Instance Type*** select ***Free***. In the ***Environment Variables*** section click ***Add Environment Variable*** to create all the environment variables in the ***.env*** file, but with values shown below.

Environment Variable	*Value*
SERVER_ENV	production

CLIENT_URL	https://kambaz-next-js-cs5610-sp25.Vercel.app
SERVER_URL	kambaz-node-server-app-cs5610-sp25.onrender.com
SESSION_SECRET	super secret session phrase

For the ***CLIENT_URL*** environment variable, use the URL of your React Web application deployed in Vercel, not mine as shown above. For ***SERVER_URL***, use the domain name in the ***Name*** field, but make sure it has the postfix ***.onrender.com*** as shown above. Note that after the application deploys, Render might modify the domain name slightly if the domain is already taken. You'll need to edit the environment variable so that its value is the actual domain name. Make sure that ***SERVER_URL*** does not contain ***http://*** or ***https://*** as a prefix.

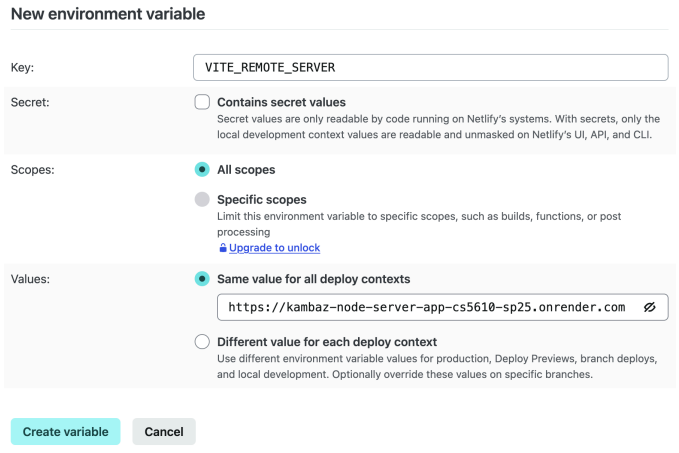


Click ***Deploy Web Service*** to deploy the server. In the deploy screen take a look at the logs. You can click on ***Maximize*** to see the logs better. Look for a ***Build successful*** message. You can click on ***Minimize*** to minimize the logs. If the deployment fails, fix whatever the logs complaint about, commit and push changes to GitHub, and try deploying again by selecting ***Deploy last commit*** from the ***Manual Deploy*** drop menu at the top right. If the deployment succeeds a URL appears at the top of the screen. Navigate to the URL, e.g., https://kambaz-node -server-app-cs5610-sp25.onrender.com and confirm you get the same greeting you would get locally, e.g., ***Welcome to Full Stack Development!*** Also confirm you can get the array of courses from the remote API, e.g., https://kambaz-node-server-app-cs5610-sp25.onrender.com/ api/courses. Finally, make sure you can get the list of modules for at least one of the courses, e.g., https://kambaz-node-server-app-cs5610-sp25.onrender.com/api/courses/RS101/modules. ***NOTE:*** the actual URL might be different based on the actual name you chose for the application.

If the name chosen was not unique, Render will modify the name of the domain so that it's unique. It might append a random string at the end of the name, e.g., https://kambaz-node-server-app-cs5610-sp25-***qpwoeiru***.onrender.com. If this is the case, modify the ***SERVER_URL*** environment variable by clicking ***Environment*** on the left sidebar of the Render dashboard. Make sure all environment variables have the correct values and edit if necessary. To edit ***SERVER_URL***, click ***Edit*** and replace the value with the actual domain name as shown below. Do not include the protocol ***https://***, or extra slashes. Click on ***Save, rebuild, and deploy*** when done.

5.4.4 Configuring Remote Environment in Vercel

Now that the Node.js HTTP server is running remotely on Render.com, the React Web application running on Vercel needs to be configured to integrate with the server running on Render.com. Currently the React Web application is configured to connect to the local Node Express server, but when the Web application is running on ***Vercel*** it needs to connect to the remote Node Express server running on ***Render*** or ***Heroku***. Configure ***Vercel*** defining environment variable ***VITE_REMOTE_SERVER*** so that it references the remote server running on ***Render*** or ***Heroku***.



To configure environment variables from the Vercel dashboard, navigate to ***Site configuration***, then ***Environment variables***, select ***Add a variable***, and then select ***Add a single variable***. In the ***New environment variable*** form here on the right, enter ***VITE_HTTP_SERVER*** in the ***Key*** field, and then in the ***Values*** field, copy and paste the root URL of the application running on Render or Heroku, e.g., ***https://kambaz-node -server-app.onrender.com***, and click ***Create variable***. Note that here we do

want the protocol ***https://***, but not in the ***SERVER_URL*** environment variable in Render.com. Redeploy the React application and confirm that the ***Dashboard*** renders the courses from the remote Node server and ***Modules*** still renders the modules for the selected course. Also confirm all the labs still work when running on Vercel. Using the ***Network*** tab in the ***Inspector*** in the ***Development Tools*** of the browser, make sure that none of the requests use ***http://localhost:3000***.

5.5 Conclusion

In this chapter we learned how to create HTTP servers using the Node.js JavaScript framework. We implemented RESTful services with the Express library and practiced sending, retrieving, modifying, and updating data using HTTP requests and responses. We then learned how to integrate React Web applications to HTTP servers implementing a client server architecture. In the next chapter we will add database support to the HTTP server so we can store data permanently.

5.6 Deliverables

As a deliverable, make sure you complete all the lab exercises, course, modules, and assignment routes on the server, as well as the client, and component refactoring on the React project. For both the React and Node repositories, all your work should be done in a branch called ***a5***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to ***Vercel*** and ***Render*** (or ***Heroku***) and confirm they integrate. All the lab exercises should work remotely just as well as locally. The Kambaz Dashboard should display the courses from the server as well as the modules, and assignments. In ***TOC.tsx***, add a link to the new GitHub repository and a link to the root of the server running on ***Render*** or ***Heroku***. As a deliverable in ***Canvas***, submit the URL to the ***a5*** branch deployment of your React application running on Vercel.

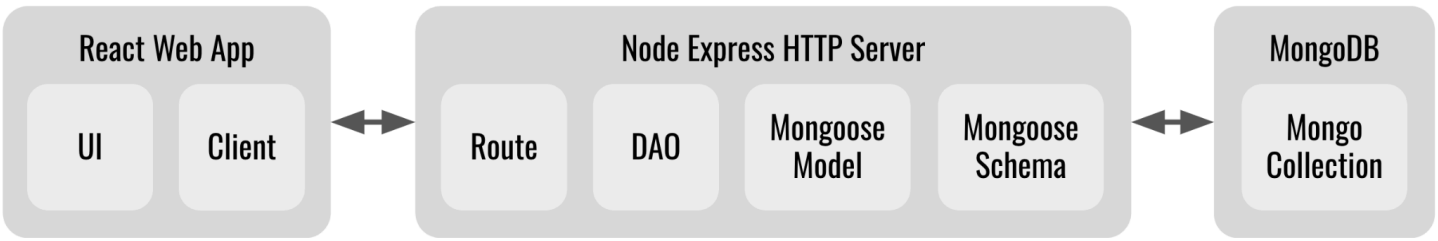
Chapter 6 - Integrating React with MongoDB

There are two main categories of databases: ***relational databases*** and ***non-relational databases***. Relational databases such as ***MySQL***, ***SQL Server***, and ***Postgre***, store data in ***tables*** containing ***records*** of the same ***type***, e.g., a table called ***courses*** would contain records representing all the ***courses***, and a ***users table*** would contain all the ***users*** of an application. Records are represented as ***rows*** in the tables where each ***column*** stores data for attributes specific to the type of the table, e.g., the rows in the ***courses*** table might have columns such as ***name***, ***description***, ***startDate***, ***endDate***, etc. Some of the columns might refer, or ***relate*** to other records in other tables such as the ***instructor*** column in the ***courses*** table might refer to, or relate to a particular row in the ***users*** table signifying that that particular user is the ***instructor*** of that particular course. Rows in one table relating to rows in another table is where ***relational*** databases get their name. The ***structured query language*** or ***SQL***, is a computer language commonly used to interact with relational databases. The ***query*** in ***SQL*** generally means to ***ask for***, or ***retrieve*** data that matches some criteria, often written as a ***boolean expression*** or ***predicate***.

More recently there has been a growing interest in representing and storing data using alternative strategies which have collectively come to be referred to as ***non relational databases***, or ***NoSQL databases***. Non relational databases such as ***MongoDB***, ***Firebase***, and ***Couchbase***, store their data in ***collections*** containing ***documents*** which are roughly analogous to ***tables*** and ***records*** in their relational counterparts. The biggest difference though is that the columns, or ***fields*** in the rows in relational databases generally can only contain ***primitive data types***, e.g., simple strings, numbers, dates, and booleans, whereas the fields, or ***properties*** in non relational documents can be arbitrarily ***complex data types***, e.g., strings, numbers, booleans, dates as well as combinations of these in complex objects containing arrays of objects of arrays, etc. The other big difference is that relational databases require the structure, or ***schema*** of the data to be explicitly described before storing any data, whereas non relational databases do not require predefined schemas. Instead, non relational databases delegate this responsibility to the applications using the database. The ***structure***, or schema in relational databases is where ***structured query language*** gets its name.

In the previous chapter we learned how to create an HTTP server with Node.js and integrated it with a React Web user interface application to store the application state on the server. In this chapter we expand on this idea to store the data to ***MongoDB***, a popular non relational database. The first section demonstrates how to download, install and use a local instance of the MongoDB database. The next section covers how to use the ***Mongoose*** library to integrate and program a MongoDB database with a Node.js server application. The final section describes how to deploy the database to ***Mongo Atlas***, a remote MongoDB database hosted as a cloud service.

The following figure illustrates the overall architecture of what we'll be building in this chapter. From right to left, we'll first create a MongoDB database called ***kambaz*** where we'll create several ***collections*** such as ***users***, ***courses***, ***modules***, ***assignments***, etc. We'll use the ***Mongoose*** library to connect to the database programmatically from a Node server. A Mongoose schema will describe the structure of the collections in the MongoDB database and a Mongoose model will implement generic ***CRUD*** operations. We'll create higher level functions in ***Data Access Objects*** (***DAO***s) that operate on the database, and expose those operations through Express routes as RESTful Web APIs. A React Web app will integrate with the RESTful API through a client that will allow the user interface to interact with the database.

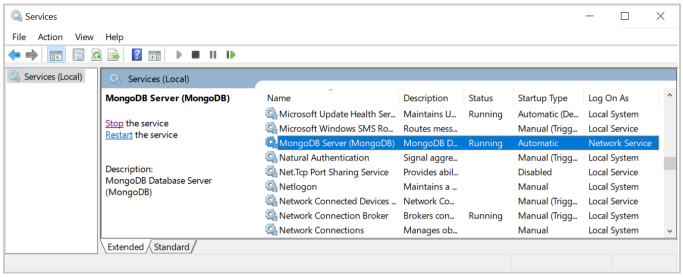


6.1 Working with a Local MongoDB Instance

MongoDB is one of an increasingly popular family of non relational databases. Data is stored in collections of documents usually formatted as JSON objects which makes it very convenient to integrate with JavaScript based frameworks such as Node.js and React. This section describes how to install, configure and get started using MongoDB.

6.1.1 Installing and Configuring MongoDB

To get started, download MongoDB for free selecting the latest version for your operating system, and click ***Download***. Run the installer and, if given the choice, choose to ***run the database as a service*** so that you don't have to bother having to restart the database sever every time you login or restart your computer. The ***MongoDB*** database will automatically start whenever you start your computer. On Windows, confirm the database is running by searching for ***MongoDB*** in the ***Services*** dialog. On macOS, confirm the database is running by clicking the ***MongoDB*** icon in the ***Systems Settings*** dialog. The service dialog gives you controls to start and stop the database, but it should already be configured to start automatically when you restart your computer.



6.1.1.1 Installing MongoDB Manually (optional)

On ***macOS*** you can install ***MongoDB*** using ***brew*** by typing the following at the command line

```
brew install mongodb-community
atlas setup
```

Alternatively you can unzip the MongoDB server from the downloaded archive to a local file system and add the right commands to your operating system ***PATH*** environment variable. On ***macOS***, unzip the file into ***/usr/local*** which creates a directory such as ***/usr/local/mongodb-macos-x86_64-5.0.3*** (your version might differ). To be able to execute the database related commands, add the path to the ***.bash_profile*** or ***.zshrc*** file located in your home directory. Add the following line in the configuration file as shown below. Your actual version might differ.

~/.bash_profile or *~/.zshrc*

```
export PATH="$PATH:/usr/local/mongodb-macos-x86_64-5.0.3/bin"
```

If the ***.bash_profile*** or ***.zshrc*** file does not exist in your home directory, create it as a plain text file, but with no extensions and a period in front of it. Configure it as shown above and then ***restart your computer***.

On ***Windows***, unzip the file into ***C:\Program Files***. To configure environment variables on ***Windows*** press the ***Windows + R*** key combination to open the ***Run*** prompt, type ***sysdm.cpl*** and press ***OK***. In the ***System Properties*** window that appears, press the ***Advanced*** tab and then the ***Environment Variables*** button. In the ***Environment Variables*** configuration window select the ***Path*** variable and press the ***Edit*** button. Copy and paste the path of the ***bin*** directory in the ***mongodb*** directory you unzipped the MongoDB download, e.g., ***"C:\Program Files\mongodb-macos-x86_64-5.0.3\bin"***. The actual path might differ. Press ***OK*** and ***restart the computer***.

6.1.1.2 Starting MongoDB from the Command Line

If you installed ***MongoDB*** as a service, it is already running in the background and can be configured and restarted in ***Windows*** from the ***Services*** dialog or from the ***System Settings*** on ***macOS***. Alternatively you can start the MongoDB server

from the command line using the ***mongod*** executable in the ***bin*** directory where you installed ***MongodDB***. First you'll need to create a ***data*** folder where the server will store all its data. You can create a data folder in your home directory as shown below.

```
cd  ~
mkdir  data
```

When you start ***MongoDB***, you'll need to tell it where the ***data*** folder is with the ***dbpath*** option. If you installed MongoDB on ***Windows*** in ***C:\Program Files\mongodb-macos-x86_64-5.0.3***, you can start MongoDB from your home directory as shown below.

```
cd  ~
C:\Program Files\mongodb-macos-x86_64-5.0.3\bin\mongod  --dbpath  data
```

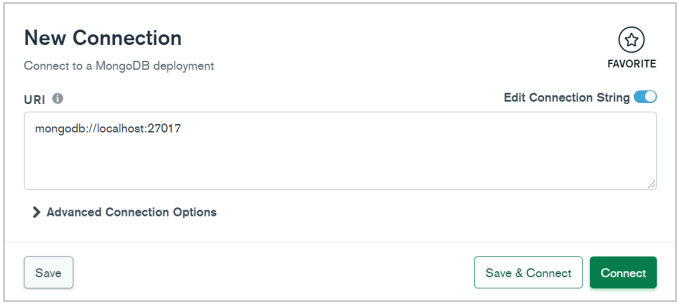
Make sure to include the ***dbpath data*** option to tell ***MongoDB*** where to find the ***data*** directory.

If you installed ***MongoDB*** on ***macOS*** in ***/usr/local/mongodb-macos-x86_64-6.0.5***, you can start ***MongoDB*** from your home directory as shown below.

```
cd  ~
/usr/local/mongodb-macos-x86_64-6.0.5/bin/mongod  --dbpath data
```

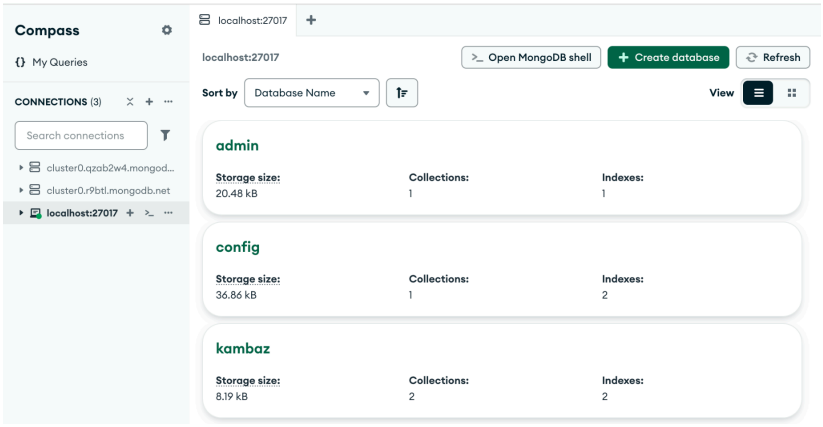
6.1.2 Using MongoDB Compass to Interact with MongoDB

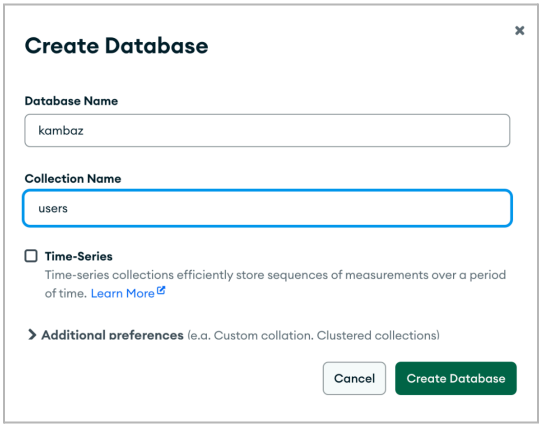
Your installation should have installed ***MongoDB Compass***, a user interface client to the MongoDB database. If not, ***MongoDB Compass*** can be downloaded from MongoDB's download page. You can start ***Compass*** from your applications folder, or search for it in your operating system's search feature. On ***macOS*** bring up ***Spotlight*** by pressing the magnifying glass on the top right menu bar, or press the ***Command*** (⌘) and ***Spacebar***. Type MongoDB ***Compass*** in the search bar and select the application from the result list. On ***Windows*** press the ***Window key*** to bring up the search field, type MongoDB ***Compass***, and select the application from the result list. When Compass comes up, confirm that the connection string ***mongodb://127.0.0.1:27017*** appears in the New Connection screen, and press ***Connect*** to connect to MongoDB.



6.1.3 Creating a MongoDB Database

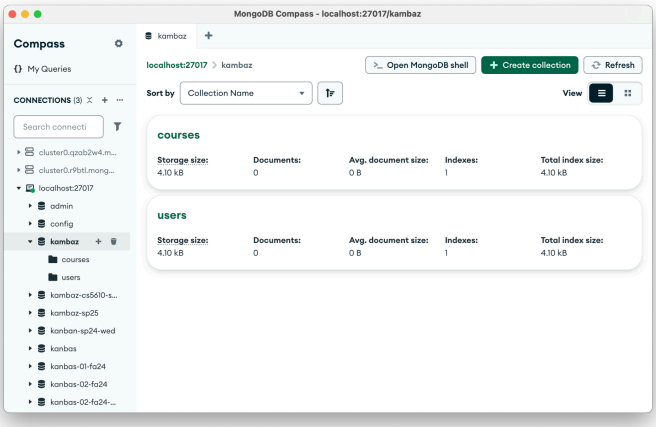
Once connected to a running MongoDB server, click on the connection on the left side bar and then click the ***Create database*** button in the tab on the right. In the ***Create Database*** dialog that appears, name your database ***kambaz*** and your first collection as ***users***. Click ***Create Database*** to create the ***kambaz*** database.

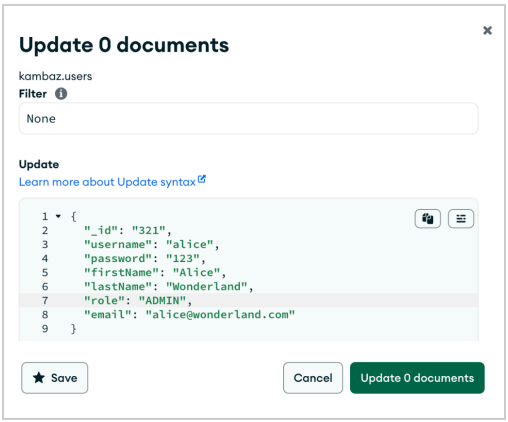




6.1.4 Inserting and Retrieving Data with Compass

In MongoDB, data is organized into ***collections***, which are analogous to ***tables*** in relational databases. Data contained in collections are referred to as ***documents***, which are analogous to ***records*** in relational databases. To create, or ***insert*** documents into a collection in a ***MongoDB*** database using ***Compass***, select the database on the left sidebar and then select the collection to insert documents into. For instance, select the ***kambaz*** database and then the ***users*** collection as shown below. On the right side, selected ***ADD DATA*** and then ***Insert document***. In the ***Insert Document*** dialog that appears, insert the document as shown below. Click ***Insert*** to insert the document and confirm the document inserted as expected.





Instead of inserting one document at a time, entire JSON files can be imported all at once. Import the users.json file we used in earlier chapters under the ***Database*** directory of the React project. To import click ***ADD DATA***, but now select ***Import JSON or CSV file***. Navigate to the location of ***users.json***, select the file and click ***Import***. Confirm the users are imported. Also find the following collections and import the JSON files linked to each of the collection names. Confirm all collections are imported: modules.json, assignments.json, courses.json.

6.2 Programming with a MongoDB Database

In the previous section we practiced interacting with the ***MongoDB*** database through the Compass graphical interface. This is all and good to make occasional simple manual updates and queries to confirm the data behaves as expected, but to applications need to interact with the database programmatically with libraries such as ***Mongoose***. The following sections describe how to install, configure, and connect a Node.js application to a ***MongoDB*** database server using the ***Mongoose*** library. The final section discusses how to configure the application to integrate to a ***MongoDB*** database

hosted in the MongoDB's ***Atlas*** cloud service. Do all your work in a new GitHub branch called ***a6*** in both the React and Node.js projects.

6.2.1 Installing and Connecting to a MongoDB Database

The ***Mongoose*** library implements a set of APIs and abstractions for applications to interact with a MongoDB database. To use the ***Mongoose*** library, install it from the root of the Node.js project as shown below.

```
$ npm  install  mongoose
```

To connect to the database server programmatically, import the Mongoose library and then use the ***connect*** function as shown below. The URL in the ***connect*** function is called the ***connection string*** and is currently referring to a ***MongoDB*** server instance running in the ***localhost*** machine (the current laptop or desktop) listening at port ***27017*** and the ***kambaz*** database existing in that server. In a later section we'll revisit the connection string and configure it to connect to a database server running in a remote machine hosted by MongoDB's ***Atlas*** cloud service.

index.js

```
import express from "express";
import mongoose from "mongoose";                          // load the mongoose library
...
const CONNECTION_STRING = "mongodb://127.0.0.1:27017/kambaz"
mongoose.connect(CONNECTION_STRING);                      // connect to the kambaz database
const app = express();
...
```

6.2.2 Configuring Connection Strings as Environment Variables

Instead of hard coding the ***connection string*** in the source code, it is better to configure it as an ***environment variable*** and then reference it from the code. This will come in handy when the server application is deployed to a remote service such as ***Render*** or ***Heroku*** and the ***connection string*** can be configure to reference the online remote database running on MongoDB's ***Atlas*** cloud service. In the ***.env*** file, declare the following connection string environment variable.

.env

```
SERVER_ENV=development
CLIENT_URL=http://localhost:3000
SERVER_URL=http://localhost:4000
SESSION_SECRET=super secret session phrase
DATABASE_CONNECTION_STRING=mongodb://127.0.0.1:27017/kambaz
```

Then in ***index.js***, read the ***connection string*** as shown below.

index.js

```
import "dotenv/config";
import express from "express";
import mongoose from "mongoose";
...
const CONNECTION_STRING = process.env.DATABASE_CONNECTION_STRING || "mongodb://127.0.0.1:27017/kambaz"
mongoose.connect(CONNECTION_STRING);
const app = express();
app.use(cors({...}));
app.use(session({...}));
app.use(express.json());
...
app.listen(process.env.PORT || 4000);
```

6.2.3 Implementing Mongoose Schemas and Models

As mentioned earlier, non relational database do not require specifying the structure, or schema of the data stored in collections like relational databases do. That responsibility has been delegated to applications using non relational databases. Once a Node.js application establishes a connection to a MongoDB database, the ***Mongoose*** API declares datatypes ***Schemas*** and ***Models*** to interact with collections. Mongoose ***Schemas*** describe the structure of the data being stored in the database and it's used to validate the data being stored or modified through the Mongoose library. The ***schema*** shown below describes the structure for the ***users*** collection imported earlier. Create the schema in a ***Users*** directory in your Node.js projects.

Kambaz/Users/schema.js

```
import mongoose from "mongoose";                                  // load the mongoose library
const userSchema = new mongoose.Schema({                          // create the schema
    _id: String,                                                  // primary key name is _id of type String
    username: { type: String, required: true, unique: true },     // String field that is required and unique
    password: { type: String, required: true },                   // String field that in required but not unique
    firstName: String,                                            // String fields
    email: String,                                                // with no additional
    lastName: String,                                             // configurations
    dob: Date,                                                    // Date field with no configurations
    role: {
      type: String,                                               // String field
      enum: ["STUDENT", "FACULTY", "ADMIN", "USER"],              // allowed string values
      default: "USER",                                            // default value if not provided
    },
    loginId: String,
    section: String,
    lastActivity: Date,
    totalActivity: String,
  },
  { collection: "users" }                                         // store data in "users" collection
);
export default userSchema;
```

6.2.4 Implementing Mongoose Models

Mongoose ***models*** implement a low level API to interact with MongoDB collections programmatically. Models provide ***CRUD*** (***Create Read Update Delete***) functions such as: ***find()***, ***create()***, ***updateOne()***, ***removeOne()***, etc. In ***Users/model.js*** below, create a Mongoose model from the users schema. The functions provided by Mongoose models are deliberately generic because they can interact with any collection configured in the schema. In the next section we'll create a ***data access object*** that implements higher level functions specific to the domain of ***kambaz***.

Kambaz/Users/model.js

```
import mongoose from "mongoose";                                  // load mongoose library
import schema from "./schema.js";                                 // load users schema
const model = mongoose.model("UserModel", schema);                // create mongoose model from the schema
export default model;                                             // export so it can be used elsewhere
```

6.2.5 Retrieving Data from MongoDB with Mongoose

The Mongoose model created in the previous section provides low level functions such as ***find***, ***create***, ***updateOne***, and ***deleteOne***, that are deliberately vague since they need to be able to operate on any collection. It is good practice to wrap these low level generic functions within higher level functions that are specific to the use cases of the specific projects. For instance instead of just using the generic ***find()*** function, it would be preferable to use something such as ***findUsers()***, ***findUserById()*** or ***findUserByUsername()***. A previous chapter implemented a ***data access object*** using arrays declared in

the ***Database/index.js*** files. This chapter refactors the ***DAO***s so they use an actual database. The following ***Users/dao.js*** re-implements the ***CRUD*** operations for the ***users*** collection written in terms of the low level Mongoose model operations.

Kambaz/Users/dao.js

```
import model from "./model.js";
import db from "../Database/index.js";
export const createUser = (user) => {} // implemented later
export const findAllUsers = () => model.find();
export const findUserById = (userId) => model.findById(userId);
export const findUserByUsername = (username) =>  model.findOne({ username: username });
export const findUserByCredentials = (username, password) =>  model.findOne({ username, password });
export const updateUser = (userId, user) =>  model.updateOne({ _id: userId }, { $set: user });
export const deleteUser = (userId) => model.deleteOne({ _id: userId });
```

6.2.6 Implementing APIs to Interact with MongoDB from a React Client Application

DAOs implement an interface between an application and the low level database access, providing a high level API to the rest of the application hiding the details and idiosyncrasies of using a particular database vendor. Likewise routes implement an interface between the HTTP network world and the JavaScript functional programming world by converting a stream of bits from a network connection request into a set of objects, maps, and function event handlers that participate in the client/server architecture of a multi tiered application.

The following sections demonstrate implementing the most ***CRUD*** (***Create Read Update*** and ***Delete***) database operations including:

- Retrieving all documents
- Retrieving documents by predicate
- Retrieving documents by primary Key
- Deleting a document
- Updating a document
- Creating a new documents

6.2.6.1 Refactoring Account Routes

Previous chapters implemented account routes such as ***signin*** and ***signup*** shown below. Since the ***DAO*** implementations used data structures imported from the local file system, the operations were ***synchronous***. Now that the ***DAO*** is interacting with a database, the operations are ***asynchronous*** and must be tagged with the ***async***/***await*** keywords as shown below. Confirm ***Signin***, ***Signup***, and ***Profile*** screens work as before. Following the examples below for the ***signin*** and ***signup*** functions, add the ***async*** keyword to all other router functions and add the ***await*** keyword to all calls to ***dao*** functions.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
export default function UserRoutes(app) {
  const signin = async (req, res) => {
    const { username, password } = req.body;
    const currentUser = await dao.findUserByCredentials(username, password);
    if (currentUser) {
      req.session["currentUser"] = currentUser;
      res.json(currentUser);
    } else {
      res.status(401).json({ message: "Unable to login. Try again later." });
    }
  };
  const signup = async (req, res) => {
    const user = await dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json({ message: "Username already taken" });
```

```
      return;
    }
    const currentUser = await dao.createUser(req.body);
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  };
  ...
  app.post("/api/users/signin", signin);
  app.post("/api/users/signup", signup);
}
```

6.2.6.2 Retrieving All Documents from MongoDB with Mongoose

DAOs implement high level data operations based on lower level ***Mongoose*** models. The ***Mongoose*** model ***find*** function retrieves all documents from a collection. The higher level ***findAllUsers*** function below uses the lower level ***find*** function to retrieve all the users from the users collection.

Kambaz/Users/dao.js

```
import model from "./model.js";
export const findAllUsers = () => model.find();
```

Routes implement ***REST***ful ***Web API***s that user interface clients can use to interact with server functionality. The route implemented below uses the ***findAllUsers*** function implemented by the DAO to retrieve all the users from the database. The route responds with the collection of users retrieved from the database. Confirm the route works by navigating to ***http://localhost:4000/api/users*** with the browser.

Kambaz/Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  const findAllUsers = async (req, res) => {
    const users = await dao.findAllUsers();
    res.json(users);
  };
  app.get("/api/users", findAllUsers);
  ...
}
```

Meanwhile in the React user interface application, in ***app/(Kambaz)/Account/client.ts***, implement the ***findAllUsers*** function shown below to send a ***GET request*** to the server and ***await*** for the server's ***response*** containing an array of users in the ***data*** property.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
export const findAllUsers = async () => {
  const response = await axiosWithCredentials.get(USERS_API);
  return response.data;
};
...
```

To display the array of users from the database, refactor the ***PeopleTable*** component to accept an optional ***users*** parameter, instead of retrieving the users from the local file system. Remove any filters since data access rules is best handled at the server.

src/Courses/People/Table/page.tsx

```
import React, { useState, useEffect } from "react";
// import * as db from "../../Database";
// import { useParams } from "react-router-dom";
export default function PeopleTable({ users = [] }: { users?: any[] }) {
  // const { cid } = useParams();
  // const { users, enrollments } = db;
  return (
    <div id="wd-people-table">
      <table className="table table-striped">
        ...
        <tbody>
          {users
            .filter((usr) => enrollments.some((enrollment) =>
              enrollment.user === usr._id && enrollment.course === cid))
            .map((user: any) => ( ... ))}
        </tbody>
      </table>
    </div>);}
```

Create a new ***Users*** screen that fetches the users from the database and displays it with the ***PeopleTable*** component as shown below.

app/(Kambaz)/Account/Users.tsx

```
import { useState, useEffect } from "react";
import { useParams } from "react-router";
import PeopleTable from "../Courses/People/Table";
import * as client from "./client";
export default function Users() {
 const [users, setUsers] = useState<any[]>([]);
 const { uid } = useParams();
 const fetchUsers = async () => {
   const users = await client.findAllUsers();
   setUsers(users);
 };
 useEffect(() => {
   fetchUsers();
 }, [uid]);
 return (
   <div>
     <h3>Users</h3>
     <PeopleTable users={users} />
   </div>
);}
```

Add a new ***Users*** route to the ***Account*** screen as shown below.

app/(Kambaz)/Account/page.tsx

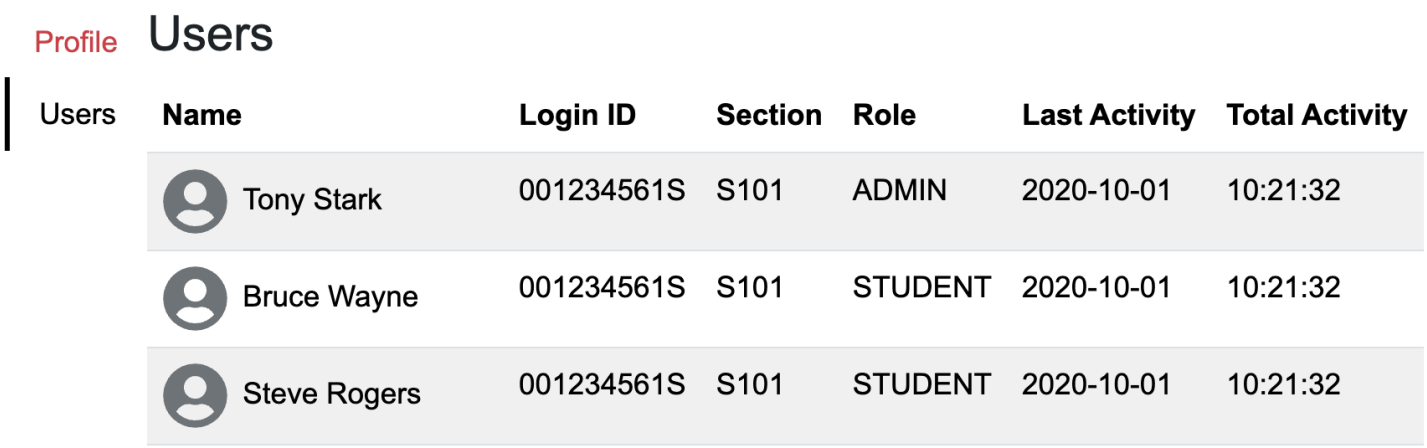
```
...
<Routes>
  <Route path="/" element={ <Navigate to={currentUser ? "/Kambaz/Account/Profile" : "/Kambaz/Account/Signin" } /> } />
  <Route path="/Signin" element={<Signin />} />
  <Route path="/Profile" element={<Profile />} />
  <Route path="/Signup" element={<Signup />} />
  <Route path="/Users" element={<Users />} />
</Routes>
...
```

Add a ***Users*** link to the ***Accounts Navigation*** sidebar that navigates to the ***Users*** screen only if the logged in user has an ***ADMIN*** role. To test, use ***Compass*** to update the ***role*** of an existing user or create a new user with ***ADMIN*** role, signin as the ***ADMIN*** user, and navigate to the ***Users*** screen. Confirm that all users are displayed.

src/Account/Navigation.tsx

```
import { Link, useLocation } from "react-router-dom";
import { useSelector } from "react-redux";
export default function AccountNavigation() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const { pathname } = useLocation();
 const active = (path: string) => (pathname.includes(path) ? "active" : "");
 return (
   <div id="wd-account-navigation" className="list-group">
     ...
     {currentUser && currentUser.role === "ADMIN" && (
       <Link href={`/Kambaz/Account/Users`} className={`list-group-item ${active("Users")}`}> Users </Link> )}
   </div>
);}
```

Logged in as an ADMIN, navigate to the new ***Users*** screen and confirm it displays all the users as shown below.



6.2.6.3 Retrieving Documents by Predicate from MongoDB with Mongoose

In the ***User***'s ***DAO***, implement ***findUsersByRole*** that filters the ***users*** collection by the ***role*** property as shown below. Mongoose model's ***find*** function takes as argument a JSON object used to pattern match documents in the collection. The ***{role: role}*** object means that documents will be filtered by their ***role*** property that matches the value ***role***.

Kambaz/Users/dao.js

```
export const findUsersByRole = (role) => model.find({ role: role }); // or just model.find({ role })
```

In the ***User routes***, refactor the ***findAllUsers*** function so that it parses the ***role*** from the query string, and then uses the ***DAO*** to retrieve users with that particular role.

Kambaz/Users/routes.js

```
  const findAllUsers = async (req, res) => {
    const { role } = req.query;
    if (role) {
      const users = await dao.findUsersByRole(role);
      res.json(users);
      return;
    }
    const users = await dao.findAllUsers();
    res.json(users);
  };
```

In the React user interface application, add ***findUserByRole*** in the client so that it encodes the ***role*** in the query string of the URL as shown below.

app/(Kambaz)/Account/client.ts

```
export const findUsersByRole = async (role: string) => {
  const response = await
    axios.get(`${USERS_API}?role=${role}`);
  return response.data;
};
```

In the ***Users*** screen, add a dropdown that invokes a ***filterUsers*** event handler function with the selected ***role***. The function updates a ***role*** state variable and requests from the server the list of users filtered by their role. Confirm that selecting various roles actually filters the users by their role.

app/(Kambaz)/Account/Users.tsx

```
export default function Users() {
  const [users, setUsers] = useState<any[]>([]);
  const [role, setRole] = useState("");
  const filterUsersByRole = async (role: string) => {
    setRole(role);
    if (role) {
      const users = await client.findUsersByRole(role);
      setUsers(users);
    } else {
      fetchUsers();
    }
  };
  const fetchUsers = async () => { ... };
  useEffect(() => { ... }, []);
  return (
    <div>
     <h3>Users</h3>
      <select value={role} onChange={(e) =>filterUsersByRole(e.target.value)}
              className="form-select float-start w-25 wd-select-role" >
        <option value="">All Roles</option>    <option value="STUDENT">Students</option>
        <option value="TA">Assistants</option> <option value="FACULTY">Faculty</option>
        <option value="ADMIN">Administrators</option>
      </select>
      ...
    </div>
);}
```

Assistants

Name	Login ID	Section	Role
Natasha Romanoff	001234564S	S101	TA
Aragorn Elessar	001234568S	S101	TA

Now practice filtering users by their ***first*** or ***lastName*** by creating a ***regular expression*** used to pattern match the ***firstName*** or ***lastName*** fields of the documents in the ***users*** collection.

Kambaz/Users/dao.js

```
export const findUsersByPartialName = (partialName) => {
  const regex = new RegExp(partialName, "i"); // 'i' makes it case-insensitive
  return model.find({
    $or: [{ firstName: { $regex: regex } }, { lastName: { $regex: regex } }],
  });
};
```

In the the ***User routes***, parse a ***name*** parameter from the ***query string*** and use it to find users whose first or last names partially match the ***name*** parameter.

Kambaz/Users/routes.js

```
  const findAllUsers = async (req, res) => {
    const { role, name } = req.query;
    if (role) { ... }
    if (name) {
      const users = await dao.findUsersByPartialName(name);
      res.json(users);
      return;
```

```
    }
    const users = await dao.findAllUsers();
    res.json(users);
  };
```

In the React client application, implement the ***findUserByPartialName*** client function as shown below which encodes a ***name*** in the ***query string*** the server can use to filter users by their ***first*** and ***lastName***.

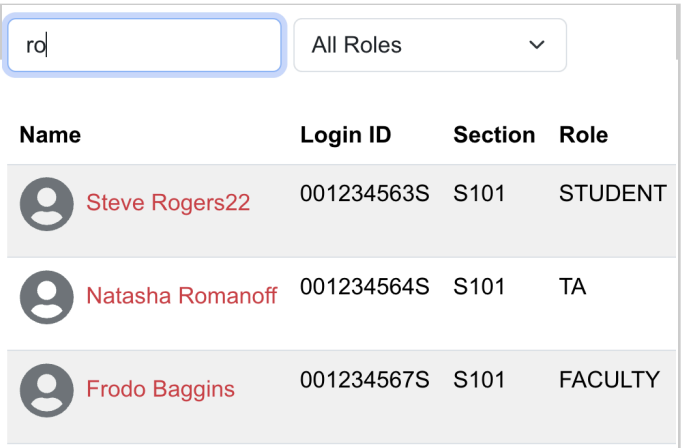
app/(Kambaz)/Account/client.ts

```
export const findUsersByPartialName = async (name: string) => {
  const response = await axios.get(`${USERS_API}?name=${name}`);
  return response.data;
};
```

In the ***Users*** screen, create a new ***name*** state variable and corresponding ***input*** field used to invoke the ***findUsersByPartialName*** client function and update the ***users*** state variable with a subset of users that match the ***name***. Confirm that typing a name in the input field actually filters the users by their first or last name. Note that the current implementation does not consider a combination of filtering by ***role*** and by ***name***. Feel free to explore how you would go about filtering by both.

app/(Kambaz)/Account/Users.tsx

```
export default function Users() {
  const [users, setUsers] = useState<any[]>([]);
  const [role, setRole] = useState("");
  const [name, setName] = useState("");
  const filterUsersByName = async (name: string) => {
    setName(name);
    if (name) {
      const users = await client.findUsersByPartialName(name);
      setUsers(users);
    } else {
      fetchUsers();
    }
  };
  ...
  return (
    <div>
     <h3>Users</h3>
      <FormControl onChange={(e) => filterUsersByName(e.target.value)} placeholder="Search people"
             className="float-start w-25 me-2 wd-filter-by-name" />
      ...
    </div>
  );
}
```



6.2.6.4 Retrieving Documents by Primary Key from MongoDB with Mongoose

A common database operation is to retrieve documents by their primary key. The ***DAO*** function below retrieves a ***user*** document by its primary key.

Kambaz/Users/dao.js

```
export const findUserById = (userId) => model.findById(userId);
```

Make the ***findUserById*** DAO function available as a RESTful Web API as shown below.

Kambaz/Users/routes.js

```
  const findUserById = async (req, res) => {
    const user = await dao.findUserById(req.params.userId);
    res.json(user);
  };
  app.get("/api/users/:userId", findUserById);
```

The user interface can then interact with the server using the ***findUserById*** client function shown below.

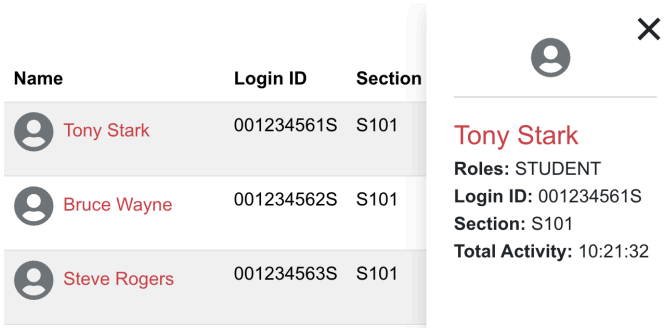
app/(Kambaz)/Account/client.ts

```
export const findUserById = async (id: string) => {
  const response = await axios.get(`${USERS_API}/${id}`);
  return response.data;
};
```

In a new ***PeopleDetails*** component, use the client's ***findUserById*** function to retrieve the user when a faculty clicks on the user's name. Parse a ***uid*** path parameter and use it to retrieve the user by their ID when the component loads. If the ***uid*** does not exist, return ***null*** so that the component does not render on the screen. In ***useEffect***, add ***uid*** as a dependency so that if the component re-renders if you click on another user while the component is still displaying.

app/(Kambaz)/Courses/[cid]/People/Details.tsx

```
import { useEffect, useState } from "react";
import { FaUserCircle } from "react-icons/fa";
import { IoCloseSharp } from "react-icons/io5";
import { useParams, useNavigate } from "react-router";
import Link from "next/link";
import * as client from "../../Account/client";
export default function PeopleDetails() {
  const { uid} = useParams();
  const [user, setUser] = useState<any>({});
  const navigate = useNavigate();
  const fetchUser = async () => {
    if (!uid) return;
    const user = await client.findUserById(uid);
    setUser(user);
  };
  useEffect(() => {
    if (uid) fetchUser();
  }, [uid]);
  if (!uid) return null;
  return (
    <div className="wd-people-details position-fixed top-0 end-0 bottom-0 bg-white p-4 shadow w-25">
      <button onClick={() => navigate(-1)} className="btn position-fixed end-0 top-0 wd-close-details">
        <IoCloseSharp className="fs-1" /> </button>
      <div className="text-center mt-2"> <FaUserCircle className="text-secondary me-2 fs-1" /> </div><hr />
      <div className="text-danger fs-4 wd-name"> {user.firstName} {user.lastName} </div>
      <b>Roles:</b>           <span className="wd-roles">         {user.role}         </span> <br />
      <b>Login ID:</b>        <span className="wd-login-id">      {user.loginId}      </span> <br />
      <b>Section:</b>         <span className="wd-section">       {user.section}      </span> <br />
      <b>Total Activity:</b>  <span className="wd-total-activity">{user.totalActivity}</span> </div> ); }
```



Add a ***close button*** rendered as an ***X*** at the top left that hides the component by navigating back to the ***Users*** screen as shown above. Confirm that clicking on the the name of a user displays the user's details. Also confirm that closing the ***PeopleDetails*** hides the component.

app/(Kambaz)/Courses/[cid]/People/Table/page.tsx

```
...
import * as client from "../../Account/client";
import PeopleDetails from "./Details";
export default function PeopleTable() {
  ...
  return (
    <div id="wd-people-table">
```

```
      <PeopleDetails />
      ...
      {users
        .map((user) => (
          <tr key={user._id}>
            <td className="wd-full-name text-nowrap">
              <Link href={`/Kambaz/Account/Users/${user._id}`} className="text-decoration-none">
                <FaUserCircle className="me-2 fs-1 text-secondary" />
                <span className="wd-first-name">{user.firstName}</span>{" "}
                <span className="wd-last-name">{user.lastName}</span>
              </Link>
            </td>
            ...
          </tr>
      ))}
      ...
    </div>
);}
```

In the ***Account*** screen, add a ***Route*** that encodes the ***uid*** path parameter in the ***URL*** that renders the same ***Users*** screen.

app/(Kambaz)/Account/page.tsx

```
...
<Routes>
  <Route path="/" element={ <Navigate to={currentUser ? "/Kambaz/Account/Profile" : "/Kambaz/Account/Signin" } /> } />
  <Route path="/Signin" element={<Signin />} />
  <Route path="/Profile" element={<Profile />} />
  <Route path="/Signup" element={<Signup />} />
  <Route path="/Users" element={<Users />} />
  <Route path="/Users/:uid" element={<Users />} />
</Routes>
...
```

6.2.6.5 Deleting a Document in MongoDB with Mongoose

To delete user documents from the ***users*** MongoDB collection, implement the ***deleteUser*** operation as shown below. The ***DAO*** function below removes a single ***user*** document from the database based on its primary key.

Kambaz/Users/dao.js ***S***

```
export const deleteUser = (userId) => model.deleteOne({ _id: userId });
```

The route below makes the ***deleteUser*** operation available as a RESTful Web API for integration with the user interface which encodes the id of the user to remove as a path parameter.

Kambaz/Users/routes.js ***S***

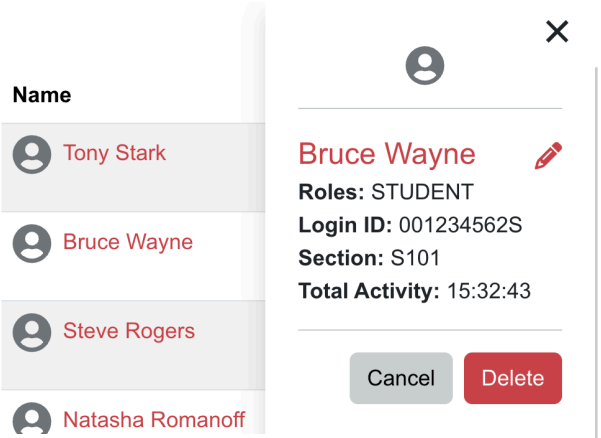
```
  const deleteUser = async (req, res) => {
      const status = await dao.deleteUser(req.params.userId);
      res.json(status);
  };
  app.delete("/api/users/:userId", deleteUser);
```

In the React Web App, implement client function that integrates with the ***deleteUser*** route in the server.

app/(Kambaz)/Account/client.ts

```
export const deleteUser = async (userId: string) => {
  const response = await axios.delete( `${USERS_API}/${userId}` );
  return response.data;
};
```

In the ***PeopleDetails*** component add buttons ***Cancel*** and ***Delete*** as shown below. The ***Delete*** button invokes a new ***deleteUser*** event handler function with ***uid***, the ID of the user to delete. Pass a reference to ***fetchUsers*** as a parameter so that ***PeopleDetails*** can notify ***PeopleTable*** that a user has been remove and that the list of users must be updated.



app/(Kambaz)/Courses/[cid]/People/Details.tsx

```
...
import { useNavigate, useParams } from "react-router";
export default function PeopleDetails() {
  const navigate = useNavigate();
  const deleteUser = async (uid: string) => {
    await client.deleteUser(uid);
    navigate(-1);
  };
  ...
  return (
    <div className="...">
      ...
      <hr />
      <button onClick={() => deleteUser(uid)} className="btn btn-danger float-end wd-delete" > Delete </button>
      <button onClick={() => navigate(-1)}
              className="btn btn-secondary float-start float-end me-2 wd-cancel" > Cancel </button>
    </div>
);}
```

Use the ***client***'s ***deleteUser*** to remove the user, and navigate back to hide the details component. The ***Cancel*** button just hides the details without removing any documents. Confirm that clicking the ***Cancel*** and ***Delete*** buttons actually work.

6.2.6.6 Updating a Document in MongoDB with Mongoose

The ***Mongoose update*** function updates documents in ***MongoDB*** databases. In the ***User***'s ***DAO***, implement ***updateUser*** as shown below to update a single document by first identifying it by its primary key, and then updating the matching fields in the ***user*** parameter.

Kambaz/Users/dao.js S

```
import model from "./model.js";
export const updateUser = (userId, user) => model.updateOne({ _id: userId }, { $set: user });
```

In the ***User***'s ***routes***, make the ***DAO*** function available as a RESTful Web API as shown below. Map a route that accepts a user's primary key as a path parameter, passes the ID and request body to the DAO function and responds with the status.

Kambaz/Users/routes.js S

```
export default function UserRoutes(app) {
  ...
  const updateUser = async (req, res) => {
    const { userId } = req.params;
    const userUpdates = req.body;
    await dao.updateUser(userId, userUpdates);
    const currentUser = req.session["currentUser"];
   if (currentUser && currentUser._id === userId) {
     req.session["currentUser"] = { ...currentUser, ...userUpdates };
   }
    res.json(currentUser);
  };
  app.put("/api/users/:userId", updateUser);
  ...
}
```

In the React client application, the client function ***updateUser*** sends user updates to the server to be saved to the database.

app/(Kambaz)/Account/client.ts

```
export const updateUser = async (user: any) => {
  const response = await axiosWithCredentials.put(`${USERS_API}/${user._id}`, user);
  return response.data;
};
```

In the ***PeopleDetails*** component, add a ***name*** state variable to edit the ***first*** and ***last*** name of the user. Also add a ***editing*** state variable to toggle the input field that edits the ***name***. With the ***navigate*** routing function, navigate back to ***PeopleTable*** once the update is done. Create a new ***saveUser*** function that splits the ***name*** state variable into the ***firstName*** and ***lastName*** and sends an updated version of the user to the server. Also update the local ***user*** state variable, turn off editing, and ***navigate*** back to the ***PeopleTable***.

app/(Kambaz)/Courses/[cid]/People/Details.tsx

```
...
import { FaPencil } from "react-icons/fa6";
import { FaCheck, FaUserCircle } from "react-icons/fa";
export default function PeopleDetails() {
  ...
  const [name, setName] = useState("");                          // to edit the user's first and last name
  const [editing, setEditing] = useState(false);                 // whether we are editing or not
  const saveUser = async () => {                                 // to save updates to user's name
    const [firstName, lastName] = name.split(" ");               // split the name into an array and get first
    const updatedUser = { ...user, firstName, lastName };        // and last and create new version of user overriding
    await client.updateUser(updatedUser);                        // first and last. Send update to server
    setUser(updatedUser);                                        // update local copy of the user
    setEditing(false);                                           // we're done editing
    navigate(-1);                                                // go back to PeopleTable
  };
  ...
  return (
      ...
      <div className="text-danger fs-4">
        {!editing && (                                           // if not editing show pencil icon
          <FaPencil onClick={() => setEditing(true)}             // clicking pencil turns on editing and hides pencil
              className="float-end fs-5 mt-2 wd-edit" /> )}
        {editing && (
          <FaCheck onClick={() => saveUser()}                    // if editing show check mark. Clicking check turns
              className="float-end fs-5 mt-2 me-2 wd-save" /> )}  // off editing, saves and hides check
        {!editing && (                                           // if not editing show first and last name
          <div className="wd-name"                               // clicking on name turns on editing
               onClick={() => setEditing(true)}>
            {user.firstName} {user.lastName}</div>)}
        {user && editing && (                                    // if editing show input field to edit name
          <FormControl className="w-50 wd-edit-name"
            defaultValue={`${user.firstName} ${user.lastName}`}  // name is initially concatenation of first and last
            onChange={(e) => setName(e.target.value)}            // update name as we type
            onKeyDown={(e) => {
              if (e.key === "Enter") { saveUser(); }}}/>)}       // save the user if Enter key is pressed
      </div>
      ...
);}
```

Add ***pencil*** and ***check*** icons to turn ***editing*** on and off. Hide each icon based on the ***editing*** boolean state variable. Clicking the name of the user also turns ***editing*** on. If ***editing*** is on, hide the user's name and instead display an input field that shows the current user's ***firstName*** and ***lastName*** and edits the ***name*** state variable. Pressing the ***Enter*** key saves the updated user's details. Confirm users can be edited. On your own, add the ability to edit a user's email and role. Use an input field of type ***email*** to edit the email. To edit the user's role, use a dropdown similar to the one used to filter users by their role.

6.2.6.7 Creating New Documents in MongoDB with Mongoose

In the ***User***'s ***DAO***, implement the ***createUser*** function as shown below to insert a new ***user*** object into the ***users*** collection.

Kambaz/Users/dao.js ***S***

```
export const createUser = (user) => {
  const newUser = { ...user, _id: uuidv4() };
  return model.create(newUser);                          // insert new user into the database
}
```

Make sure the incoming ***user*** object does not have an ***_id*** property since it can interfere with the database insert operation. In the ***User***'s ***routes***, make the DAO operation available as a RESTful Web API for the user interface to interact with. The new user is posted to the route in the request's body. The DAO ***createUser*** function inserts the new user into the database and returns the newly inserted ***user*** which is sent back to the user interface in the response.

Kambaz/Users/routes.js ***S***

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  ...
  const createUser = async (req, res) => {
    const user = await dao.createUser(req.body);
    res.json(user);
  };
  app.post("/api/users", createUser);
  ...
}
```

In the React client application, implement a ***createUser*** client function to interact with the route created above. Post the new user object to the server as shown below.

app/(Kambaz)/Account/client.ts

```
export const createUser = async (user: any) => {
  const response = await axios.post(`${USERS_API}`, user);
  return response.data;
};
```

In the ***Users*** screen, implement a new ***createUser*** event handler that sends a new user object to be inserted in the database. Use default values for the fields as shown and confirm that clicking the new ***+ People*** button actually creates the new user. Optionally implement editing those fields in the ***PeopleDetails*** component.

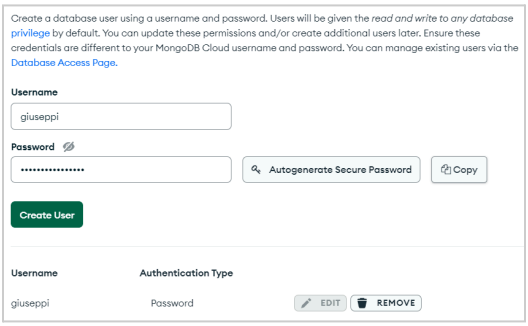
app/(Kambaz)/Account/Users.tsx

```
export default function Users() {
  const { uid } = useParams();
  const [users, setUsers] = useState<any[]>([]);
  const [role, setRole] = useState("");
  const [name, setName] = useState("");
  const createUser = async () => {
    const user = await client.createUser({
      firstName: "New",
      lastName: `User${users.length + 1}`,
      username: `newuser${Date.now()}`,
      password: "password123",
      email: `email${users.length + 1}@neu.edu`,
      section: "S101",
      role: "STUDENT",
    });
    setUsers([...users, user]);
  };
  return (
    <div>
      <button onClick={createUser} className="float-end btn btn-danger wd-add-people">
        <FaPlus className="me-2" />
        Users
```

```
      </button>
      ...
    </div>
);}
```

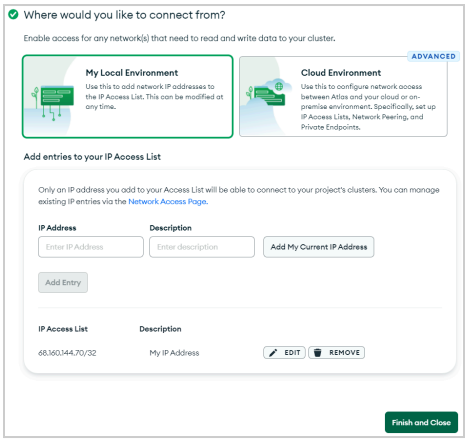
6.3 Integrating with MongoDB Hosted in Atlas Cloud Service

When you run your server on your development environment, it should be connecting to a MongoDB instance running on the same local development computer. When you deploy the server on a remote server such as ***Render***, ***Heroku*** or ***AWS***, the server needs to connect to a database that is also hosted on a public site. ***MongoDB Atlas Cloud Service*** provides a hosted database service where a MongoDB instances run on public servers, and they provide a connection string to integrate our Node.js application. This section describes setting up and deploying the database online and then integrating with it from our Node.js server running on Heroku.



6.3.1 Setting up MongoDB Atlas

To get started, head over to https://www.mongodb.com/ and click on ***Sign in*** at the top right corner. Login with your ***Google*** account or click on ***Sign Up*** to create an account with an email and password. If you get a validation email, confirm it and login. Answer any general questions if asked during the sign up process. In the ***Deploy your cluster*** screen choose a ***Free*** plan for now which should be enough for this course. Name your cluster ***Kambaz***. In the ***Provider*** section, choose any of the cloud providers and in the ***Region*** section choose a region close to your geographic area, for instance ***AWS*** and ***North Virginia***, and then click ***Create Deployment***. In the ***Connect to Kabas*** screen, in the ***Create a database user*** section create credentials to login to your database. In the ***Username*** and ***Password*** fields, type credentials you'll remember later since these are the credentials ***mongoose*** will use to login to the database from your Node.js server application when running on Render or Heroku. If you forget these credentials you'll need to create new ones later. I went with ***giuseppi*** and ***supersecretpassword*** :) and clicked ***Create Database User***.



6.3.1.1 Connecting to a Remote Database from Compass

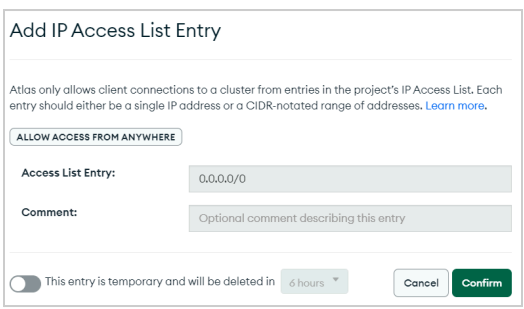
Then click the ***Choose a connection method*** button, and in the ***Access your data through tools***, select ***Compass***. In the ***Connecting with MongoDB Compass*** screen, in the ***Copy the connection string, then open MongoDB Compass*** section, copy the connection string which should look something like so

```
mongodb+srv://giuseppi:supersecretpassword@kambaz.jxui0bc.mongodb.net/
```

Click ***Done***. From ***Compass*** select ***Connect***, ***New Window***, paste the connection string in the ***URI*** field, and then click connect. Following the same steps used earlier, import the ***courses***, ***modules***, and ***users*** into the remote database.

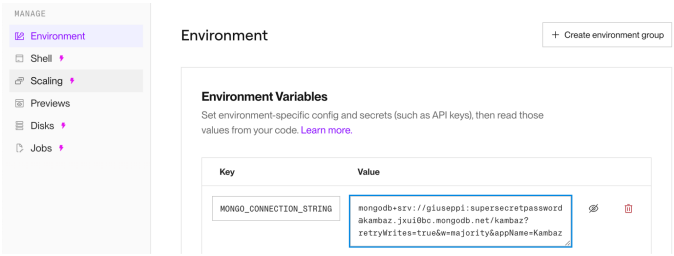
6.3.1.2 Connecting from Node.js

In the ***Atlas*** main window click on ***Network Access*** and in the ***Network Access*** screen ***IP Access List*** tab, select ***+ ADD IP ADDRESS***. In the ***Add IP Access List Entry*** dialog click on ***ALLOW ACCESS FROM ANYWHERE***. This



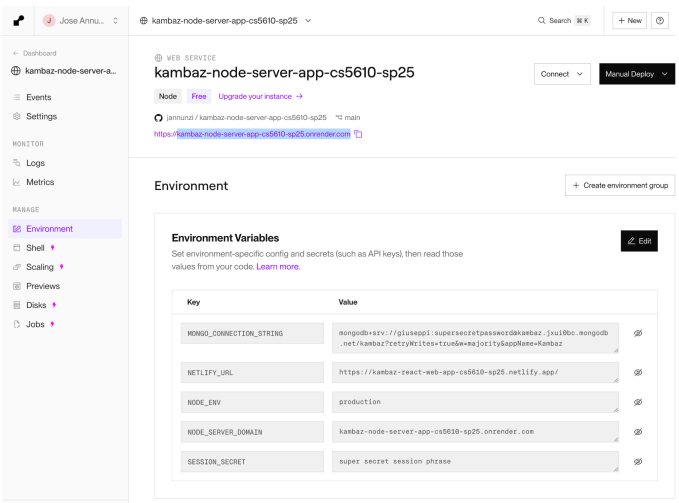
will add ***0.0.0.0/0*** to the ***Access List Entry***, allowing any computer to connect. Click ***Confirm*** and verify the new entry appears in the ***Network Access*** screen. Click ***Database*** on the left and in the ***Clusters*** screen, click ***Connect***. In the ***Connect to Kambaz*** dialog, in the ***Connect to your application***, select ***Drivers and version*** section, confirm the ***Driver*** is set to ***Node.js*** and ***Version*** is set to ***5.5 or later***. In the ***Add your connection string into your application code*** section, copy the the URL. It should look similar to the following.

Node MongoDB Database Connection String
`mongodb+srv://giuseppi:<password>@kambaz.jxui0bc.mongodb.net/kambaz?retryWrites=true&w=majority&appName=Kambaz`



Commit and push your code to a branch called ***a6*** and deploy the server application to a new remote service running on ***Render***, ***Heroku***, or ***AWS***. Make sure not to deploy to the server for prior chapters since TAs might still be grading it. In the new remote server, configure an environment variable called ***DATABASE_CONNECTION_STRING*** with the URL value above. For instance, in the ***Render*** dashboard click on ***Environment***, type ***DATABASE_CONNECTION_STRING*** in the ***Key*** field and the URL in the ***Value*** field. Replace the ***<password>*** with the actual password created in an earlier step. Commit and deploy the React application to a new ***a6*** branch. Configure the ***VITE_HTTP_SERVER*** to point to the new remote server. For the environment variables to take effect, you might need to redeploy and/or restart the remote Node server on Render as well as the remote React application server on Vercel. ***NOTE***: the ***DATABASE_CONNECTION_STRING*** environment variable must include the name of the database at the end of the path, e.g., between the last slash (/) and the question mark (?). The above example highlights the name of the database ***kambaz*** in red.

6.3.2 Configuring Session in Remote Servers



The local Node server was configured to support multiple sessions. Similarly, the remote server also needs to be configured to support sessions. In ***Render.com***, navigate to the ***Environment*** section of the application dashboard. Click ***Add Environment Variable***, type the name of the variables in the ***Key*** column and the variable's value in the ***Value*** column. Repeat for each of the environment variables as shown here on the right. When environment variables change, the server must be restarted by clicking ***Manual Deploy*** and then ***Deploy latest commit***. Below is an example of the environment variables used to configure a remote server running on ***Render.com***. Use the same ***Environment Variable*** keys shown on the left column below. Don't use the values shown on the right column below. Instead use the values for your Mongo Database, your Vercel remote server, and your remote Node server. Note: ***SERVER_URL*** should not start with ***https://***, remove it if present.

Environment Variable	*Value*
DATABASE_CONNECTION_STRING	mongodb+srv://giuseppi:supersecretpassword@kambaz.jxui0bc.mongodb.net/***kambaz***?retryWrites=true&w=majority&appName=Kambaz
CLIENT_URL	https://a6--kambaz-next-js-sp25.Vercel.app
SERVER_URL	kambaz-node-server-app-sp25.onrender.com
SERVER_ENV	production
SESSION_SECRET	what ever

6.4 Integrating the Kambaz Web Application with a Database

The current ***Kambaz*** implementation renders various courses, modules and assignments using ***JSON*** files. The files are wrapped into a Database data structure that first lived in the React application and then in the Node server. It is time for the data to live where it belongs. This section demonstrates migrating the ***courses*** and ***modules*** into corresponding collections in the ***kambaz*** MongoDB database. Create the ***Mongoose*** schemas, models and DAOs, and refactor the ***REST***ful Web APIs to ***CRUD*** courses and modules in the database. Confirm that all ***courses*** and ***modules CRUD*** functionality works as expected. Optionally also migrate the assignments into a collection and confirm that all assignment ***CRUD*** operations work.

6.4.1 Storing Courses in a Database

The current server application implements RESTful Web APIs to access courses stored in the ***courses.js*** file. This section replaces the source of the courses from a ***MongoDB courses*** collection. The basic operations on any data source are create, read, update and delete, colloquially referred to as ***CRUD*** operations. The following sections demonstrate how to implement several ***CRUD*** operations for courses.

6.4.1.1 Retrieving Courses from a Database

To access the ***courses*** collection created earlier, create a ***mongoose schema*** file as shown below. The schema file describes the ***constraints*** of the ***documents*** stored in a ***collection***, such as the names of the ***properties***, their data types, and the name of the collection where the documents will be stored.

Kambaz/Courses/schema.js ***S***

```
import mongoose from "mongoose";
const courseSchema = new mongoose.Schema({
   _id: String,
   name: String,
   number: String,
   credits: Number,
   description: String,
 },
 { collection: "courses" }
);
export default courseSchema;
```

Using the ***mongoose schema*** file, create a ***mongoose model*** file as shown below. Mongoose models provide functions to interact with the collection such as ***find()***, ***create()***, ***updateOne()***, and ***deleteOne()***. The ***CourseModel*** in the mongoose model declares a unique name that can be used as a reference from other mongoose schemas.

Kambaz/Courses/model.js ***S***

```
import mongoose from "mongoose";
import schema from "./schema.js";
const model = mongoose.model("CourseModel", schema);
export default model;
```

Using the ***mongoose model***, create a ***DAO*** file that implements various ***CRUD*** operations to interact with the ***courses*** collection. Start by refactoring the ***findAllCourses()*** function to retrieve all the courses from the database instead of the ***Database*** file as shown below. The ***model.find()*** function returns an array containing all the courses documents in the ***courses*** collection.

Kambaz/Courses/dao.js *S*

```
// import Database from "../Database/index.js";
import model from "./model.js";

export function findAllCourses() {
//   return Database.courses;
 return model.find();
}
...
```

The functions in ***mongoose models*** all return ***promises*** allowing ***asynchronous*** communication with the MongoDB server. In the Courses ***routes*** file, declare all router functions as asynchronous by adding the ***async*** keyword in front of the router functions as shown below. Also add the ***await*** keyword in front of all asynchronous calls of the ***DAO*** functions such as ***dao.findAllCourses()*** as shown below.

Kambaz/Courses/routes.js *S*

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const findAllCourses = async (req, res) => {
    const courses = await dao.findAllCourses();
    res.send(courses);
  }
  ...
}
```

In the React project, refactor the ***fetchAllCourses*** client function so that it uses a version of ***axios*** that includes cookies in the request, as shown below. Confirm you can access all courses from the browser: http://localhost:4000/api/courses.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const fetchAllCourses = async () => {
 const { data } = await axiosWithCredentials.get(COURSES_API);
 return data;
};
```

In the ***Kambaz*** root component, import ***course client*** and refactor the ***fetchCourse*** function to fetch all the courses, as shown below.

app/(Kambaz)/page.tsx

```
import { useEffect, useState } from "react";
import * as courseClient from "./Courses/client";
export default function Kambaz() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const [courses, setCourses] = useState<any[]>([]);
 const fetchCourses = async () => {
   try {
     const courses = await courseClient.fetchAllCourses();
     setCourses(courses);
   } catch (error) {
     console.error(error);
   }
 };
 useEffect(() => {
   fetchCourses();
 }, [currentUser]);
 return ( ... );
}
```

In the ***Dashboard*** screen, remove the ***filter*** since filtering is best implemented on the server. Start the database, server, and client application, sign in on the React application, and navigate to the Dashboard screen. Confirm that all courses are rendered.

app/(Kambaz)/Dashboard/page.tsx

```
export default function Dashboard({ ... }) {
 return (
   <div id="wd-dashboard">
     <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
     <div id="wd-dashboard-courses" className="row">
       <div className="row row-cols-1 row-cols-md-5 g-4">
         {courses
           .filter((course) => ( ... ))
           .map((course) => ( ... ))}
       </div>
   </div>
);}
```

6.4.1.2 Inserting Courses into a Database

Refactor the ***DAO***'s ***createCourse()*** function to insert new courses into the database with the ***mongoose model*** as shown below.

Kambaz/Courses/dao.js ***S***

```
import model from "./model.js";
export function createCourse(course) {
  const newCourse = { ...course, _id: uuidv4() };
  return model.create(newCourse);
// const newCourse = { ...course, _id: uuidv4() };
// Database.courses = [...Database.courses, newCourse];
// return newCourse;
}
...
```

Refactor the Courses routes file by adding keywords ***async*** and ***await*** before the route and DAO functions as shown below.

Kambaz/Courses/routes.js ***S***

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const createCourse = async (req, res) => {
    const course = await dao.createCourse(req.body);
    res.json(course);
  }
  app.post("/api/courses", createCourse);
  ...
}
```

In the React Courses client file, refactor the ***createCourse*** client function so that it uses the axios instance with credentials as shown below.

Kambaz/Courses/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const createCourse = async (course: any) => {
 const { data } = await axiosWithCredentials.post(COURSES_API, course);
```

```
 return data;
};
```

Refactor the Kambaz root component so that it uses the ***createCourse*** client function in the Course client file as shown below. Confirm that new courses are inserted in the courses collection in the database.

Kambaz/index.js

```
import { useEffect, useState } from "react";
import * as courseClient from "./Courses/client";
export default function Kambaz() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const [course, setCourse] = useState<any>({ ... });

 const addNewCourse = async () => {
   // const newCourse = await userClient.createCourse(course);
   const newCourse = await courseClient.createCourse(course);
   setCourses([...courses, newCourse]);
 };
 ...
 return ( ... );
}
```

6.4.1.3 Deleting Courses from the Database

Refactor the ***deleteCourse() DAO*** function to delete courses from the database by using the courses model as shown below.

Kambaz/Courses/dao.js

```
export function deleteCourse(courseId) {
 return model.deleteOne({ _id: courseId });
}
```

In the Courses routes file, refactor the route that deletes courses by adding keywords ***async*** and ***await*** in front of the route and ***DAO*** functions as shown below.

Kambaz/Courses/routes.js

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const deleteCourse = async (req, res) => {
    const { courseId } = req.params;
    const status = await dao.deleteCourse(courseId);
    res.send(status);
  }
  app.delete("/api/courses/:courseId", deleteCourse);
 ...
}
```

In the React application, refactor the ***deleteCourse()*** client function so that it uses the axios instance with credentials.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const deleteCourse = async (id: string) => {
 const { data } = await axiosWithCredentials.delete(`${COURSES_API}/${id}`);
 return data;
};
```

```
...
```

Refactor the ***Kambaz*** root component by adding keywords ***async*** and ***await*** in front of the ***deleteCourse*** event handler function and Course client function as shown below. Confirm that courses are removed from the ***courses*** collection.

app/(Kambaz)/index.ts

```
import { useSelector } from "react-redux";
import { useEffect, useState } from "react";
import * as courseClient from "./Courses/client";
export default function Kambaz() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const [courses, setCourses] = useState<any[]>([]);
 const deleteCourse = async (courseId: string) => {
   const status = await courseClient.deleteCourse(courseId);
   setCourses(courses.filter((course) => course._id !== courseId));
 };
 ...
```

6.4.1.4 Updating Courses in the Database

In the ***Courses DAO***, refactor the ***updateCourses*** function to update courses in the database with the ***model.updateOne()*** function as shown below.

Kambaz/Courses/dao.js ***S***

```
import model from "./model.js";
export function updateCourse(courseId, courseUpdates) {
 return model.updateOne({ _id: courseId }, { $set: courseUpdates });
// const { courses } = Database;
// const course = courses.find((course) => course._id === courseId);
// Object.assign(course, courseUpdates);
// return course;
}
```

In the ***Courses routes***, refactor the routing function by adding ***async*** and ***await*** keywords before the routing and DAO function calls as shown below.

Kambaz/Courses/routes.js ***S***

```
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const updateCourse = async (req, res) => {
    const { courseId } = req.params;
    const courseUpdates = req.body;
    const status = await dao.updateCourse(courseId, courseUpdates);
    res.send(status);
  }
  app.put("/api/courses/:courseId", updateCourse);
  ...
}
```

In the ***Courses client*** file, refactor the client function to use ***axios*** with credentials as shown below. Confirm that updating the title of a course is persisted in the database.

app/(Kambaz)/Courses/client.ts

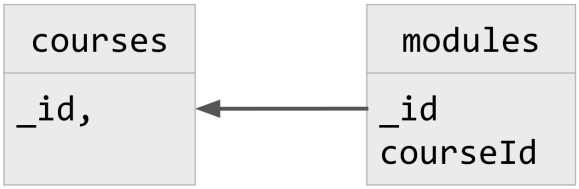
```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
```

```
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const updateCourse = async (course: any) => {
 const { data } = await axiosWithCredentials.put(`${COURSES_API}/${course._id}`, course);
 return data;
};
```

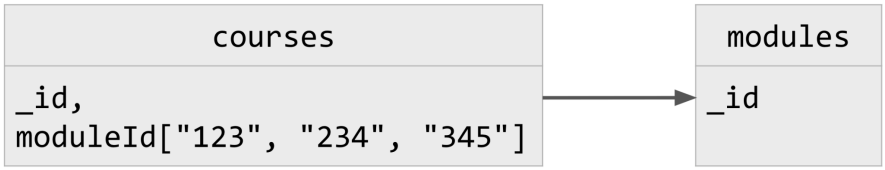
6.4.2 Persisting Modules in a Database as One to Many Relations with Courses

In ***Kambaz***, each course contains several modules, establishing a ***one to many relationship***. The relationship is implemented by each module having a field that refers to the course they below to. This section demonstrates how to use Mongoose to implement ***one to many relationships***. In UML, one to many relationships can be illustrated as shown below.

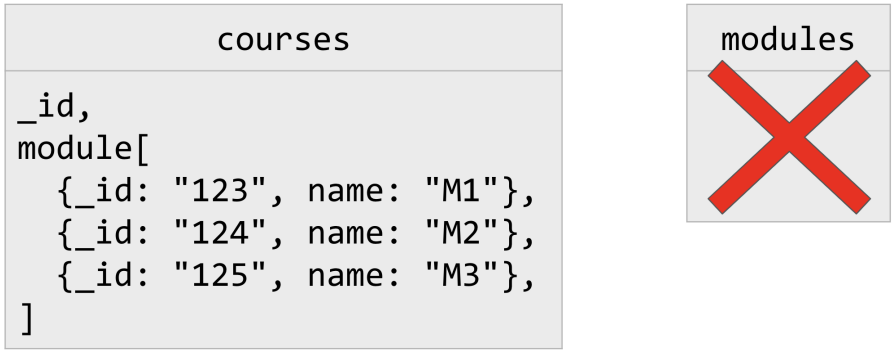
The ***courses*** collection is said to be on the one side of a one to many relation and the ***modules*** collection is said to be on the many side of a one to many relationship. The relationship describes that each course is related to many modules. It is often useful to think of the relationship as a ***parent child relationship*** describing it as ***courses are the parents of many modules***. Another way to think of the relationship is as an ownership relationship as in ***courses have many modules***. The easiest way to implement one to many relationships in both relational and non-relational databases, is to use foreign keys referencing related records. In the diagram below, module documents contain field ***courseId*** whose value is the ***_id*** of some course document the module belongs to.



In non-relational databases there are two additional alternatives to implement one to many relationships. One way is to include an array of foreign keys in the parent document that reference all the child documents. In the diagram below documents in the ***courses*** collection contain a ***moduleId*** array that contains the values of primary keys of child module documents. The keys in the ***moduleId*** array can be used to retrieve the actual documents from the ***modules*** collection.



Another alternative implementation is do away entirely of the collection on the many side (***modules***) and embed the documents in the collection on the one side (***courses***). In the diagram below the ***modules*** collection is removed and the documents are instead embedded in the corresponding parent course document in a ***module*** array.



6.4.2.1 Declaring One to Many Relationships

Although there many alternatives to implement one to many relationships, this section describes the simplest implementation where the collection on the many side (***modules***) defines a foreign key referencing documents on the one side (***courses***). In Mongoose, relationships are implemented by declaring fields as references and providing the name of the model that it is related to. To demonstrate, create the ***schema*** file below that describes the data structure of ***module*** documents stored in a collection called ***modules***. The ***course*** field is declared as type ***String*** to store a ***primary key*** value. The ***ref*** property set to the value ***CourseModel*** is the same value as the name of the ***courses model*** stored in the ***courses*** collection, declared in an earlier section. The ***ref*** property establishes that the ***primary key*** stored in ***course*** refers to a document stored in the ***courses*** collection, effectively implementing a ***one to many*** relation.

Kambaz/Modules/schema.js S

```
import mongoose from "mongoose";
const schema = new mongoose.Schema(
  {
    _id: String,
    name: String,
    description: String,
    course: { type: String, ref: "CourseModel" },
  },
  { collection: "modules" }
);
export default schema;
```

In a ***Modules model*** file implement a ***mongoose model*** to interact with the ***module*** collection in the database. The ***model*** implements various functions to implement ***CRUD*** operations such as ***find()***, ***create()***, ***deleteOne()***, ***updateOne()***.

Kambaz/Modules/model.js S

```
import mongoose from "mongoose";
import schema from "./schema.js";
const model = mongoose.model("ModuleModel", schema);
export default model;
```

6.4.2.2 Retrieving Modules for a Course

Import the ***modules.json*** file into the ***MongoDB*** database making sure that the module's ***course*** field are set to the same values of ***_id*** fields of courses in the ***courses*** collection. For instance, there should be a course whose ***_id*** is set to ***RS101***, and the modules whose ***_id*** are ***M101***, ***M102***, and ***M103*** should have have their ***course*** field set to ***RS101***. Refactor the ***Modules DAO*** so that it uses the ***mongoose model*** to retrieve modules for a course from the database as shown below.

Kambaz/Modules/dao.js S

```
import model from "./model.js";
export function findModulesForCourse(courseId) {
 return model.find({ course: courseId });
 // const { modules } = Database;
 // return modules.filter((module) => module.course === courseId);
}
```

In the ***Courses routes***, refactor the routing function by adding ***async*** and ***await*** keywords before the routing and DAO function calls as shown below.

Kambaz/Courses/routes.js ***S***

```
import * as modulesDao from "../Modules/dao.js";
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const findModulesForCourse = async (req, res) => {
    const { courseId } = req.params;
    const modules = await modulesDao.findModulesForCourse(courseId);
    res.json(modules);
  }
  app.get("/api/courses/:courseId/modules", findModulesForCourse);
  ...
}
```

In the ***Courses client*** file, refactor the client function to use ***axios*** with credentials as shown below. Confirm that modules are retrieved and displayed from the database for the correct course.

app/(Kambaz)/Courses/client.js

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const findModulesForCourse = async (courseId: string) => {
 const response = await axiosWithCredentials.get(`${COURSES_API}/${courseId}/modules`);
 return response.data;
};
```

Add a ***setModules*** reducer function to the modules reducer to populate the reducer's modules from the database.

app/(Kambaz)/Courses/[cid]/Modules/reducer.ts

```
import { createSlice } from "@reduxjs/toolkit";
import { db } from "../../Database";
const initialState = {
 modules: db.modules,
};
const modulesSlice = createSlice({
 name: "modules",
 initialState,
 reducers: {
   ...
   setModules: (state, { payload: modules }) => {
     state.modules = modules;
   },
 },
});

export const { addModule, deleteModule, updateModule, editModule, setModules } =
 modulesSlice.actions;
export default modulesSlice.reducer;
```

In the ***Modules*** screen, use the new ***findModulesForCourse*** client function to fetch the modules for the course and populate the modules in the reducer using the ***setModules*** reducer function. Navigate to various courses and confirm that the modules displayed are the ones for the course.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { addModule, editModule, updateModule, deleteModule, setModules } from "./reducer";
import { useSelector, useDispatch } from "react-redux";
import { useState, useEffect } from "react";
import * as courseClient from "../client";

export default function Modules() {
 const { cid } = useParams();
 const { modules } = useSelector((state: any) => state.modulesReducer);
 const dispatch = useDispatch();
 const fetchModulesForCourse = async () => {
   const modules = await courseClient.findModulesForCourse(cid!);
   dispatch(setModules(modules));
 };
 useEffect(() => {
   fetchModulesForCourse();
 }, [cid]);
 ...
 return ( ... );
}
```

6.4.2.3 Creating Modules for a Course

Refactor the ***Modules DAO*** so that it uses the ***mongoose model*** to insert a new module into the database as shown below.

Kambaz/Modules/dao.js ***S***

```
import Database from "../Database/index.js";
import model from "./model.js";
export function createModule(module) {
 const newModule = { ...module, _id: uuidv4() };
 return model.create(newModule);
 // Database.modules = [...Database.modules, newModule];
 // return newModule;
}
```

In the ***Courses routes***, refactor the routing function by adding ***async*** and ***await*** keywords before the routing and DAO function calls as shown below.

Kambaz/Courses/routes.js ***S***

```
import * as modulesDao from "../Modules/dao.js";
import * as dao from "./dao.js";
export default function CourseRoutes(app) {
  ...
  const createModuleForCourse = async (req, res) => {
    const { courseId } = req.params;
    const module = {
      ...req.body,
      course: courseId,
    };
    const newModule = await modulesDao.createModule(module);
    res.send(newModule);
  }
  app.post("/api/courses/:courseId/modules", createModuleForCourse);
  ...
}
```

In the ***Courses client*** file, refactor the client function to use ***axios*** with credentials as shown below. Confirm that new modules are created in the database and that the ***course*** field is set to the ***primary key*** of the parent ***course***.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;
export const createModuleForCourse = async (courseId: string, module: any) => {
 const response = await axiosWithCredentials.post(
   `${COURSES_API}/${courseId}/modules`,
   module
 );
 return response.data;
};
```

In the ***Modules*** component, refactor the screen to post a new module when the user adds a new module. Use the new client function to send the new module to the server and then add the module to the reducer to render on the screen.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { addModule, setModules } from "./reducer";
export default function Modules() {
 const { cid } = useParams();
 const [moduleName, setModuleName] = useState("");
 const dispatch = useDispatch();
 const addModuleHandler = async () => {
   const newModule = await courseClient.createModuleForCourse(cid!, {
     name: moduleName,
     course: cid,
   });
   dispatch(addModule(newModule));
   setModuleName("");
 };
 return (
   <div>
     <ModulesControls
       addModule={addModuleHandler}
       moduleName={moduleName}
       setModuleName={setModuleName}
     />
     <br /> ...
);}
```

6.4.2.4 Deleting Modules

Refactor the ***Modules DAO*** so that it uses the ***mongoose model*** to delete modules from the database as shown below.

Kambaz/Modules/dao.js ***S***

```
import model from "./model.js";
export function deleteModule(moduleId) {
 return model.deleteOne({ _id: moduleId });
 // const { modules } = Database;
 // Database.modules = modules.filter((module) => module._id !== moduleId);
}
```

In the ***Modules routes***, refactor the routing function by adding ***async*** and ***await*** keywords before the routing and DAO function calls as shown below.

Kambaz/Modules/routes.js ***S***

```
import * as modulesDao from "./dao.js";
```

```
export default function ModuleRoutes(app) {
  const deleteModule = async (req, res) => {
    const { moduleId } = req.params;
    const status = await modulesDao.deleteModule(moduleId);
    res.send(status);
  }
  app.delete("/api/modules/:moduleId", deleteModule);
  ...
}
```

In the ***Modules client*** file, refactor the client function to use ***axios*** with credentials as shown below. Confirm that modules are removed from the database.

app/(Kambaz)/Courses/[cid]/Modules/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const MODULES_API = `${HTTP_SERVER}/api/modules`;
export const deleteModule = async (moduleId) => {
 const response = await axiosWithCredentials.delete(
   `${MODULES_API}/${moduleId}`
 );
 return response.data;
};
```

Refactor the ***Modules*** screen to notify the server when a module is removed and then update the user interface to match the database.

app/(Kambaz)/Courses/[cid]/Modules/client.ts

```
import { deleteModule } from "./reducer";
import * as courseClient from "../client";
import * as modulesClient from "./client";
...
export default function Modules() {
 ...
 const deleteModuleHandler = async (moduleId: string) => {
   await modulesClient.deleteModule(moduleId);
   dispatch(deleteModule(moduleId));
 };
 ...
 return (
   <div>
    ...
       <ModuleControlButtons moduleId={module._id}
               deleteModule={(moduleId) => deleteModuleHandler(moduleId)}
               editModule={(moduleId) => dispatch(editModule(moduleId))} />
    ...
   </div>
);}
```

6.4.2.5 Updating Modules

Refactor the ***Modules DAO*** so that it uses the ***mongoose model*** to update modules in the database as shown below.

Kambaz/Modules/dao.js

```
import model from "./model.js";
export function updateModule(moduleId, moduleUpdates) {
 return model.updateOne({ _id: moduleId }, moduleUpdates);
// const { modules } = Database;
// const module = modules.find((module) => module._id === moduleId);
// Object.assign(module, moduleUpdates);
```

```
// return module;
}
```

In the ***Modules routes***, refactor the routing function by adding ***async*** and ***await*** keywords before the routing and DAO function calls as shown below.

Kambaz/Modules/routes.js ***S***

```
import * as modulesDao from "./dao.js";
export default function ModuleRoutes(app) {
  ...
  const updateModule = async (req, res) => {
    const { moduleId } = req.params;
    const moduleUpdates = req.body;
    const status = await modulesDao.updateModule(moduleId, moduleUpdates);
    res.send(status);
  }
  app.put("/api/modules/:moduleId", updateModule);
  ...
}
```

In the ***Modules client*** file, refactor the client function to use ***axios*** with credentials as shown below. Confirm that modules are updated in the database.

app/(Kambaz)/Courses/[cid]/Modules/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const MODULES_API = `${HTTP_SERVER}/api/modules`;
...
export const updateModule = async (module: any) => {
 const { data } = await axiosWithCredentials.put(
   `${MODULES_API}/${module._id}`,
   module
 );
 return data;
};
```

Refactor the ***Modules*** screen to notify the server when a module is updated and refresh the user interface to match the database.

app/(Kambaz)/Courses/[cid]/Modules/page.tsx

```
import { editModule, updateModule } from "./reducer";
import * as courseClient from "../client";
import * as modulesClient from "./client";

export default function Modules() {
 const updateModuleHandler = async (module: any) => {
   await modulesClient.updateModule(module);
   dispatch(updateModule(module));
 };
 return (
   <div>
    ...
       {!module.editing && module.name}
       {module.editing && (
         <input onChange={(e) =>
                  updateModuleHandler({ ...module, name: e.target.value }) }
                onKeyDown={(e) => {
                  if (e.key === "Enter") {
                    updateModuleHandler({ ...module, editing: false });
                  }
                }}
                value={module.name}/>
```

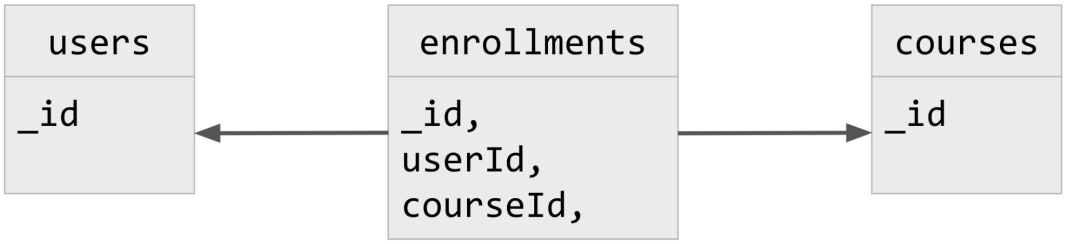
```
        )}
    ...
   </div>
);}
```

6.4.3 Persisting Enrollments in a Database as Many to Many Relations

In Kambaz, a user can be enrolled in several courses, and a course can have many enrollments. An enrollment establishes a relationship between a user and a course. Since there can be many enrollments where many users can be enrolled (or associated) in many courses, the enrollments relation is referred to as a ***many to many relation***. Many to many relationships can be represented in UML as shown in the following diagram capturing the fact that many users are related to many courses.

Implementing the diagram above in relational databases is challenging since it implies that each record in the ***users*** collection contains several references to records in the ***courses*** collection and vice versa. It is often easier to understand the implementation by using an intermediate collection that captures the relationship between each record in the original collections. For instance the UML diagram below uses a new ***enrollments*** collection that refactors the ***many to many*** relationship as two ***one to many*** relationships. The new collection is often referred to as a ***mapping table*** or ***collection***.

This new UML representation is easier to implement as the original collections ***users*** and ***courses*** no longer need to know anything about each other and instead a new ***enrollments*** collection can establish the relationship by capturing references to the records that related, e.g., a record entry in the ***enrollments*** collection captures what users are enrolled in what courses. Practially the ***enrollments*** collection declares fields ***userId*** and ***courseId*** to record references of the documents that are related to each other, as shown below.



The following sections describes how to implement ***many to many*** relationships using ***mongoose***.

6.4.3.1 Declaring Enrollments as a Many to Many Relationship

The ***Courses model*** below, implemented earlier, declares ***CourseModel*** as the name of the model for ***course*** documents stored in the ***courses*** collection. This name can be used to establish relationships between ***models*** and ***collections***.

Kambaz/Courses/model.js S

```
import mongoose from "mongoose";
import schema from "./schema.js";
const model = mongoose.model("CourseModel", schema);
```

```
export default model;
```

Similarly the ***Users model*** below declares the name of the model as ***UserModel*** for ***user*** documents stored in the ***users*** collection.

Kambaz/Users/model.js ***S***

```
import mongoose from "mongoose";                    // load mongoose library
import schema from "./schema.js";                   // load users schema
const model = mongoose.model("UserModel", schema);  // create mongoose model from the schema
export default model;                               // export so it can be used elsewhere
```

In a new ***schema*** file shown below, implement a ***many to many Enrollments*** relationship that relates ***user*** and ***course*** documents stored in the ***users*** and ***courses*** collections, referred to by their model names ***CourseModel*** and ***UserModel*** respectively.

Kambaz/Enrollments/schema.js ***S***

```
import mongoose from "mongoose";
const enrollmentSchema = new mongoose.Schema(
 {
   _id: String,
   course: { type: String, ref: "CourseModel" },
   user:   { type: String, ref: "UserModel"   },
   grade: Number,
   letterGrade: String,
   enrollmentDate: Date,
   status: {
     type: String,
     enum: ["ENROLLED", "DROPPED", "COMPLETED"],
     default: "ENROLLED",
   },
 },
 { collection: "enrollments" }
);
export default enrollmentSchema;
```

Create an ***Enrollments model*** file as shown below, to ***CRUD*** enrollment documents in an ***enrollments*** collection.

Kambaz/Enrollments/model.js ***S***

```
import mongoose from "mongoose";
import schema from "./schema.js";
const model = mongoose.model("EnrollmentModel", schema);
export default model;
```

Create an ***Enrollments DAO*** file that implements operations that create enrollments, deletes enrollments, and filters enrollments by either a course or user. The following sections describe each of the operations in detail.

Kambaz/Enrollments/dao.js ***S***

```
import model from "./model.js";
export async function findCoursesForUser(userId) {
 const enrollments = await model.find({ user: userId }).populate("course");
 return enrollments.map((enrollment) => enrollment.course);
}
export async function findUsersForCourse(courseId) {
 const enrollments = await model.find({ course: courseId }).populate("user");
 return enrollments.map((enrollment) => enrollment.user);
}
export function enrollUserInCourse(user, course) {
 return model.create({ user, course, _id: `${user}-${course}` });
}
```

```
export function unenrollUserFromCourse(user, course) {
 return model.deleteOne({ user, course });
}
```

6.4.3.2 Retrieving Courses for Enrolled Users

Enrollments establish a ***many to many relationship*** between ***users*** and ***courses***. A common operation consists of finding which documents in one collection are related to documents in the other collection. For instance, given a particular user we would like to determine which courses are related to that user, e.g., which courses is a user enrolled in. The ***findCoursesForUser()*** function below retrieves the ***enrollment*** documents for a given ***user***. The ***enrollments*** documents would contain the ***primary keys*** for the ***user*** and ***course*** documents being referenced. The ***populate()*** function tells ***mongoose*** to use the value of the ***primary keys*** to fetch the actual document referenced by the ***key***. The ***populate("course")*** function replaces the ***course primary key*** value in the ***enrollments*** document with the actual ***course*** document from the ***courses*** collection corresponding to the ***key***'s value. The ***enrollments.map()*** operation unwraps the ***enrollments*** array and returns a new array with just the ***course*** objects.

Kambaz/Enrollments/dao.js S

```
import model from "./model.js";
export async function findCoursesForUser(userId) {
 const enrollments = await model.find({ user: userId }).populate("course");
 return enrollments.map((enrollment) => enrollment.course);
}
...
```

In the ***Users routes***, implement a route function ***findCoursesForUser*** to retrieves courses for a given user. First make sure there's a logged in user. If the logged in user's role is ***ADMIN***, then respond with all the courses. Use the user's ID to retrieve the user's courses using the ***findCoursesForUser*** in the ***DAO*** implemented above.

Kambaz/Users/routes.js S

```
import * as dao from "./dao.js";
import * as courseDao from "../Courses/dao.js";
import * as enrollmentsDao from "../Enrollments/dao.js";
export default function UserRoutes(app) {
 const findCoursesForUser = async (req, res) => {
   const currentUser = req.session["currentUser"];
   if (!currentUser) {
     res.sendStatus(401);
     return;
   }
   if (currentUser.role === "ADMIN") {
     const courses = await courseDao.findAllCourses();
     res.json(courses);
     return;
   }
   let { uid } = req.params;
   if (uid === "current") {
     uid = currentUser._id;
   }
   const courses = await enrollmentsDao.findCoursesForUser(uid);
   res.json(courses);
 };
 app.get("/api/users/:uid/courses", findCoursesForUser);
 ...
}
```

In the React project, create ***findCoursesForUser()*** client function to request the ***courses*** for a given ***user ID***.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
export const findCoursesForUser = async (userId: string) => {
  const response = await axiosWithCredentials.get(`${USERS_API}/${userId}/courses`);
  return response.data;
};
```

In the ***Kambaz*** root component, implement an ***enrolling*** boolean state variable that toggles true/false to either display all courses or only the courses the current user is enrolled in. Implement event handler functions findCoursesForUser and fetchCourses as shown below to only fetch the courses a user is enrolled in or, also fetch all the courses. If enrolling, use both sets of courses to toggle an ***enrolled*** flag in the courses that can be used to display an ***Enroll***/***Unenroll*** button for each course. Pass ***enrolling*** and ***setEnrolling*** to the ***Dashboard*** so that it knows how to render the courses.

app/(Kambaz)/page.tsx

```
import { useSelector } from "react-redux";
import { useEffect, useState } from "react";
import * as courseClient from "./Courses/client";
import * as userClient from "./Account/client";
export default function Kambaz() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const [courses, setCourses] = useState<any[]>([]);
 const [enrolling, setEnrolling] = useState<boolean>(false);
 const findCoursesForUser = async () => {
   try {
     const courses = await userClient.findCoursesForUser(currentUser._id);
     setCourses(courses);
   } catch (error) {
     console.error(error);
   }
 };
 const fetchCourses = async () => {
   try {
     const allCourses = await courseClient.fetchAllCourses();
     const enrolledCourses = await userClient.findCoursesForUser(
       currentUser._id
     );
     const courses = allCourses.map((course: any) => {
       if (enrolledCourses.find((c: any) => c._id === course._id)) {
         return { ...course, enrolled: true };
       } else {
         return course;
       }
     });
     setCourses(courses);
   } catch (error) {
     console.error(error);
   }
 };
 ...
 useEffect(() => {
   if (enrolling) {
     fetchCourses();
   } else {
     findCoursesForUser();
   }
 }, [currentUser, enrolling]);
 ...
 return (
   ...
   <Dashboard courses={courses} course={course} setCourse={setCourse}
              addNewCourse={addNewCourse} deleteCourse={deleteCourse} updateCourse={updateCourse}
              enrolling={enrolling} setEnrolling={setEnrolling}/>
   ...
);}
```

In the ***Dashboard*** screen, use the ***enrolling*** and ***setEnrolling*** to implement a new button that toggles the ***enrolling*** boolean state variable. If ***enrolling*** is true, show a ***Enroll***/***Unenroll*** button for each course. If the ***enrolled*** flag is true, show ***Unenrolled*** and ***Enroll*** otherwise.

app/(Kambaz)/Dashboard/page.tsx

```
...
export default function Dashboard({ ..., enrolling, setEnrolling }
: { ...; enrolling: boolean; setEnrolling: (enrolling: boolean) => void;}) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  ...
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">
        Dashboard
        <button onClick={() => setEnrolling(!enrolling)} className="float-end btn btn-primary" >
          {enrolling ? "My Courses" : "All Courses"}
        </button>
      </h1>
      ...
      {courses.map((course) => (
        ...
        <div className="card-body">
          <h5 className="wd-dashboard-course-title card-title">
            {enrolling && (
              <button className={`btn ${ course.enrolled ? "btn-danger" : "btn-success" } float-end`} >
                {course.enrolled ? "Unenroll" : "Enroll"}
              </button>
            )}
            {course.name}
          </h5>
          ...
        </div>
      ))}
      ...
    </div>
  ...
);}
```

6.4.3.3 Enrolling / Unenrolling

In the ***Enrollments DAO***, implement ***enrollUserInCourse*** and ***unenrollUserFromCourse*** functions as shown below. The ***enrollUserInCourse*** function inserts a new ***enrollment*** document in the ***enrollments*** collection creating a relation between a user and the course they are enrolled in. The ***unenrollUserFromCourse*** function deletes an existing ***enrollment*** document from the ***enrollments*** collection, removing the relation between a user and the course they were enrolled in.

Kambaz/Enrollments/dao.js ***S***

```
import model from "./model.js";
export async function findCoursesForUser(userId) {
 const enrollments = await model.find({ user: userId }).populate("course");
 return enrollments.map((enrollment) => enrollment.course);
}
export function enrollUserInCourse(user, course) {
 const newEnrollment = { user, course, _id: `${user}-${course}` };
 return model.create(newEnrollment);
}
export function unenrollUserFromCourse(user, course) {
 return model.deleteOne({ user, course });
}
```

In the ***Users routes*** implement ***post*** and ***delete*** routes that create or removes an enrollment using the corresponding ***DAO*** functions.

Kambaz/Users/routes.js *S*

```
import * as enrollmentsDao from "../Enrollments/dao.js";
export default function UserRoutes(app) {
 const findCoursesForUser = async (req, res) => { ... };
 const enrollUserInCourse = async (req, res) => {
   let { uid, cid } = req.params;
   if (uid === "current") {
     const currentUser = req.session["currentUser"];
     uid = currentUser._id;
   }
   const status = await enrollmentsDao.enrollUserInCourse(uid, cid);
   res.send(status);
 };
 const unenrollUserFromCourse = async (req, res) => {
   let { uid, cid } = req.params;
   if (uid === "current") {
     const currentUser = req.session["currentUser"];
     uid = currentUser._id;
   }
   const status = await enrollmentsDao.unenrollUserFromCourse(uid, cid);
   res.send(status);
 };
 app.post("/api/users/:uid/courses/:cid", enrollUserInCourse);
 app.delete("/api/users/:uid/courses/:cid", unenrollUserFromCourse);
 app.get("/api/users/:uid/courses", findCoursesForUser);
 ...
}
```

In the React application, create client functions ***enrollIntoCourse*** and ***unenrollFromCourse*** that request the server to enroll or unenroll a user from a course. Encode the primary keys of the ***user*** and ***course*** as part of the path.

app/(Kambaz)/Account/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
export const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
export const USERS_API = `${HTTP_SERVER}/api/users`;
export const findCoursesForUser = async (userId: string) => { ... }
export const enrollIntoCourse = async (userId: string, courseId: string) => {
 const response = await axiosWithCredentials.post(`${USERS_API}/${userId}/courses/${courseId}`);
 return response.data;
};
export const unenrollFromCourse = async (userId: string, courseId: string) => {
 const response = await axiosWithCredentials.delete(`${USERS_API}/${userId}/courses/${courseId}`);
 return response.data;
};
```

In the ***Kambaz*** root component, implement ***updateEnrollment()*** event handler which calls the client functions ***enrollIntoCourse*** or ***unenrollFromCourse*** based on whether enrolled is true of false, as shown below. Pass the ***updateEnrollment()*** function to the ***Dashboard*** screen.

app/(Kambaz)/page.tsx

```
import * as userClient from "./Account/client";
export default function Kambaz() {
 const { currentUser } = useSelector((state: any) => state.accountReducer);
 const [enrolling, setEnrolling] = useState<boolean>(false);
 const findCoursesForUser = async () => { ... };
 const updateEnrollment = async (courseId: string, enrolled: boolean) => {
   if (enrolled) {
     await userClient.enrollIntoCourse(currentUser._id, courseId);
   } else {
     await userClient.unenrollFromCourse(currentUser._id, courseId);
   }
   setCourses(
     courses.map((course) => {
       if (course._id === courseId) {
```

```
        return { ...course, enrolled: enrolled };
      } else {
        return course;
      }
    })
  );
};
const fetchCourses = async () => { ... }
...
useEffect(() => { ... }
...
return (
  ...
  <Dashboard courses={courses} course={course} setCourse={setCourse}
             addNewCourse={addNewCourse} deleteCourse={deleteCourse} updateCourse={updateCourse}
             enrolling={enrolling} setEnrolling={setEnrolling} updateEnrollment={updateEnrollment}/>
  ...
);}
```

In the ***Dashboard*** screen implement an ***onClick*** event handler that calls ***updateEnrollment()*** with the ***course***'s ***ID*** and toggles the ***course***'s ***enrolled*** flag, causing the user to enroll or unenroll accordingly. Confirm that users can enroll and unenroll from any course.

app/(Kambaz)/Dashboard/page.tsx

```
...
export default function Dashboard({ ..., enrolling, setEnrolling, updateEnrollment }
: { ...; enrolling: boolean; setEnrolling: (enrolling: boolean) => void;
        updateEnrollment: (courseId: string, enrolled: boolean) => void }) {
  const { currentUser } = useSelector((state: any) => state.accountReducer);
  ...
  return (
    <div id="wd-dashboard">
      ...
      {courses.map((course) => (
        ...
        <div className="card-body">
          <h5 className="wd-dashboard-course-title card-title">
            {enrolling && (
              <button onClick={(event) => {
                        event.preventDefault();
                        updateEnrollment(course._id, !course.enrolled);
                       }}
                       className={`btn ${ course.enrolled ? "btn-danger" : "btn-success" } float-end`} >
                {course.enrolled ? "Unenroll" : "Enroll"}
              </button>
            )}
            {course.name}
          </h5>
          ...
        </div>
      ))}
      ...
    </div>
  ...
);}
```

6.4.3.4 Enrolling the Author

When a user creates a course, they should be automatically be enrolled in the course, otherwise they would not see the course in their ***Dashboard*** when they signin. In the ***Courses routes***, in the route that creates new courses, use the ***ID***s of the signed in user and the new course to enroll the user in the new course. Use the code below as a guide. Confirm that users that create courses are enrolled in the new course.

Kambaz/Courses/routes.js ***S***

```
import * as enrollmentsDao from "../Enrollments/dao.js";

export default function CourseRoutes(app) {
  ...
  const createCourse = async (req, res) => {
    const course = await dao.createCourse(req.body);
    const currentUser = req.session["currentUser"];
    if (currentUser) {
      await enrollmentsDao.enrollUserInCourse(currentUser._id, course._id);
    }
    res.json(course);
  }
  ...
}
```

6.4.3.5 Retrieving a Course's Students (On Your Own)

The ***Users*** screen implement earlier uses the ***PeopleTable*** to display all the users in the database. The ***People*** route implemented earlier under the ***Courses*** screen, should display the users in the current course. Reimplement the ***People*** route so that the ***PeopleTable*** only displays the users that are enrolled in a particular course when navigating to the ***People*** link. In the ***Courses routes*** use the ***enrollments DAO findUsersForCourse*** function to retrieve the users enrolled in a course.

Kambaz/Courses/routes.ts ***S***

```
import * as enrollmentsDao from "../Enrollments/dao.js";
export default function CourseRoutes(app) {
  ...
  const findUsersForCourse = async (req, res) => {
    const { cid } = req.params;
    const users = await enrollmentsDao.findUsersForCourse(cid);
    res.json(users);
  }
  app.get("/api/courses/:cid/users", findUsersForCourse);
  ...
}
```

In ***Courses client*** implement ***findUsersForCourse()*** client function to retrieve the users for a given course. Use the ***Courses routes*** and ***client*** to display the users enrolled in a course when navigating to a course's ***People*** route.

app/(Kambaz)/Courses/client.ts

```
import axios from "axios";
const axiosWithCredentials = axios.create({ withCredentials: true });
const HTTP_SERVER = import.meta.env.VITE_HTTP_SERVER;
const COURSES_API = `${HTTP_SERVER}/api/courses`;

export const findUsersForCourse = async (courseId: string) => {
 const response = await axios.get(`${COURSES_API}/${courseId}/users`);
 return response.data;
};
```

6.4.4 Assignments (On Your Own)

Implement ***schema***, ***model***, ***DAO***, ***routes***, and ***client*** files so that the ***Assignments*** and ***AssignmentsEditor*** screens display assignments stored in a database. Users should be able to display assignments in a course, create new assignments, update assignments, and delete existing assignments.

6.5 Deliverables

As a deliverable, make sure you complete all the lab exercises, mongoose schemas, models, DAOs, React components, and they behave as described. For both the React and Node repositories, all your work should be done in a branch called ***a6***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to ***Vercel*** and ***Render*** (or ***Heroku***) and confirm they integrate. If you are using ***Render***, it does not create a separate branch deployment like ***Vercel*** so you'll have to deploy to an entirely different Web service so that you don't trample the previous assignment while the TAs are still grading. The Node server application running on Render will need to be configured to interact with the ***a6 Vercel*** branch deployment. All the exercises should work remotely just as well as locally. The Kambaz Dashboard should display the courses and modules from the database. As a deliverable in ***Canvas***, submit the URL to the ***a6*** branch deployment of your React application running on Vercel.

7 References

1. HTML 5 Specification - https://html.spec.whatwg.org
2. MDN HTML Web Docs - https://developer.mozilla.org/en-US/docs/Web/HTML
3. The World Wide Web Consortium (W3C) - https://www.w3.org