Final Project Presentation

Code review of Book Connect Web App by Rehumile Sekoto

User Stories for Book-Connect

Book Connect is a social media platform that a targeted for book enthusiasts.

Below are the what users want from Book Connect Web App:

- To view a list of books previews by the title and author.
- An image associated with all the book previews.
- The option of reading a summary of a book.
- The option of seeing the date that the book was published.
- To filter the book by author.
- To filter the book by genre.
- To toggle between light and dark mode.

Viewing book previews by title and author with Image

In order to view the books on the web page, I have created a function called `createPreviewFragment` which will take the books object stored in the `data.js` file and create an HTML element that adds the title, author and image of the book from the `createPreview` function and then loop through each book element and add to a document fragment which will be added to the webpage by means of finding the DOM node that will be appended to.

```
/**
 * This function will take the books object and the
 number of books required per page and the page number.
 It will create a document Fragment whereby all the book
 previews made from the createPreview function will be
 looped through and add to the document fragment
 * @param {Array} booksArray
 * @param {number} bookRange
 * @returns {DocumentFragment}
 */
const createPreviewFragment = (booksArray, bookRange) => {
    const previewFragment = document.
    createDocumentFragment()
    let extractedBooks = booksArray.slice(bookRange[0],
    bookRange[1])

    for (let i =0; i < extractedBooks.length;i++) {
        const {author,  id, image, title} = extractedBooks
        [i]

        const preview = createPreview( {author, id,
        image, title})
        previewFragment.appendChild(preview)
    }

return previewFragment
}
```

Reading a summary of a Book

To read a summary of the book, I have used an event listener which will start up the `bookPreviewHandler` when a user clicks on a book. To determine which book the user is clicking on, I check the bubble path using the `event.compsedPath()` function. The path will be looped over until the book element with a preview ID is found. Once it is found, I add the image, title, subtitle with the author's name and the year of when it was published along with a short description of book. Then the user will see a overlay appear of the book's summary.

```
const bookPreviewHandler= (event) => {

    let pathArray = Array.from(event.path || event.
    composedPath())
    let active = null;

    for (const node of pathArray) {
const previewId = node.dataset.preview
        if (active)
         break;

        for (const singleBook of books) {
            if (singleBook.id === previewId) {
                active = singleBook
            }
        }
    }

    if (!active) {return}

    html.preview.active.style.display= 'block';
    html.preview.blur.src = active.image
    html.preview.image.src = active.image
    html.preview.title.innerText = active.title
    html.preview.subtitle.innerText = `${authors[active.
    author]} (${new Date(active.published).getFullYear()})`
    html.preview.description.innerText = active.description

}
```

Filter Books by genre, author or title

For the user to filter the books by genre, author or title, They will have to click on the `search` button. Once clicked on, an input field for a user to type in a title and 2 drop down menus with the genre categories and author names will already be populated from using the DropDown Function that I created will appear. An event listener will call the `searchSubmitHandler` function which will take the user's input from the `event.target`. Thereafter I use a `for..of loop` to loop through the `books` object to check if there is a title match or an author match or a genre match. If there is a match then the books will be pushed to a empty array and will then be display on the web page using the `createPreviewFragment` function.

```
const searchSubmitHandler = (event) => {
    html.search.overlay.style.display = 'none'
    event.preventDefault()

    const formData = new FormData(event.target)
    const filters = Object.fromEntries(formData)
    const selectedGenre = filters.genre
    const selectedAuthor = filters.author
    const selectedTitle = filters.title
    let results = []

    if (selectedTitle === "" && selectedGenre === 'any' && selectedAuthor === 'any') {
        results = books
    } else {

        for (const book of books) {
            const macthesTitle = selectedTitle.trim() === '' ||
            book.title.toLowerCase().includes(selectedTitle.toLowerCase());

            const matchesAuthor = selectedAuthor === 'any' || book.author === selectedAuthor;

            let matchesGenre = false;

            for (const genre of book.genres) {
              if (selectedGenre === 'any') {
                matchesGenre = true;
              } else if (genre === selectedGenre) {
                matchesGenre = true;
              }
            }

            if (macthesTitle && matchesAuthor && matchesGenre) {
              results.push(book);
            }
```

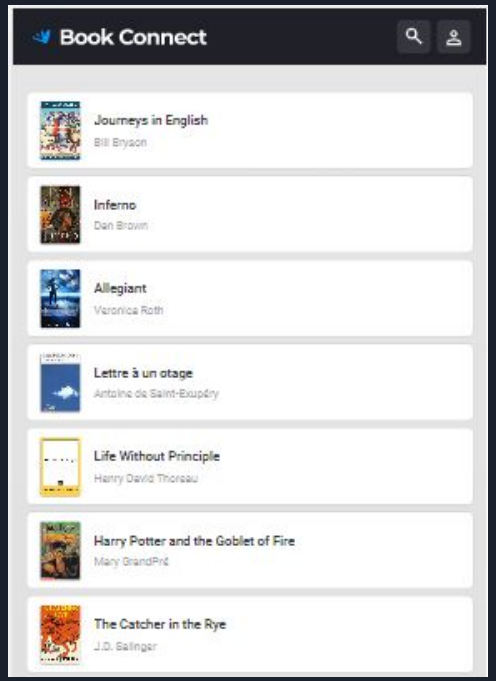
Toggling between dark and light mode

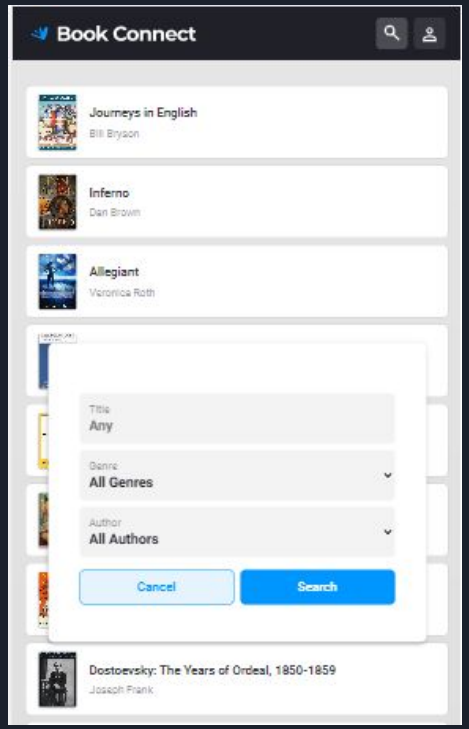
For a user to toggle between a dark or light mode, they will have to click on the `Settings` button on the top right corner of the web page. Once clicked on the `settingsSubmitHandler` will then be invoked. This function starts off by preventing any default executions and then gets the user input from the `event.target` . If user clicks on dark or light mode the document will set the properties of `--color-dark` or `--color-light` to the specific colour from the `css` object literal which contains the RGB values

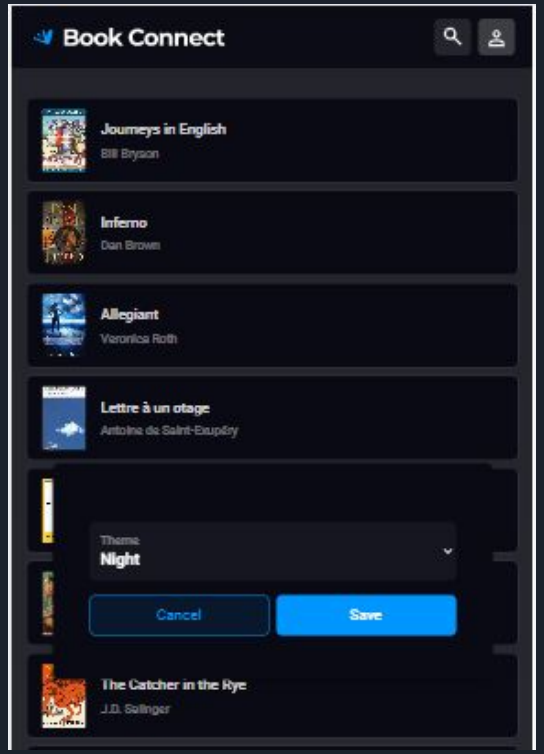
```
/**
 * This event handler will fire when the user changes value
 of dropdown in settings and clicks on the "Save" button. It
 can toggle between a light mode and dark mode so that a user
 can use the app comfortably at night.
 * @param {Event} event
 */
const settingsSubmitHandler = (event) => {
    event.preventDefault()
    const formData = new FormData(event.target)
    const result = Object.fromEntries(formData)

    document.documentElement.style.setProperty
    ('--color-dark', css[result.theme].dark);
    document.documentElement.style.setProperty
    ('--color-light', css[result.theme].light);
   html.settings.overlay.style.display = 'none';
}
```

User interface of Book Connect App







Overview of recommendations for Book Connect

- Using a standard naming conventions for variables and functions
- Documenting code using JSDocs
- Organising code
- Reusing code
- Declaring Variables
- Using indentation

Naming Conventions

- Using a standard naming convention for the variables or functions created is a great way to improve the readability of the code.
- Issues with not having a standard naming convention can result in other developers who try to debug your work end up being confused and not knowing what is what.
- It is best to also use names that are self-explanatory
- It is always ideal to guess what the function does or what the variable is referring simply by reading the name.

Do this:

```
const populateDropDownMenu = (objectSource, formLabel, formSource) => {
```

Instead of:

```
const menu = (objectSource, formLabel, formSource) => {
```

Documenting and Commenting

- Commenting or documenting specific parts of your code is an important practice as it will help other developers see what is not so straight forward.
- The codebase of Book Connect app was extremely hard to understand and comprehend which became a long process.
- I recommend using comments or JS Docs to explain unclear code such as what a particular function is. This will certainly be helpful in the future for maintenance and helping future developers familiarize themselves more quickly with the codebase.

Example

```
/**
 * The purpose of this function is to add the values of
the genres and authours objects into the form select.
Although it is for the population of the form select of
the authors and genre object. This function can be
reused for other form selects
 * @param {Object} objectSource - object from where the
values will be created
 * @param {String} formLabel - The title of the form
select
 * @param {Node} formSource - Dom node which where the
values will be appended
 */
```

Organising your code

- Organising your code is an essential practice as it helps with the overall readability and maintainability of the codebase.
- It will be easier to navigate through sections and identify the purpose of each section which in turn makes it more efficient to make changes or add features to the codebase.
- When code is well organised and consistent, collaborating with other developers on the code will be seamless.
- Debugging the code will be a much easier process as you will be able to locate and fix bugs
- On the 2 next slides, I recommend a way on how you can organise the code.

Data Section

```
// Data

//The varibale will be used to store matches of current filter settings from b
let matches = books

//This variable will be used to as the current page of books being display and
let page = 1;
```

Functions Section

```
// Functions

/**
 * A function that takes a book as an object literal and converts it into an HTML element th
elements individually prevents the JS having to re-render the entire DOM every time a new b
 * @param {object} props
 * @returns {HTMLElement}
 */
    const createPreview = (props) => {
    const {author,  id, image, title} = props


        let BookElement = document.createElement('button')
        BookElement.classList = 'preview'
        BookElement.setAttribute('data-preview', id)
```

Event Handlers Section

```
// Event Handlers

/**
 * This event handler will fire when the user changes value of dropdown in settin
 a light mode and dark mode so that a user can use the app comfortably at night.
 * @param {Event} event
 */
const settingsSubmitHandler = (event) => {
    event.preventDefault()
    const formData = new FormData(event.target)
    const result = Object.fromEntries(formData)

    document.documentElement.style.setProperty('--color-dark', css[result.theme]
```

Event Listeners Section

```
// EVENT LISTENERS

html.settings.form.addEventListener('submit', settingsSubmitHandler)

html.search.cancel.addEventListener('click', headerSearchCancelHandler)

html.other.search.addEventListener('click', headerSearchHandler)

html.other.settings.addEventListener('click', headerSettingsHandler)

html.settings.cancel.addEventListener('click', headerSettingsCancelHandler)

html.list.button.addEventListener('click', showMoreHandler)
```

Reusing Code

- Reusing code is a good practice in coding.
- It will save developers a lot of time and effort from rewriting code over and again whereas you can just reuse code that is already written
- Reusing code allows for consistency throughout the project.
- It is also easier to maintain as changes or updates made are automatically applied to all the other instances.
- For this codebase, I would recommend creating a object literal for the html elements that need references so instead of repeating the same code over and over, a reference for the HTML is available for you to just look up an HTML element
- The next slide is an example of the object literal.

HTML Object literal example

```
export const html = {
  list: {
      item: document.querySelector('[data-list-items]'),
      message: document.querySelector('[data-list-message]'),
      button : document.querySelector('[data-list-button]')},
  preview : {
     active: document.querySelector('[data-list-active]'),
     blur : document.querySelector('[data-list-blur]'),
     image: document.querySelector('[data-list-image]'),
     title: document.querySelector('[data-list-title]'),
     subtitle: document.querySelector('[data-list-subtitle]'),
     description: document.querySelector('[data-list-description]'),
     close: document.querySelector('[data-list-close]')
  },
  search : {
      overlay : document.querySelector('[data-search-overlay]'),
      form: document.querySelector('[data-search-form]'),
      title: document.querySelector('[data-search-title]'),
      genre : document.querySelector('[data-search-genres]'),
      author: document.querySelector('[data-search-authors]'),
      cancel: document.querySelector('[data-search-cancel]')
  },
  settings : {
      overlay: document.querySelector('[data-settings-overlay]'),
      form: document.querySelector('[data-settings-form]'),
      theme: document.querySelector('[data-settings-theme]'),
      cancel: document.querySelector('[data-settings-cancel]')
  },
  other: {
      settings: document.querySelector('[data-header-settings]'),
      search: document.querySelector('[data-header-search]')
  }
  }
```

Declaring Variables

- Variables in the codebase were declared implicitly which means they did not use the `const` or `let` keywords which will return undefined.
- To avoid unexpected scoping issues or prevent Javascript code from breaking, I recommend declaring the variables

Incorrect version

```
matches = books
page = 1;

if (!books && !Array.isArray(books)) throw new Erro
if (!range && range.length < 2) throw new Error('Ra

day = {
    dark: '10, 10, 20',
    light: '255, 255, 255',
}

night = {
    dark: '255, 255, 255',
    light: '10, 10, 20',
}

fragment = document.createDocumentFragment()
```

Correct version

```
let matches = books
let page = 1;

/**
 * This object literal stores the settir
 the dark and night mode in 'RGB' form.
 css settings when user chooses between
 */
const css = {
    day: {
        dark: '10, 10, 20',
        light: '255, 255, 255',
    },

    night :{
        dark: '255, 255, 255',
        light: '10, 10, 20',
    }
    }
```

Use Indentation

- Using indentation consistently within the codebase will help make the code more readable and easier to understand.
- The code is less cluttered and helps identify different blocks of code such as loops, if statements or functions.
- On the left is an example of how I would indentate the code

Incorrect version

```
if display.length < 1
data-list-message.class.add('list__message_show')
else data-list-message.class.remove('list__message_show')
```

Correct version

```
if (matches.length === 0) {
    html.list.message.style.display = 'block'
} else {
   html.list.message.style.display = 'none'
}
```

Conclusion

Overall, having good naming conventions, documenting or commenting code, organising code in a consistent way, reusing code, declaring variables and using indentation are all good practices to implement in your codebase as it helps make it more readable, maintainable and efficient which saves time and effort in the future.