# DWA_08 Discussion Questions

In this module you will continue with your “Book Connect” codebase, and further iterate on your abstractions. You will be required to create an encapsulated abstraction of the book preview by means of a single factory function. If you are up for it you can also encapsulate other aspects of the app into their own abstractions.

To prepare for your session with your coach, please answer the following questions. Then download this document as a PDF and include it in the repository with your code.

---

1. What parts of encapsulating your logic were easy?

Encapsulating the logic into a factory function was relatively straightforward. The process involved identifying the internal state and functionality that needed to be encapsulated, and then creating the factory function that returned an object with public methods and properties.

Here are the parts that were relatively easy in the encapsulation process:

- Identifying the internal state: It was clear that the `page` and `matches` variables were part of the internal state that needed to be encapsulated. These variables were used within the module to maintain the current page number and the list of books that match the search filters.

- Creating private functions: The existing code already had separate functions for creating button elements, initializing the list, updating the "Show more" button, and handling various event callbacks. These functions could be kept as private functions within the factory function to encapsulate the logic.

- Returning public methods and properties: By returning an object with the necessary public methods and properties from the factory function, the encapsulated module exposes only the desired functionality to the outside world. This provides a clear interface for interacting with the encapsulated module.

- Instantiating and using the module: Creating an instance of the module using the factory function and calling the public methods or accessing public properties of the instance was straightforward. This allowed for the modular and organized usage of the encapsulated logic.

---

2. What parts of encapsulating your logic were hard?

While encapsulating the logic into a factory function was relatively straightforward, there were a few parts that posed challenges during the process:

- Identifying dependencies: In some cases, it was challenging to determine the dependencies of certain functions or pieces of code. Understanding which variables or data were being accessed or modified by different functions was crucial for encapsulation. Analyzing the code and its interactions required careful examination to ensure all dependencies were accounted for.

- Managing event handlers: The code included several event handlers that were bound to specific elements. When encapsulating the logic, it was important to ensure that the event handlers remained functional and maintained their proper context. Managing event handlers within the encapsulated module required thoughtful consideration and potentially using techniques like binding or arrow functions to preserve the correct `this` context.

- Handling interdependencies between functions: Some functions in the original code relied on shared variables or state. Ensuring that these functions could access and modify the necessary shared data within the encapsulated module required careful design and coordination. This involved considering the order of function execution and maintaining the proper sequence of operations.

- Testing and debugging: Encapsulating code into a single factory function can make testing and debugging more challenging. Ensuring that the encapsulated module functions correctly and handles edge cases as expected requires thorough testing. Debugging can also be more complex since the encapsulated code may have interconnected functions and dependencies that need to be traced.

- Balancing abstraction and readability: Encapsulation can introduce additional layers of abstraction, which can make the code harder to understand if not done carefully. It's essential to strike a balance between encapsulation and readability, ensuring that the encapsulated code remains clear and maintainable.

3. Is abstracting the book preview a good or bad idea? Why?

Abstracting the book preview into a separate component or function can be a good idea in many cases. Here are some reasons why it can be beneficial:

- Reusability: By abstracting the book preview, you can create a reusable component or function that can be used in multiple places throughout your application. This promotes code reuse and reduces duplication. If you have multiple sections or pages that require book previews, encapsulating the preview logic allows you to easily use it wherever needed.

- Modularity: Abstracting the book preview promotes modular design. It isolates the functionality related to displaying the book information and provides a clear separation of concerns. This makes the codebase easier to understand, maintain, and update. Changes or updates to the book preview can be made in one place, improving code consistency and reducing the likelihood of introducing bugs.

- Encapsulation of UI logic: The book preview likely involves UI-related functionality, such as creating HTML elements, setting attributes, and handling events. Encapsulating this logic in a separate component or function improves the organization and readability of the code. It allows you to focus on the specific functionality of the preview without cluttering other parts of the codebase.

- Testing: When the book preview is abstracted, it becomes easier to write unit tests specifically targeting that component or function. Testing isolated components or functions can be simpler and more focused, allowing for better test coverage and easier debugging.