RATIONALE

Singleton Design

In this project we employ the singleton design pattern. This design pattern makes sure that a single class is responsible for creating objects instantiated from other classes, and these objects can be accessed without having to instantiate another object from the class. This allows us to keep the data and reuse those data in a session without having to re query the data again and again. In assignment 2 we explained that the traditional web-app is stateless and therefore we have to employ different methods to preserve the data to use for subsequent requests. This way of using the singleton design pattern makes it much easier to access the data that is saved for the duration of the session for different functions within the app. However this design pattern doesn't come without its disadvantages, by employing the singleton design pattern we cannot test each individual class separately and for this it doesn't achieve the separation of concern design principle. The singleton class, in our case it is the Dashboard class, has too many responsibilities and so the design violates the Single Responsibility Principle.

Observer Design Pattern

In the previous assignment, we were unsuccessful in implementing the observer design pattern for the project. In this new assignment, we implemented an observer pattern in Blazor way for syncing components. In particular, we define an OnChange action along with functions that front-end components can interact with in the IDashboardInterface. Then, we inject the concrete class implementing that interface under name DashboardService into the Dashboard component. The Dashboard component has a function OnInitialized which will be invoked when the component is initialized after having received its initial parameters. This function will call StateHasChanged(), which will notify the component that its state has changed and render for the first time.
To handle UI events in child components, the methods in IDashboardService are registered with the property EventCallback<T> defined as parameter in the child component. This method will call back to the parent component along with the parameter, which will be processed by the method registered earlier. After that, this method will invoke OnChange in the Dashboard component using NotifyStateChanged(), which causes the main component and its child components to be rerendered. Blazor is also smart enough just to re-render changes. Using this pattern, our application can update data in N interval time and on events without refreshing pages.
The disadvantage of this implementation has a potential disadvantage is that the component would be kept in memory forever and cause memory leak. Therefore, I had to implement IDisposable so that the component can unsubscribe from the event when it's removed

Facade Design Pattern and Package Design

The DashboardService class implementing IDashboardService can be considered as a Facade design pattern in the sense that it provides an interface for the front-end, which is the client here, to communicate with the Dashboard models and back-end controllers in a simplified way. It helps us to hide the complex implementations of models and back-end controllers and only provide to the client the version that is read only and cannot be modified directly.
Using this pattern, we create a balance between Common Closure Principle (CCP) and Common Reuse Principle (CRP). In detail, we group models into one package and controllers into one package as an API. Then, in DashboardService we group two of Dashboard as a representation for

whole models and API as a representation for controllers using HTTP requests to communicate with. This helps us decouple client code from subsystem while still make sure the size of the package & number of packages