**Review**

Objective 1

**User Story:** As a buyer I want to see when my offer has been accepted so that I can proceed with the property purchase.

For this objective I extended the current offer functionality by creating a MyOffersViewModelBuilder class in the builder section that returns a new MyOffersViewModel.

The class is locked down to the role of the buyer and allows the buyer to view a list of offers for each property they have made an offer on including the status of whether the offer was accepted, rejected or still in a pending state.

I extended this functionality further by allowing the buyer to make another offer where an existing offer had been rejected.

Objective 2

**User Story:** As a buyer I want to book a viewing appointment at a property so that I can determine whether I would like to make an offer.

For this objective I created BookViewingViewModelBuilder, BookViewingCommanHandler and BookViewingViewModel classes in the Property Controllers section allowing a buyer the ability to book a viewing at a property.

I also extended this functionality by creating a Viewings Controller section allowing buyers to view all of the viewings they have booked including the status of the viewing (Pending, Confirmed or Unavailable). For each viewing marked by the seller as unavailable, the buyer can choose to book a different viewing date.

This functionality also allows the seller to view all of the viewings for each property and to confirm the viewing or mark the viewing as unavailable.

Problems Fixed

Below is a summary of some of the issues I found with the site that I have fixed/improved:

* A security vulnerability would allow a buyer to navigate to the OnProperty action method allowing them to view all offers on a seller’s property. This was fixed by adding an authorization attribute for the role of seller.
* Authorization attributes were also added to other insecure methods.
* As a seller I was able to manipulate hidden form value fields meaning that I could accept or reject an offer for another seller by changing the Property ID or Offer ID client side. This could have been fixed by encrypting the hidden form values on the client but this would have caused Security through Obscurity concerns. It was resolved at the controller level by validating that the OfferId belonged to the Property entity.
* AntiForgeryTokens were added to forms missing them to protect against Cross-Site Request Forgery vulnerabilities.
* Extra unit tests were added.
* Friendly messages were added to views that returned no data.
* Comments were added in the code for guidance purposes for other developers working on the project.
* Checks for nullable objects were added to prevent an exception bubbling up to the view.
* Extra security defences were added to the web config such as custom headers to protect against the risk of a Click-Jacking attack, custom errors turned on and defending against malicious 3rd party scripts or scripts that attack an older browser version with the use of a Content Security Policy.

Future Improvements

* To ensure a more scalable, flexible and maintainable application, the application should have a better separation of concerns, making use of components and relationships being more loosely coupled. Calls to the database should be migrated into a separate data layer making use of patterns such as the repository pattern wrapped in unit of work calls to abstract the data layer from the business layer. A business or service layer would be beneficial to further extract business logic away from the web project.

The inclusion of Dependency Injection in the project helps the project achieve an inversion of control pattern and fulfils the Dependency Inversion principle within SOLID as classes should not need to know the concrete implementation of derived classes. However, methods in the web layer should be injected with an interface type from a business or service layer as oppose to an interface linked directly to the ApplicationDbContext class.

Further SOLID principles could be utilised such as the open/closed principle by extending classes as oppose to modifying them. Using a base Repository<T> class that contains base CRUD functionality and then utilising extended repository classes that are more orientated towards the business model than the base CRUD functionality could be an example.

Using small interfaces in a business or service layer related to specific controller actions could also be used to achieve the interface segregation principle.

Calls to the database could be made asynchronously ensuring the UI layer does not need to wait for a thread to be retuned ensuring a better user experience.

Depending on the business model, another approach could be to introduce a Microservices architecture that would allow independent deployment and development of key areas, improved fault isolation, system resilience and more scalable application.

* A notification system would be good to implement that would email/text both the buyer and seller when an offer has been made, accepted or rejected and when a viewing has been made, confirmed or marked as unavailable. Further intelligence could also be implemented informing the buyer of other properties within the same area that could be of interest to them.
* Better cross-cutting concerns could be achieved by creating an exception handling implementation and auditing process within the application.
* Other business model functionality such as suggested asking price of a property and further steps to purchase the property could also be implemented.