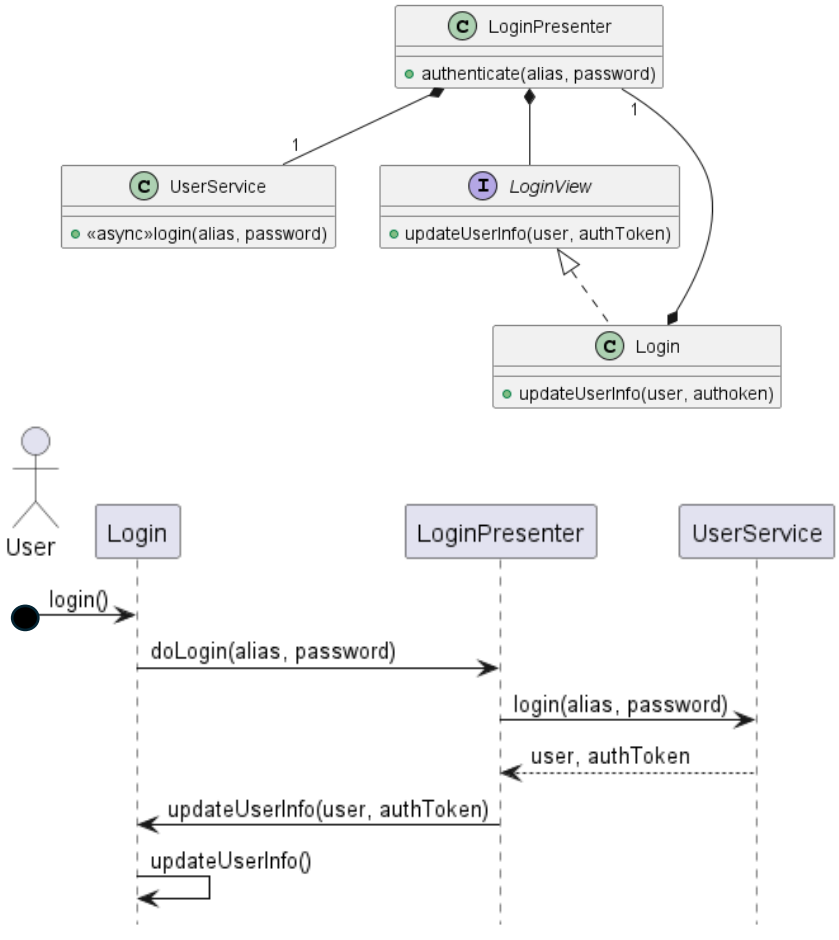
Question 1:

A. The service classes (UserService, StatusService, FollowService) are the Model part of the architecture. The React Components (Login.tsx, PostStatus.tsx, etc.) are the View part of the architecture. The Presenter classes (LoginPresenter, PostStatusPresenter, etc.) are the Presenter part of the model. The way it works is that the View calls its presenter class when it needs to do something with data. The Presenter defines an interface that the View implements in order for the Presenter to update the View. This is where the Observer pattern is seen, the View being an observer of the Presenter to know when to change. The Presenter also talks to the service classes to actually get new data and update the data.

B. One reason is that it allows for a lot of abstraction. When each React component had the functionality to grab data and update data, there was a lot of duplication throughout the code. With the MVP architecture, we have removed a lot of that duplication, moving some of the functionality to shared Presenter classes that each component can then just call. Another thing is that we have removed the need for hard coded data access from each component. We moved all of the data access into the Service classes, allowing us to easily swap out the implementation of this data access without having to change anything in the View or Presenter layers.



Question 2:

A. I used the template method in implementing the functionality of the ItemPresenter and all of its children classes (FeedPresenter, StoryPresenter, FollowerPresenter, FolloweePresenter). The ItemPresenter held a common algorithm for grabbing more items from the data, then displaying those items and updating the page. The ItemPresenter implemented the template, delegating the actual implementation of the getMoreItems method to each of its children.

B. Before I changed this in my code, each of those aforementioned children presenters had to implement the same logic in their own class, only differing in the actual function to retrieve the new data. When I moved this functionality to a parent class, it made it so that each child presenter only had to implement the one different part of the functionality. So now, instead of having a super similar function, save for one line, in each child presenter, I have a single function that calls a different line depending on the child.