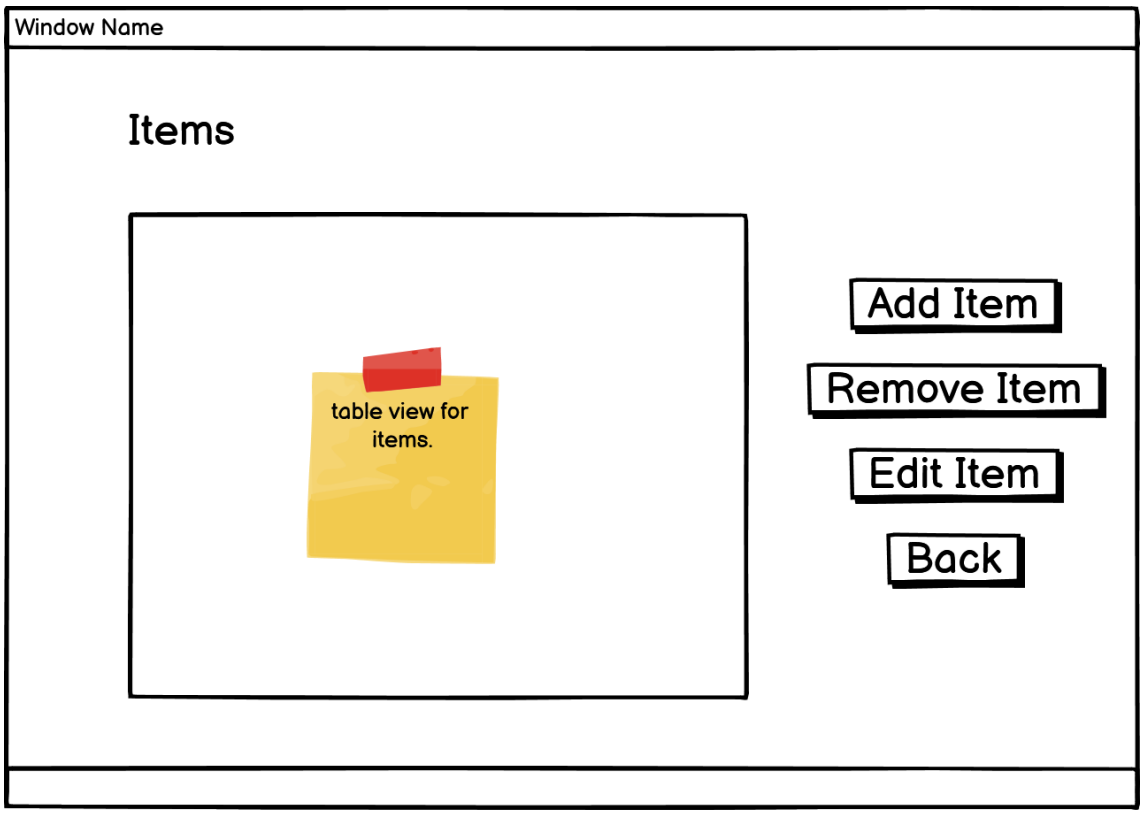
DATABASE VIEW DOCUMENTATION

DESIGN CHOICES:

For the use case of this sprint (Add an Item), it was relevant to get started on the Database View. This view features a Table with all of the items stored inside the database and a few buttons: Add, Remove or Edit Item, and go Back. The only button with functionality, for now, is Add Item. This is because the use case features nothing about removing an Item or editing it.

The Back button exists for navigability – it will take the user back to the Login page.

The Add Item button will open another window: the Add Item window.



IMPLEMENTATION:

There is not much to this window.

The ViewModel has, as instance variables, two fields: an ArrayList with all the items, a field for the Item currently chosen (no functionality yet) and, of course, the Model object.

When a DatabaseViewModel is created, the model is instantiated to the one given as an argument to the constructor, the allItems list is fetched from the model and the chosenItem, for now, is set to null because it is not relevant to the current use case.

```
15    public DatabaseViewModel(Model model)
16    {
17      this.model = model;
18      allItems = new ArrayList<>();
19      try {
20        allItems = model.getAllExistingItems().getAllItems();
21      }
22      catch (Exception e) {
23        e.printStackTrace();
24      }
25
26      chosenItem = null;
27    }
```

The ViewModel also has a reset() button that will fetch the existing items all over again. This method is made to be called whenever it is known that there was an update in the database.

```
31    public void reset() {
32      try
33      {
34        allItems = model.getAllExistingItems().getAllItems();
35      }
```

Moreover, it has a getter for all the Items so that they can be injected into the Table View inside the View Controller (which will follow in the next page).

```
43    public ArrayList<Item> getAllItems() { return allItems; }
```

There is also not much to the ViewController. Since the view has a Table View with Columns, these needed to be injectable FXML fields as shown in the picture below:

```
12  public class DatabaseViewController extends ViewController
13  {
14    @FXML private TableView itemTable;
15    @FXML private TableColumn idCol;
16    @FXML private TableColumn nameCol;
17    @FXML private TableColumn typeCol;
18    @FXML private TableColumn priceCol;
19    private DatabaseViewModel viewModel;
```

And the next screenshot is self-explanatory: in the init() method, the Item objects are fetched from the ViewModel and inserted into the table.

```
22  @Override protected void init()
23  {
24    viewModel = getViewModelFactory().getDatabaseViewModel();
25
26    idCol.setCellValueFactory(new PropertyValueFactory<Item, Integer>( s: "id"));
27    nameCol.setCellValueFactory(new PropertyValueFactory<Item, String>( s: "name"));
28    typeCol.setCellValueFactory(new PropertyValueFactory<Item, String>( s: "type"));
29    priceCol.setCellValueFactory(
30        new PropertyValueFactory<Item, Double>( s: "price"));
31
32    ObservableList<Item> observableListItem = FXCollections.observableArrayList(
33        viewModel.getAllItems());
34
35    itemTable.setItems(observableListItem);
36  }
```

The ViewController also has a reset() method, which calls reset() on ViewModel and updates the table items accordingly. *This was necessary because we did not know how to bind an ArrayList to a TableView.*

```
38  public void reset()
39  {
40    viewModel.reset();
41    ObservableList<Item> observableListItem = FXCollections.observableArrayList(
42        viewModel.getAllItems());
43    itemTable.setItems(observableListItem);
44  }
```

That is all.