OVERVIEW:

- The Data Entry Input field allows customizable (and scalable) attributes for the Picking App text input fields.
- Allows for standardized UI behavior, appearance, and Success/Error states
- This document details how to implement this input field in layout xml files

SET UP DataEntryViewModel

- In your ViewModel, add DataEntryInterface as an implementation.

```
class LoginViewModel @Inject constructor(
    private val pickingApi: PickingApi,
    private val activityService: ActivityService,
    private val authenticationPayloadWrapper: AuthenticationPayloadWrapper
) : ViewModel(), DataEntryInterface {
```

  - This gives the ViewModel access to Optional methods (see DataEntryInterface file)

- Create variable to instantiate DataEntryViewModel

```
val dataEntryViewModel = MutableLiveData(DataEntryViewModel())
```

- Call various setup methods to customize the input (such as Hint String, the input type, etc)
  - See DataEntryViewModel for various "set" options
  - NOTE: You can have multiple DataEntryViewModels, each one tied to a separate input.

DATA BIND – Layout XML

- In the layout .xml file, create a `<data>` object and set a `<variable>`

```
<data>

    <variable
        name="viewModel"
        type="com.reece.pickingapp.viewmodel.LoginViewModel" />
</data>
```

- To add a data_entry input field, `<include>` it in your layout

```
<include
    android:id="@+id/username_layout"
    layout="@layout/data_entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:dataEntryInterface="@{ viewModel }"
    app:dataEntryViewModel="@{ viewModel.dataEntryViewModel }"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

  - NOTE: The `dataEntryInterface` value is set to the `viewModel` variable created at the top
    - This ViewModel already is implementing the `DataEntryInterface`
    - Doing this gives the included data_entry input the ability to use the optional methods for `afterTextChanged`, `onEditorAction`, and `onTextChanged`
    - see data_entry.xml for how these are implemented.

  - NOTE: The `dataEntryViewModel` value is set to the `DataEntryViewModel` variable in the viewModel. This links this specific input with all the BindingAdapter methods (see `DataEntryAdapters`).
    - All these methods are what set the various UI elements and stylings for the InputLayout as well as icons, focus state, etc.
    - Each individual input will need its own `DataEntryViewModel` to track its various States and values.

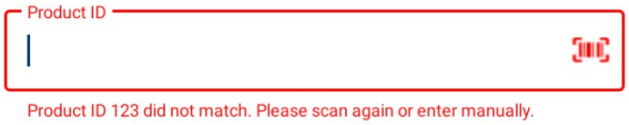
DATA BIND – Fragment

- In the fragment create binding var

```
private lateinit var binding: FragmentSignInBinding
```

- In onCreateView()
  - set the binding's viewModel (this is the <data><variable>.. set up in the layout's .xml)
  - set the binding's lifecycleOwner to the fragment's viewLifecycleOwner
    - Doing this will automatically remove all the Observers for DataBinding when the Fragment is deallocated.

```
binding = FragmentSignInBinding.inflate(inflater)
binding.viewModel = viewModel
binding.lifecycleOwner = viewLifecycleOwner
```

Example InputFields	DataEntryViewModel set States
	• InputFocusState.Default() • InputState.Default() • DataEntryFieldType.*SCANNABLE_PRODUCT*
	• InputFocusState.Focussed()
	• InputFocusState.Focussed() • InputState.Error(message = …)
	• InputState.Success(message = …) • setInputCanBeCleared(false)
	• InputState.Success(message = …) • setInputCanBeCleared(true)