Sesión # 7 Componente Práctico

Patrones de manejo de estado - BLoC

1. Para el siguiente componente práctico usted como instructor debe guiar al tripulante en el desarrollo y escritura del código.

2. Lea el enunciado del tripulante

3. Revise el repositorio **bloc-sesion7-solucion**. Este repositorio ya cuenta con todo el componente desarrollado y es la base para que usted como instructor le indique a los tripulantes qué realizar. ESTE ES EL REPOSITORIO PARA INSTRUCTORES.

4. Ejecute la solución y muéstrele a los tripulantes

5. Acceda al template https://github.com/EjemplosMisionTic2022/bloc-sesion7 . ESTE ES LA PLANTILLA BASE PARA ESTUDIANTES Y PARA QUE USTED MUESTRE LA SOLUCIÓN.

6. Lea lo solicitado al estudiante, explique y guíe paso a paso su solución en base a su repositorio solución.

   Archivos a modificar en el Componente Práctico:

bloc.dart

Se deben implementar las siguientes respuestas a los eventos:

```
if (event is UpdateDecimalEvent) {
      _decimal = Converter.adjustValue(_decimal,
event.digit);
      _binary = Converter.dec2bin(_decimal);

      yield ConverterState(
        _decimal,
        _binary,
      );
    }

    if (event is UpdateBinaryEvent) {
      _binary = Converter.adjustValue(_binary,
event.digit);
      _decimal = Converter.bin2dec(_binary);

      yield ConverterState(
        _decimal,
        _binary,
      );
    }

    if (event is ResetEvent) {
      _binary = "0";
      _decimal = "0";

      yield ConverterState(
        _decimal,
        _binary,
      );
    }
```

7. Diferencias entre BLoC y Provider:

Provider pattern	Explanation	Bloc Pattern	Explanation

Consumer<LoginProvider>(builder: (context, provider, child) {
 return TextField(
  decoration: InputDecoration(
   labelText: "Email",
   errorText: provider.email.error,
  ),
  onChanged: (String value) {
 	
   provider.changeEmail(value);
 	
  },
 );
}),
	1. Consumer is a widget, which is responsible for rebuilding the Textfield widget.
	2. LoginProvider is having the ChangeNotifier which keeps notifying the Consumer widget to rebuild the child widget(In our case it's a text field).
	3. provider.email.error is a variable of type ValidationModel which updates itself when provider.changeEmail(value) is called. It's all done by notifyListeners().
	4. The change in **Consumer** occurs when there is change in LoginProvider variables notified by notifyListeners()
		StreamBuilder<String>(
		  stream: _bloc.emailStream,
		  builder: (context, snapshot) {
		   return TextField(
		    decoration: InputDecoration(
		     labelText: "Email",
		     errorText: snapshot.error,
		    ),
		    onChanged: (String value) {
		
		     _bloc.emailOnChange(value);
		    },
		   );
		  })
			1. StreamBuilder is a widget which is responsible for rebuilding the Textfield widget.
			2. ***_bloc.emailStream*** is a stream from **BehaviourSubject** in LoginBloc that updates the text field widget.
			3. **snapshot.error** is updated by ***_bloc.emailOnChange*** method by on every hit of keyboard.
			4. The change in **StreamBuilder** occurs when there is a change in the value of Stream or Sink.

void changeEmail(String value){
 	
  if (ValidatorType.*email*.hasMatch(value)){
   _email=ValidationModel(value,null);
 	
  } else if (value.isEmpty){
   _email=ValidationModel(null,null);
 	
  } else {
   _email=ValidationModel(null, "Enter a valid email");
 	
  }
  notifyListeners();
 	
 }
	1. This **changeEmail** method in **LogiProvider** is updated whenever there is hit by keyboard in email text field.
	2. **_email** is the variable updated when the conditions in the method are met.
	3. **notifyListeners()** keeps on notifying when there is a change.
		StreamTransformer validateEmail() {
		 return StreamTransformer<String, String>.fromHandlers(
		   handleData: (String email, EventSink<String> sink) {
		    if (ValidatorType.*email*.hasMatch(email)){
		     sink.add(email);
		
		    } else if (email.isEmpty){
		     sink.addError(null);
		
		    } else {
		     sink.addError("Enter a valid email");
		
		    }
		   }
		);
		}
			1. **validateEmail** is the method inside **LoginBloc** class and is called whenever there is a text change in email text field.
			2. **validateEmail** updates the **emailStream**. As a result, Streambuilder is updated.

Consumer<LoginProvider>(builder: (context, provider, child) {
 return RaisedButton(
  color: Colors.*blue*,
  disabledColor: Colors.*grey*,
  child: Text(
   'Submit',
   style: TextStyle(color: Colors.*white*),
  ),
  onPressed: (!provider.isValid)
    ? null
    : () {
      provider.submitLogin();
     },
 );
})
	1. **RaisedButton** is disabled or enabled by getting the value of
	2. provider.isValid.
	3. Consumer rebuilds the **RaisedButton** based on the value present in **provider.isValid**
		StreamBuilder<bool>(
		  stream: _bloc.submitValid,
		  builder: (context, snapshot) {
		   return RaisedButton(
		    color: Colors.*blue*,
		    disabledColor: Colors.*grey*,
		    child: Text(
		     'Submit',
		     style: TextStyle(color: Colors.*white*),
		    ),
		    onPressed: (snapshot.hasData && snapshot.data)
		      ? _bloc.submitLogin
		      : null,
		   );
		  })
			In Bloc, StreamBuilder updates the RaisedButton, either enabled or disabled based on submitValid

bool get isValid {
 if (_password.value != null && _email.value != null){
  return true;
 } else {
  return false;
 }
}
	The change in the value of parameter *isValid* depends upon the value of *_email.value and _password.value* in LoginProvider class.	Stream<bool> get submitValid => Rx.*combineLatest2*(_isEmailValid.stream, _isPasswordValid.stream, (isEmailValid, isPasswordValid) {
		 if(isEmailValid is bool && isPasswordValid is bool) {
		  return isEmailValid && isPasswordValid; } return false;
		});
			The change in the value of parameter *submitValid* depends upon the value of **isEmailValid.stream and _isPasswordValid.stream**, in LoginBloc class.