Flutter 2

Navigator, Provider

Pontificia Universidad Javeriana
Departamento de Ingeniería de Sistemas
Profesor: Carlos Andrés Parra
E-mail: ca.parraa@javeriana.edu.co

Agenda

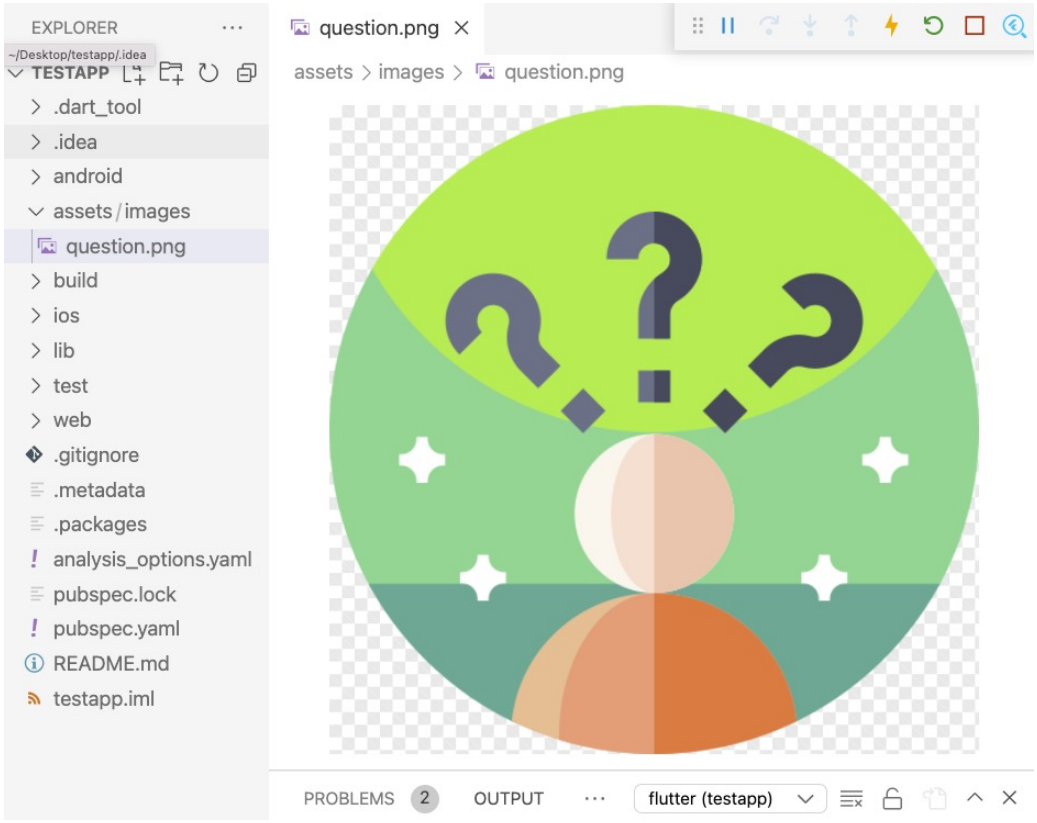
- Elementos básicos
  - Imágenes
  - Controladores
  - Acceso a elementos de la interfaz
- Demo Guess Game
- Navegación
  - NamedNavigator
- Manejo de Estado
  - Provider

Imágenes

- Para almacenar las imágenes en la aplicación, se agregan a una carpeta dentro del proyecto y se referencian desde el archivo pubspec.yaml:

```
# To add assets to your application, add an assets section, like this:
assets:
- assets/images/question.png
# - images/a_dot_ham.jpeg
```

pubspec.yaml

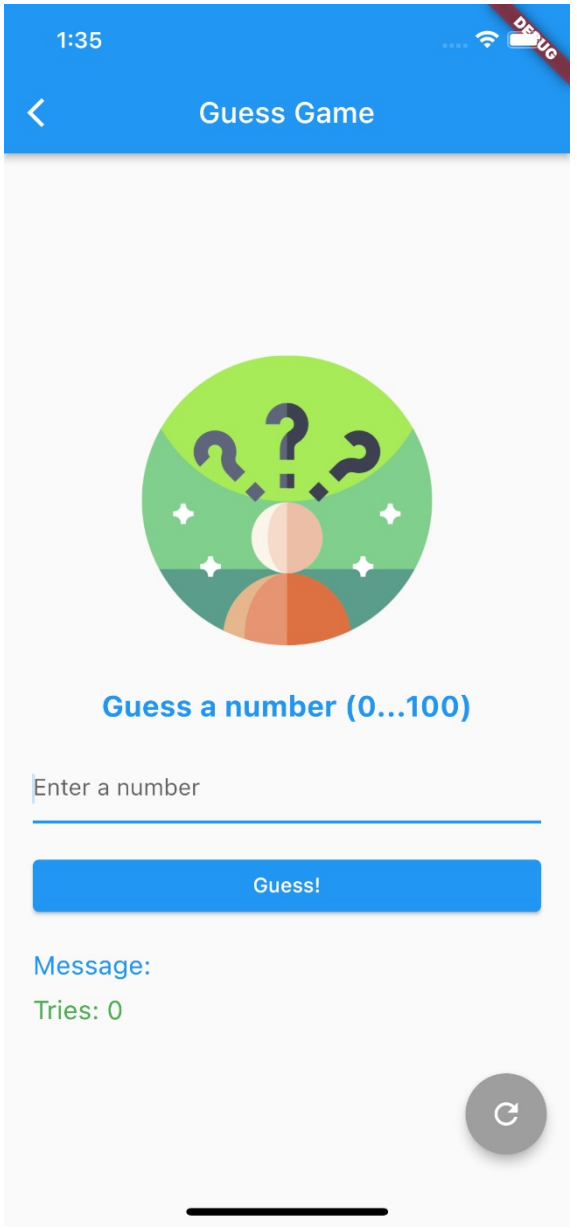


https://www.flaticon.com/premium-icon/question_4219829?term=question&page=1&position=2&page=1&position=2&related_id=4219829&origin=search

Imágenes

- Referenciar la imagen desde el código:

```
SizedBox(
    width: 200,
    height: 200,
    child: Image.asset(
    "assets/images/question.png")
),
```

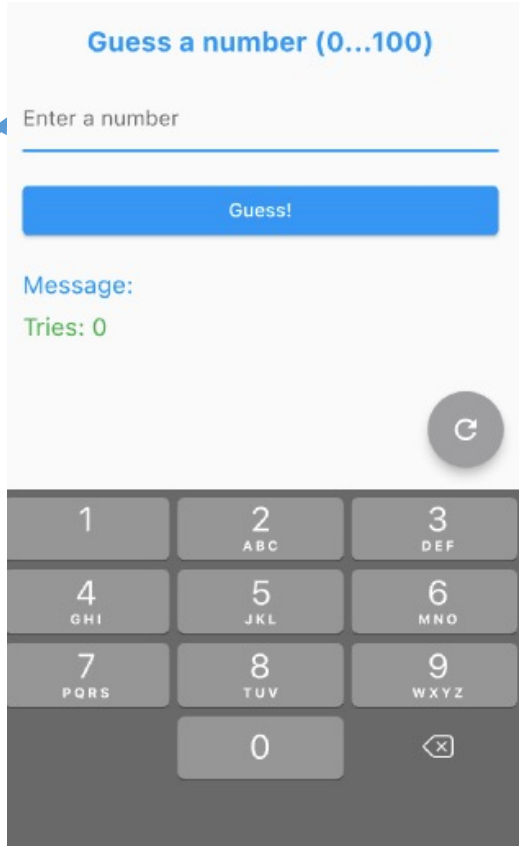


TextFields and controllers

- Para acceder al valor de un campo de texto en flutter, es necesario definir un controlador, y asignarlo al campo

```
final TextEditingController _controller
= TextEditingController();
```

```
TextField(
    enabled: !finished,
    controller: _controller,
    decoration: const InputDecoration(
        hintText:  'Enter a number'),
    keyboardType: TextInputType.number,
),
```

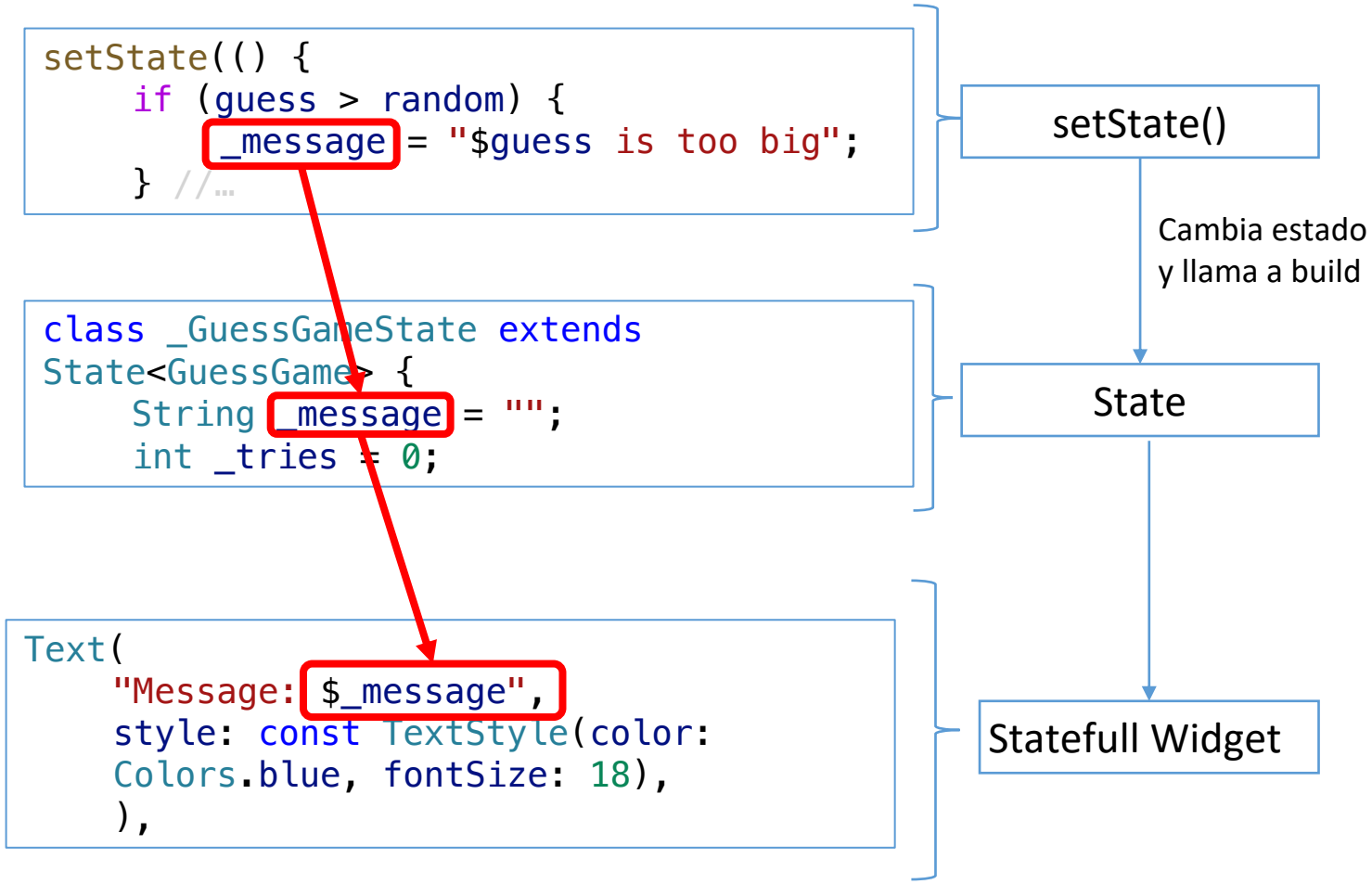


TextFields y Controladores

- Para saber lo que el usuario ingresó en el campo se accede a la propiedad *text* del controlador

```
if (_controller.text.isNotEmpty) {
    _tries++;
    int guess = int.parse(_controller.text);
    //…
```

Manejo del Estado



Acceso a elementos de la interfaz

- Dado que flutter se encarga del método build para construir el árbol de elementos, éstos no se pueden modificar de forma programática desde otras funciones de la clase.
- En este caso para modificar un valor de la interfaz, por ejemplo deshabilitar un botón, hay que hacerlo a través de la modificación del estado del widget.
- Primero hay que definir que el botón depende del estado y luego modificar el estado con *setState* cuando sea necesario para lograr el efecto deseado.

Deshabilitar un elemento de la GUI

- Estado

```
class _GuessGameState extends State<GuessGame> {
bool finished = false;
//…
```

- Widgets dentro de build

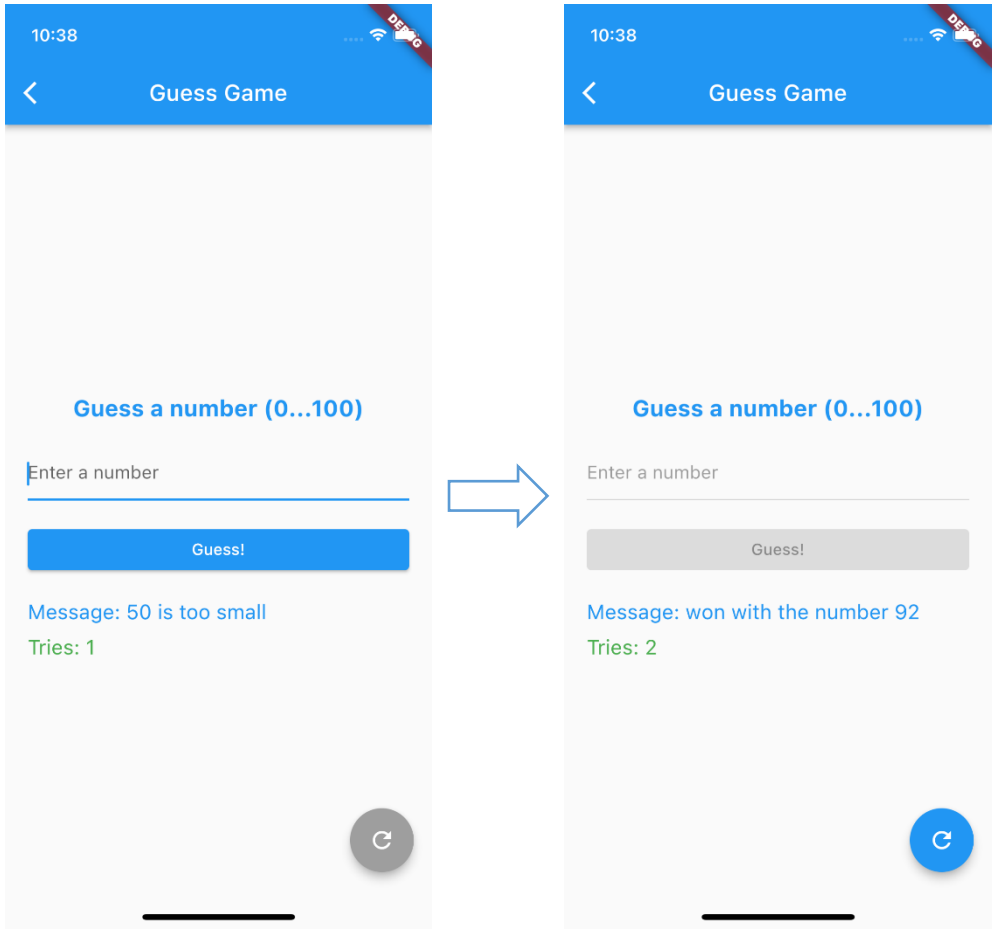
```
TextField(
    enabled: !finished,
    controller: _controller,
    keyboardType: TextInputType.number,
),
```

```
SizedBox(
    width: double.infinity,
    child: ElevatedButton(
    onPressed: finished ? null : _play,
    child: const Text("Guess!"))),
```

Deshabilitar un elemento de la GUI

- Modificar el estado para que Flutter pinte de nuevo los elementos

```
setState(() {
    //..
    finished = true;
}
```



Agenda

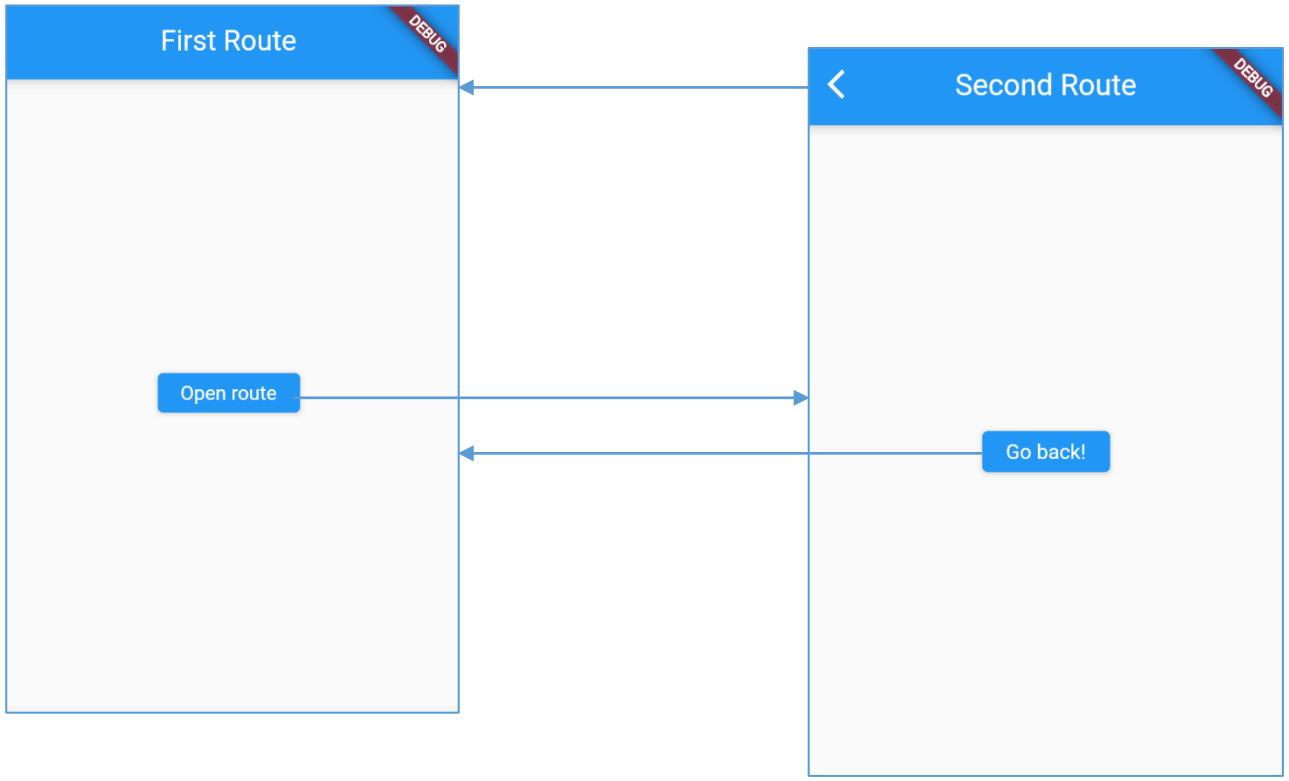
- Elementos básicos
  - Imágenes
  - Controladores
  - Acceso a elementos de la interfaz
- Navegación
  - NamedNavigator
- Manejo de Estado
  - Provider

Agenda

- Elementos básicos
  - Imágenes
  - Controladores
  - Acceso a elementos de la interfaz
- Demo Guess Game
- **Navegación**
  - NamedNavigator
- Manejo de Estado
  - Provider

Navigator

- Para hacer una transición entre dos pantallas se usa el objeto Navigator de Flutter.



https://flutter.dev/docs/cookbook/navigation/navigation-basics

Navigator
Transición entre pantallas

- En el origen se puede usar el método push para iniciar la transición hacia la pantalla (Widget) de destino

```dart
import 'package:testapp/guess_game.dart';

//…
onPressed: () {
    Navigator.push(context,
    MaterialPageRoute(builder: (context) => const GuessGame()));
},
```

Origen

```dart
class GuessGame extends StatefulWidget {
    const GuessGame({Key? key}) : super(key: key);

    @override
    _GuessGameState createState() =>  _GuessGameState();
} //…
```

Destino

Transición entre pantallas

- También se pueden enviar datos entre las pantallas. En este caso los datos llegan como parámetros al constructor del widget de destino

```
import 'package:first_app/fiboList.dart';

void calculate(){
    Navigator.push(
        context,
        MaterialPageRoute(builder: (context) =>
        FiboList(int.parse(myController.text))),
    );
}
```

```
class FiboList extends StatelessWidget {
final int upperLimit;
   FiboList(this.upperLimit);
//…
```

Named Navigator

- Si en la aplicación se tienen muchas pantallas, es ideal definir todas las rutas posibles, asignarles un nombre y luego a través del *Navigator*, ir a una pantalla en específico.
- Para esto es necesario primero definir las rutas.
- Cada ruta debe aparecer identificada, tipicamente en el primer widget de la aplicación: MaterialApp

Named Navigator

- En el widget de la aplicación se pueden definir las rutas y la ruta inicial a cargar.
  Cada ruta puede estar en un archivo diferente pero deben importarse.

```
import 'login.dart';
import 'friends.dart';
import 'map.dart';
void main() {
    runApp(MaterialApp(
            title: "MyApp",
            debugShowCheckedModeBanner: false,
            //home: LoginWidget(),
            initialRoute: '/login',
            routes: {
                '/login': (context) => LoginWidget(),
                '/friends': (context) => FriendsList(),
                '/map': (context) => MapView(),
            },
        )
    ));
}
```

Named Navigator

- Una vez se tengan las rutas, se puede utilizar el método pushedNamed( ) del Navigator:

```
void updateUI() {
    if (validateLogin()) {
      _userEmail = FirebaseAuth.instance.currentUser?.email;
      Navigator.of(context).pushNamed('/map');
    }
}
```

Ejercicio

- Defina 3 pantallas, en la primera pantalla defina un campo de texto, un botón *dropdown* y dos botones que permitan ir a las pantallas 2 y 3 como se muestra en la figura.
- Defina las rutas y las transiciones a través del Navigator.
- Cuando se pulse el botón 1, lance la pantalla 2 y muestre el contenido del cuadro de texto.
- Cuando se pulse el boton 2, lance la pantalla 3 y muestre la selección del botón *dropdown*.

DropdownButton class - material library - Dart API (flutter.dev)

Agenda

- Elementos básicos
  - Imágenes
  - Controladores
  - Acceso a elementos de la interfaz
- Navegación
  - NamedNavigator
- **Manejo de Estado**
  - Provider

Manejo del estado

- Si bien se puede usar *setState* como se vio en la sesión inicial, esta práctica no es recomendable para aplicaciones más complejas que manejen diferentes estados en varias pantallas con muchos widgets.
- Para esto se puede usar una libreria como **Provider** para separar la responsabilidad del manejo del estado de cada widget y definir un funcionamiento basado en eventos.
- Provider se puede entender como la implementación de un patron Observer para el estado de la aplicación.

https://pub.dev/packages/provider

Manejo del Estado
Provider

Pasos para usar Provider

1. Agregar dependencias
2. Definir el modelo para el estado
3. Registrarlo en el contexto
4. Operaciones sobre el estado
    1. Watch
    2. Read
    3. Select

Agregar dependencias

- Dentro del archivo *pubspec.yaml*, es necesario agregar la dependencia de Provider:

```
dev_dependencies:
 flutter_test:
  sdk: flutter
 provider: ^6.0.0
```

- Para actualizar las dependencias (si no se hace automáticamente), se pueden ejecutar los comandos:

  - `flutter clean`
  - `flutter pub get`

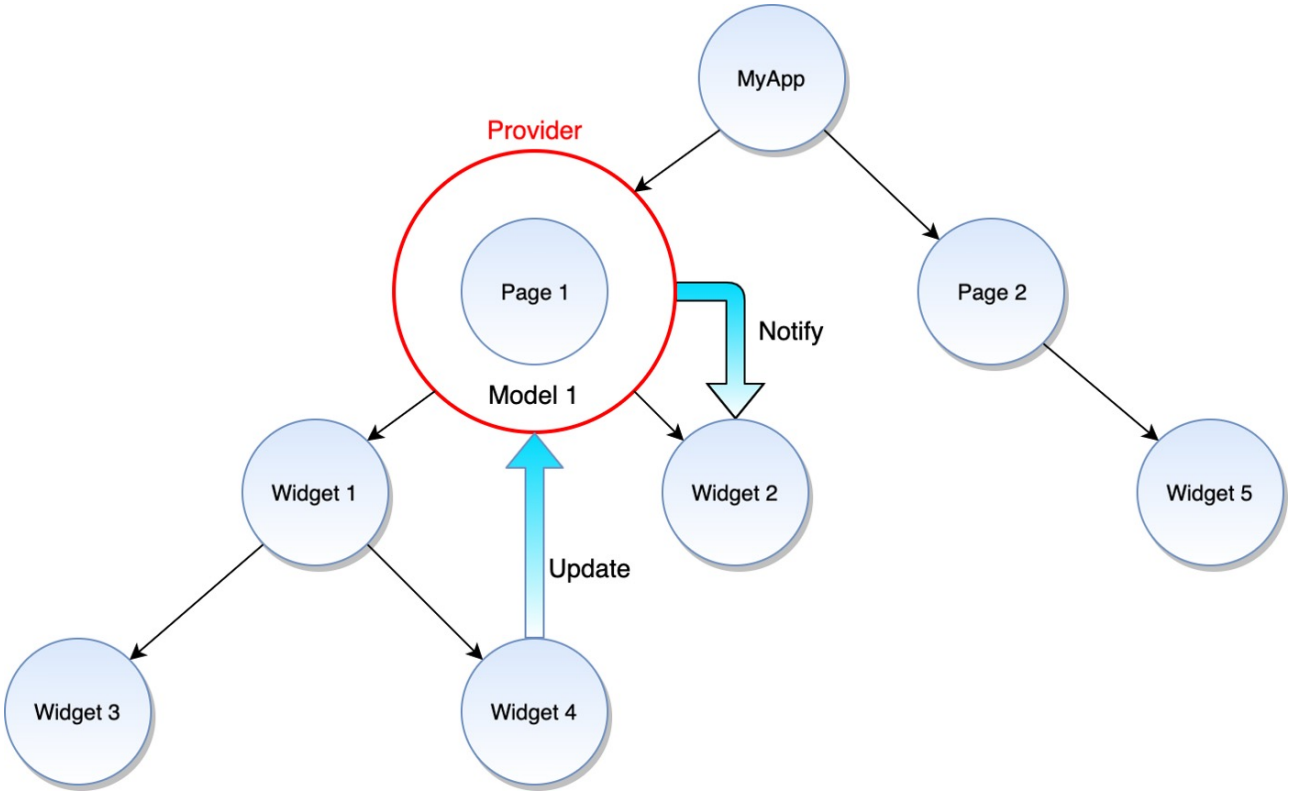
Definir el modelo para el estado

- El modelo corresponde a una o varias clases que extienden de *ChangeNotifier*. En cada modificación del estado, se debe llamar a *notifyListeners()*

```
class GuessState extends ChangeNotifier {

    String _message = "";

    String get message => _message;

    set message(String message) {
        _message = message;
        notifyListeners();
    }
}
```

Registrarlo estado - contexto

- Hay que identificar un ancestro común a todos los widgets que dependen del estado qué se esta definiendo. Si toda la aplicación depende del estado, este Widget puede ser MaterialApp



https://meysam-mahfouzi.medium.com/understanding-state-management-using-providers-in-flutter-7154c6968a3f

Registar estado

- En este caso se envuelve al widget de MaterialApp con ChangeNotifierProvider. Este recibe dos parámetros, el constructor del estado y la clase original que se quiere envolver.

```
import 'package:provider/provider.dart';

void main() => runApp(ChangeNotifierProvider(
    create: (context) => GuessState(),
    child: const MaterialApp(
        title: "FlutterApp",
        home: HomeWidget(),
    )
));
```

Acceder al estado

Hay tres métodos para acceder al estado en Provider:

- **watch:** se usa cuando se quiere acceder al modelo y además se queire reconstruir el widget cuando el modelo cambie
- **read:** se usa cuando se quiere acceder al modelo, pero no se quiere reconstruir el widget cuando el modelo cambie
- **select:** igual que **watch**, pero sólo para una parte del modelo

Acceder al estado

- **watch** y **select** se pueden usar sólo dentro del método *build* de un widget
- **read** no se puede usar en el método *build*, se usa típicamente en lo métodos *callback* referenciados en el *onPress* de un botón por ejemplo
- Cuando se llama a *notifyListeners()* en un modelo, todos los métodos build que hayan accedido al modelo usando **watch** o **select** se vuelven a invocar.

Acceder al estado

- Dentro de un método build:

```
@override
Widget build(BuildContext context) {
var gameState = context.watch<GuessState>();
return Scaffold(
    appBar: AppBar(
    //…

    Text("Message: " + gameState.message,
        style: const TextStyle(
            color: Colors.blue,
            fontSize: 18),
    ),
    //…
```

Acceder al estado

- Dentro de un método callback:

```
void _newGame(BuildContext context) {
    context.read<GuessState>().finished = false;
    context.read<GuessState>().newRandom();
    context.read<GuessState>().message = "";
    context.read<GuessState>().counter = 0;
}
```

Ejercicio final sesión 2 Flutter

- Agregue el manejo de estado con provider al juego GuessGame!

Para continuar con Flutter!

1. Acceso a Hardware
2. Mapas y localización
3. Backends
   - Firebase, Parse
4. Animaciones
5. Pushed notifications
6. …