07

Lidando com os estados do snapshot

Transcrição

Conseguimos evitar os problemas comuns do `FutureBuilder()` como o caso de carregar o primeiro valor como referência nula.

É importante tomar alguns cuidados em relação ao que é apresentado à usuária ou usuário. Por exemplo, se entramos na lista de contatos por meio do Dashboard no emulador, temos uma lista em branco seguida de outra com os itens. Enquanto carrega, faz sentido apresentarmos o ícone animado de progresso criado anteriormente que foi retirado após a última modificação. Nesta etapa, focaremos nesta questão para entendermos as **possibilidades** de devolver um conteúdo que faz sentido em relação ao cenário atual.

Para isso, precisamos ter acesso à presente situação de `future:` e callback, além de obter seus **estados da conexão** por meio de uma propriedade de `snapshot` chamada `.connectionState`. Dentro desta, há constantes indicando seu status que nos ajudam a tomar uma decisão para o momento.

Implementamos isso através da abordagem comum que usa `switch()`; se receber `snapshot.connectionState`, exigirá a implementação de todas as situações possíveis, pois caso haja um cenário no qual precisamos devolver outra coisa, já estaremos cientes para colocar algo que faça sentido ou até mesmo nada.

Para conhecer as demais possibilidades, usamos o próprio IntelliJ IDEA para nos ajudar através de "Alt + Enter" na linha de `switch()` após `builder:`, para então selecionar a opção "*Add missing case clauses*" que adiciona as cláusulas ausentes.

Com isso, nos são apresentadas quatro opções possíveis da conexão entre `snapshot` e `future:`. O primeiro `case` de `ConnectionState.none:` significa que `future:` ainda não foi executado, e costumamos colocar algum tipo de `Widget` que permite clique ou ação dos usuários e comece `future:`, para que o estado saia e execute as demais ações. Neste cenário, podemos implementar o botão que inicializa o `future:`, por exemplo.

Como não é nosso caso, podemos retirá-lo removendo seu comentário `// TODO: Handle this case`. O segundo `ConnectionState.waiting:` é um estado no qual verificamos se o `future:` ainda está carregando e não foi finalizado, e podemos devolver nosso progresso. Portanto, pegamos o bloco de código de `return Center()` e devolvemos ao `ConnectionState.waiting:`.

```
body: FutureBuilder(
    initialData: List(),
    future: Future.delayed(Duration(seconds:1)).then((value) => findAll()),
    builder: (context, snapshot) {
        switch(snapshot.connectionState){

        case ConnectionState.none:
            break;

            case ConnectionState.waiting:
                return Center(
                    child: Column(
                        mainAxisAlignment: MainAxisAlignment.center,
                        crossAxisAlignment: CrossAxisAlignment.center,
                        children: <Widget>[
```

```
                        CircularProgressIndicator(),
                        Text('Loading')
                    ], // <Widget>[]
                ), //Column
            ); // Center
            break;

        case ConnectionState.active:
            // TODO: Handle this case
            break;

        case ConnectionState.done:
            // TODO: Handle this case.
            break;

// código omitido

), // FutureBuilder
```

Testamos para ver o funcionamento desta abordagem no emulador.

Com o progresso apresentado, vemos que os estados são bem precisos, e inclusive percebemos várias execuções justamente por ter todos os estados.

Precisamos identificar os próximos casos, sendo que o terceiro `ConnectionState.active:` significa que nosso `snapshot` possui um dado disponível, mas o `future:` ainda não foi finalizado. Serve para quando utilizamos outra referência conhecida como `stream:` que trabalha com processamentos assíncronos e traz pedaços de um carregamento assíncrono, no caso do progresso de um *download* por exemplo.

Novamente não é nosso caso, então retiramos seu comentário. Por fim, temos `ConnectionState.done` que devolve nossa lista. Pegamos seu bloco e transferimos para esta situação.

```
body: FutureBuilder(
    initialData: List(),
    future: Future.delayed(Duration(seconds:1)).then((value) => findAll()),
    builder: (context, snapshot) {
        switch(snapshot.connectionState){

        case ConnectionState.none:
            break;

            case ConnectionState.waiting:
                return Center(
                    child: Column(
                        mainAxisAlignment: MainAxisAlignment.center,
                        crossAxisAlignment: CrossAxisAlignment.center,
                        children: <Widget>[
                            CircularProgressIndicator(),
                            Text('Loading')
                        ], // <Widget>[]
                    ), //Column
                ); // Center
                break;

            case ConnectionState.active:
```

```
                break;

            case ConnectionState.done:
                return ListView.builder(
                    itemBuilder: (context, index) {
                        final Contact contact = contacts[index];
                        return _ContactItem(contact);
                    },
                    itemCount: contacts.lenght,
                ); //ListView.builder
                break;

// código omitido

), // FutureBuilder
```

Assim, entregamos o comportamento esperado. É importante tomar cuidado de novo, pois estamos devolvendo os cenários comuns que tendem a ocorrer no `snapshot`, mas devemos devolver um valor padrão também caso não aconteça. Mesmo sendo impossível, é necessário colocar um retorno nulo, pois `snapshot` sempre atende um dos casos.

Se houver alguma situação onde o `return null` é executado, a abordagem comum que impeça a apresentação da tela de erro é inserir `text()`. Por mais que não seja alcançado, é uma boa prática evitar retornos nulos para `Widget` e telas de falha. Por isso, inserimos alguma mensagem genérica como `'Unknown error` para que a usuária ou usuário nos notifique de algum eventual problema.

Isso tende a não acontecer justamente porque todos os cenários foram preenchidos e somente estes serão executados no `snapshot`, que sempre terá algum dos estados.

Com este controle mais fino, podemos testar o aplicativo novamente.

Desta forma, entregamos algo que faz mais sentido ao usuário que clica no botão "Contacts", visualiza o ícone de progresso e acessa a lista de contatos, e assim fechamos a parte de `FutureBuilder()`.

Conseguimos carregar nossos itens com `FutureBuilder()`, entregando as possibilidades esperadas. Podemos então retirar o `.delayed`, visto que este não faz mais sentido, e testar mais uma vez para analisar o resultado.

Em seguida, faremos a integração do formulário para podermos gerar os contatos.