FICHA DE EXPECTATIVA DE RESPOSTA DA PROVA ESCRITA

CONCURSO	
Edital:	059/2023 (16/05/2023)
Carreira:	PROFESSOR DO MAGISTERIO SUPERIOR
Unidade Acadêmica:	DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
Área de Conhecimento:	SISTEMAS MÓVEIS E DISTRIBUÍDOS

CRITÉRIOS DE AVALIAÇÃO PARA TODAS AS QUESTÕES DISCURSIVAS
Clareza e propriedade no uso da linguagem
Coerência e coesão textual
Domínio dos conteúdos, evidenciando a compreensão dos temas objeto da prova
Domínio e precisão no uso de conceitos
Coerência no desenvolvimento das ideias e capacidade argumentativa

Questão 1: **Valor (0,00 a 3,00)**

O Flutter é uma tecnologia open-source para, entre outras coisas, desenvolvimento de aplicativos para dispositivos móveis. A tecnologia oferece diversas soluções para a gerência de estados de um aplicativo; algumas “nativas”, disponíveis na distribuição padrão da tecnologia; outras disponíveis por meio de bibliotecas desenvolvidas por terceiros. Entre as diversas soluções estão o Stateful Widget, o Provider, o BloC e o Riverpod (uma biblioteca dart, independente do Flutter, mas que pode ser utilizada com o framework). Explique duas dessas soluções citadas. Sua resposta deve incluir:
a) um parágrafo explicando cada solução em linhas gerais;
b) explanações sobre como e em que situações usar cada uma das soluções;
c) trechos de código para ilustrar o uso prático de cada uma das soluções; e
d) eventuais limitações e pontos positivos de cada uma das soluções.

Resposta Esperada:
(bastam duas dessas explicações na resposta da prova)

StatefulWidget

- solução básica e “nativa” para gerência de estados
- widget que possui um estado mutável e que deve ser redesenhado a cada mudança
- usado em casos mais simples, onde o próprio componente gerencia localmente seu estado.
- Em geral, deve ser criada uma extensão da classe StatefulWidget, sobrescrevendo o método createState. Este, por sua vez, deve retornar uma extensão da classe State, que sobrescreve o método build e retorna a composição de widgets que deve formar a interface do componente. Chamar o método setState nesse objeto state marca o StatefulWidget que o criou para ser redesenhado (renderizado novamente).

Pontos positivos

solução “nativa” e bem conhecida, disponível na distribuição padrão e com bastante material didático online;
requer uso de mecanismos simples de herança e substituição para funcionar
(há outras possibilidades, analisar caso a caso)

Limitações

serve mais para aplicativos simples, onde componentes gerenciam localmente seu próprio estado
em geral requer criação de duas novas classes, com herança e substituição, o que deixa o código um tanto burocrático em se considerando que tratamos de cenários mais simples
solução atrelada ao flutter, muitas vezes com dados, lógica e componentes de interface com o usuário misturados na classe que herda de State, o que dá mais trabalho para testes
usar o StatefulWidget na base da árvore da interface gráfica do app pode causar diversas renderizações desnecessárias e impactar negativamente o desempenho
(há outras possibilidades, analisar caso a caso)

Exemplo
(na correção, detalhes de implementação devem ser desconsiderados, apenas as partes relevantes para a gerência de estado devem ser levadas em conta)

```
class CounterApp extends StatefulWidget {
```

```
  @override
  _CounterAppState createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter App - StatefulWidget'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('Counter Value: $_counter'),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _incrementCounter,
              child: Text('Increment'),
            ),
          ],
        ),
      ),
    );
  }
}

void main() {
  runApp(MaterialApp(
    home: CounterApp(),
  ));
}
```

Provider

- Gerente de estados para apps de tamanho intermediário, que requerem gerência de estados mais complexa que os StatefulWidgets
- permite que um estado seja compartilhado, consumido e alterado por uma sub-árvore de widgets que estejam sob um ChangeNotifierProvider
- o ChangeNotifierProvider é responsável por injetar objetos que representam o estado nos diversos componentes da sub-árvore.
- Além do já citado ChangeNotifierProvider, para usar o Provider precisamos de uma extensão de ChangeNotifier (que encapsula e propaga o modelo/estado, e a lógica de negócio) e de um widget Consumer (parametrizável com a classe concreta do objeto que representa o estado) para receber estados propagados e (re)renderizar-se com base neles. É possível ler e interagir com o objeto ChangeNotifier injetado pelo ChangeNotifierProvider, sem a necessidade de redesenhar o widget, através do Provider.of, passando-se o parâmetro listen como false.

Pontos positivos

> O ChangeNotifier é uma classe básica, dá para testá-la (testar a lógica, em outras palavras) independentemente de componentes de interface gráfica
> Menos burocrático que o StatefulWidget (requer uma herança apenas, no casos mais convencionais, sem necessidade de sobrescrição) e, ainda assim, contempla casos de aplicativos com gerência de estados mais complexa (através da injeção de dependência)
> (há outras possibilidades, analisar caso a caso)

Limitações

> O Consumer (e o Provider, como um todo) depende do tipo (a classe) do estado e não do objeto em si que representa o estado, então há problemas se for necessário definir diversos objetos de um mesmo tipo para representar estados distintos numa mesma sub-árvore (o Riverpod, uma espécie de reimplementação do provider com características extras, contorna essa limitação)
> (há outras possibilidades, analisar caso a caso)

Exemplo
(na correção, detalhes de implementação devem ser desconsiderados, apenas as partes relevantes para a gerência de estado devem ser levadas em conta)

```
class Counter with ChangeNotifier {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    _counter++;
    notifyListeners();
  }
}

class CounterApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => Counter(),
      child: MaterialApp(
        home: Scaffold(
          appBar: AppBar(
            title: Text('Counter App - Provider'),
          ),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Consumer<Counter>(
                  builder: (context, counter, child) => Text('Counter Value: ${counter.counter}'),
                ),
                SizedBox(height: 20),
                ElevatedButton(
                  onPressed: () {
                    Provider.of<Counter>(context, listen: false).increment();
                  },
                  child: Text('Increment'),
                ),
              ],
            ),
          ),
        ),
      ),
    );
  }
}

void main() {
  runApp(CounterApp());
}
```

BloC

- gerente de estados que separa a lógica de negócios da interface do usuário
- pode ser usado em sistemas de maior porte, com gerência de estados mais complexa.
- utiliza Streams para gerenciar a comunicação do estado da aplicação
- em versões mais recentes, o uso explícito de Streams e eventos por parte do desenvolvedor pode ser contornado, utilizando-se objetos Cubit com o método emit.
- Em geral usa-se um objeto Cubit, que encapsula a lógica de negócio, emitindo novos estados quando necessário, e um BlocProvider, que injeta um mesmo objeto Cubit numa sub-árvore de componentes gráficos abaixo dele. Abaixo de um BlocProvider, o Cubit pode ser acessado através do contexto (parâmetro context nos métodos build) ou de widgets específicos que devem ser redesenhados de acordo com as mudanças de estado (BlocBuilder e BlocSelector, por exemplo)

Pontos positivos

escalabilidade - pode ser usado em sistemas de maior porte
tem mecanismos diversos para evitar renderizações desnecessárias, como o atributo buildWhen do BlocBuilder ou o próprio widget BuidSelector
separação clara entre lógica e interface gráfica
(há outras possibilidades, analisar caso a caso)

Limitações

Versões mais antigas com curva de aprendizado maior, pois requerem uso extensivo e explícito de Streams, com definição, ainda, de diversas classes de eventos
(há outras possibilidades, analisar caso a caso)

Exemplo

(Código com Cubit/emit, mas poderia ser com Streams/Eventos. Na correção, detalhes de implementação devem ser desconsiderados, apenas as partes relevantes para a gerência de estado devem ser levadas em conta)

```
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
  void decrement() => emit(state - 1);
}

void main() => runApp(CounterApp());

class CounterApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => CounterCubit(),
        child: CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Counter')),
      body: BlocBuilder<CounterCubit, int>(
        builder: (context, count) => Center(child: Text('$count')),
      ),
      floatingActionButton: Column(
        crossAxisAlignment: CrossAxisAlignment.end,
        mainAxisAlignment: MainAxisAlignment.end,
        children: <Widget>[
          FloatingActionButton(
            child: const Icon(Icons.add),
            onPressed: () => context.read<CounterCubit>().increment(),
          ),
          const SizedBox(height: 4),
          FloatingActionButton(
            child: const Icon(Icons.remove),
            onPressed: () => context.read<CounterCubit>().decrement(),
          ),
        ],
      ),
    );
  }
}
```

Riverpod

- Oferece uma abordagem simplificada (em termos de quantidade de código escrito pelo desenvolvedor) e declarativa para lidar com o estado da aplicação
- fornece, em sua versão mais atual, uma sintaxe concisa para injetar e acessar dependências
- alternativa mais poderosa em relação ao Provider tradicional e menos burocrática em relação a outras bibliotecas para uma abordagem complexa de gerência de estados.
- Uma forma de usar o riverpod seria: definir uma classe de estado, anotando-a com @riverpod, estendendo uma classe gerada pela biblioteca e sobrescrevendo o método build, que inicializa o estado. Nesta classe, é possível acrescentar métodos de negócio tantos quantos forem necessários. Criar uma classe que estenda ConsumerWidget sempre que precisar acessar o estado. A classe deve sobrescrever o build, recebendo um parâmetro a mais (além do context), que é um WidgetRef. Usar o WidgetRef, dentro do ConsumerWidget, para acessar o estado e interagir com ele (vide exemplo).

Pontos positivos

- Permite que classes de negócio consumam estados de outras classes de negócio (além dos componentes de interface, claro)
- Independente do flutter (biblioteca dart pura), o que facilita testes das classes de negócio e da comunicação entre elas, por não depender de componentes de alto nível
- Bem compreendidas as anotações e mecanismos de geração de código decorrentes delas, a gerência de estados pode se fazer de forma bastante sintética (vide exemplo, adiante)
- (há outras possibilidades, analisar caso a caso)

Limitações

Curva de aprendizado um pouco maior em relação a soluções menos robustas. Há muitos tipos de provedores de estado disponíveis (e mais código “boiler plate”) nas versões mais antigas, e geração de código nas versões mais recentes, características que requerem mais tempo para compreensão do uso da biblioteca como um todo (há outras possibilidades, analisar caso a caso)

Exemplo
(Exemplo pode ser em versão mais antiga. Na correção, detalhes de implementação devem ser desconsiderados, apenas as partes relevantes para a gerência de estado devem ser levadas em conta)

```
void main() {
  runApp(
    const ProviderScope(child: MyApp()),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: Home());
  }
}

@riverpod
class Counter extends _$Counter {
  @override
  int build() => 0;

  void increment() => state++;
}

class Home extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Scaffold(
      appBar: AppBar(title: const Text('Counter example')),
      body: Center(
        child: Text('${ref.watch(counterProvider)}'),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => ref.read(counterProvider.notifier).increment(),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Questão 2: **Valor (0,00 a 2,50)**

No contexto da programação web, descreva as diferenças entre Static Site Generation (SSG), Server Side Rendering (SSR). Sua resposta deve incluir:
a) características essenciais de cada um dos tópicos;
b) motivações e cenários para o uso de cada abordagem; e
c) exemplos de código para cada abordagem (codifique considerando o framework Next.js ou Nuxt.js). A abordagem SSG deve usar rotas dinâmicas na montagem das páginas.

Resposta Esperada:
- A abordagem SSG gera as páginas no build do sistema web.
- Dados para a geração do conteúdo são requisitados no build e as páginas são montadas, gerando, antecipadamente, as páginas que serão servidas sob requisição do cliente/navegador.
- Vantagens incluem um carregamento mais rápido no cliente (as páginas já estão construídas antes da requisição), uma carga de processamento menor no servidor e uma melhora no SEO (Search Engine Optimization).
- A abordagem SSR consiste em processar dados, fazer fetch de conteúdo (em bases de dados ou API’s) e gerar páginas a pós cada requisição, antes de enviar o resultado final ao cliente.
- Principal vantagem é a flexibilidade
- Numa pesquisa aberta, onde o usuário pode buscar por quaisquer palavras num dado sistema, é difícil pré-renderizar essas páginas pois não se pode prever todas as possibilidades de pesquisa de todos os usuários do sistema. Nesse caso, o processamento dinâmico, a cada requisição, seria mais indicado.

SSG em Next.js ficaria algo do tipo:

(criar um arquivo [id].js num subdiretório de pages, atentando para o uso de colchetes)

```
export default function TheBook({data}){
    return (
        <div>
            <div>{data.title} --- {data.author}</div>
        </div>
    )
}

export async function getStaticPaths(){
    return {
        paths:[
            {params: {id: "tt0095801"}},
            {params: {id: "tt0033152"}},
            {params: {id: "tt0015400"}},
            {params: {id: "tt0041149"}},
            {params: {id: "tt0044388"}},
            {params: {id: "tt0098746"}},
            {params: {id: "tt0046322"}},
            {params: {id: "tt0046497"}},
            {params: {id: "tt0044389"}}
        ],
        fallback: true
    }
}


export async function getStaticProps({ params }) {
    const res = await fetch(`https://api.livros.com/?t=${params.id}`)
    const data = await res.json();
    return {
      props: {
        data
      }
    }
}
```

SSR em Next.js

```
//arquivo pages/books.js
export default function BooksPage({ data }) {
  return (
    <div>
     {data.map( (m) => <div>{m.title} --- {m.author}</div> )}
    </div>
  );
}

export async function getServerSideProps(context){
  const res = await fetch(`https://api.exemplo.com/livros`)
  const data = await res.json()
  return {
     props: {
       data
     }
  }
}
```

SSG em Nuxt

```
//no arquivo nuxt.config.js
export default {
  generate: {
    routes: dynamicRoutesGenerator
  }
}


async function dynamicRoutesGenerator() {
  const dynamicData = await fetchData();
  const dynamicRoutes = dynamicData.map(item => `/dynamic/${item.id}`);
  return dynamicRoutes;
}

async function fetchData() {
  // buscar e retornar dados de rotas dinâmicas
}
```

Nesse caso, seria necessário, ainda, um arquivo _id.vue num diretório /pages/dynamic (dynamic é escolha do desenvolvedor). Esse arquivo teria o conteúdo da página, algo como:

```
<template>
  <div>
    <h1>{{ book.title }}</h1>
    <p>{{ book.author }}</p>
  </div>
</template>

<script>


export default {
  async asyncData({ params }) {
    const books = await fetchData();
    const book = books.find(p => p.id === params.id);
    return { books };
  }
};
</script>
```

SSR em Nuxt

Num arquivo convencional (pages/index.vue, por exemplo).

```
<template>
  <div>
    <h1>Books</h1>
    <ul>
      <li v-for="book in books" :key="book.id">
        {{ book.title }}
      </li>
    </ul>
  </div>
</template>

<script>
export default {
  async asyncData() {
    const books = await fetchBooks();
    return { books };
  }
};
</script>
```

Questão 3: **Valor (0,00 a 2,50)**

Escreva um componente (em React puro ou React Native) que acessa uma API e se utiliza dos hooks useState e useEffect. Explique o fluxo de execução do código deste componente quando ele for usado numa página qualquer. Atente, em sua explicação, para os aspectos de programação assíncrona e para os detalhes de funcionamento dos hooks utilizados. O código do componente deve contemplar as seguintes características:
a) o próprio componente busca dados numa API. Use a URL "https://exemplo.com/livros";
b) a requisição à API retorna texto no formato JSON referente a um array de diversos livros.
Cada livro tem as propriedades nome e autor, ambas no formato string;
c) deve ser retornado, inicialmente, um componente de texto com a mensagem
"Carregando..."; e
d) após a chegada dos dados da requisição, o componente deve ser redesenhado. Desta feita, os dados de todos os livros devem ser exibidos num componente de listagem.

Resposta Esperada:
A ideia geral do código é essa abaixo. Outros componentes gráficos poderiam ser utilizados, e mesmo marcas do HTML básico com React puro.

```
import React, { useEffect, useState } from 'react';
import { View, Text, FlatList, ActivityIndicator } from 'react-native';

const BookList = () => {
  const [books, setBooks] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchBooks = async () => {
      try {
        // Assuming the API returns an array of book objects
        const response = await fetch('https://exemplo.com/livros');
        const bookData = await response.json();
        setBooks(bookData);
```

```
        setLoading(false);
      } catch (error) {
        console.error('Error fetching books:', error);
        setLoading(false);
      }
    };

    fetchBooks();
  }, []);

  return (
    <View>
      {loading ? (
        <Text>Carregando…</Text>
      ) : (
        <FlatList
          data={books}
          keyExtractor={(item, index) => index.toString()}
          renderItem={({ item }) => (
            <View>
              <Text>Nome (ou título): {item.nome}</Text>
              <Text>Autor: {item.autor}</Text>
              <View style={{ borderBottomWidth: 1, borderBottomColor: '#ccc' }} />
            </View>
          )}
        />
      )}
    </View>
  );
};

export default BookList;
```

A execução:

Na primeira execução do componente, teremos o estado books como array vazio e o estado loading como true. O componente retorna um componente de texto com mensagem “Carregando…”
após a primeira renderização a callback function passada para o useEffect é chamada. O array vazio, segundo parâmetro do useEffect, garante que essa função de callback (que será uma espécie de efeito colateral da primeira renderização) será invocada apenas após a primeira renderização
são realizadas chamadas assíncronas (não bloqueiam a execução do app ou componente) dentro da função de callback do useEffect (optou-se pelo uso do await, poderiam ter sido usadas promessas). Quando os resultados dessas chamadas estiverem disponíveis, o estado do componente é modificado através das chamadas a setBooks e setLoading (ou apenas setLoading, no caso de haver erro)
a mudança no estado do componente causa uma nova renderização. Nesta nova renderização, em a requisição tendo acontecido com sucesso, o useState vai retornar um array de livros para o estado books e false para o estado loading. O componente resultante nessa renderização será o FlatList descrito no código

---

Questão 4: **Valor (0,00 a 2,00)**

O framework open-source Flutter utiliza uma hierarquia Widgets como parte central para representação de interfaces com o usuário.
Escolha algum ponto específico dessa hierarquia que faz uso de polimorfismo e explique, detalhadamente, como o polimorfismo é utilizado nesse ponto escolhido por você. Deixe claro na sua resposta:
a) A(s) classe(s) envolvidas e métodos envolvidos, destacando se há tipos ou métodos abstratos ;
b) Onde há potencial comportamento polimórfico e como pode se dar esse comportamento;
c) O que o comportamento polimórfico proporciona a quem utiliza o framework; e
d) Que implementações devem ser feitas por quem vai se valer desse comportamento polimórfico.

Resposta Esperada:
- Há diversos pontos da hierarquia relacionados ao polimorfismo.
- Em linhas gerais, a herança de classes e sobrescrição/substituição de métodos podem fazer com que uma chamada feita pelo framework seja direcionada para um código escrito pelo desenvolvedor, promovendo o comportamento polimórfico.
- Podem ser citados alguns exemplos básicos, como StatelessWidget ou o StatefulWidget, ambas classes abstratas.
- No StatelessWidget deve-se criar uma nova classe que estenda a própria StatelessWidget e que sobrescreva o método (abstrato) build. É, basicamente, o necessário para quem utiliza o framework poder encaixar seus próprios widgets na árvore de componentes gráficos que representa a interface com o usuário. Assim, os componentes nativos do framework poderão funcionar com componentes do desenvolvedor, o que se dá através do polimorfismo.
- No caso do StatefulWidget, temos potencial comportamento polimórfico na extensão do próprio StatefulWidget com sobrescrição do método (abstrato) createState e na extensão da classe State com sobrescrição do método (abstrato) build. O efeito disso é que em qualquer parte da interface gráfica onde seja possível usar um Widget será possível encaixar um objeto da classe que estende o StatefulWidget, e os componentes gráficos criados na árvore que representa a interface com o usuário serão os que resultam da chamada ao método build do objeto State retornado pelo createState.

NATAL, 16 de Outubro de 2023 às 15:46.

Assinado digitalmente em 16/10/2023 15:21	Assinada digitalmente em 16/10/2023 15:41	Assinado digitalmente em 16/10/2023 15:31
FABRICIO VALE DE AZEVEDO GUERRA	JOSÉ AMANCIO MACEDO SANTOS	RODRIGO REBOUÇAS DE ALMEIDA
PRESIDENTE	1° EXAMINADOR	2° EXAMINADOR