

Relatório

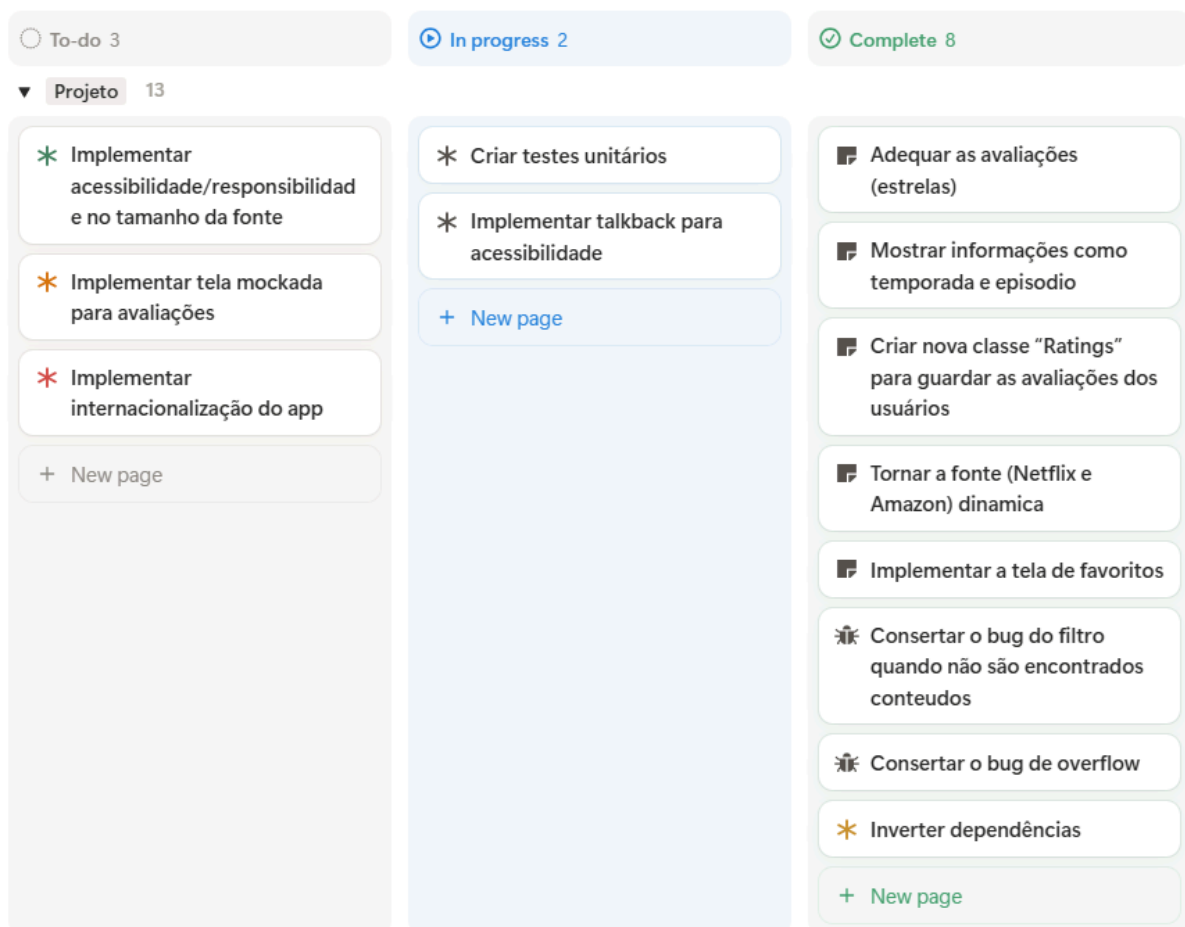
Desafio de Desenvolvimento Mobile

Laboratório Bridge, 2025

Juliana Miranda Bosio

1. Introdução e Organização do Trabalho

Para a realização do desafio de desenvolvimento mobile, organizei meu fluxo de trabalho utilizando a metodologia **Kanban** feita no Notion, o que me permitiu acompanhar visualmente as tarefas pendentes, em andamento e concluídas. Essa abordagem ajudou a estruturar o projeto em etapas claras, garantindo que cada requisito descrito no PDF fosse atendido, tendo em conta o prazo. Além disso, durante a implementação, utilizei **anotações TO-DO no código** sempre que encontrava pequenas inadequações ou pontos que poderiam ser melhorados.



2. Contexto e problema

2.1 Avaliações inconsistentes

Um dos problemas identificados foi que **as avaliações dos conteúdos não estavam sendo exibidas de forma coerente com a escala usada**, o que causava ruído na experiência do usuário. Para resolver isso, defini que a nota deveria refletir diretamente a quantidade de estrelas, em uma **escala de 0 a 5**. Essa escolha foi inspirada em produtos consolidados, como a rede social *Letterboxd*, e fundamentada em um princípio essencial do livro *“Não me faça pensar”*, de Steve Krug: nosso cérebro se acostuma com padrões simples e consistentes, tornando a interação mais intuitiva.

Com base nessa decisão de design, modifiquei os seguintes widgets no código:

- **content_card_widget.dart**
Ajustei as variáveis `fullStars` e o parâmetro da booleana `hasHalfStar`, dobrando seus valores. Com isso, a quantidade de estrelas exibida no card da tela principal passou a ser proporcional à nota na escala de 0 a 5.
- **hero_section_widget.dart**
Na hero section da página de detalhes, alterei a nota máxima para a nova escala padrão, exibindo valores de 0.5 em 0.5, como “4.5/5”.
- **user_ratings_widget.dart**
Refinei a lógica de exibição da nota no card de avaliações dos usuários, garantindo que a proporção entre nota e quantidade de estrelas se mantivesse consistente, de forma semelhante à lógica usada nos cards da tela principal.

2.2 Exibição de temporada e episódio para séries

Outro ponto identificado foi a **ausência de informações específicas para séries**, como temporada e episódio atual, o que limitava a clareza da experiência. Para resolver isso, adicionei uma **tag no card da tela principal** que exibe essas informações de forma compacta.

```
102 Widget _buildDurationTag() {
103   String tagText;
104   if (content.type == MediaType.series) {
105     final season = content.seasons != null ? 'T${content.seasons}' : '';
106     final episode = content.episodes != null ? 'E${content.episodes}' : '';
107     tagText = [season, episode].where((e) => e.isNotEmpty).join(' • ');
108     if (tagText.isEmpty) tagText = 'Série';
109   } else {
110     tagText = content.duration.isNotEmpty ? content.duration : 'Filme';
111   }
112
113   return Container(
114     padding: EdgeInsets.symmetric(horizontal: 8, vertical: 4),
115     decoration: BoxDecoration(
116       color: AppTheme.mutedText.withValues(alpha: 0.2),
117       borderRadius: BorderRadius.circular(4),
118       border: Border.all(
119         color: AppTheme.mutedText.withValues(alpha: 0.3),
120       ),
121     ),
122     child: Text(
123       tagText,
124       style: AppTheme.darkTheme.textTheme.bodySmall?.copyWith(
125         color: AppTheme.mutedText,
126         fontWeight: FontWeight.w500,
127       ),
128     ),
129   );
130 }
```

Durante os testes iniciais, percebi que a tag poderia gerar **overflow dentro do card**. Para corrigir, utilizei um widget `Expanded` em conjunto com a remoção de um `Spacer`, garantindo melhor gerenciamento do espaço disponível.

Posteriormente, notando que ficou vaga a informação na tela mais crucial, a de **detalhes da série**, inseri o método `_buildSeriesDetailsTag()`, que adiciona, ao lado da sinopse, a quantidade total de temporadas e episódios se o conteúdo for do tipo série. Esse widget também recebeu suporte a acessibilidade, fornecendo uma descrição semântica mais detalhada para leitores de tela.

```

25 Widget _buildSeriesDetailsTag() {
26   if (widget.medium.type != MediaType.series) {
27     return const SizedBox.shrink();
28   }
29
30   // Lógica para o texto visual (ex: "T5 • E62")
31   final parts = <String>[];
32   if (widget.medium.seasons != null) parts.add('T${widget.medium.seasons}');
33   if (widget.medium.episodes != null) parts.add('E${widget.medium.episodes}');
34   final tagText = parts.join(' • ');
35
36   if (tagText.isEmpty) return const SizedBox.shrink();
37
38   // TalkBack: (ex: "5 temporadas com 62 episódios")
39   final semanticParts = <String>[];
40   if (widget.medium.seasons != null) {
41     final seasonText = widget.medium.seasons == 1 ? 'temporada' : 'temporadas';
42     semanticParts.add('${widget.medium.seasons} $seasonText');
43   }
44   if (widget.medium.episodes != null) {
45     semanticParts.add('com um total de ${widget.medium.episodes} episódios');
46   }
47   final semanticLabel = semanticParts.join(' ');
48
49   return Semantics(
50     label: semanticLabel,
51     child: Container(
52       padding: EdgeInsets.symmetric(horizontal: 2.w, vertical: 0.5.h),
53       decoration: BoxDecoration(
54         color: AppTheme.secondaryDark.withAlpha(150),
55         borderRadius: BorderRadius.circular(6),
56       ),
57       child: Text(
58         tagText,
59         style: AppTheme.darkTheme.textTheme.labelMedium?.copyWith(
60           color: AppTheme.mutedText,
61         ),
62       ),
63     ),
64   );
65 }

```



2.3 Reestruturação da representação de dados de avaliação de conteúdo

Na implementação inicial, as avaliações eram tratadas apenas como um número simples (`double rating`). Essa abordagem era limitada, pois não permitia representar adequadamente o contexto de quem avaliou, o comentário associado ou a data da avaliação. Além disso, dificultava a manutenção e impedia futuras expansões, como exibir um histórico ou destacar resenhas de usuários.

Para resolver esse problema, criei a classe `Rating`, capaz de representar avaliações de forma estruturada, incluindo atributos como `userId`, `userName`, `score`, `comment` e `date`. Dessa forma, cada avaliação é um objeto completo, e não apenas um valor numérico isolado.

```
1 class Rating {
2     final int userId;
3     final String userName;
4     final double score;
5     final String? comment;
6     final DateTime? date;
7
8     Rating({
9         required this.userId,
10        required this.userName,
11        required this.score,
12        this.comment,
13        this.date,
14    });
15
16    factory Rating.fromJson(Map<String, dynamic> json) {
17        return Rating(
18            userId: json['userId'] ?? 0,
19            userName: json['userName'] ?? 'Usuário',
20            score: json['score'] != null ? (json['score'] as num).toDouble() : 0.0,
21            comment: json['comment'],
22            date: json['date'] != null ? DateTime.parse(json['date']) : null,
23        );
24    }
25
26    Map<String, dynamic> toJson() {
27        return {
28            'userId': userId,
29            'userName': userName,
30            'score': score,
31            'comment': comment,
32            'date': date?.toIso8601String(),
33        };
34    }
35 }
```

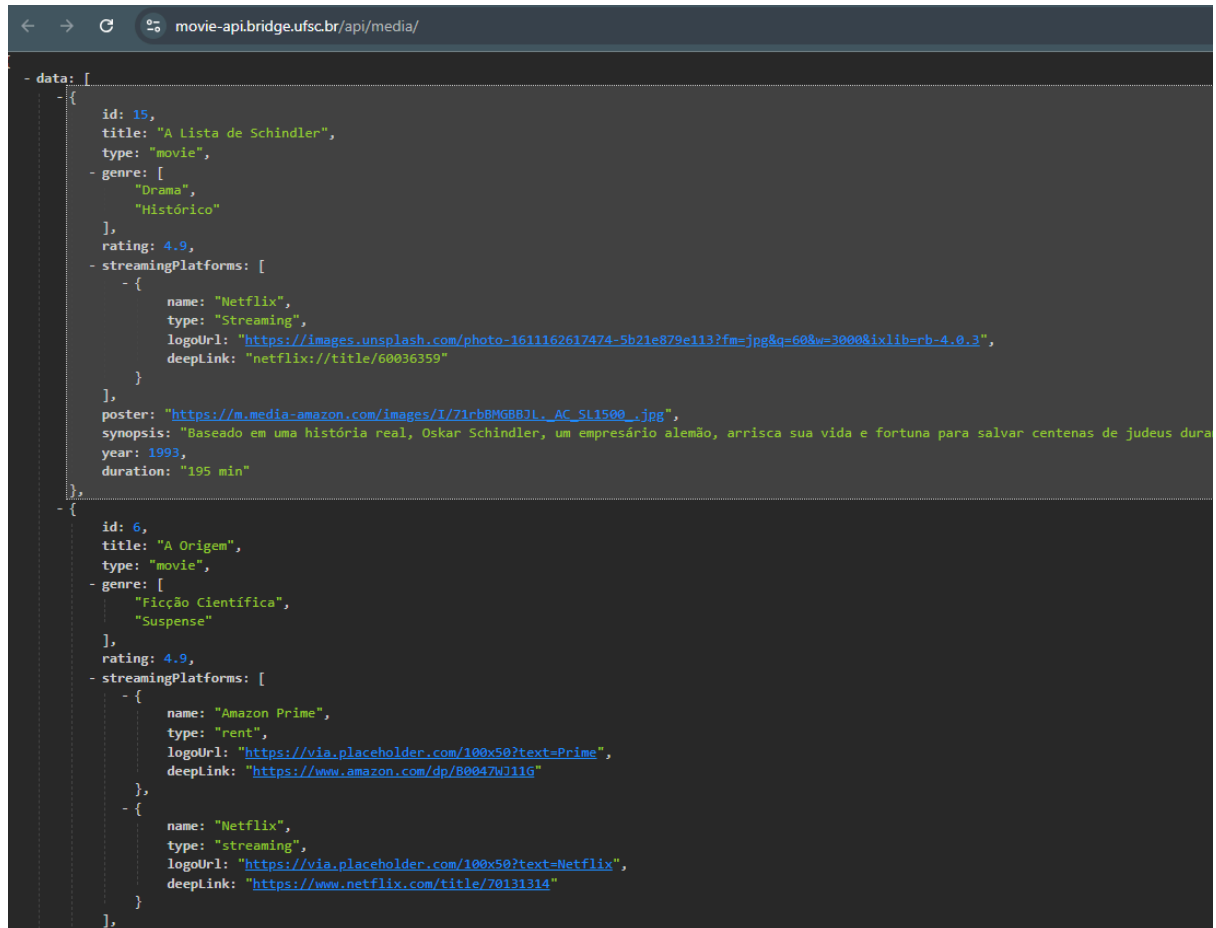
Além disso, adaptei a classe `Medium` para incluir um novo atributo obrigatório: uma lista de Ratings. No `factory Medium.fromJson`, foi adicionada a lógica para mapear corretamente as avaliações recebidas da API.

Com essa refatoração, o modelo de domínio passou a ser mais expressivo e preparado para novas expansões. Agora é possível não apenas exibir a nota média, mas também mostrar **comentários de usuários, datas de avaliação e múltiplas opiniões** sobre um mesmo conteúdo, enriquecendo a experiência do aplicativo e abrindo espaço para funcionalidades futuras.

2.4 Uso dos dados do endpoint `/media`

Outro problema identificado foi o uso de **dados hardcoded** no código, enquanto a API já disponibilizava informações completas na rota oficial:

<https://movie-api.bridge.ufsc.br/api/media/>



```
- data: [
  - {
    id: 15,
    title: "A Lista de Schindler",
    type: "movie",
    - genre: [
      "Drama",
      "Histórico"
    ],
    rating: 4.9,
    - streamingPlatforms: [
      - {
        name: "Netflix",
        type: "Streaming",
        logoUrl: "https://images.unsplash.com/photo-1611162617474-5b21e879e113?fm=jpg&q=60&w=300&ixlib=rb-4.0.3",
        deepLink: "netflix://title/60036359"
      }
    ],
    poster: "https://m.media-amazon.com/images/I/71rb8MGBB7L._AC_SL1500_.jpg",
    synopsis: "Baseado em uma história real, Oskar Schindler, um empresário alemão, arrisca sua vida e fortuna para salvar centenas de judeus dura",
    year: 1993,
    duration: "195 min"
  },
  - {
    id: 6,
    title: "A Origem",
    type: "movie",
    - genre: [
      "Ficção Científica",
      "Suspense"
    ],
    rating: 4.9,
    - streamingPlatforms: [
      - {
        name: "Amazon Prime",
        type: "rent",
        logoUrl: "https://via.placeholder.com/100x50?text=Prime",
        deepLink: "https://www.amazon.com/dp/B0047WJ116"
      },
      - {
        name: "Netflix",
        type: "streaming",
        logoUrl: "https://via.placeholder.com/100x50?text=Netflix",
        deepLink: "https://www.netflix.com/title/70131314"
      }
    ]
  },
],
```

Ao inspecionar a resposta da API, percebi que existia o campo **streamingPlatform**, trazendo informações de onde cada conteúdo estava disponível. No entanto, esses dados não estavam sendo processados pela aplicação, na classe **Medium**, o que resultava em uma interface limitada e inconsistências na representação.

Para corrigir isso, fiz as seguintes alterações:

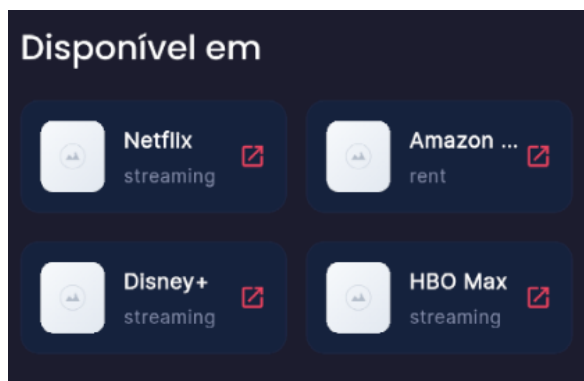
- Criei a classe **StreamingPlatform**, responsável por representar de forma clara as plataformas de streaming retornadas pela API.

```

1 class StreamingPlatform {
2   final String name;
3   final String type;
4   final String logoUrl;
5   final String deepLink;
6
7   StreamingPlatform({
8     required this.name,
9     required this.type,
10    required this.logoUrl,
11    required this.deepLink,
12  });
13
14  factory StreamingPlatform.fromJson(Map<String, dynamic> json) {
15    return StreamingPlatform(
16      name: json['name'] ?? 'Plataforma',
17      type: json['type'] ?? 'Streaming',
18      logoUrl: json['logoUrl'] ?? 'https://images.unsplash.com/photo-1611162617474-5b21e879e113?fm=jpg&q=60&w=3000&ixlib=rb-4.0.3',
19      deepLink: json['deepLink'] ?? '',
20    );
21  }
22 }

```

- Ajustei o modelo **Medium** e seu **fromJson** para mapear corretamente esse novo campo.
- Os dados fixos (*hardcoded*) da aplicação foram substituídos por chamadas dinâmicas à API, permitindo que o conteúdo seja atualizado em tempo real.
- Como parte da **refatoração**, a ordem dos atributos na classe de modelo **Medium** foi ajustada para corresponder à ordem dos campos na resposta da API. Essa padronização aumenta a clareza do código e simplifica o processo de debug e manutenção.
- Atualizei a UI para exibir essa informação:
 - **No card da tela principal**, mostrei as plataformas de streaming de forma resumida.
 - **Na hero section da página de detalhes**, exibi as plataformas de forma destacada, permitindo ao usuário identificar facilmente onde assistir o conteúdo.
 - OBS.: perceba o erro na imagem a direita, falo mais sobre na seção de testes da classe **StreamingPlatform**.



2.5 Tela de favoritos e persistência

Um dos requisitos principais era oferecer ao usuário a possibilidade de salvar conteúdos como favoritos e acessá-los em uma tela dedicada. Para implementar essa funcionalidade, trabalhei em duas frentes: a **organização do estado** e a **integração com a interface**.

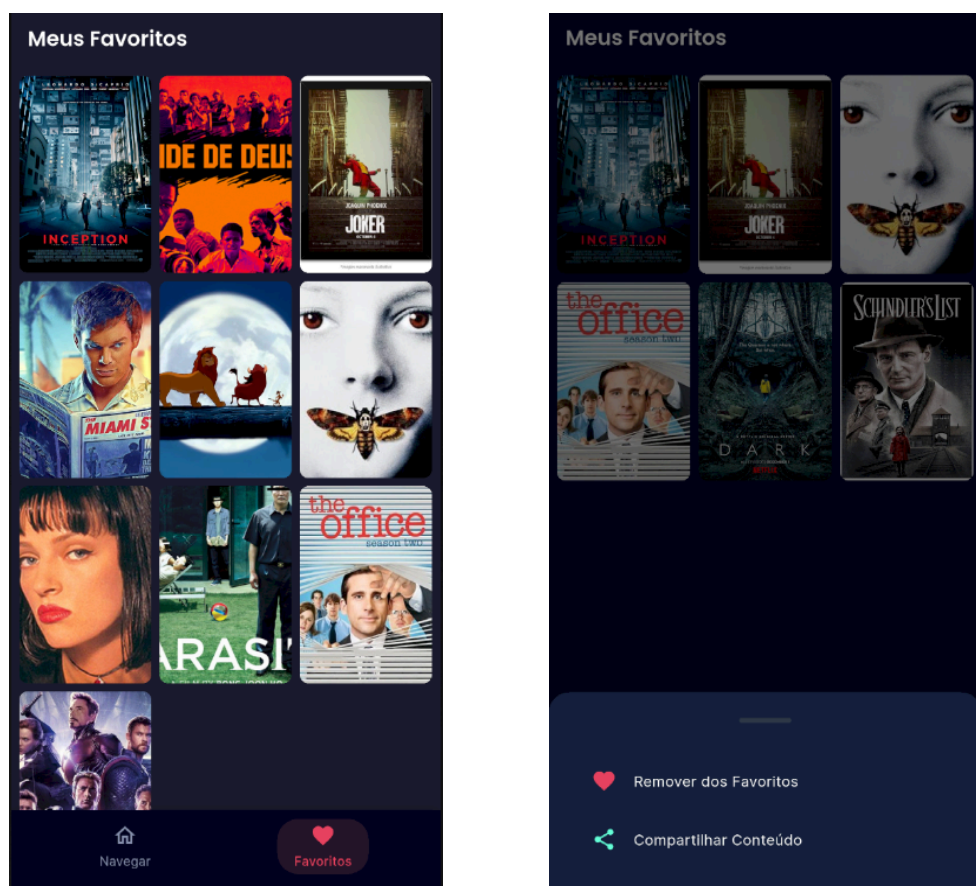
Primeiro, criei um **FavoritesProvider**, responsável por manter a lista de itens favoritos. Essa classe centraliza toda a lógica de adicionar, remover e verificar se um conteúdo já está salvo, garantindo uma fonte única de verdade acessível em qualquer tela do aplicativo. O provider foi injetado no topo da árvore de widgets, no `main.dart`, de modo que todas as telas compartilham o mesmo estado.

```
1  import 'package:flutter/material.dart';
2  import '../model/medium.dart';
3
4  class FavoritesProvider extends ChangeNotifier {
5    final List<Medium> _favorites = [];
6
7    List<Medium> get favorites => _favorites;
8
9    bool isFavorite(Medium content) => _favorites.contains(content);
10
11    void addFavorite(Medium content) {
12      if (!_favorites.contains(content)) {
13        _favorites.add(content);
14        notifyListeners();
15      }
16    }
17
18    void removeFavorite(Medium content) {
19      _favorites.remove(content);
20      notifyListeners();
21    }
22
23    void toggleFavorite(Medium content) {
24      if (isFavorite(content)) {
25        removeFavorite(content);
26      } else {
27        addFavorite(content);
28      }
29    }
30  }
```

```
24  runApp(
25    ChangeNotifierProvider(
26      create: (context) => FavoritesProvider(),
27      child: MyApp(),
28    ),
29  );
```

Na sequência, integrei esse gerenciamento de favoritos à interface. A tela de detalhes passou a reagir dinamicamente ao estado do provider: o botão de favoritar agora muda de ícone e cor conforme o item já está salvo ou não. Essa lógica também foi aplicada aos botões de ação reutilizáveis, que ficaram mais limpos, sem depender de estado interno.

Com a base pronta, construí a **FavoritesScreen**, onde o usuário pode visualizar sua coleção. Para dar mais impacto visual, organizei os pôsteres em uma grade de três colunas, inspirada no layout do *Letterboxd*. Além disso, cada pôster já foi envolvido por um widget **Semantics** com uma descrição falada, garantindo acessibilidade para leitores de tela. Ao tocar em um pôster, o usuário é levado para a página de detalhes correspondente.



Um ponto de atenção foi que as imagens retornadas pela API não vinham todas no mesmo tamanho. Isso dificultou manter um alinhamento perfeito entre os *posters* na grade.

Posteriormente, implementei também o botão `_buildQuickActions()`, já existente na tela principal, permitindo que o usuário acesse ações rápidas ao clicar e segurar em um item da tela de favoritos. Essa inclusão repetiu um pouco de código, o que sei não ser uma boa prática, mas considerei válido no contexto do desafio para garantir consistência de interação entre as telas. Em uma refatoração futura, esse trecho deve ser extraído para um componente compartilhado, reduzindo a duplicação.

3. Implementações Adicionais (Bônus)

Além dos requisitos funcionais obrigatórios, dediquei um esforço extra para implementar práticas de qualidade que tornassem a aplicação mais sustentável, testável e inclusiva. Não segui uma ordem rígida: fui alternando entre testes, ajustes de arquitetura e melhorias de acessibilidade conforme os pontos mais críticos apareciam durante o desenvolvimento. Dentre eles:

- **Testes Unitários:** Cobertura de testes para a lógica de negócio e a camada de serviço.
- **Acessibilidade:** Refinamentos na interface para garantir uma experiência de uso completa para usuários de leitores de tela (TalkBack).
- **Arquitetura e Inversão de Dependência (DI):** Refatoração da camada de acesso a dados para um design desacoplado e testável.
- **Usabilidade:** Alguns adicionais que melhoram a experiência do usuário.

3.1 Testes unitários

FavoritesProvider

O primeiro passo foi criar um grupo de **testes unitários para o FavoritesProvider**, já que ele concentra a lógica de adicionar e remover itens da lista de favoritos. Esse teste garantiu que todas as operações básicas de persistência e de verificação (`isFavorite`, `addFavorite`, `removeFavorite`, `toggleFavorite`) funcionassem corretamente, além de assegurar que o estado fosse notificado e refletido na interface. Veja um trecho exemplar:

```
7 void main() {
8   group('FavoritesProvider', () {
9     test('Adicionar um item à lista de favoritos', () {
10      // Arrange
11      final favoritesProvider = FavoritesProvider();
12      final testMedium = Medium(
13        id: 1,
14        title: 'Teste - O Filme',
15        type: MediaType.movie,
16        genres: [],
17        rating: 5.0,
18        streamingPlatforms: [],
19        synopsis: 'synopsis',
20        year: 2025,
21        duration: '90 min',
22        ratings: []
23      );
24
25      // Act
26      favoritesProvider.addFavorite(testMedium);
27
28      // Assert
29      // Esperamos que contenha o testMedium
30      expect(favoritesProvider.favorites.contains(testMedium), isTrue);
31      // Esperamos que a lista não esteja vazia
32      expect(favoritesProvider.favorites.isNotEmpty, isTrue);
33    });
34  });
35 }
```

FilterData

Implementei também um conjunto de testes para a classe **FilterData**, validando a conversão de filtros em parâmetros de consulta (`toQueryParams`), e também a geração de mensagens amigáveis para o usuário (`toReadableString`).

No primeiro caso, os testes asseguraram que filtros aplicados — como tipo, gêneros ou nota mínima — eram convertidos corretamente em parâmetros de URL.

Medium

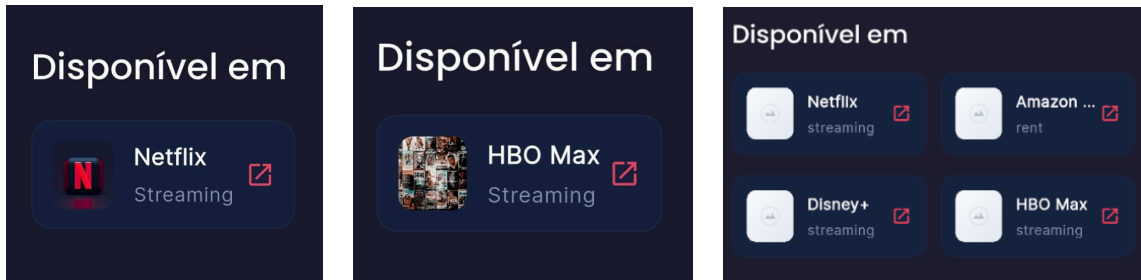
Também implementei testes unitários para o mapeamento de JSON em objetos **Medium**, garantindo que a desserialização funcionasse corretamente tanto para filmes quanto para séries. Foram validados cenários como:

- **Filme completo:** verificar se todos os campos obrigatórios (id, título, tipo, poster, ano, etc.) são parseados corretamente.
- **Campos opcionais ausentes:** assegurar que, quando o campo **poster** não existe no JSON, o atributo correspondente fica como **null** sem quebrar o parsing.
- **Série com temporadas e episódios:** conferir se valores de **seasons** e **episodes** são atribuídos corretamente quando o tipo é **series**.
- **Filme sem temporadas/episódios:** garantir que, quando o tipo é **movie**, esses campos permaneçam nulos.

```
27 test('Lidar com campos opcionais ausentes (ex: poster)', () {
28     // Arrange
29     // Este JSON não tem a chave 'poster'.
30     final Map<String, dynamic> json = {
31         'id': 2, 'title': 'Filme Sem Poster', 'type': 'movie', 'genre': [],
32         'rating': 4.0, 'streamingPlatforms': [], 'synopsis': '',
33         'year': 2024, 'duration': '100 min', 'ratings': []
34     };
35
36     // Act
37     final medium = Medium.fromJson(json);
38
39     // Assert
40     expect(medium.poster, isNull);
41 };
```

StreamingPlatform

Notei que as imagens das plataformas não estavam sendo renderizadas corretamente em alguns casos, por isso, implementei também testes para a classe **StreamingPlatform**, garantindo que a desserialização do JSON funcionasse corretamente em diferentes cenários.



Validei casos com todos os campos preenchidos, com valores nulos e com dados parciais, assegurando que os placeholders definidos no modelo fossem aplicados de forma consistente. Todos os testes passaram, confirmando que a lógica de parsing está correta. **Embora ainda haja um comportamento inesperado na interface** (exibindo `no-image.png`), não houve tempo hábil para depurar essa discrepância.

Testes mockados do Dio (ideia futura)

Embora eu tenha implementado testes unitários para classes de modelo e provedores, ainda não cheguei a criar testes mockados para o cliente HTTP Dio, que seria uma evolução na cobertura do projeto.

Com a adoção de injeção de dependência, o **MediaService** já está preparado para receber uma instância de **Dio** substituída em tempo de teste. Isso permitiria simular chamadas à API sem depender de rede real.

Essa estratégia tornaria a camada de comunicação testável, aumentando a confiança em futuras refatorações. Apesar de não ter sido implementada dentro do prazo do desafio, é um próximo passo para fortalecer a qualidade de projetos assim.

3.2 Arquitetura e Inversão de Dependência (DI)

Durante o desenvolvimento, percebi que o código original estava bastante acoplado à implementação do cliente HTTP. Cada função global que fazia chamadas à API, como `getMediaPage()`, criava uma nova instância de `Dio` dentro de si. Essa abordagem trazia alguns problemas: além da repetição de código e da criação desnecessária de várias instâncias, qualquer configuração de rede precisava ser replicada manualmente e não havia como testar a camada de serviço sem realizar requisições reais.

Para resolver isso, refatorei a comunicação com a API em torno de uma arquitetura mais limpa. Criei a classe `MediaService`, que passou a concentrar toda a lógica de chamadas externas. Essa classe deixou de instanciar diretamente o `Dio` e passou a receber a dependência pelo construtor, aplicando o princípio da **injeção de dependência**.

```
9 class MediaService {
10   final Dio _dio;
11
12   MediaService({required Dio dio}) : _dio = dio;
13   Future<Page<Medium>> getMediaPage({
14     int page = 1,
15     FilterData? filterData,
16   }) async {
17     final response = await _dio.get(
18       Endpoints.media(),
19       queryParameters: {
20         'page': page,
21         if (filterData != null) ...filterData.toQueryParams(),
22       },
23     );
24
25     if (response.statusCode == 200) {
26       final jsonResponse = (response.data as Map<String, dynamic>);
27       return Page<Medium>.fromJson(
28         jsonResponse,
29         (json) => Medium.fromJson(json),
30       );
31     } else {
32       throw Exception('Failed to load movies');
33     }
34   }
35
36   Future<List<Actor>> getMediumCast(int id) async {
37     final response = await _dio.get(Endpoints.cast(id: id));
38
39     if (response.statusCode == 200) {
40       List jsonResponse = response.data;
41
42       return jsonResponse.map((actor) => Actor.fromJson(actor)).toList();
43     } else {
44       throw Exception('Failed to load movie cast');
45     }
46   }
47 }
```

Para gerenciar essas instâncias de forma prática, adotei o pacote `get_it`, que funciona como um Service Locator. Configurei um arquivo `locator.dart` onde registrei o `Dio` e o próprio `MediaService` como **singletons**, garantindo que toda a aplicação use apenas uma instância de cada um. O locator foi inicializado no

`main()`, e os widgets que antes chamavam funções globais foram refatorados para obter o `MediaService` diretamente pelo locator.

Essa mudança trouxe diversas vantagens: o código ficou desacoplado e testável, já que agora conseguiríamos injetar um `Dio` simulado em testes unitários, a configuração de rede ficou centralizada e a manutenção passou a ser mais simples, pois trocar o cliente HTTP no futuro exige alteração em apenas um ponto.

3.3 Acessibilidade

TalkBack

O objetivo aqui foi garantir que pessoas que utilizam leitores de tela, como o TalkBack, pudessem navegar e compreender os conteúdos com clareza.

Comecei adicionando **descrições de controles e elementos visuais** que não possuíam texto explícito. Botões compostos apenas por ícones, como os de voltar, compartilhar ou filtrar, receberam o parâmetro `tooltip` com descrições curtas e diretas, algumas já tinham, porém estavam em inglês. No caso do botão de favoritos, a mensagem foi dinâmica, alternando entre “Adicionar aos favoritos” e “Remover dos favoritos” conforme o estado do item. Na tela de favoritos, cada pôster de filme ou série foi envolvido com `Semantics`, recebendo uma etiqueta que anuncia informações adicionais — duração no caso de filmes e quantidade de temporadas para séries.

Na **tela principal**, o widget de card de conteúdo recebeu uma etiqueta semântica completa:

```
369 return Semantics(  
370   label: '${content.title}, gêneros: ${content.genres.join(', ')}, avaliação: ${content.rating} de 5.',  
371   button: true,  
372   onTapHint: 'ver os detalhes',  
373   onTap: () => _onContentTap(content),  
374   child: ExcludeSemantics(  
375     child: ContentCardWidget(  
376       content: content,  
377       onTap: () => _onContentTap(content),  
378       onFavorite: () => _onFavorite(content),  
379       onShare: () => _onShare(content),  
380     ),  
381   ),  
382 );
```

Assim, o leitor de tela anuncia título, gêneros e avaliação, além de indicar claramente que o item é clicável e qual ação será tomada.

Na **FavoriteScreen**, cada pôster também foi envolvido em `Semantics`, garantindo que a navegação fosse clara, uma vez que o ícone do filme não tinha nenhuma informação textual que pudesse ser lida pelo TalkBack.

```

124   child: Semantics(
125       label: medium.title,
126       onTapHint: 'ver detalhes',
127       button: true,
128       child: ClipRRect(
129         borderRadius: BorderRadius.circular(8.0),
130         child: CustomImageWidget(
131           imageUrl: medium.poster ?? '',
132           fit: BoxFit.cover,
133         ),
134       ),
135     ),

```

Também refinei o **contexto e estado de componentes customizados**. O [ContentTypeToggleWidget](#), responsável por alternar entre “Todos”, “Filmes” e “Séries” na tela de filtro, passou a informar se uma opção está selecionada e sua posição dentro do grupo (ex.: “Opção 2 de 3”).

Durante os testes, identifiquei situações de **fala redundante**, em que o leitor de tela repetia a etiqueta customizada e depois lia todos os textos internos do widget. Para corrigir esse problema, utilizei [ExcludeSemantics](#), garantindo uma narração mais limpa e sem repetições.

A acessibilidade é um trabalho contínuo. Novos testes ainda serão necessários para avaliar a experiência em diferentes cenários, ajustar etiquetas e refinar a navegação, porém fico impedida pelo prazo do desafio.

Escalabilidade de Fonte

Foi identificado um ponto crítico na arquitetura inicial do aplicativo, no arquivo `main.dart`, o escalonamento dinâmico de fontes. O código forçava o fator de escala de texto (`textScaler`) a um valor estático de `1.0`, fazendo com que o aplicativo ignorasse as preferências de tamanho de fonte definidas pelo usuário nas configurações do sistema operacional.

Essa implementação representa uma barreira de usabilidade significativa, especialmente para usuários com baixa visão ou idosos, que dependem de fontes maiores para uma leitura confortável.

Embora a correção completa — que envolve a remoção deste bloqueio e o ajuste de todos os layouts para garantir que não haja overflows visuais — não tenha sido implementada devido às restrições de tempo do desafio, a atenção a este problema é registrada como um ponto de alta prioridade para a evolução da acessibilidade do projeto.

```
33 class MyApp extends StatelessWidget {
34   @override
35   Widget build(BuildContext context) {
36     return Sizer(builder: (context, orientation, screenType) {
37       return MaterialApp(
38         title: 'CineList',
39         theme: AppTheme.lightTheme,
40         darkTheme: AppTheme.darkTheme,
41         themeMode: ThemeMode.dark,
42         builder: (context, child) {
43           return MediaQuery(
44             data: MediaQuery.of(context).copyWith(
45               // TO-DO: remover o tamanho da fonte estático
46               textScaler: TextScaler.linear(1.0),
47             ),
48             child: child!,
49           );
50         },
51         debugShowCheckedModeBanner: false,
52         routes: AppRoutes.routes,
53         onGenerateRoute: AppRoutes.onGenerateRoute,
54         initialRoute: AppRoutes.initial,
55       );
56     });
57   }
58 }
```

3.4 Usabilidade

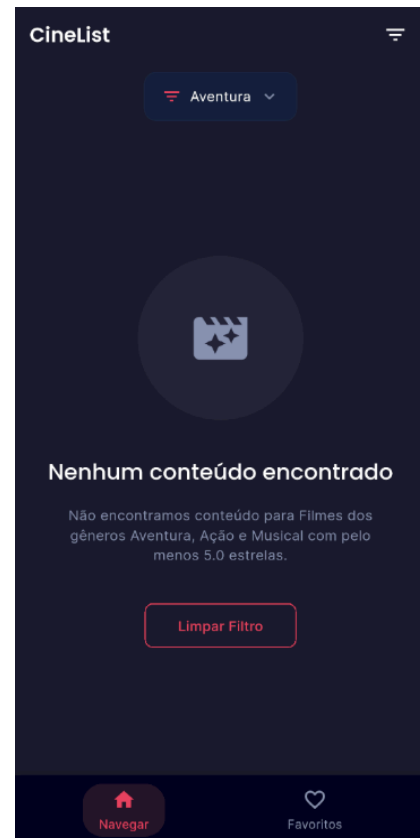
Além das funcionalidades previstas, implementei pequenas melhorias de **usabilidade** que tornam a experiência mais agradável.

Nenhum conteúdo encontrado no filtro

Um exemplo foi no sistema de filtros: quando o usuário aplicava filtros e nenhum conteúdo era encontrado, a mensagem exibida anteriormente era pouco amigável, com códigos crus que não faziam sentido fora do contexto técnico.

Para resolver isso, adicionei à classe `FilterData` o método `toReadableString()`, responsável por gerar uma frase descritiva mais natural. A função monta dinamicamente uma mensagem em português, considerando os filtros aplicados (tipo, gênero, ano e nota mínima).

```
55 String toReadableString() {
56     if (isEmpty) {
57         return 'Nenhum filtro aplicado.';
58     }
59
60     List<String> parts = [];
61
62     // Indica o tipo do conteúdo
63     if (type != null) {
64         parts.add('para ${type!.ptBrName}');
65     }
66
67     // Adiciona os gêneros (ex: "do gênero Ação e Drama")
68     if (genre != null && genre!.isNotEmpty) {
69         if (genre!.length == 1) {
70             parts.add('do gênero ${genre![0]}');
71         } else {
72             String lastGenre = genre!.removeLast();
73             parts.add('dos gêneros ${genre!.join(', ')} e ${lastGenre}');
74         }
75     }
76
77     // Adiciona o ano (ex: "de 1993")
78     if (year != null) {
79         parts.add('de $year');
80     }
81
82     // Adiciona a avaliação (ex: "com mais de 5.0 estrelas")
83     if (rating != null) {
84         parts.add('com pelo menos ${rating} estrelas');
85     }
86
87     // Junta as strings
88     return 'Não encontramos conteúdo ${parts.join(' ')}.';
89 }
```



Assim, em vez de mensagens genéricas, o usuário recebe algo mais claro, como:

“Não encontramos conteúdo para Séries do gênero Ação e Drama com pelo menos 4.5 estrelas.”

4. Conclusão

O desenvolvimento deste desafio foi uma oportunidade valiosa para consolidar práticas de programação, arquitetura, engenharia de software e usabilidade no contexto mobile com Flutter.

Gostaria de deixar um agradecimento ao Laboratório Bridge pela oportunidade de participar deste desafio. E por fim, espero que este material seja útil como referência na avaliação.

5. Referências

- Krug, S. ***Don't Make Me Think: A Common Sense Approach to Web Usability***. New Riders, 2014.
- **Documentação oficial** do Flutter. <https://flutter.dev/docs>
- Package **provider** – Flutter. <https://pub.dev/packages/provider>
- Package **get_it** – Flutter. https://pub.dev/packages/get_it
- Package **flutter_test** – Flutter.
https://api.flutter.dev/flutter/flutter_test/flutter_test-library.html
- Flutter & Dart - The Complete Guide [2025 Edition].
<https://www.udemy.com/course/learn-flutter-dart-to-build-ios-android-apps/?couponCode=KEEPLARNINGBR>
- Flutter. ***Building in Accessibility with Flutter (Flutter Interact '19)***. YouTube. Disponível em: <https://youtu.be/bWbBgbmAdQs>. Acesso em: set. 2025.
- Flutter Mapp. ***Flutter Unit Testing - Fast & Simple***. YouTube. Disponível em: <https://youtu.be/jSaoTC1ULB8>. Acesso em: set. 2025.
- Flutter Guys. ***Dependency Injection in Flutter - You HAVE to Use it !***. YouTube. Disponível em: <https://youtu.be/2YEkdAE0-j4>. Acesso em: set. 2025.
- FilledStacks. ***Flutter Dependency Inversion For Beginners | Complete Guide***. YouTube. Disponível em: <https://youtu.be/vBT-FhgMaWM>. Acesso em: set. 2025.