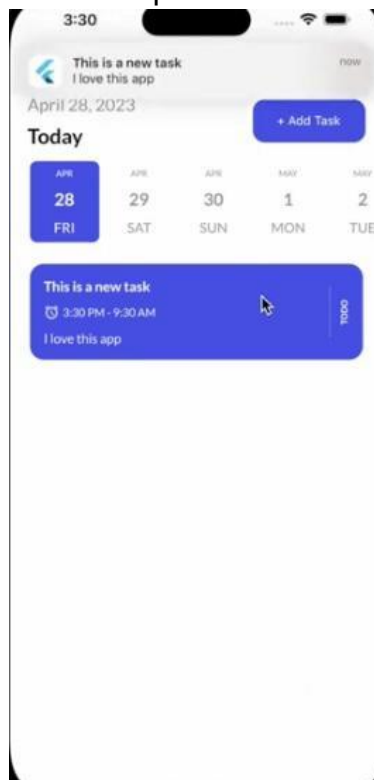


DESENVOLVIMENTO DO APP FLUTTER

Jonathan Gomes RA: 82311794
Giuliano Poyatos RA: 823128723
Rafael Oliveira RA: 12524145204
Natalia Barbosa RA: 1282312705
Gustavo Novais Lima RA: 823114572


IMPLEMENTAÇÃO DAS TELAS

Tela Principal



Tela de Adicionar/Editar Tarefa

3:29


< 

Add Task



Title

Note


Date


Start Time **End Time**




Remind

Repeat


Color

Create Task

Tabela de Configuração de tarefa

3:29


< 

Add Task



Title

Note


Date


Start Time **End Time**




Remind

Repeat

Color

Create Task

ENTER TIME

3

:


30

Hour

Minute

AM

PM



CANCEL

OK

FUNCIONALIDADES

Tela Principal (Lista de Tarefas)

Este código é uma implementação de uma tela principal (HomePage) de um aplicativo de gerenciamento de tarefas usando o framework Flutter. A aplicação permite ao usuário adicionar, visualizar e gerenciar tarefas diárias. Aqui está um resumo de como o código foi estruturado:

Dependências: O código importa várias bibliotecas e pacotes, como:

flutter/material.dart para a interface do usuário,
date_picker_timeline para um seletor de datas,
get para gerenciamento de estado e navegação,
google_fonts para fontes personalizadas,
flutter_svg para mostrar imagens SVG, etc.

Estrutura de Estado: A HomePage é um StatefulWidget com o estado gerenciado por _HomePageState. O estado mantém informações como a data selecionada (_selectedDate) e o controlador de tarefas (_taskController).

Notificações: O código também configura um serviço de notificações usando a classe NotifyHelper. Isso permite exibir notificações para o usuário, por exemplo, quando uma tarefa é concluída ou o tema da aplicação é alterado.

Barra de Data: O widget _dateBar exibe um seletor de data que permite ao usuário escolher um dia. O formato da data é personalizado com a ajuda do pacote GoogleFonts.

Adição de Tarefa: A barra de adicionar tarefa (_addTaskBar) contém um botão que navega para a página de adicionar tarefa (AddTaskPage). Quando uma tarefa é adicionada, a lista de tarefas é atualizada.

Exibição de Tarefas: O método _showTasks exibe as tarefas em uma lista. As tarefas são filtradas com base na data selecionada. Se não houver tarefas, uma mensagem personalizada é exibida.

Animação: O código utiliza o pacote flutter_staggered_animations para animar a exibição das tarefas, tornando a interface mais dinâmica.

Interação com Tarefas: Quando uma tarefa é clicada, um "bottom sheet" (uma área deslizando na parte inferior da tela) é mostrado, oferecendo opções como marcar a tarefa como concluída ou excluí-la.

Alteração de Tema: O aplicativo suporta alternância entre modo claro e escuro, e essa troca é gerenciada por ThemeService com notificações sobre a mudança.

Serviços Externos:

O TaskController gerencia as tarefas (adicionar, excluir, concluir).

O NotifyHelper gerencia as notificações locais para lembrar o usuário sobre tarefas agendadas.

Fluxo de uso:

O usuário vê a data atual e as tarefas relacionadas a ela.
Ele pode adicionar tarefas, marcar como concluídas ou excluir.
O tema pode ser alterado entre claro e escuro.
As notificações são agendadas para lembrar o usuário sobre tarefas.
Este código usa práticas comuns em Flutter, como gerenciamento de estado com GetX, animações para uma interface mais agradável e notificação local para lembretes de tarefas.

Estrutura Básica:

O AddTaskPage é uma StatefulWidget, com seu estado gerenciado pela classe _AddTaskPageState.

A página contém um formulário onde o usuário pode inserir detalhes sobre a tarefa, como título, nota, data, hora de início e término, lembretes, repetição, e cor associada à tarefa.

Controllers e Variáveis:

O controlador TaskController é utilizado para manipular as tarefas no banco de dados.

Usam-se TextEditingController para capturar texto dos campos de entrada (título e nota).

A data e hora atuais são usadas como valores iniciais para os campos de data e hora.

O tempo de início e término são formatados utilizando o DateFormat.

Campos do Formulário:

Título e Nota: São campos de entrada para o nome e descrição da tarefa.

Data: Um botão abre um seletor de data para que o usuário escolha a data da tarefa.

Horas de Início e Término: São campos para o horário de início e término da tarefa, com um seletor de hora.

Lembrete: Um menu suspenso permite que o usuário escolha o tempo de lembrete antes da tarefa (em minutos).

Repetição: Um menu suspenso permite escolher a frequência de repetição da tarefa (Nunca, Diariamente, Semanalmente, Mensalmente).

Cor: O usuário pode escolher a cor da tarefa, com três opções (primária, rosa, amarela), representadas por círculos coloridos.

Ações do Usuário:

Salvar Tarefa: Quando o usuário preenche os campos e clica no botão "Criar Tarefa", o sistema valida se todos os campos obrigatórios foram preenchidos. Se estiverem, a tarefa é salva no banco de dados. Caso contrário, um aviso é exibido.

Navegação: O botão "Criar Tarefa" chama o método _validateInputs, que verifica se os campos estão preenchidos antes de adicionar a tarefa. Se bem-sucedido, retorna para a página anterior.

Interatividade:

Seleção de Hora e Data: O método `_getDateFromUser` exibe um seletor de data, enquanto o método `_getTimeFromUser` exibe um seletor de hora para definir o horário de início e término da tarefa.

Mudança de Cor: Quando o usuário clica em uma das opções de cor (representada por círculos coloridos), a cor da tarefa é alterada. A cor escolhida é armazenada em `_selectedColor`.

Interface Visual:

O layout é organizado em um `SingleChildScrollView` para permitir o deslocamento da tela caso o teclado seja exibido.

A interface exibe todos os campos de entrada e inclui ícones interativos para selecionar a data e hora.

AppBar e Navegação:

Um AppBar simples é usado com um ícone de voltar à página anterior. O fundo da barra e o estilo visual da página se ajustam conforme o tema (claro ou escuro).

Validando e Salvando a Tarefa:

Após a validação, a tarefa é adicionada ao banco de dados através do `TaskController` com o método `_addTaskToDB`. A tarefa é criada com os dados coletados e salva em um formato `Task`.

Herança de `GetXController`:

O `TaskController` herda de `GetXController`, que é uma classe fornecida pelo pacote `GetX` para gerenciamento de estado e ciclo de vida do controlador.

O `onReady()` é substituído para garantir que a lista de tarefas seja carregada assim que o controlador estiver pronto, chamando o método `getTasks()`.

2. Variável `taskList`:

`taskList` é uma lista observável de tarefas (`RxList<Task>`) que será usada para armazenar todas as tarefas carregadas do banco de dados.

Como é uma lista reativa (`RxList`), qualquer mudança nela (como adicionar, excluir ou atualizar tarefas) será automaticamente refletida na interface do usuário que estiver ouvindo essa variável.

3. Método `addTask`:

O método `addTask` adiciona uma nova tarefa ao banco de dados.

Ele recebe um objeto `Task` e o passa para o `DBHelper.insert()` para inserção no banco de dados. O `DBHelper` provavelmente é uma classe que abstrai a interação com o banco de dados (SQLite, por exemplo).

4. Método `getTasks`:

O método `getTasks` busca todas as tarefas armazenadas no banco de dados.

Ele chama `DBHelper.query()` para recuperar os dados e, em seguida, mapeia esses dados para objetos `Task`, que são adicionados à lista `taskList` utilizando `taskList.assignAll()`.

O uso de `assignAll` é uma maneira de atualizar a lista com novos dados. O `RxList` observa essas alterações e atualiza a interface do usuário automaticamente.

5. Método `deleteTask`:

O método `deleteTask` recebe uma tarefa, chama `DBHelper.delete(task)` para excluir essa tarefa do banco de dados e, em seguida, recarrega a lista de tarefas chamando `getTasks()`.

6. Método `markTaskCompleted`:

Este método marca uma tarefa como concluída. Ele recebe um id da tarefa, chama `DBHelper.update(id)` para atualizar o status da tarefa no banco de dados (possivelmente alterando o campo `isCompleted` ou outro campo relevante), e então recarrega a lista de tarefas chamando `getTasks()` novamente.

Resumo do Funcionamento:

Gerenciamento de Estado com GetX: O `taskList` é uma lista reativa. Quando qualquer mudança ocorre nela (como adicionar, excluir ou atualizar tarefas), a interface do usuário que está ouvindo essa lista será atualizada automaticamente.

Interação com o Banco de Dados: O `DBHelper` é responsável por realizar as operações no banco de dados, como inserção, consulta, exclusão e atualização das tarefas.

Ciclo de Vida do `GetXController`: Quando o controlador é preparado (no método `onReady()`), ele chama o método `getTasks()` para carregar as tarefas da tabela no banco de dados, garantindo que os dados sejam exibidos quando a página for carregada.

Observações Importantes:

A classe `TaskController` assume que há uma classe `Task` que possui métodos como `fromJson` (para criar um objeto `Task` a partir de um `Map<String, dynamic>`) e que o `DBHelper` é responsável por todas as operações CRUD (Criar, Ler, Atualizar, Excluir) no banco de dados.

O uso de GetX facilita a gestão do estado reativo no aplicativo, fazendo com que a interface do usuário seja atualizada de forma eficiente sempre que os dados mudam.

Este controlador é fundamental para a gestão das tarefas, garantindo que o banco de dados seja acessado corretamente e que a interface do usuário seja reativa a essas mudanças.

ESTILO

O código fornecido define uma estrutura para os temas e estilos de texto utilizados no aplicativo Flutter, adaptados para o modo claro e escuro, usando o pacote GetX para facilitar a troca entre os modos e o pacote GoogleFonts para personalização das fontes. Aqui está uma explicação detalhada do que cada parte faz:

1. Definição de Cores:

Cores Primárias e Secundárias:

`bluishClr`: Um tom de azul (utilizado como a cor principal do aplicativo).

`yellowClr`: Um tom de amarelo, possivelmente utilizado como uma cor de destaque.

`pinkClr`: Um tom de rosa, que pode ser usado para elementos como botões ou seleções.

`primaryClr`: Definido como `bluishClr`, é utilizado como a cor principal em ambas as versões de tema.

darkGreyClr: Um tom escuro de cinza, provavelmente usado para o fundo no modo escuro.

darkHeaderClr: Cor do cabeçalho no modo escuro (um tom de cinza escuro).

2. Temas:

Tema Claro (light) e Tema Escuro (dark):

Utiliza a classe ThemeData para definir as cores e o brilho do tema.

O primaryColor é atribuído a bluishClr para ambos os temas, enquanto o colorScheme é ajustado para se adequar aos modos claro e escuro (superfície branca para o claro e superfície escura para o escuro).

O método brightness define se o tema será claro (Brightness.light) ou escuro (Brightness.dark).

3. Estilos de Texto:

Uso de GoogleFonts: O código usa o pacote GoogleFonts para estilizar o texto. A fonte Lato foi escolhida e aplicada em vários tipos de texto.

Estilos de Texto: São definidos diferentes estilos para cabeçalhos, subtítulos, corpo do texto, etc. A cor do texto é ajustada automaticamente para o modo claro ou escuro, com base em Get.isDarkMode, que verifica se o aplicativo está no modo escuro.

Estilos Específicos de Texto:

headingTextStyle: Usado para os títulos principais. A fonte tem tamanho 24, com peso em negrito.

subHeadingTextStyle: Usado para subtítulos, com um tamanho de 20 e peso médio.

titleTextStle: Para títulos menores ou seções importantes, com tamanho de 18 e peso em negrito.

subTitleTextStle: Usado para subtítulos menores ou texto explicativo, com tamanho 16.

bodyTextStyle: Usado para o texto principal ou corpo, com tamanho de 14 e peso regular.

body2TextStyle: Usado para textos auxiliares ou secundários, com uma cor ligeiramente mais suave.

4. Adaptação ao Modo Claro e Escuro:

O uso de Get.isDarkMode permite que o aplicativo altere dinamicamente as cores e os estilos de texto com base no modo de exibição. Quando o aplicativo está no modo escuro, as cores de texto são ajustadas para tons mais claros (como Colors.white ou Colors.grey[200]), enquanto no modo claro, o texto é geralmente mais escuro (como Colors.black ou Colors.grey[700]).

5. Uso do GetX para Troca de Tema:

Get.isDarkMode verifica se o modo escuro está ativado no dispositivo e altera os estilos de texto e cores de fundo automaticamente. Isso facilita a alternância entre os modos claro e escuro sem a necessidade de lógica adicional, proporcionando uma experiência mais fluida para o usuário.

Resumo e Benefícios:

Gerenciamento Simples de Temas: Usando GetX e GoogleFonts, o código permite alternar facilmente entre modos claro e escuro, mantendo a consistência na interface.

Fontes Personalizadas: A biblioteca GoogleFonts foi utilizada para garantir que as fontes utilizadas sejam esteticamente agradáveis e consistentes no aplicativo.

Design Responsivo: A interface se adapta ao modo do sistema automaticamente, otimizando a experiência do usuário independentemente da preferência de tema. Esse conjunto de definições de temas e estilos de texto ajuda a manter a interface do aplicativo fluida, moderna e facilmente personalizável para diferentes condições de iluminação.

TEMA:

1. Dependências Importadas:

`package:flutter/material.dart`: Importa o pacote principal do Flutter, que contém os widgets essenciais, incluindo o `ThemeMode` e os `ThemeData` necessários para definir os temas.

`package:get_storage/get_storage.dart`: Importa a biblioteca `GetStorage` que é usada para armazenar dados localmente no dispositivo de forma simples e eficiente.

`package:get/get.dart`: Importa o pacote `GetX`, que facilita o gerenciamento de estado e outros recursos, como a mudança de temas no aplicativo.

2. Classe `ThemeService`:

A classe `ThemeService` é responsável por gerenciar a troca de tema do aplicativo e armazenar a preferência do usuário (modo claro ou escuro) nos arquivos locais usando o `GetStorage`.

3. Propriedades:

`_box`: A instância do `GetStorage` usada para armazenar dados de forma local (no armazenamento persistente do dispositivo).

`_key`: A chave de armazenamento para armazenar o valor que indica se o modo escuro está ativado ou não. No caso, a chave é `'isDarkMode'`.

4. Métodos:

`theme`:

Esse método consulta o estado do tema no armazenamento local e retorna o valor adequado do tipo `ThemeMode`. Se o método `_loadThemeFromBox()` retornar `true`, o modo escuro será retornado (`ThemeMode.dark`), caso contrário, será retornado o modo claro (`ThemeMode.light`).

`_loadThemeFromBox()`:

Esse método carrega a preferência de tema do armazenamento local. Ele verifica se existe um valor associado à chave `_key` e retorna `true` (modo escuro) ou `false` (modo claro). Caso o valor não esteja presente, ele retorna `false`, que implica no modo claro.

`_saveThemeToBox(bool isDarkMode)`:

Esse método recebe um valor booleano e o salva no armazenamento local usando a chave `_key`. Ele é responsável por persistir a preferência do usuário para que o tema permaneça o mesmo quando o aplicativo for reiniciado.

`switchTheme()`:

Esse método alterna entre os modos claro e escuro, verificando o estado atual. Se o modo escuro estiver ativo (retornado por `_loadThemeFromBox()`), ele muda para o modo claro e vice-versa. Após isso, ele atualiza a preferência do tema no armazenamento local chamando o método `_saveThemeToBox()`.

5. Como Funciona a Troca de Tema:

O `switchTheme()` verifica o estado atual do tema, usa o `Get.changeThemeMode` para alternar entre `ThemeMode.dark` e `ThemeMode.light`, e em seguida, atualiza a preferência no armazenamento local.

O `Get.changeThemeMode` é um método de `GetX` que altera o tema do aplicativo em tempo real.

6. Uso da Classe `ThemeService`:

Para utilizar o `ThemeService` no seu aplicativo, você pode instanciar a classe em qualquer parte do código e chamar o método `switchTheme()` para alternar o tema.

NOTIFICAÇÃO

O código fornecido é responsável por gerenciar e exibir notificações locais em um aplicativo Flutter. Ele utiliza a biblioteca `flutter_local_notifications` para configurar e mostrar notificações no dispositivo, bem como manipula notificações periódicas e a navegação para uma nova tela quando o usuário interage com a notificação. Vamos explicar em detalhes as funcionalidades principais do código.

Estrutura do Código

1. Dependências Importadas:

`flutter_local_notifications`: Usada para enviar notificações locais.

`flutter_native_timezone`: Para acessar o fuso horário do dispositivo.

`rxdart`: Usada para gerenciar o fluxo de dados de forma reativa.

`timezone`: Para manipulação de fusos horários ao agendar notificações.

`get`: Para facilitar a navegação e gerenciamento de estado.

2. Classe `NotifyHelper`:

A classe `NotifyHelper` é responsável por gerenciar a lógica de notificações, incluindo a inicialização, configuração do fuso horário e exibição de notificações.

Métodos da Classe `NotifyHelper`:

`initializeNotification()`:

Inicializa o plugin de notificações e configura a plataforma (Android/iOS).

Define as configurações de inicialização para o iOS e Android.

Configura a lógica para quando o usuário interagir com a notificação (via `onDidReceiveNotificationResponse`).

`requestIOSPermissions()`:

Solicita permissões específicas para o iOS, como alertas, badges e sons para notificações.

`_configureLocalTimeZone()`:

Configura o fuso horário local, essencial para o agendamento de notificações em horários específicos.

`onDidReceiveLocalNotification()`:

Esse método é chamado quando o usuário recebe uma notificação local enquanto o app está em primeiro plano. Exibe um diálogo (`CupertinoAlertDialog`) com os detalhes da notificação e possibilita a navegação para uma nova tela (`SecondScreen`).

`displayNotification()`:

Exibe uma notificação simples no Android e no iOS com o título e corpo especificados. A notificação também tem um "payload", que pode ser usado para passar dados para a tela que será aberta após a interação.

`_nextInstanceOfTenAM()`:

Retorna a próxima instância de um horário específico (hora e minutos). Se a data já passou, o método agendará a notificação para o próximo dia.

`scheduledNotification()`:

Agenda uma notificação local para um horário específico usando o método `zonedSchedule`. Esse método leva em consideração o fuso horário local e é usado para notificações que precisam ser exibidas em momentos futuros específicos, como para tarefas ou lembretes.

`_configureSelectNotificationSubject()`:

Configura um `BehaviorSubject` para escutar as respostas das notificações e navegar para a `SecondScreen` com base no "payload" da notificação.

`periodicalyNotification()`:

Exibe uma notificação periódica que se repete a cada minuto, utilizando o método `periodicallyShow`.

3. Classe `SecondScreen`:

A classe `SecondScreen` exibe os detalhes de uma tarefa ou notificação quando o usuário interage com a notificação.

Ao tocar na notificação, o usuário é redirecionado para esta tela, onde ele pode visualizar o título, descrição e data da tarefa (baseados no "payload" que foi passado na notificação).

Tela `SecondScreen`:

A tela mostra o título, a descrição e a data de uma tarefa recebida como parte da carga útil ("payload") da notificação. O payload é passado como uma string formatada, e os dados são extraídos usando o método `split("|")`.

Exemplo de Fluxo de Trabalho:

Inicialização:

A função `initializeNotification()` é chamada no início do aplicativo, configurando o sistema de notificações locais para Android e iOS.

Exibição de Notificação:

O método `displayNotification()` é chamado para mostrar uma notificação, que pode ser programada para um horário específico ou exibida imediatamente.

Agendamento de Notificação:

O método `scheduledNotification()` é utilizado para agendar notificações em horários específicos.

