Flutter Dark Mode & Theming

Learn to implement dynamic light/dark themes and consistent styling for beautiful, adaptive Flutter apps.

Website reference: haas.dev | https://dev-roast-app.vercel.app

---

Introduction

Modern apps must support **light and dark modes** for better user experience and accessibility. Flutter makes theming flexible, allowing you to **apply global styles, switch themes dynamically, and create custom theme data** for consistent UI across your app.

Step 1: Setting Up Basic Themes

Define **light and dark theme data** in `MaterialApp`:

```
MaterialApp(
  title: 'MyApp',
  theme: ThemeData(
    brightness: Brightness.light,
    primaryColor: Colors.blue,
    accentColor: Colors.orange,
  ),
  darkTheme: ThemeData(
    brightness: Brightness.dark,
    primaryColor: Colors.blueGrey,
    accentColor: Colors.amber,
  ),
  themeMode: ThemeMode.system, // Auto-switch based on device
  home: HomeScreen(),
);
```

- `themeMode` can be `ThemeMode.light`, `ThemeMode.dark`, or `ThemeMode.system`.

Step 2: Switching Themes Dynamically

- Use a **state management solution** (Provider, Riverpod, GetX) to switch themes at runtime.

Example with **Provider**:

```
class ThemeNotifier extends ChangeNotifier {
  bool isDarkMode = false;

  void toggleTheme() {
    isDarkMode = !isDarkMode;
    notifyListeners();
  }
}

// In widget
IconButton(
  icon: Icon(Icons.brightness_6),
  onPressed: () {
    Provider.of<ThemeNotifier>(context, listen: false).toggleTheme();
  },
)
```

Step 3: Custom Theme Data

- Define a **centralized theme** for colors, fonts, and button styles:

```
class AppTheme {
  static final lightTheme = ThemeData(
    primaryColor: Colors.blue,
    scaffoldBackgroundColor: Colors.white,
    textTheme: TextTheme(bodyText1: TextStyle(color: Colors.black)),
  );

  static final darkTheme = ThemeData(
    primaryColor: Colors.blueGrey,
    scaffoldBackgroundColor: Colors.black,
    textTheme: TextTheme(bodyText1: TextStyle(color: Colors.white)),
  );
}
```

- Helps maintain **consistent UI** and simplifies updates.

Step 4: Theming Widgets Individually

- Some widgets may need **custom styles independent of global theme**:

```
ElevatedButton(
```

```
ElevatedButton(
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.red,
    padding: EdgeInsets.symmetric(horizontal: 24, vertical: 12),
  ),
  onPressed: () {},
  child: Text('Custom Button'),
)
```

- Override global theme only when necessary.

Step 5: Theming Best Practices

1. **Use centralized theme data** to ensure consistency.
2. **Leverage system theme** for automatic dark/light mode.
3. **Avoid hard-coded colors**; always reference ThemeData.
4. **Test on multiple devices** to ensure colors, fonts, and contrasts are readable.
5. **Consider accessibility**: high contrast, font scaling, and color-blind friendly palettes.

Step 6: Practical Exercise

Objective: Implement a Flutter app that supports dynamic dark/light mode:

1. Create light and dark themes using ThemeData.
2. Implement a toggle button to switch themes dynamically.
3. Use centralized theme for colors, fonts, and button styles.
4. Test on devices with system dark mode enabled.

Step 7: Key Takeaways

- Dark mode improves UX and accessibility.
- Centralized ThemeData ensures **consistency and maintainability**.
- Dynamic theme switching enhances user personalization.
- Always follow best practices for **colors, fonts, and contrasts**.

Visit **haas.dev** for more resources and guides.

https://dev-roast-app.vercel.app