Недостатки go_router и их нивелирование в go_router_wrapper.

Отмеченные разработчиками недочеты библиотеки при использовании можно условно разделить на две группы:

1. **Неожиданное и непредсказуемое поведение при redirect**: В некоторых случаях go_router может неправильно обрабатывать redirect-ы, особенно если они зависят от асинхронных потоков (например, в случае с authStateChanges()).
2. **Проблемы с состоянием**: В последних версиях были замечены проблемы с сохранением состояния маршрутов, что приводит к нестабильному поведению при навигации.

Предложенные решения в go_router_wrapper:

Динамическое управление маршрутами:

Добавлены restrict-механизмы и правила переходов в динамическом режиме, что недоступно в стандартной реализации.

Их задача - безопасность переходов и контроль над доступом к определенным маршрутам. Например, можно предотвратить переход на некоторые страницы, если пользователь не авторизован, или установить особые правила для маршрутов.

Управление типами переходов (transitions):

Расширена возможность управления типами переходов между экранами, такими как push, go, popUntil, pushNamed, goNamed. Такие запросы периодически озвучиваются в сообществе, например, для гибкости управления анимациями и переходами между экранами.

Обработка информации о типе навигации:

На нехватку информации о типах навигации часто жалуются разработчики в сообществе. Ожидаемые пути применения: 1. Дебаг (логи помогают отслеживать, какие именно переходы выполняются в приложении. Это помогает выявлять и устранять ошибки, связанные, к примеру, с некорректными переходами или зацикливанием маршрутов или выявить роуты, которые вызывают замедление работы или ошибки загрузки страниц). 2. Аналитика (получение этих параметров позволяет

прикрутить ивенты для генерации отчетов и статистики по использованию приложения). 3. Отлов подозрительной активности, например, частые переходы на страницы с ограниченным доступом.

Сохранение состояния:

Стандартный go_router не предоставляет встроенного механизма для сохранения и восстановления последнего маршрута после перезапуска приложения. go_router_wrapper решает эту проблему, сохраняя последний маршрут в SharedPreferences.

Контроль над маршрутами:

Проблема стандартной реализации заключается в том, что _CustomNavigatorState преобразует RouteMatch в GoRouteState, не предоставляя RouteMatch в route.pageBuilder или route.builder. Это означает, что невозможно определить, какой маршрут строится на текущем этапе, особенно при наложении двух маршрутов друг на друга. В обсуждениях сообщества предлагалось добавить RouteMatch в реализацию, что и было сделано в нашем коде.

Реализация:

1. Restrict-механизмы (Keepers)

Принимают решение, можно ли пользователю продолжить навигацию по запрашиваемому маршруту.

Проблемы, которые решают Keepers:

1. **Контроль доступа**:

Keepers позволяют ограничить доступ к определенным маршрутам для пользователей, которые не авторизованы или не имеют достаточных прав. Например, только авторизованные пользователи могут получить доступ к экранам профиля или корзины.

2. **Управление состоянием**:

Keepers помогают управлять состоянием приложения, определяя, можно ли продолжить навигацию на основе текущего состояния. Это может

включать проверку, завершены ли все необходимые действия (например, заполнены ли обязательные формы при авторизации).

3. **Навигационные ограничения**:

Keepers могут предотвращать доступ к маршрутам, если определенные условия не выполнены. Например, если пользователь пытается перейти на страницу оформления заказа без добавления товаров в корзину.

Преимущества использования Keepers:

- **Повышенная безопасность**: **Keepers** обеспечивают безопасность, ограничивая доступ к чувствительным разделам приложения.
- **Позитивный пользовательский опыт**: За счет управления навигацией можно предотвращать ситуации, когда пользователь переходит на некорректные страницы и теряется в неведении.
- **Гибкость и контроль**: возможность управлять поведением приложения, устанавливая условия для доступа к различным маршрутам.

Дополнительная информация - в файле keeper.dart.

Пример использования - в файле examples/main.dart.

Синтаксис Keepers:

```
GoRouterWrapper(
  keepers: [Keeper(onEnter: () => ...)],
  routes: [
    GoRoute(
      path: '/',
      pageBuilder: (context, state) => HomeScreen(),
      keepers: [Keeper(onEnter: () => ...)],
    ),
    GoRoute(
      path: '/profile',
      pageBuilder: (context, state) => ProfileScreen(),
```

```
        keepers: [Keeper(onEnter: () => ...)],
      ), ],)
```

Порядок исполнения - от первого до последнего: [keeper1, keeper2, keeper3, keeper4, keeper5]

```
GoRouter(
  keepers: [ keeper1,  keeper2,  keeper3],
 routes: [
   GoRoute(
     …
      keepers: [ keeper4,  keeper5], ), ],)
```

2. Определение типов навигации

Используя класс NavigationTypeObserver, наша обертка отслеживает навигационные ивенты, определяет тип навигации (push, pop, replace, remove) с помощью determineNavigationType().

3. Сохранение состояния между переходами.

Класс RouteAwareStateHandler расширяет RouteObserver<PageRoute<dynamic>>, позволяет централизованно управлять ивентами маршрутов (push, pop, replace) и автоматически сохранять текущее состояние роута(через NavigationStateHandler). Кроме того, метод popWithValue() предусматривает ситуацию, когда, например, в ShellRoute при pop() во внешний роут значение не возвращается, на что иногда жалуются разработчики.

Пример использования - в файле examples/main.dart и go_router_wrapper.dart.

4. Добавление RouteMatch в GoRouterWrapper

Плюсы использования RouteMatch:

- **Ясность структуры маршрутов** - упрощает понимание текущей структуры маршрутов.

- **Точность** - сокращает вероятность появления ошибок, связанных с нечетким сопоставлением параметров.
- **Удобство использования** в разработке и дебаге.