

Flutter routing packages usability research report

Authors: Tao Dong (inmatrix@), John Ryan (johnpryan@), Jack Kim (jackkim9@), Mariam Hasnany (mariamhas@), Chun-Heng Tai (chunhtai@)

First published: Aug 23, 2021

Last updated: Aug 19, 2021

Table of contents

[1. Introduction](#)

[2. Developing navigation scenarios](#)

[2.1. Six common navigation scenarios](#)

[2.2. Nested routing](#)

[2.3. Writing snippets for scenarios](#)

[3. Addressing the complexity of the Router API](#)

[3.1. Path-driven routing APIs](#)

[3.2. Route guard APIs](#)

[3.3. Nested routing APIs](#)

[4. Evaluating the usability of high-level APIs](#)

[4.1. Selecting packages](#)

[4.2. Evaluation methods](#)

[4.3. Results](#)

[4.4. Summary of results](#)

[5. Discussions](#)

[5.1. Implications for designing high-level routing APIs](#)

[5.2. Guidance about choosing a routing API](#)

[5.3. Limitations of the study](#)

[6. Conclusions](#)

[7. Acknowledgments](#)

If you have any questions or comments about this research report, please post them to [this discussion thread](#) in the flutter/uxr repo on Github. Thank you.

1. Introduction

Flutter's new navigation system, exposed as the Router API (previously known as Navigator 2.0), provides many desirable enhancements and a great deal of flexibility and power, but it's also considered to be complex and hard to use by Flutter users. To take advantage of this new navigation system (for example, deep linking on the web or [deep linking](#) on mobile devices) without the API's complexity, the Flutter community proposed [many high-level routing APIs](#) available on [pub.dev](#). With the growing number of routing API choices available, Flutter users could soon have difficulty choosing a package, similar to the challenge of selecting a state management solution.

We formed a small research team to investigate the proposed community solutions and evaluate whether to recommend one of them to our users or, at least, provide guidance about how to choose a routing API. In addition to addressing this imminent problem blocking Flutter users from implementing the ideal navigation behavior for their apps, we also wanted to take the opportunity to reflect on how to improve our API design process, which contributed to the lack of balance between power and usability in the Router API.

In this investigation, we wanted to answer the following research questions (RQ):

- RQ 1. What are the core set of navigation scenarios that a high-level routing API should make easy to implement?
- RQ 2. What design choices do community routing packages make to address the complexity of the Router API?
- RQ 3. How usable are the community routing packages? Specifically, we focused on the experience of reading code written with the APIs provided by those packages.

To answer those questions, we started with developing a set of navigation scenarios that most Flutter users care about. This scenario-driven approach is important. It helps us ensure that any proposed API is evaluated based on the concrete experience of using it. After developing scenarios based on user and expert interviews, we implemented those scenarios using the Router API to establish a baseline to compare against. Then, we invited routing package authors to submit snippets implementing the same scenarios but using their respective packages. We had many API [design discussions](#) along the way to gain a better understanding about RQ2. Last, we conducted usability studies on three routing packages that were gaining traction in the community to answer RQ3.

By sharing what we found from this investigation with the Flutter team and community, we hope to achieve the following three objectives:

- Support *Flutter users* to make faster and better decisions when choosing routing APIs
- Help *Flutter package authors* improve their existing routing packages
- Inform the *Flutter team and contributors* of important usability considerations to be considered in future improvements to Flutter's routing system

The rest of the doc is largely organized in accordance with the three research questions mentioned earlier. We first describe the navigation scenarios we developed. Next, we explain how the high-level routing APIs from community routing packages go about simplifying the implementation of those scenarios. After that, we report the results of evaluating the usability of three routing packages.

Finally, we discuss takeaways for routing API designers as well as our suggestions for Flutter users who are looking to choose a routing API for their apps.

2. Developing navigation scenarios

“To optimize the overall productivity of the developers using a framework, it is crucial to invest heavily in the design of APIs that are used in the most common scenarios.”

– Cwalina & Abrams in [Framework Design Guidelines](#)

2.1. Six common navigation scenarios

Defining the intended usage scenarios is often the first step toward building a usable system.

Because members of the Flutter community were designing many high-level routing APIs in parallel, we would like to ensure that the evolution and evaluation of those proposed APIs are based on a shared understanding of what is important to the majority of Flutter users. The shared understanding and user requirements are expressed in six navigation scenarios:

- **Deep linking - path parameters:** When the user can navigate to a specific part of the app by opening a deep link with a path parameter such as `“/books/1”`.
- **Deep linking - query parameters:** When the user can navigate to a specific part of the app by opening a deep link with a query parameter such as `“/books/q?=fantasy&sort=?newest”`.
- **Sign-in routing:** When the user goes to certain routes of the app, they are redirected to a sign-in page if they haven’t logged in yet.
- **Dynamic linking:** When the user creates an item (for example, a list or a post) in the app, the app generates a unique link for the user.
- **Nested routing:** When the user navigates between routes using a bottom navigation bar, where each tab bar index corresponds to a path component (`‘/books’` or `‘/settings’`). The user can also navigate using a tab bar displayed when the first item in the bottom navigation bar is selected (`‘/books’`). The tab bar index corresponds to the next path component (`‘/all’` or `‘/new’`).
- **Skipping stacks:** When the user navigates to a section of the app and the underlying stack of screens changes. When the user presses the in-app back button, they’re shown a new screen in the page hierarchy of the new stack, rather than the previous screen. For more details about the distinction between reverse-chronological navigation and upward navigation, see the [Understanding Navigation](#) page in the Material Design documentation.

To define those scenarios, we synthesized interviews with six Flutter users who had trouble implementing navigation patterns on the web with an analysis of features provided by popular routing libraries both in Flutter’s ecosystem and in the web ecosystem. Then, we created [storyboards](#) to illustrate those scenarios and solicited feedback from the Flutter community using this [GitHub issue](#).

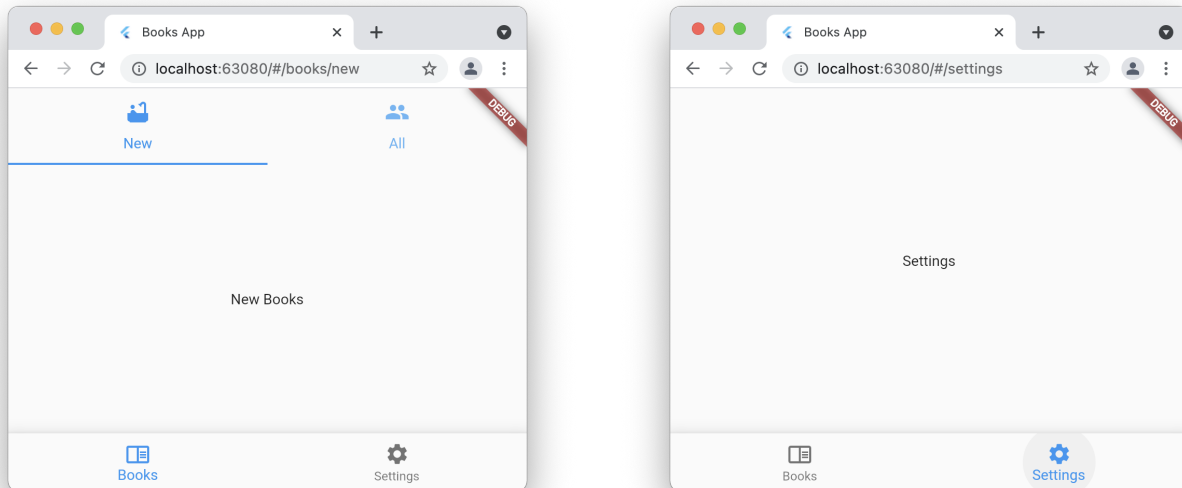
2.2. Nested routing

Though we were able to arrive at clear definitions for the first four scenarios, [our discussion](#) with community members shows that *nested routing* is an ambiguous term.

The first definition of nested routing is the ability to organize routes in a hierarchy of paths. For example, the app might organize all user-related routes under `“/home/users,”` and everything related to

articles under `"/home/articles."` In this case, all of the routes might still take up the full screen, which are referred to as "stacked routes" in some packages.

The second definition refers to using a navigation system for a section of the UI that doesn't take up the full screen. An example of this usage of the term is navigating to different routes within each tab of the app. Routing inside of each tab is independent from routing between those tabs. The storyboard we created for the nested routing scenario is mainly catered toward demonstrating this capability, as shown in the following screenshots.



Finally, the third definition allows the user to define a hierarchy of routes with relative paths making the routes in the navigation system less coupled to each other. We didn't examine this aspect of nested routing in this study.

2.3. Writing snippets for scenarios

To establish a baseline for understanding how a high-level API can make those scenarios easier to implement, the research team created snippets using the Router API for all these scenarios and then invited package authors to create equivalent implementations using their respective high-level routing APIs. The results of this exercise can be found in the [scenario_code](#) folder in the [flutter/uxr](#) repo. In the next section, we will refer to those scenarios and the code implementing them in our explanation of how high-level routing APIs mitigate the complexity of the Router API.

3. Addressing the complexity of the Router API

To understand how a high-level API might simplify the code for implementing routing and navigation, we need to first talk about where the complexity comes from in the Router API. Without using a package, the Router API requires the user to perform the following tasks:

1. Use a `ChangeNotifier` on any application state that manages your app's navigation.
2. (Optional) Make application state accessible through the `BuildContext` by using `package:provider` or `InheritedWidget`.

3. Implement a class that extends `RouteInformationParser`. This class translates between the route path in the URL bar and the application state that manages your app's navigation.
 - a. Implement `parseRouteInformation`, which translates from the URL path to the application state
 - b. Implement `restoreRouteInformation`, which translates the application state to a URL path
4. Implement a class that extends `RouterDelegate`:
 - a. Define a `GlobalKey<NavigatorState>`.
 - b. Implement a constructor that listens to the `ChangeNotifier` containing the application state.
 - c. Define a `build()` method that returns a `Navigator`. Pass the `GlobalKey` in the constructor.
 - d. In the `Navigator` constructor, implement `onPopPage`.
 - e. In the `Navigator` constructor, return a list of `Page` objects. These represent the "stack" of pages that are being displayed. Typically, the pages being displayed are determined by the current application state.
 - f. Implement `currentConfiguration`, which returns the current application state.
 - g. Implement `dispose()`. This is where you should call `dispose()` on the application state if it's a `ChangeNotifier`, and any other classes being used.

There are several notable attributes of the Router API. First, it optimizes for flexibility and power, because almost all the building blocks need to be defined and tied together by the user (the app developer). Second, route changes are driven by app state instead of the path (or URL in a web context). This is a departure from [Flutter's original navigation system](#).

The three packages that were studied (`VRouter`, `AutoRoute`, and `Beamer`) were all built on top of the Router API, but they mitigated its complexity in three ways:

1. Providing default, opinionated building blocks such as `RouterDelegate`, `RouteInformationParser`
2. Taking a path-driven routing approach and exposing a configuration API for defining path-to-route mappings
3. Providing high-level APIs for use cases such as conditional redirection (for example, sign-in routing) and nested routing

The following is an example configuration API provided by the `Beamer` package:

```
SimpleLocationBuilder(
  routes: {
    '/': (context) => BooksListScreen(
      books: books,
      onTap: (index) => context.beamToNamed('/books/$index'),
    ),
    '/books/:bookId': (context) {
      final bookId = int.parse(
        context.currentBeamLocation.state.pathParameters['bookId']!);
      return BookDetailsScreen(
        book: books[bookId],
      );
    },
  },
);
```

3.1. Path-driven routing APIs

Each package that was studied uses a path-driven approach, instead of the state-driven approach afforded by the Router API. (AutoRoute also offers a state-driven API demonstrated in the snippet for sign-in scenario.) The current pages on screen aren't based on app state, as defined and tracked in an object, but instead the state is the current route path, and is managed by the package. The current route path (for example `"/books/1"`) is the application state used to determine the pages on screen.

All three packages allow their users to configure a mapping from a path template to a WidgetBuilder (or type literal when using AutoRoute). The following snippet shows how this is done using the Beamer package:

```
final _routerDelegate = BeamerRouterDelegate(
  locationBuilder: SimpleLocationBuilder(
    routes: {
      '/': (context) => BooksListScreen(/* ... */),
      '/books/:bookId': (context) => BookDetailsScreen(/* ... */),
    },
  ),
);
```

Each package implements RouterDelegate and RouteInformationParser for the user, and automatically configures a Navigator (or multiple Navigators for nested routing) based on the path templates that were configured by the user. For example, `"/books/1"` matches the `"/books/:bookId"` path template. In Beamer, the path parameters can be accessed through the BuildContext (`context.currentBeamLocation.state.pathParameters`).

3.2. Route guard APIs

The sign-in scenario is solved by implementing a guard (also sometimes called validation or redirection). Without such an API, the user must add this logic to their implementation of the RouteInformationParser class:

```

@override
Future<AppRoutePath> parseRouteInformation(
  RouteInformation routeInformation) async {
  final uri = Uri.parse(routeInformation.location!);

  // Check if the user is signed in.
  if (!await _appState.auth.isSignedIn()) {
    return SignInRoutePath();
  }

  // Handle '/'
  if (uri.pathSegments.isEmpty) {
    return HomeRoutePath();
  }

  if (uri.pathSegments.length == 1 && uri.pathSegments[0] == 'books') {
    return BooksRoutePath();
  }

  // Handle unknown routes
  return HomeRoutePath();
}

```

As the preceding code snippet shows, the user needs to implement the `parseRouteInformation` method to validate the sign-in state (for example, sending an HTTP request to the authentication server), and then returns different `RoutePath[s]` based on the authentication result.

This parsing isn't very scalable or easy to maintain. A guard provides an abstraction over this process by giving the user a callback that can be used to redirect to a new route if a certain set of conditions aren't met. For example, `VRouter` uses a callback called `beforeEnter` to provide a way to customize this behavior:

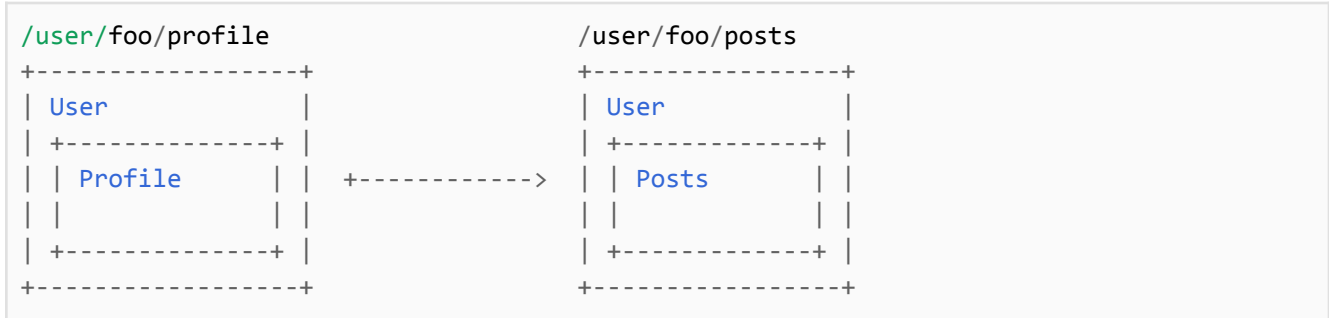
```

VGuard(
  beforeEnter: (vRedirector) async {
    if (await _appState.auth.isSignedIn()) {
      vRedirector.push('/');
    }
  },
  stackedRoutes: [
    /* Any routes defined here will not be shown if a redirection occurs in beforeEnter
    */
  ],
),

```

3.3. Nested routing APIs

Web frameworks like Vue or React support nested routes ([1](#), [2](#)), which allow their users to associate segments of a URL to a certain structure of nested components/widgets. The following image shows how a path segment (“profile” or “posts”) is related to the component being displayed (nested within the User component):



Source: [Nested routing](#) from the Vue documentation

In Vue, the behavior shown in the preceding image is implemented using the following configuration:

```
// JavaScript
const router = new VueRouter({
  routes: [
    {
      path: '/user/:id',
      component: User,
      children: [
        {
          // UserProfile will be rendered inside User's <router-view>
          // when /user/:id/profile is matched
          path: 'profile',
          component: UserProfile
        },
        {
          // UserPosts will be rendered inside User's <router-view>
          // when /user/:id/posts is matched
          path: 'posts',
          component: UserPosts
        }
      ]
    }
  ]
})
```

In the component template, the user can specify where to place the child component using a component sometimes called the Router Outlet or Router View:


```
// JavaScript
const User = {
  template: `
    <div class="user">
      <h2>User {{ $route.params.id }}</h2>
      <router-view></router-view>
    </div>
  `
}
```

A similar approach can be created in Flutter. For example, VRouter provides a `VNester` class. In the following snippet, the `VNester` object is configured to build an `AppScreen` page. The widget associated with a subroute is built by VRouter and provided to the `widgetBuilder` callback as the child parameter:

```
class BooksApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return VRouter(
      initialUrl: '/books/new',
      routes: [
        VNester(
          path: null,
          widgetBuilder: (child) => AppScreen(child: child),
          nestedRoutes: [
            VWidget(
              path: '/books/all',
              aliases: ['/books/new'],
              key: ValueKey('books'),
              widget: Builder(
                builder: (context) => BooksScreen(
                  initialSelectedTab:
                    context.vRouter.url!.contains('/new') ? 0 : 1,
                ),
              ),
              buildTransition: (animation, _, child) =>
                FadeTransition(opacity: animation, child: child),
            ),
            VWidget(
              path: '/settings',
              widget: SettingsScreen(),
              buildTransition: (animation, _, child) =>
                FadeTransition(opacity: animation, child: child),
            ),
          ],
        ),
      ],
    ),
  ],
}
```

```
    );  
  }  
}
```

4. Evaluating the usability of high-level APIs

4.1. Selecting packages

To gain a good understanding of the high-level APIs the Flutter community developed for routing, we conducted an extensive search and stopped after finding 12 of them. Then, we compared their capabilities by checking which scenarios they support. The results of this exercise can be viewed in this [feature comparison table](#).

In addition to supported scenarios, those packages also vary in other aspects, including maturity, usage, and, of course, API design choices. To narrow down the number of package we can manage in the detailed usability evaluation, we adopted the following selection criteria:

- The package supports (or has a concrete plan for supporting) all the navigation scenarios identified from our research.
- The package author is engaged in this research and is willing to contribute code snippets for the comparative analysis.
- The package represents a distinct approach toward simplifying common usage of the Router API. In other words, we might only examine one of the several packages that are similar to maximize what we learn from the research.
- The package is relatively popular based on signals such as the popularity score on pub.dev and the number of stars/contributors on Github.

After applying these criteria and considering the resource and time constraints, selected just three packages for detailed evaluation: [AutoRoute](#), [Beamer](#), and [VRouter](#). We reported the detailed selection rationale in this [GitHub post](#).

4.2. Evaluation methods

To evaluate the usability of high-level APIs implemented in the three routing packages, we conducted two studies. Each study looked at three scenarios: *deep linking by path parameters*, *sign-in routing*, and *nested routing*. The specific version of the snippets that we evaluated can be accessed in [this GitHub folder](#). The versions of VRouter, Beamer, and AutoRoute were 1.1.0, 0.12.4, and 2.1.0, respectively.

The first study was a [heuristic evaluation](#) where we developed a set of API usability rubrics and applied them to a systematic examination of routing APIs. The second study was an API walkthrough study for which we recruited 15 Flutter users and observed how they made sense of the code snippets if they found them in an online search. We describe the designs of the two studies in more detail in the following sections.

Heuristic evaluation

We developed a set of API usability heuristics based on the [Cognitive Dimensions framework](#), first introduced by the Visual Studio user experience group at Microsoft in 2005. We adapted this

framework for API usability evaluation based on our needs in this project and earlier usage of the framework within Google, arriving in seven dimensions as the core heuristics of our evaluation:

- *Role expressiveness*: Do the API names clearly communicate what the APIs do?
- *Domain correspondence*: Does the API directly map the concepts the programmer thinks in the application domain?
- *Consistency*: Is the API consistent with its own surface and across the API surface of the framework?
- *Premature commitment*: Does the API require the programmer to make some upfront decisions that are hard to reverse?
- *Abstraction level*: Does the API allow the programmer to achieve a set of common goals with just a few components or many building blocks?
- *Work-step unit*: How concise is the code for implementing common API usage scenarios?
- *Viscosity*: Does the API allow the programmer to make changes to their code easily? Is there a domino effect when refactoring code?

The [full evaluation guide](#) is published on the project's Github repository. Each package was independently evaluated by two authors of this report, and we debriefed together to consolidate our results.

API walkthrough

The heuristic evaluation generated useful hypotheses and preliminary observations we further validated in an API walkthrough study. In the walkthrough study, we recruited 15 Flutter users who previously signed up for Google's usability studies. Those 15 participants were evenly divided into three groups to read (walk through) code snippets written with the three packages, respectively.

To ensure that the feedback from this study speaks to the developer experience of both Flutter on mobile and Flutter on the web, we specifically recruited experienced Flutter users who started building web apps using Flutter. Five of the 15 participants had experience using the Router API. In the rest of this report, we use IDs, such as V-P01 (that is, the first participant in the VRouter walkthrough), when referring to individual participants.

Each study session lasted for about 90 minutes and was conducted by a moderator over Google Meet. One of the authors of this report moderated all the sessions. The moderator and the participant were the only two people in the live video call, while a few research team members observed the sessions through a live stream. A study session typically consisted of the following parts:

- Background interview – 5 minutes
- Code sample walkthrough – 75 minutes
 - Read documentation (the package's landing page on pub.dev) – 5 minutes
 - Walk through the code snippet for the first scenario – 15 minutes
 - Quiz questions – 5 minutes
 - Repeat the last two steps for the other two scenarios – 50 minutes
- Post-test interview – 5 minutes

In each walkthrough task, the participant was asked to do the following:

- Read the code for implementing a navigation scenario.
- Explain what the code does line by line.
- Describe the navigation scenario in their own words.

The moderator noted points of confusion and encouraged the participant to take a guess before consulting documentation. The quiz questions were customized to each package and scenario based on the findings of the heuristic results for the purpose of verifying known problems. For example, in the study sessions for VRouter, we had a quiz question, “What does ‘stackedRoutes’ do?”

The post-test interview gave the moderator an opportunity to discuss with the participant the overall experience of learning this new routing API through reading sample code. We asked questions derived from our heuristic evaluation framework, such as “How easy was it to map from these navigation scenarios to their code snippets?” and “How easy was it to reason its navigation behavior of the snippets?”, We also asked how likely the participant would adopt the package they examined in their own projects.

4.3. Results

In this section, we report the results from both our heuristic evaluation and the API walkthrough user study. The data source of each finding is identified by *evaluators* and *participants*, respectively.

Scenario: Deep linking by path parameters

Overall, all three packages made this scenario quite straightforward to implement. They all resulted in less than half of the code required to implement the same scenario with the Router API. In particular, AutoRoute’s implementation of this scenario is the shortest—only 92 lines of code in total, including just 17 lines specific to routing.

In our heuristic evaluation, we found all packages achieved a high-level of domain correspondence. It’s easy to map generic routing concepts in the scenario to relevant classes and methods in those packages. On the dimension of viscosity, it’s easy to expand and change code written with those packages for the purposes of adding and removing routes or changing path parameters. The rest of the results are focused on room for improvement.

VRouter

VRouter allows the user to configure the paths for different routes using a parameter in the VRouter constructor. The path parameters can be accessed from the VRouter state, which is provided through an extension method on BuildContext. See the usage of the API in the following snippet:

```
VRouter(  
  routes: [  
    VWidget(  
      path: '/',  
      widget: BooksListScreen(books: books),  
      stackedRoutes: [  

```

```

VWidget(
  path: r'book/:id(\d+)',
  widget: Builder(
    builder: (context) => BookDetailsScreen(
      book: books[int.parse(context.vRouter.pathParameters['id']!)],
    ),
  ),
),
],
),
VRouteRedirector(path: ':(.+)', redirectTo: '/'),
],
);

```

[Full snippet](#)

Overall, we found very few usability issues in the VRouter's implementation of the deep linking by path parameters scenario. In the walkthrough study, all five participants were able to form a correct mental model after reading VRouter's snippet for this scenario. They summarized the scenario accurately in their own words without seeing a demo. Participants also found it easy to read the snippet. Nonetheless, some enhancements can probably be made in the following areas to make the API more intuitive:

- **Using RegEx:** `VRouteRedirector` redirects a path pattern `:(.+)` to `/` in this scenario. Evaluators found the syntax difficult to understand. The API user might not recognize it as a regular expression (RegEx). In the walkthrough study, though most participants were able to guess the RegEx used in `VRouterDirector`, most were unsure about the path pattern defined in RegEx and expressed a general discomfort working with RegEx. Some participants consider it an overkill in this use case: *"I'm familiar with Regex, but I wouldn't necessarily want to work with it all the time"* (V-P05)¹
- **Unclear meaning of `stackedRoutes`:** Evaluators had difficulty developing a precise understanding of this API by its name. According to VRouter's [documentation](#) and the demo's behavior, `stackedRoutes` enables [Upward Navigation](#) described in Material Design. When the app user opens a deep link such as `"/book/0"` in the browser, the app pushes its parent page `"/"` first, enabling the user to use the app's back button to go upward in the app's navigation hierarchy. In hindsight, the API name makes sense, but it didn't help evaluators form a correct first impression.

In the walkthrough study, some participants conflated `stackedRoutes` with subroutes or nested routes. None of the participants seemed to get the idea about upward navigation the API enables. *"I'm assuming that this (stackedRoutes) is just a concatenation of parent with the child routers."* (V-P02)

- **Inconvenient access to path parameters:** Evaluators thought that the builder returning `BookDetailsScreen` could include the path parameters as an additional parameter, so the user wouldn't have to access it from the `BuildContext`.
- **Lacking a default for handling unknown paths:** Because handling unknown paths is a common task, a high-level API can be convenient instead of having to use RegEx.

¹ VRouter recently added [support for using wildcards](#) in path patterns to address this feedback.

Beamer

Like VRouter, Beamer allows the user to configure the paths for different routes in a single block of code. This configuration is located within the constructor of `BeamerDelegate`, an opinionated RouterDelegate the Beamer package provides. A `SimpleLocationBuilder` is used to specify the path-to-route mapping for the deep linking by path parameters scenario as the following snippet shows:

```
SimpleLocationBuilder(  
  routes: {  
    '/': (context) => BooksListScreen(  
      books: books,  
      onTap: (index) => context.beamToNamed('/books/$index'),  
    ),  
    '/books/:bookId': (context) {  
      final bookId = int.parse(  
        context.currentBeamLocation.state.pathParameters['bookId']!);  
      return BeamPage(  
        key: ValueKey('book-$bookId'),  
        popToNamed: '/',  
        child: BookDetailsScreen(  
          book: books[bookId],  
        ),  
      );  
    },  
  },  
)
```

[Full snippet](#)

Note that Beamer doesn't have an API equivalent to VRouter's `stackedRoute` nor AutoRoute's `includePrefixMatches`. Instead, parent pages were pushed into the navigation stack by default to enable upward navigation when using `SimpleLocationBuilder`. Beamer allows the user to assume finer control of this behavior by using the `BeamLocation` API, such as plugging in custom "navigation states" through `createState` and building the page stack manually through `buildPages` using the previously mentioned state.

There were just a couple of minor issues evaluators noticed in the heuristic evaluation:

- **SimpleLocationBuilder vs. BeamerLocationBuilder:** *Simple* doesn't explain the purpose of this class and how it's different from the `BeamerLocationBuilder` class. The ambiguity of what is considered simple creates a situation where the user of Beamer needs to decide which API to use up front and the cost of refactoring from one API to another could be non-trivial.
- **Inconvenient accessing path parameters:** Like VRouter, path parameters need to be accessed through `context.currentBeamLocation.state` in Beamer. Evaluators thought the builder could make path parameters readily available as an additional parameter.

- **Lacking a default for handling unknown paths:** Though Beamer provides a `notFoundRedirectNamed` parameter on `BeamerDelegate`, evaluators thought it could go further by making redirecting unknown paths to `"/` the default behavior.

In the walkthrough study, all five participants understood the snippet easily with only a few signs of uncertainty.

AutoRoute

In AutoRoute, a path parameter can be defined using a colon followed by the name of the parameter, as `"/book/:id"` shows in the following snippet:

```
@MaterialAutoRouter(
  replaceInRouteName: 'Screen,Route',
  routes: <AutoRoute>[
    AutoRoute(path: "/", page: BooksListScreen),
    AutoRoute(path: "/book/:id", page: BookDetailsScreen),
    RedirectRoute(path: "*", redirectTo: "/")
  ],
)
class $AppRouter {}
```

This parameter in the path is then matched to a widget constructor argument of the same name annotated by `pathParam@`:

```
class BookDetailsScreen extends StatelessWidget {
  final int id;
  BookDetailsScreen({@pathParam required this.id});

  @override
  Widget build(BuildContext context) { /***/}
}
```

[Full snippet](#)

AutoRoute's snippet for this scenario was concise and easy to follow in general. All five participants were able to form a correct mental model after reading the AutoRoute snippet for deep linking. They also reacted positively to the conciseness of the code.

As the only package that relies on code generation in this study, it's worth noting that most participants had a negative perception of code generation in general. The two reasons they cited were worries about lacking control of the codegen process and how difficult it would be to debug the generated code if it breaks. Two of the five participants said they personally wouldn't mind, but they said their team wouldn't like it.

In addition to this general hesitancy about using code generation, we identified a few specific usability issues:

- **Unfamiliar codegen input and output:** `$AppRouter` is an empty class with a potentially large annotation as the app grows [L37]. Evaluators were concerned that users might have trouble understanding that this is the input for the code generator and what code would be generated.



In the walkthrough study, some code generated parts made participants pause to figure out what the code did. For example, A-P05 paused while reading the `$AppRouter{}` declaration. Their initial reaction was: *"This is new to me. I don't know what it is."*


- **Unclear parameter names:** Two parameter names caused confusion. The first parameter was `replaceInRouteName` on the `@MaterialAutoRouter` class. Evaluators had to check documentation to understand what this parameter did [L30]. In the walkthrough study, most participants were somewhat confused by what `replaceInRouteName` meant, but they understood its meaning after consulting the documentation.

The second parameter with a confusing name was `includePrefixMatches` [L47]. The value of the parameter had a significant effect on the app's navigation behavior. When it's set to be `true`, the app would push all routes matching the prefix to the navigation stack. In the walkthrough, most participants were quite confused by how `includePrefixMatches` worked and what it meant. They also found the documentation vague.

- **Lacking a default for handling unknown paths.** Like the other two packages, handling unknown paths needs to be specified explicitly. Evaluators thought redirection to the root route could be the default behavior.

Summary

To compare the usability of these three packages for implementing deep linking, we broke down the scenario into a few key programming tasks and summarized the differences and similarities across those packages in the following table. An  exclamation point indicates a usability concern described earlier in this section, and a  checkmark suggests a usability win.

Tasks	VRouter	Beamer	AutoRoute
Configure "/" to display BooksListScreen	Add a VWidget object to the 'routes' list in VRouter and set the "path" parameter to "/" and the "widget" parameter to a BooksListScreen object.	Add a key/value entry to the 'routes' map in SimpleLocationBuilder. Set the key to "/" and the value to a WidgetBuilder that returns BooksListScreen.	Add an AutoRoute object to the @MaterialAutoRouter annotation and set the "path" parameter to "/" and "page" parameter to the BooksListScreen type literal.
Configure paths matching "/book/:id" to display the BookDetailsScreen	Add a VWidget and configure the path parameter to match "book/:id(\d+)" and the value to a Builder() widget that builds BookDetailsScreen. Uses RegEx 	Add a key/value entry to the 'routes' map and set the key to "/books/:bookId" and the value to a WidgetBuilder that returns the BookDetailsScreen.	Add an AutoRoute object and configure the "path" to "/book/:id" and the value to the BookDetailsScreen type literal.
Obtain the ":id"	Use	Use	Use the @pathParam

path parameter	context.vRouter.pathParameters['id'].	context.currentBeamLocation.state.pathParameters['bookId'].	annotation in the BookDetailsScreen constructor. This made the code more concise without affecting readability. ✓
Handle unknown paths	Add a VRouteRedirector object to the 'routes' list in VRouter constructor and configure the ":(.+)" pattern to redirect to "/".	Set the "notFoundRedirectNamed" parameter in BeamerRouterDelegate to "/" This provides convenience for a common operation. ✓	Add a RedirectRoute object to the @MaterialAutoRouter annotation and configure "*" to redirect to "/"
Display the BookDetailsScreen on top of the BooksListScreen in the navigation stack to enable upward navigation .	Place the "/book/:id" VWidget in the <code>stackedRoutes</code> list of the "/" VWidget. The meaning of <code>stackedRoutes</code> was found to be unclear. ⚠	Default behavior of SimpleLocationBuilder	Configure the RouteInformationParser by setting <code>includePrefixMatches</code> to true on the <code>_appRouter.defaultRouteParser()</code> constructor to make the app push all screens that whose routes are a matching prefix of the current route (for example "/" matches "/book/:id", so it is included in the Navigator's stack of pages) when opening a deep link. Participants had trouble making sense of the <code>includePrefixMatches</code> parameter name. ⚠

To sum up, all three packages make this deep linking scenario easy to understand and we only found a handful of usability issues. The evaluators and study participants ran into fewest usability issues with Beamer, while AutoRoute provided the most concise code for this scenario. The differences across the three packages are more noticeable in their respective implementations of the next two scenarios.

Scenario: Sign-in routing

VRouter

VRouter provides a `VGuard` class that takes actions when a new route is accessed, before it's displayed to the app user. In this scenario, VGuard is used to prevent a path from being displayed if the user isn't signed in, and to skip the "/signIn" screen if they are. See the usage of the API in the following snippet:

```

class _BooksAppState extends State<BooksApp> {
  final AppState _appState = AppState(MockAuthentication());

  @override
  Widget build(BuildContext context) {
    return VRouter(
      routes: [
        VGuard(
          beforeEnter: (vRedirector) async {
            if (await _appState.auth.isSignedIn()) {
              vRedirector.push('/');
            }
          },
        ),
        stackedRoutes: [
          VWidget(
            path: '/signIn',
            widget: Builder(
              builder: (context) => SignInScreen(
                onSignedIn: (Credentials credentials) async {
                  await _appState.signIn(credentials.username, credentials.password);
                  context.vRouter.push('/');
                },
              ),
            ),
          ),
        ],
      ),
    VGuard(
      beforeEnter: (vRedirector) async {
        if (!await _appState.auth.isSignedIn()) {
          vRedirector.push('/signIn');
        }
      },
      stackedRoutes: [
        VWidget(
          path: '/',
          widget: Builder(
            builder: (context) => HomeScreen(
              onSignOut: () async {
                await _appState.signOut();
                context.vRouter.push('/signIn');
              },
            ),
          ),
        ),
        stackedRoutes: [
          VWidget(path: 'books', widget: BooksListScreen()),
        ],
      ),
    ],
  ),

```

```

    ),
  ],
);
}
}

```

[Full snippet](#)

Overall, both participants and evaluators were able to understand the code and concepts with some help from the documentation. However, we observed the following usability issues:

- **Unintuitive double VGuard setup:** The main point of confusion for participants was the double VGuard setup in sign-in routing. Three of the five participants didn't understand what the VGuards were doing until the second VGuard and questioned the purpose of needing both. *One participant said, "Then we've also got [another] VGuard beforeEnter. What is the difference between these two?" (V-P01) Another participant said, "I do wonder why there are two vGuards" but later understood that: "I think I figured it out as I went. My initial thought was that you put a guard around more of it and would have more logic around that." (V-P03)*
- **Trouble identifying the route being guarded:** Due to the indentations and the distances between the line a VGuard was defined and the route it was applied to, some participants had trouble identifying the guarding relationships quickly. *"I can't tell what this is actually guarding" (V-P02)*
- **Unclear how to redirect routes within VGuard:** For a couple of participants, the confusion stemmed from having to use vRedirector in `VGuard.beforeEnter`: *"What is the purpose of vRedirector? Why can't I use VRouter.push() in beforeEnter? ... Apparently you need to use vRedirector to redirect otherwise it won't do it correctly. It's not exactly intuitive." (V-P05)* This issue was noticed by evaluators in the heuristic evaluation as well.

AutoRoute

AutoRoute offers two ways to redirect the app user based on the sign-in state of the app. The first method, demonstrated in the following snippet, is based on declarative routing, where routes are managed in a Dart list, and the last element on the list is the route currently visible. The app developer can use [collection if](#) to build and modify the list of routes based on the sign-in state of the app. This is achieved by using the `AutoRouterDelegate.declarative` method as the following snippet shows:

```

MaterialApp.router(
  routerDelegate: AutoRouterDelegate.declarative(
    _appRouter,
    routes: (_) => [
      if (appState.isSignedIn)
        AppStackRoute()
      else
        SignInRoute(
          onSignedIn: _handleSignedIn,
        ),
    ],
  ),
)

```

```

        routeInformationParser:
            _appRouter.defaultRouteParser(includePrefixMatches: true));
    }

    Future _handleSignedIn(Credentials credentials) async {
        await appState.signIn(credentials.username, credentials.password);
    }
}

// can be replaced with the shipped in widget
// EmptyRouterWidget
class AppStackScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return AutoRouter();
    }
}

```

[Full snippet](#)

AutoRoute also supports [Route Guards](#), but this wasn't studied. Its usage is demonstrated in [this post](#) on Github.

Overall, evaluators found that less code was required than expected to achieve this scenario, but observed a few issues with understanding this snippet even with documentation.

The first question we'd like to address is: how usable is the `AutoRouterDelegate.declarative` API? Our findings are mixed on that question. On the one hand, participants found **the redirection logic defined in `AutoRouterDelegate.declarative` easy to understand**. All five participants were able to articulate how the sign-in logic works in their own words. For example, A-P02 described it this way, "so if you're signed in, it would use the app stock route. And, if you're not, it would use the signing route." A-P01 thought "The logic here is very simple."

On the other hand, **the API name "declarative" didn't resonate** with both the evaluators and the study participants. In the heuristic evaluation, evaluators were concerned that the user might not be able to associate the API name with the navigation behavior they want to implement. Introducing the concept of declarative routing, which can be novel to many users, might be overkill for implementing this scenario. In the walkthrough study, what *declarative* meant and whether this API is declarative was a point of confusion for some participants. This was reflected in A-P01's less than coherent comment on the API's name:

*"Well, it's not really declarative. Because, usually in declarative ways, you would just declare the routes and everything would work out. This is not declarative here. This is imperative (pointing to the if...element). **This is a flow** (emphasis ours). But I guess this part, this part here is declarative."*

There are a few additional areas of improvement:

- **Unclear purpose of `AppStackScreen`:** In the heuristic evaluation, evaluators noted that the widget defined in the sign-in scenario immediately returns an `AutoRouter()` object without defining its content. This was confirmed as the main source of confusion for four of the five participants in the walkthrough study, and they didn't find the documentation helpful. For example, A-P05 said, *"I don't know why this [AutoRouter()] is empty."*
- **Unintuitive route model:** The routes are modeled as a hierarchy, which might not fit the API user's mental model of the sign-in scenario. Both A-P02 and A-P03 found the hierarchy of the routes confusing and thought the sign-in scenario was doing nested routing. *"[I'm] not sure why this app needs nested routing" (A-P02)*
- **Confusing code-generated classes (for example, `AppStackRoute`):** Two of the five participants didn't understand where `AppStackRoute` came from. *"AppStackRoute doesn't exist anywhere else." (A-P01)* *"I don't see this [AppStackRoute] defined anywhere, nor does the SignInRoute. Oh, OK, these are the auto-generated ones" (A-P02)*

Beamer

Similar to `VRouter`, `Beamer` also has the concept of `Guards` to use during authentication, which allow the user to change the route path if a condition isn't met. In this scenario, they are used to prevent a path from being displayed if the app user isn't signed in, and to skip the sign-in screen if they are. One notable difference between `BeamGuards` and `VGuards` is that `BeamGuards` are defined separately from the path template configuration (that is, the `guards` parameter and the `routes` parameter, respectively), while `VGuards` were embedded in the path template configuration.

```
class _BooksAppState extends State<BooksApp> {
  final Authentication _auth = MockAuthentication();
  late final List<BeamGuard> _guards;
  late final BeamerRouterDelegate _delegate;
  bool _isSignedIn = false;

  @override
  void initState() {
    super.initState();
    _guards = [
      BeamGuard(
        pathBlueprints: ['/signin'],
        guardNonMatching: true,
        check: (_, __) => _isSignedIn,
        beamToNamed: '/signin',
      ),
      BeamGuard(
        pathBlueprints: ['/signin'],
        check: (_, __) => !_isSignedIn,
        beamToNamed: '/',
      )
    ];
    _delegate = BeamerRouterDelegate(
      guards: _guards,
      locationBuilder: SimpleLocationBuilder(
```

```

routes: {
  '/': (context) => HomeScreen(
    onGoToBooks: () => Beamer.of(context).beamToNamed('/books'),
    onSignOut: () => _auth
      .signOut()
      .then((value) => setState(() => _isSignedIn = false)),
  ),
  '/signin': (context) => SignInScreen(
    onSignedIn: (credentials) => _auth
      .signIn(credentials.username, credentials.password)
      .then((value) => setState(() => _isSignedIn = value)),
  ),
  '/books': (context) => BooksListScreen(),
},
),
);
}
/* ... */
}

```

[Full Snippet](#)

In the walkthrough study, participants were excited about the prospects of BeamGuard. *“This looks really cool”*(B-P02) and *“That’s very nifty. It’s much more promising after seeing sign-in than seeing a deep link scenario.”*(B-P03).



Nonetheless, evaluators and study participants found issues when it came to the actual usage of these classes and the decisions that needed to be made. Most of these issues were related to the `guardNonMatching` parameter on `BeamGuard`.












- **Unclear meaning of `guardNonMatching`:** Evaluators found this parameter name confusing. It’s unclear if it reverses the behavior of the guard. It was also hard to map to a generic concept in routing, and the documentation was lacking. Similar to the evaluators, most participants had to consult the documentation several times to understand how it worked. *“I need to see the doc to understand what it means.”* (B-P05)
- **Complicated redirection logic resulted from using `guardNonMatching`:** The decision tree for setting up sign-in is complex, and would need major refactoring of the logic if set incorrectly or requirements change. This complexity mostly stems from the use of `guardNonMatching`, which implicitly changes the semantic of another parameter `pathBlueprints`. For example, when `guardNonMatching` is false (the default value), the `pathBlueprints` is a list of protected path patterns. However, when `guardNonMatching` is true, the route guard will protect any path *except* those specified in `pathBlueprints`. The interplay between these two parameters isn’t obvious and hard to keep track of, which contributed to the next two issues.
- **Confusion with `pathBlueprints`:** Four of the five participants were initially confused by the `pathBlueprints` API name. After reading the snippet and the documentation again, they understood that it was a way of defining the paths to guard. There was still confusion as to why in the snippet they were the same and if it was like RegExp in routing. *“Interesting they*

both have the same pathBlueprints”(B-P03) “The meaning is somewhat lost to me” (B-P04) “I’m not sure what that is.”(B-P05)

- **Double BeamGuards:** B-P01 was initially confused by having two BeamGuards for the “/signin” path: “...then you have two BeamGuard definitions. I don’t understand why. Basically they are both pointing to ‘/signin.’” While this is similar to the “double VGuards” issue we reported earlier for VRouter, the reason that caused the confusion here might be different. This issue is related to how `guardNonMatching` works. The first BeamGuard had `guardNonMatching` set to `true`, which effectively applied the guard to any routes except the ones specified by `pathBlueprints`.

Summary

To compare the usability of these three packages for implementing sign-in, we broke down the scenario into a few key programming tasks and summarized the differences and similarities across those packages in the following table. An  exclamation point indicates a usability concern described earlier in this section, and a  checkmark suggests a usability win.

Tasks	VRouter	Beamer	AutoRoute
Set up structure for routes	Place two VGuard objects in the VRouter <code>routes</code> list. Place a VWidget in each VGuard's <code>stackedRoutes</code> list, one for '/signin' and one for '/'. 	Use SimpleLocationBuilder with a flat list of routes to set up for redirection. Straightforward route structure. 	Use @MaterialAutoRouter with an AutoRoute object that has a page (AppStackScreen) with children paths and pages. Unclear purpose of <code>AppStackScreen</code> .  Unintuitive route hierarchy. 
Handle redirection logic to proper path/route based on auth status	Configure the VGuard's <code>beforeEnter</code> callback to call <code>vRedirector.push('/signin')</code> or to <code>vRedirector.push('/')</code> after the call to <code>isSignedIn()</code> completes. Double VGuards purpose was unclear.  Trouble identifying the route being guarded.  <code>vRedirector.push()</code> was confusing. 	Create a BeamGuard object configured to change the route to '/signin' if <code>_isSignedIn</code> is false or change the route to '/' if <code>_isSignedIn</code> is true. <code>guardNonMatching</code> was found to be confusing.  <code>pathBlueprints</code> was confusing.  Complex redirection logic. 	Use <code>AutoRouterDelegate.declarative</code> , setting the <code>routes</code> parameter to a list containing <code>SignInRoute</code> or setting the <code>routes</code> parameter to a list containing <code>AppStackRoute</code> . Redirection steps were straightforward.  <code>AutoRouterDelegate.declarative</code> naming was confusing. 
Redirect to proper route when app user logged out	Call <code>context.vrouter.push("/signOut")</code> after the call to <code>signOut()</code> completes.	set the BooksApp State's <code>_isSignedIn</code> flag after <code>SignOut()</code> completes	Rebuild the <code>AutoRouterDelegate.declarative()</code> when AppState changes by calling

	Inconsistency in using vrouter.push() vs vRedirector.push(). ⚠️		appState.addListener() => setState({})
--	---	--	---

To sum up, all three packages offer explicit APIs for guarding routes and most participants were able to guess, if not fully understand initially. However, participants ran into and noted different kinds of usability problems when trying to understand the snippets. Here, we share a few general observations.

First, participants appreciated a compact syntax for specifying redirection logic. In this regard, both VRouter and AutoRoute offer familiar syntaxes for specifying route guarding and redirection logics by leveraging Dart language features such as conditional expressions and [collection if](#), respectively.

Second, it seems to be easier to reason about the app's navigation flow when the guarding and redirection logic is centralized in one code block as in the AutoRoute snippet. In contrast, VRouter's approach—wrapping route configuration within a VGuard widget—leads to a relatively large vertical distance between a VGuard and the Route it applies to, making it harder to quickly understand the redirection logic.

Third, participants wanted API names to be straightforward and describe what they do instead of invoking unnecessary metaphors or concepts. Several API names in Beamer, such as `pathBlueprints` and `guardNonMatching`, as well as the AutoRoute's `declarative`, did not help convey their purposes and intended usage.

Finally, participants were confused when similarly named APIs were used in different contexts. This issue was mostly noted in VRouter's walkthrough and heuristic evaluation with regard to different ways of navigating to a route by name.

Scenario: Nested routing

In this section, we report findings on each package's core facilities for supporting nested routing under the heading *main results*, and other observations as *additional results*.

VRouter

VRouter provides a VNester object, which is used to configure nested routes:

```
class BooksApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return VRouter(
      initialUrl: '/books/new',
      routes: [
        VNester(
          path: null,
          widgetBuilder: (child) => AppScreen(child: child),
          nestedRoutes: [
```



```

VWidget(
  path: '/books/all',
  aliases: ['/books/new'],
  key: ValueKey('books'),
  widget: Builder(/* ... */),
  buildTransition: (animation, _, child) =>
    FadeTransition(opacity: animation, child: child),
),
VWidget(
  path: '/settings',
  widget: SettingsScreen(),
  buildTransition: /* ... */,
),
],
),
],
);
}
}

```

[Full snippet](#)

VNester wraps the widget for the current route in the widget returned by `widgetBuilder`. For example, when the app user visits `"/books/all"`, the widget for that route becomes the `widgetBuilder` callback's child parameter. Behind the scenes, it builds a `Navigator` with the pages corresponding to the widget for the current route. Note that the build transition can be customized.

Main results:

- **VNester:** The key question in `VRouter`'s support for the nested routing scenario is: can users understand how `VNester` works? Based on the data from the walkthrough study, the majority of participants (3 of 5) understood the mechanism of inserting a nested-route widget into a parent route through `VNester`'s `widgetBuilder` parameter. For example, here is how V-P05 described it in his own words:

"Okay, so this (widgetBuilder) is the thing I was talking about where I can wrap any of the children with a scaffold or some sort of container widget." (V-P05)

Nonetheless, one of the participants didn't clearly articulate how `VNester` worked, and another had trouble understanding it: *"I'm unclear how that navigation is nested within the top level of the app."* (V-P02) In addition, several participants paused at `path:null` on the `VNester` class and found it confusing. The differences between `stackedRoutes` and `nestedRoutes` were not clear to V-P05. He also wondered if `nestedRoutes` allows more than one child at a time.

Overall, the basic idea behind `VNester` seems to be reasonable to understand, but the API can use some polish to make the connection between the child parameter of `widgetBuilder` and `nestedRoutes` clearer and avoid the confusion of assigning null to path.

Additional results:

- **Unclear meaning of `beforeUpdate`:** In VRouter's snippet, a `VWidgetGuard.beforeUpdate` was used to animate the tabs on the Books screen when the path changes between "books/new" and "books/all" (L116). Evaluators initially had trouble understanding the behavior of `beforeUpdate`, even after reading the API docs. Multiple participants in the walkthrough study also had trouble intuiting the meaning of `beforeUpdate`. For example, V-P02 said, "I'm not sure without looking at the documentation." While V-P05 did figure out what the code did correctly, he found it "very not intuitive... It basically looks like it's a lot of work to hook up routing to things that aren't by default supported."
- **VWidgetGuard vs. VGuard:** Evaluators found it difficult to map this `VWidgetGuard` class to a generic concept and purpose in the context of the Nested Navigation scenario. Evaluators found potential confusion between `VWidgetGuard` and `VGuard`. In the walkthrough study, participants also noted that these classes do similar things but were named differently.
- **Using Builder in VWidget widget parameter:** Evaluators felt that the `widget` parameter should be changed to `builder`. This puts an extra burden on the user to decide when to use a `Builder` widget or a regular widget.

Beamer

To support nested routing, Beamer allows the user to use a `Beamer` widget as a descendant of the parent `Beamer` widget further down the widget tree. The `Beamer` widget dynamically builds its subtree based on the active route. In the following snippet, the `Beamer` widget uses a `SimpleLocationBuilder` to determine if the body should show `BooksScreen` or `SettingsScreen`.

```
class _HomeScreenState extends State<HomeScreen> {
  final _innerBeamer = GlobalKey<BeamerState>();

  @override
  Widget build(BuildContext context) {
    final beamerState = Beamer.of(context).state;
    return Scaffold(
      body: Beamer(
        key: _innerBeamer,
        routerDelegate: BeamerRouterDelegate(
          transitionDelegate: NoAnimationTransitionDelegate(),
          locationBuilder: SimpleLocationBuilder(
            routes: {
              '/books/*': (context) => BooksScreen(),
              '/settings': (context) => SettingsScreen(),
            },
          ),
        ),
      ),
      bottomNavigationBar: BottomNavigationBar(
        currentIndex: beamerState.uri.path == '/settings' ? 1 : 0,
        onTap: /* ... */
        items: /* ... */
      ),
    );
  }
}
```

```
    },  
  );  
}  
}
```

[Full snippet](#)

Main results:

- **Embedding Beamer directly in the widget tree:** How well did participants understand how the Beamer widget worked in the context of nested routing? First, all participants were able to articulate that Beamer would show either the Books Route or the Settings route based on which path was matched. In addition, a few participants demonstrated their understanding by stating that Beamer supplied the widget for the body of the HomeScreen's scaffold. For example, *"So Beamer is given to the body of the scaffold."* (B-P05) Finally, some participants also noted the scope of the two Beamer widgets in the snippet. *"The HomeScreen gets the Beamer created in MaterialApp router; and everything in BooksScreen and SettingsScreen gets the sub-Beamer that is in the Body"* (B-P02)

In general, participants had little trouble understanding how Beamer worked. Placing it directly in the scaffold, a choice different from VRouter's, seemed to help users understand how the BooksScreen and SettingsScreen routes fit into the HomeScreen's widget tree. *"It seems pretty easy. I can see [it's] a pretty linear approach to navigation."* (B-P01) Nonetheless, a side effect of passing Beamer directly to the AppScreen's body was that one of the participants wondered if Beamer would produce any visual effects or not: *"I cannot, you know, visualize how this looks... if it [Beamer] has a visual representation or if it's just the rules that handle the navigation."* (B-P04)

Additional results:

- **Path pattern specification:** Evaluators were unclear about why `"/*/"` was used as the pattern match for the top level of nesting. This issue was noted by multiple participants in the walkthrough as well. For example, B-P03 commented, *"I'd be like, 'well, that's like it's just always going to the home screen, but why not just have a star?'"*
- **updateRouteInformation:** Evaluators found it unclear why `updateRouteInformation` (now just `update` in Beamer 0.13) must be used for nested routes. It was unclear by reading the documentation what was meant by *"the route propagating to the root router delegate"*, and what was meant by *"this is solved via `notifyListeners` in non-nested navigation"* from the API docs. In the walkthrough study, B-P03 and B-P04 also found it hard to understand what `updateRouteInformation` did. The most common guess was that it would update the URL displayed by the browser's address bar.

AutoRoute

AutoRoute allows the user to define a tree of AutoRoute objects in the `@MaterialAutoRouter` annotation. Subroutes are specified as children of an AutoRoute object:

```
@MaterialAutoRouter(  
  routes: [
```

```

replaceInRouteName: 'Screen,Route',
routes: <AutoRoute>[
  AutoRoute(
    page: AppScreen,
    path: "/",
    children: [
      RedirectRoute(path: "", redirectTo: "books/new"),
      AutoRoute(path: 'books/:tab', page: BooksScreen),
      AutoRoute(path: 'settings', page: SettingsScreen),
    ],
  ),
  RedirectRoute(path: "*", redirectTo: "/")
],
)
class $AppRouter {}

```

Inside the `AppScreen` widget, an `AutoTabsRouter` widget is used to configure the widget each subroute will display. The widget corresponding to the subroute is passed into the builder as the child parameter.

```

class AppScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return AutoTabsRouter(
      routes: [BooksRoute(), SettingsRoute()],
      duration: Duration(milliseconds: 400),
      builder: (context, child, animation) {
        final tabsRouter = context.tabsRouter;
        return Scaffold(
          body: FadeTransition(child: child, opacity: animation),
          bottomNavigationBar: BottomNavigationBar(
            currentIndex: tabsRouter.activeIndex,
            onTap: tabsRouter.setActiveIndex,
            items: /* ... */
          ),
        );
      },
    );
  }
}

```

[Full snippet](#)

Main results:

- **AutoTabsRouter:** The key to `AutoRoute`'s implementation of the nested routing scenario is the `AutoTabsRouter` widget. How intuitive is this API? In the walkthrough study, all five participants described the behavior enabled by `AutoTabsRouter` accurately. In particular, A-P01 gave a specific explanation about how the builder worked: *"Basically this [child] is the route itself, which could be one of these: BooksRoute or SettingsRoute."* A-P03 reacted especially

enthusiastically to this widget, “You have a very very handy widget called *AutoTabsRouter* that handles the state of the bottom navigation bar for me, which is very, very helpful in my opinion.”

While the behavior of *AutoTabsRouter* was not hard to guess, A-P02 was disappointed by the lack of API documentation for the widget, and that made it difficult to confirm what he thought the widget does. In the heuristic evaluation, evaluators also had trouble confirming what the class does due to lack of documentation.

- **Redirection in nested routing setup:** Some participants were confused about how the redirection would work. For example A-P02 said, “This is a bit confusing now. I’m not sure why we have a *RedirectRoute* down here with an asterisk and this *RedirectRoute* here with nothing. So what’s the point of this redirect?”
- **Full route hierarchy required to navigate:** Evaluators found it verbose that `context.navigateTo` requires the full route hierarchy starting from “*AppRoute*”.

```
context.navigateTo(  
  AppRoute(  
    children: [  
      BooksRoute(tab: _tabs.elementAt(index)),  
    ],  
  ),  
);
```

- **Unclear when to use the children parameter of the *AutoRoute* class:** Evaluators found it unclear when to use *AutoRoute.children*, and whether a flat list of *AutoRoute* objects would work.

Additional results:

- **Route transitions:** Evaluators were looking for a way to provide a default route transition, for example, provide a global route animation to be applied everywhere.
- **Not defining inner routes explicitly:** A-P05 expected the two routes within the *BooksScreen* to be defined explicitly in the route configuration instead of using a path parameter (that is, ‘books/:tab’).
- **Confusing code-generated classes:** “It’s unclear how *BooksRoute* gets created. Where does it come from?” (A-P01 didn’t realize some classes were created by codegen.)

Summary

To compare the usability of these three packages for implementing nested routing, we broke down the scenario into a few key programming tasks and summarized the differences and similarities across those packages in the following table. An ⚠️ exclamation point indicates a usability concern described earlier in this section, and a ✅ checkmark suggests a usability win.

Tasks	VRouter	Beamer	AutoRoute
Show ‘/books/new’ as the initial route.	Set the <code>initialUrl</code> parameter VRouter to ‘/books/new’.	Set the <code>initialPath</code> of BeamerRouterDelegate to ‘/books/new’.	Configure a <i>RedirectRoute</i> to redirect from “” to “/books/new”.

Display a BottomNavigation Bar.	Create a VNester object, configured to build the AppScreen, which displays the bottom navigation bar.	Create a SimpleLocationBuilder to direct all routes matching "/*/*" to HomeScreen, which displays the bottom navigation bar. ⚠ SimpleLocationBuilder vs BeamerLocationBuilder ⚠ Route configuration ("/*/*")	Create an AutoRoute object, configured to direct all routes with the "/" prefix in their path to the BooksScreen.
Set the BottomNavigation Bar index.	Set the bottom navigation bar index based on whether or not the url contains '/books'.	Set the bottom navigation bar index is based on whether or not the url contains '/settings'.	Use an AutoTabsRouter widget configured with the generated BooksRoute and SettingsRoute(). ⚠ AutoTabsRouter's API doc was lacking. ✅ One participant liked that this API is decoupled from the BottomNavigationBar.
Change the route when the bottom navigation bar index changes.	Call vRouter.push() in the BottomNavigationBar's onTap callback.	Call .beamToNamed() in the BottomNavigationBar's onTap callback.	Call context.tabsRouter.setActiveIndex in the BottomNavigationBar's onTap callback.
Display either the BooksScreen or SettingsScreen in the body of the Scaffold, depending on the route.	Create a VNester object. Set the nestedRoutes parameter to a list of VWidget objects, one for '/books/all' and one for '/settings'. Set the widgetBuilder parameter to a callback that takes a child widget and returns an AppScreen widget. The child widget is the widget returned by the widget parameter of the active VWidget in the nestedRoutes list. ⚠ "path:null" in VNester	Pass an inner Beamer widget to the body of the home screen. ✅ Set the routerDelegate parameter to a BeamerRouterDelegate with a map of routes, one for '/books/*' and one for '/settings'. Display a BottomNavigationBar outside the inner Beamer widget.	Create an AutoRoute object in the @MaterialAutoRouter annotation. Set the children parameter to a list of AutoRouteObjects, one for 'books/:tab' and one for '/settings'. Set the page parameter to the AppScreen widget. The child widget is the widget associated with the active AutoRoute object in the children list. ⚠ Unclear when to use the children parameter of the AutoRoute class.

	<p>⚠ Differences between stackedRoutes and nestedRoutes</p> <p>⚠ Using Builder in VWidget widget parameter</p>		
Set the TabBar index when the route changes to '/books/new' or '/books/all'.	<p>Use a VWidget that handles '/books/all' and '/books/new' using the 'aliases' parameter.</p> <p>Configure the selected tab of the BooksScreen depending on if the url contains '/new' by using a VGuard.</p> <p>⚠ Unclear meaning of beforeUpdate</p> <p>⚠ VRedirector vs VRouteRedirector</p> <p>⚠ VWidgetGuard vs VGuard</p> <p>⚠ Using VRouter or vRedirector with VWidgetGuard</p>	Use the inner Beamer widget and BeamerRouterDelegate that handles '/books/*'. Configure the selected tab of BooksScreen depending on if the url contains 'all' .	<p>Use an AutoRoute object that handles 'books/:tab'. Use a @PathParam annotation in the BooksScreen to handle the inner tab bar index.</p> <p>⚠ Not defining inner routes explicitly</p>
Change the route to '/books/new' or '/book/all' when the TabBar index changes.	Call vRouter.push().	<p>Call Beamer.of(context).updateRouteInformation() (now update() in Beamer 0.13).</p> <p>⚠ updateRouteInformation documentation</p>	<p>Call context.navigateTo() with the AppRoute object that was generated, containing the BooksRoute configured with the selected tab.</p> <p>⚠ The full route hierarchy is required to navigate.</p>

On the positive side, all three packages provide nested routing APIs participants by and large figured out without much trouble. Beamer seemed to have the most straightforward way of inserting a subroute into its parent page's widget tree by avoiding the builder pattern. AutoRoute's high-level API for managing routes in a tabbed UI was also liked by some participants.

Though most issues were not showstoppers in our heuristic evaluation nor the walkthrough study for this scenario, we noticed some recurring sources of confusing: having different APIs for similar purposes (for example, VRedirector vs VRouteRedirector), requiring some seemingly low-level operations (Beamer's updateRouteInformation method), and lacking convenience (for example, full

route hierarchy required for navigating to a nested route in AutoRoute). Some of those issues were not unique to the nested routing scenario.

4.4. Summary of results

The following table summarizes all the usability issues we found in our research. We assigned a severity rating to each issue according to the following scale:

- **Cosmetic:** The problem is an inconvenience, but it doesn't lead to user confusion or errors.
- **Minor:** The problem might result in occasional confusion or errors in the initial use. The user can learn and remember the correct use of the API easily.
- **Major:** The problem could result in frequent user confusion or errors in sustained use, or the user might not be able to easily understand how the API works after consulting the documentation.
- **Catastrophe:** The problem could lead to critical errors that are hard to prevent or recover from in the program.

Issue	Package	Scenario	Severity
Using RegEx	VRouter	Deep linking	Minor
Unclear meaning of stackedRoutes	VRouter	Deep linking	Minor
Inconvenient access to path parameters	VRouter, Beamer	Deep linking	Cosmetic
Lacks a default for handling unknown paths	VRouter, Beamer, AutoRoute	Deep linking	Cosmetic
SimpleLocationBuilder vs. BeamerLocationBuilder	Beamer	Deep linking	Minor
Unfamiliar codegen input and output	AutoRoute	Deep linking	Minor
Unclear parameter names (for example, replaceInRouteName, includePrefixMatches)	AutoRoute	Deep linking	Major
Unintuitive double VGuard setup	VRouter	Sign in	Minor
Trouble identifying the route being guarded	VRouter	Sign in	Minor
Unclear how to redirect routes within VGuard	VRouter	Sign in	Minor
Unintuitive API name declarative	AutoRoute	Sign in	Minor
Unclear purpose of AppStackScreen	AutoRoute	Sign in	Major
Unintuitive route model	AutoRoute	Sign in	Minor
Confusing code-generated classes (for example, AppStackRoute)	AutoRoute	Sign in	Minor
Unclear purpose of guardNonMatching	Beamer	Sign in	Major
Complicated redirection logic resulted from using guardNonMatching	Beamer	Sign in	Major
Confusion with pathBlueprints	Beamer	Sign in	Minor
Double BeamGuards	Beamer	Sign in	Minor

path:null	VRouter	Nested routing	Minor
Unclear meaning of beforeUpdate	VRouter	Nested routing	Major
VWidgetGuard vs. VGuard	VRouter	Nested routing	Minor
Using Builder in VWidget widget parameter	VRouter	Nested routing	Minor
Path pattern specification (/*/*)	Beamer	Nested routing	Cosmetic
updateRouteInformation	Beamer	Nested routing	Major
Lacks documentation for AutoTabsRouter	AutoRoute	Nested routing	Minor
Redirection in nested routing setup	AutoRoute	Nested routing	Minor
Full route hierarchy required to navigate	AutoRoute	Nested routing	Cosmetic
Unclear when to use the children parameter of the AutoRoute class	AutoRoute	Nested routing	Major
Route transitions	AutoRoute	Nested routing	Cosmetic
Not defining inner routes explicitly	AutoRoute	Nested routing	Cosmetic
Confusing code-generated classes (BooksRoute)	AutoRoute	Nested routing	Minor

5. Discussions

In this section, we discuss our takeaways from this research, including implications for designing high-level routing APIs and suggestions for choosing a routing API. Additionally, we acknowledge the limitations of this research project, and outline potential future work by the Flutter contributor community including the Flutter team.

5.1. Implications for designing high-level routing APIs

For anyone who's looking to design or improve a high-level routing API for Flutter, we'd like to share a few lessons that we learned from evaluating the three packages and comparing them with Router in this research.

1. Path-driven routing is intuitive.

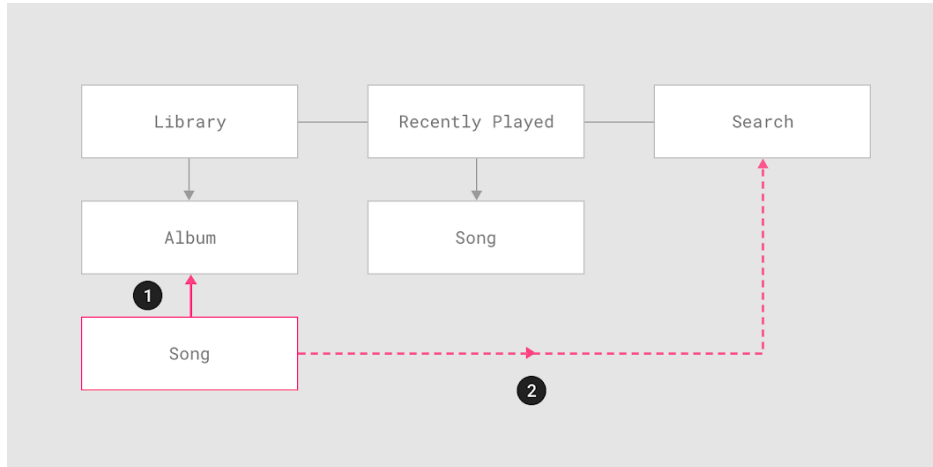
All three packages offer a mechanism to define path-to-route mappings as the source of truth for the app's routing behavior. Such mappings can often be expressed within a single code block, enabling users to reason about the app's routing behavior easily. This is similar to how Flutter's original navigation system works with named routes but significantly different from Router, which allows the app to determine which pages are visible based on arbitrary app state variables. Note that, both Beamer and AutoRoute also provide state-driven routing (also known as declarative routing) for specialized use cases (for example, sign-in and skipping stacks) or complex apps.

2. RegEx is too complex for expressing simple path patterns.

Participants universally disliked the fact that RegEx was used to define path matching patterns in VRouter's snippets. It introduced an element of uncertainty in code comprehension, because many participants weren't confident in their ability to correctly interpret RegEx.

3. Enabling upwards navigation can be hard to express through API names.

The Material Design [Understanding Navigation](#) page defines two distinct ways to navigate in reverse: *upward* and *reverse chronological*. In the following diagram, upward navigation is when the user navigates upward in the hierarchy (1), and reverse chronological is when the user navigates to the previously visited screen (2). In Flutter, the in-app back button is always upward, and the browser back button is always reverse chronological.



All these packages support upward navigation. Specifically in the deep linking scenario, when a deep link (for example, `"/book/0"`) takes the user to a specific route, the app can choose to automatically build and insert all the routes on the current route's hierarchical chain (in this case, the book list route for `"/"`) to enable upward navigation by the app's back button.

However, this behavior proves to be hard to express with an intuitive API. Both VRouter's `stackedRoutes` and AutoRoute's `includePrefixMatches` did not resonate with study participants. Beamer worked around this problem by making this behavior the default when using its `SimpleLocationBuilder`.

4. Similarly named APIs can be confusing.

When walking through the VRouter snippets, participants weren't sure about the differences between a few sets of similarly named APIs, including `VGuard` vs. `VWidgetGuard` and `vRedirector.push` vs. `vRouter.push`. It would be difficult for users to remember which API should be used in which situation. We recommend API designers either give them more specific names for the user to recognize different use cases, or combine them so that the user doesn't have to choose.

5. Plain and descriptive API naming is important.

Participants had trouble inferring what the API does when the name leans on an unfamiliar metaphor such as `pathBlueprints`, or the name introduces a more complex concept than the user case calls for, for example, `AutoRouterDelegate.declarative`.

6. Route Guard as a capability explicitly designed for is appreciated.

Participants in our study were excited to see route guards were integral parts of the high-level API in the routing packages. This concept seems to be well known to Flutter users and has become part of their expectation in choosing routing packages.

7. The organization and syntax for specifying redirection logic matters.

There are two important API design choices when it comes to the syntax for specifying redirection logic in routing. The first choice is about *where* the logic is specified. Is it centralized in a code block separate from route definitions, or is it scattered inline alongside route definitions? The sign-in routing snippets provided by Beamer and AutoRoute are examples of the former approach, while the snippet by VRouter demonstrates the latter (for example, wrapping protected routes in a VGuard). Using a centralized code block for expressing redirection logic seems to have readability advantages, because it's easier to see the routing flow at a glance.

The second design choice is about *how* the logic for redirection is specified. It's important to provide a syntax users are already familiar with. That often means leveraging the [conditional expressions](#) provided by the programming language. For example, AutoRoute allows the user to use [collection if](#) to specify the value of the `AutoRouteDelegate.declarative` parameter, and VRouter uses plain if statements in the `beforeEnter` callback function.

Our research also suggests a few factors that can undermine the readability of the guard configuration code:

- Implicitly inverting the redirection condition. For example, Beamer's `guardNonMatching` turns the BeamerGuard's `pathBlueprints` a block list into an allow list, when it's set to `true`, confusing most study participants and evaluators and resulting in two nearly identical BeamerGuards for the `signIn` path.
- Having a long vertical distance and multiple layers of indentations between the guard definition and the route it applies to (for example, VGuard).
- Having separate route guards specify redirection logic for the same state variable (for example, `isSignedIn`)

8. A clear way to see where route nesting happens improves readability.

All three packages provide a mechanism to embed a subroute in its parent page's scaffold. However, it seemed to be easiest for participants to make sense of that relationship when the subroute was placed directly into the parent page's widget tree using the `Beamer` widget instead of using a builder pattern in VRouter and AutoRoute. Figuring out what the `child` parameter was in the builder took some extra effort. Nonetheless, defining subroutes as part of the Widget tree can cause issues with Guards because, at the start of the build phase, it isn't known which routes are allowed and which aren't. For more discussion, see [slovnicki/beamer#372](#).

9. Codegen has usability tradeoffs.

While codegen helps AutoRoute achieve the most concise code among the three packages we evaluated, it comes with hard-to-ignore downsides. First, there was a general perception of anxiety about giving up control when using codegen among our study participants, even before they read the code snippets. Some believed that using codegen would make it difficult to convince their teammates even if they got comfortable using a codegen-based solution. Second, codegen leads to classes that aren't explicitly defined by the user (for example, `BooksRoute`, `AppStackRoute`), and our study shows that this can negatively affect code comprehension, especially for users new to the package. Therefore, API designers must consider the tradeoffs carefully when using codegen.

5.2. Guidance about choosing a routing API

Our research shows that the three packages we evaluated are all successful in providing useful abstractions and defaults to reduce the amount of code and the number of building blocks needed to implement common routing scenarios, compared with using Router directly. As much as we wanted to help Flutter users efficiently decide which routing API would fit their needs best, the race is simply too close to justify recommending a single package this time. It was especially difficult because these three packages showed strengths and areas of improvements in different ways. Moreover, due to the public and participatory nature of this research study, packages continued to evolve and sometimes converge on their designs as we provided feedback and engaged authors in discussions.

Though we aren't recommending a single package for all scenarios this time, we believe there is still value to elaborate on how you—Flutter users—might consider a package's features and usability for implementing different scenarios and suggest a package that you might want to check out first, whenever the signal is strong enough for us to say so. While package suggestions might be short lived, as this space continues to evolve, it's our sincere hope that you can use the key factors and the design choices that made an API more or less usable, as discovered in this research, to assess both updated and new packages for Flutter routing in the future.

Choosing a routing API depends on your app's requirements, so our suggestions are organized by scenarios. At the end of this section, we also offer our general guidance about selecting a routing API.

Scenario: Deep linking

Our recommendation: AutoRoute, Beamer, or VRouter

Deep linking is a common feature of modern apps. It's especially important when your Flutter app targets the web. So, we consider this to be a foundational requirement. All three packages we evaluated make deep linking easy to implement, and their APIs for supporting this scenario are similar.

AutoRoute achieved the most concise code for implementing this scenario (93 LoC) than Beamer (105 LoC) and VRouter (119 LoC) by leveraging annotations and codegen. It also made accessing path parameters very convenient through annotating widget constructor arguments. However, adopting a codegen-based package introduces additional build steps to a Flutter project, and it might raise concerns about debugging complexity, as we heard from our study participants. Unless your project is already using codegen for other purposes, this tradeoff needs to be carefully considered.

All things considered, we don't have a strong reason to recommend one package over others for this scenario. The bottom line is that your app will likely be well served by any of the three packages we evaluated if the main motivation is to enable deep linking.

Scenarios: Deep linking and sign-in

Our recommendation: VRouter

If sign-in is important to your app, in addition to deep linking, it's important to consider both the organization and the syntax of the code specifying route guarding and redirection logic as well as how

the hierarchy of routes is modeled. To this end, we recommend trying out VRouter first, due to its versatility and natural hierarchy of routes.

VRouter provides two ways of specifying route guarding and redirection logic. In the snippet we studied, a protected route can be nested in a `VGuard` class where redirection logic can be expressed. While this makes sense and follows the familiar widget composition pattern in the Flutter framework, we observed some initial readability frictions, potentially caused by the visual distance between `VGuard` and the route it was applied to. Later we learned that VRouter also provides an alternate way of writing the redirection logic all in one place instead of inline with the route definition, as demonstrated in [this gist](#).

AutoRoute's syntax for specifying redirection logic in its `AutoRouteDelegate.declarative` was easy to understand by study participants. However, the route hierarchy was unintuitive, and some participants initially questioned the need for nesting routes in a sign-in scenario. Beamer might require the most work to improve its route guard API. For one thing, it uses a set of parameters on `BeamGuard` to express the redirection logic instead of using familiar conditional expressions. For another, both the name and the behavior of `guardNonMatching` are quite vague and hard to make sense of.

Scenarios: Deep linking and nested routing

Our recommendation: Beamer or VRouter

If your app's design calls for nested routing, then the most important aspect of a routing API is the mechanism for configuring a route to be inserted into its parent widget tree based on the relative path associated with it. This logic should be easy to understand when reading the code. This is, again, a close race, and we end up having a tie between Beamer and VRouter.

In Beamer's snippet for nested routing, it's very easy to see which part of the widget tree is supplied by a subroute, because it's identified by a Beamer widget directly in the main route's build method. The code can be read linearly without making any effort to match a `child` parameter with a subtree when a builder pattern is used, as in both VRouter and AutoRoute. However, using a nested `Beamer` object in the widget tree invalid routes aren't known until that inner Beamer is built. Therefore, determining whether the route is valid requires a partial build of the widget tree. This causes widgets that were intended by the developer not to be built, to be built and quickly disposed of, because the route turned out to be invalid. For more discussion, see [slovnicki/beamer#372](#).

VRouter's support for nesting routes is enabled by a builder within `VNester`. While it took some time for a couple of participants to get used to it, it allows for expressing the nesting behavior logically and flexibly. In addition, it offers the ability to set a specific animation for each route.

AutoRoute's support for nested routing is not bad either. Some participants appreciated the high-level API `AutoTabRouter` it provided for implemented tabbed routing, though evaluators had reservations about the API name and its lack of documentation. There are two main concerns that made us hesitant to recommend it:

1. The nesting of routes, especially routes for the tabs in the books screen, was not obvious in its route configuration.
2. Users had more trouble understanding generated classes in this scenario as the complexity increased.

General considerations for choosing a routing API

Though the specific research results could become out of date as the packages evolve, the methodology and the high-level findings from this research might still be useful to guide your package evaluations for at least months, if not years, to come. Here are what we suggest you do:

- Define the navigation scenarios you care about. Try to implement those scenarios with packages under your consideration, so you can compare them directly.
- Focus on the key technical concepts for each scenario. For example, the syntax for specifying redirection is crucial to the sign-in route, and the readability of this syntax ought to have more weight than other aspects of the scenario.
- Apply heuristics to evaluate API usability. Employing the [heuristics](#) that we developed and tested in this research is a quick but effective way to systematically examine the usability of an API. We found a high degree of agreement between heuristic evaluation results and user testing (the walkthrough study).
- Consider the return on investment from learning a new package. Do you expect to use the package in many places in your app, or you just need it to make a few things work? Do you expect to use the package in many future projects? A package with a large API surface might be extra hard to master, if you don't need to use it frequently.

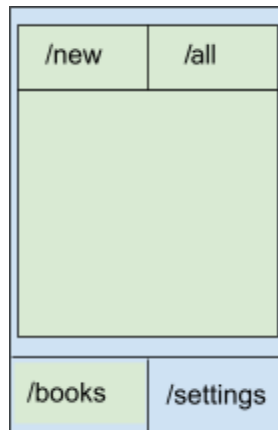
5.3. Limitations of the study

We made our best effort to understand the design choices for creating Flutter routing APIs and their effect on API usability, but our investigation isn't exhaustive nor are our conclusions definitive. There are a few important limitations of the study that you need to be aware of.

First, this study focused largely on code comprehension instead of code composition. Both are important aspects of API usability. Participants were asked to read through a snippet and provide their interpretation and comments, but they weren't asked to write or make changes to code, with the exception of the viscosity dimension in the heuristic evaluation. This means that there could be underlying usability issues that were not exposed by this study.

Second, the scalability of routing packages was not examined in this study. It was not studied whether or not these APIs can be composed from separate parts. For example, how two separate teams working on separate routes might combine their projects into a single app. We also observed that the advantage of using high-level routing APIs decreased in complex scenarios such as nested routing, but we didn't test those APIs with even more complex scenarios to find out when the user might be better off using Router directly.

Third, the navigation scenarios we defined and used to study routing APIs weren't exhaustive, and some might have important variants that we didn't consider. For example, the nested routing scenario was defined as an inner tab bar wrapped by a bottom navigation bar:



This does not specify certain behaviors that a Flutter app might require:

- Is there a transition animation when a new page is displayed, or is the widget immediately replaced? For example, navigating from `/books` to `/settings`.
- What is the initial route? `/books/new` or `/books`?
- Should the app use relative or absolute paths to navigate from `'/books/new'` to `/books/all` when the inner tab bar index changes? `VRouter` and `Beamer` use absolute paths, and `AutoRoute` uses a generated class and `AutoTabsRouter`.

Finally, this is a qualitative, small-sample study. While testing with five users is often sufficient to uncover the major usability issues², we couldn't quantitatively compare those APIs on usability metrics.

6. Conclusions

In this research study, we set out to answer three questions. Here, we provide summarized answers to them:

- RQ 1. *What are the core set of navigation scenarios that a high-level routing API should make easy to implement?* We defined six scenarios that are common and should be well supported by any high-level routing API.
- RQ 2. *What design choices do community routing packages make to address the complexity of the Router API?* Community routing packages provided high-level APIs to enable path-driven routing, route guards, and nested routing. In general, there are more similarities than differences in their design choices.
- RQ 3. *How usable are the community routing packages?* All three packages we evaluated are easier to use than Router. However, we identified dozens of usability issues of various degrees of severity that should be addressed as those packages continue to mature.

We also summarized the key lessons learned from examining routing packages available at this time. We hope those lessons can inform the design of high-level routing APIs in the future.

² [Why You Only Need to Test with 5 Users](#). Nielsen Norman Group.

7. Acknowledgments

This study wouldn't have been possible without the support from the Flutter developer community, especially the routing package authors who contributed example code, actively engaged in online discussions, and provided feedback on the research design and findings. We also thank all the Flutter users who participated in this research for their time.