ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

502071

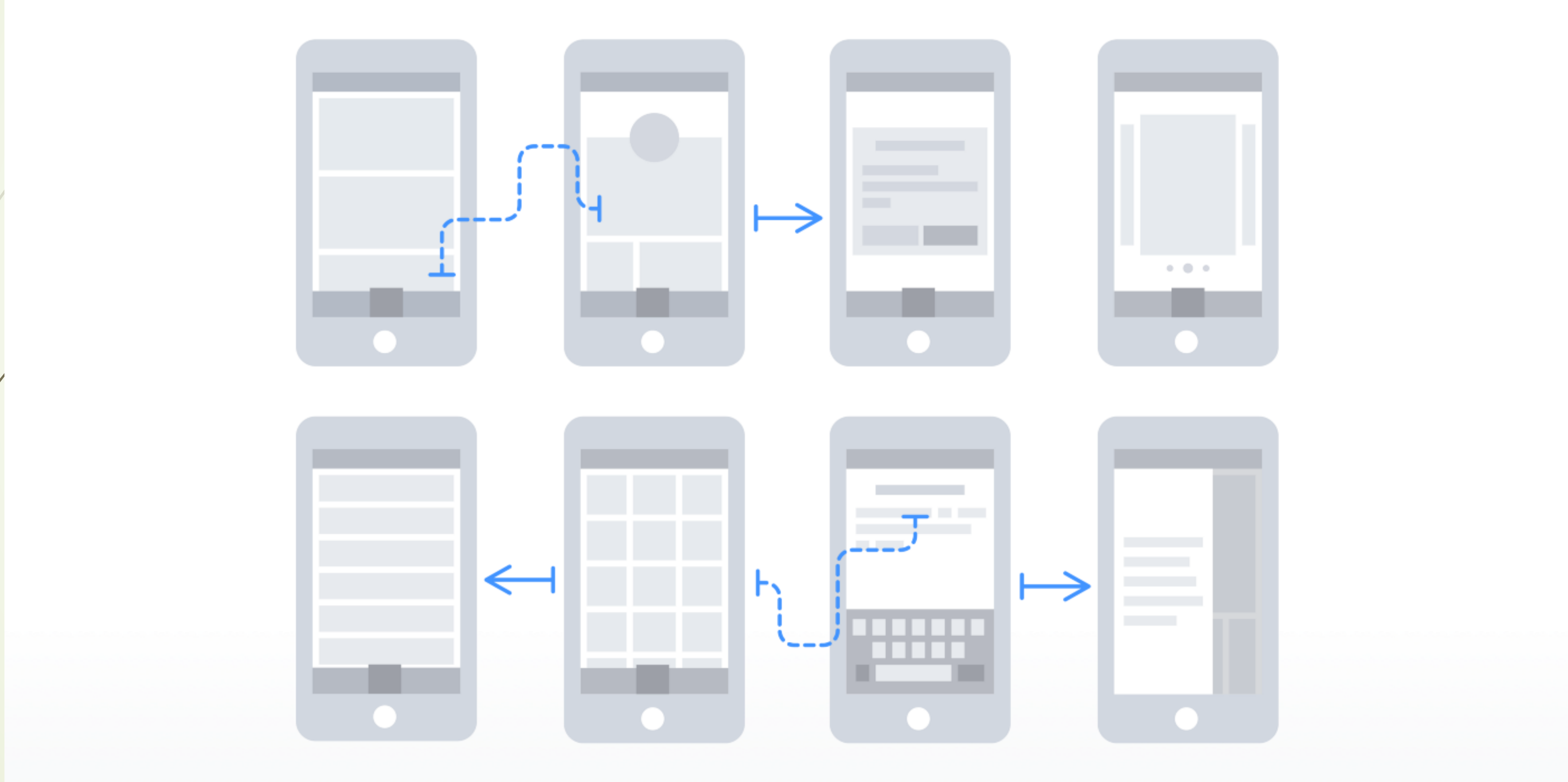
Cross-Platform Mobile Application Development

Navigation and Routing

1

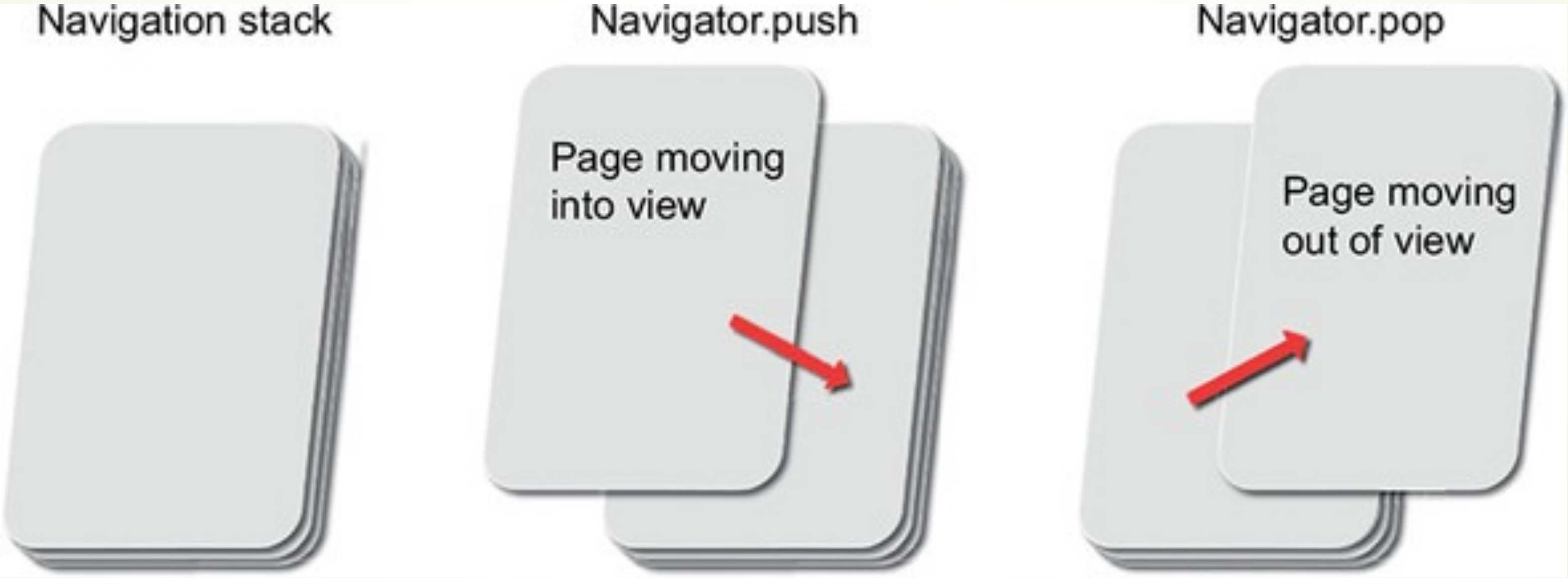
Navigation and Routing

Introduction



Navigator in Flutter

- Navigation in Flutter is achieved through a widget called Navigator. The Navigator widget manages a stack of screens or pages, which are referred to as routes.
- When a user navigates to a new screen, the new route is pushed onto the top of the stack, and when they go back, the current route is popped off the stack.



Navigator in Flutter

- **Navigating to a New Screen**
  - To navigate to a new screen in Flutter, you need to define a new route and push it onto the Navigator's stack. Here's an example:

```
Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => MyNewScreen()),
);
```

  - In this example, we use the push method of the Navigator widget to push a new route onto the stack. The first argument is the BuildContext, which is required to access the Navigator widget. The second argument is a MaterialPageRoute, which specifies the new route's builder method that creates and returns the new screen widget.

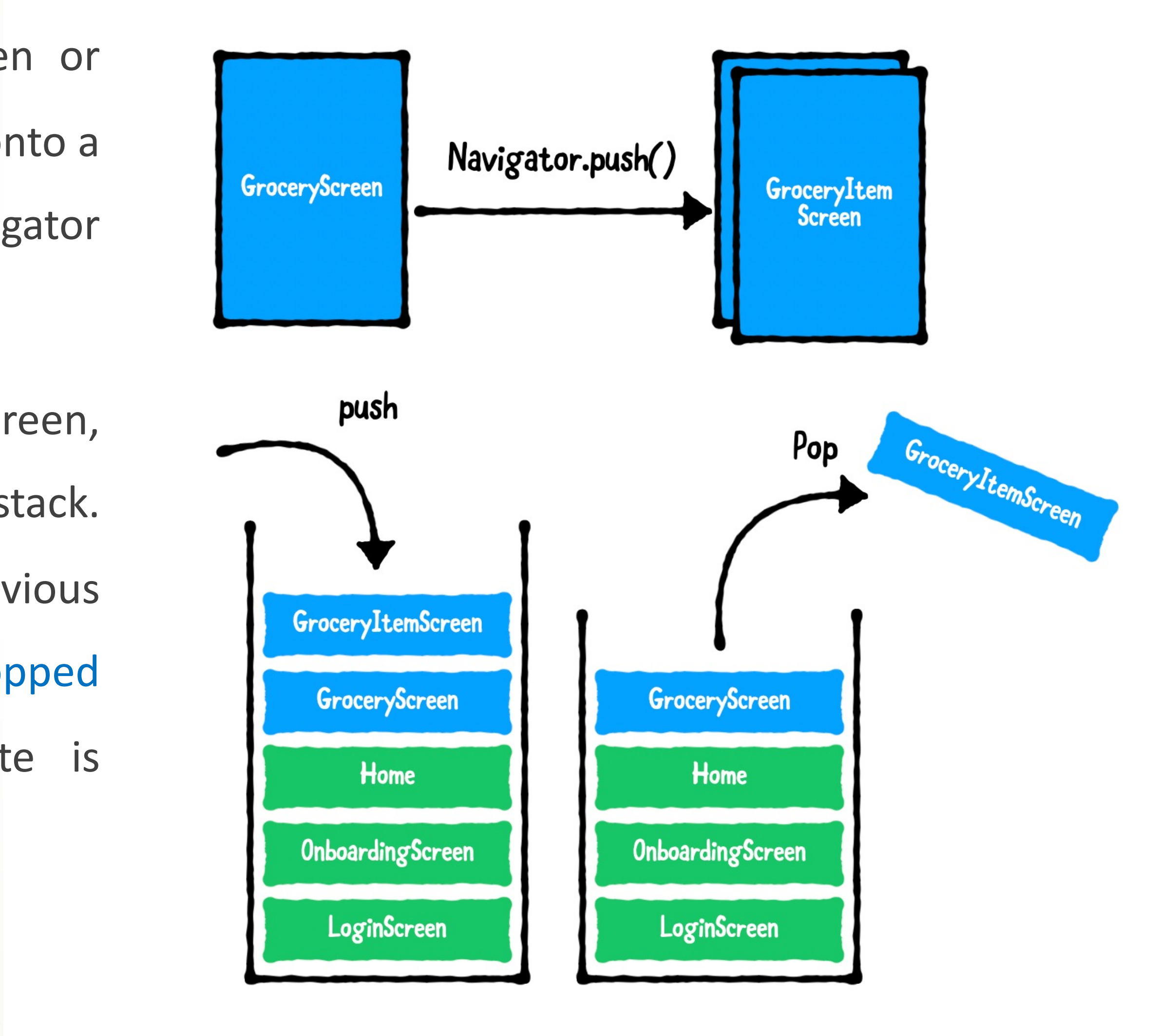
Navigator in Flutter

- In Flutter, a route is a way to represent a screen or page in your mobile application. Each route is associated with a unique name that can be used to navigate to it from other parts of the app.
- In Flutter, you can define a route as a widget that represents a screen or page. This widget can contain any other Flutter widgets to build the user interface for that screen. Once you've defined a route widget, you can register it with the application's routing table using a unique name.

```
class MyNewScreen extends StatelessWidget {
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(...),
            body: Container(...),
        );
    }
}
```

Navigator in Flutter

- When a user navigates to a new screen or page, a new route is created and pushed onto a stack of routes maintained by the Navigator widget.
- The current route is displayed on the screen, while the previous routes remain in the stack. When the user wants to go back to a previous screen or page, the topmost route is popped off the stack, and the previous route is displayed.



Navigator in Flutter

- To navigate back to the previous screen, we can use the pop method:

```
ElevatedButton(
    child: Text('Close screen’),
    onPressed: () {
        Navigator.pop(context);
    },
)
```

Send & Receive data

Sending data between routes

- To send data from one route to another, we need to pass the data as arguments when navigating to the new route. Here is an example of how to do this:

```
class SecondScreen extends StatelessWidget {
    final String data;
    SecondScreen({required this.data});
    // ...
}
```

- Navigate to the second screen and pass the data as an argument:

```
Navigator.push(context, MaterialPageRoute(
        builder: (context) => SecondScreen(data: 'Hello World'),
    ),
);
```

Sending data between routes

- **Sending data between routes**
  - In the second screen, retrieve the data by accessing the arguments:

```
class SecondScreen extends StatelessWidget {
    final String data;
    SecondScreen({required this.data});

    Widget build(BuildContext context) {
        return Text(data);
    }
}
```

Sending data between routes

- **Sending data between routes**
  - In the second screen, retrieve the data by accessing the arguments:

```
class SecondScreen extends StatefulWidget {
    final String data;
    ...
}

class _SecondScreenState extends State<SecondScreen> {
    Widget build(BuildContext context) {
        return Text(widget.data);
    }
}
```

Receiving data returned from a route

- To receive data from a previous route, we can use the Navigator.pop method to return data to the previous route.
- Define the data type of the data you want to receive. For example, let's say we want to receive a string value:

```
onPressed: () async {
    String? result = await Navigator.push(context, MaterialPageRoute(
            builder: (context) => SecondScreen(),
        ),
    );
);
```

Receiving data returned from a route

- Navigate to the second screen and return the data to the previous screen using Navigator.pop:

```
ElevatedButton(
    child: Text('Close screen’),
    onPressed: () {
        Navigator.pop(context, 'Selected Value’);
    },
)
```

Receiving data returned from a route

- In the first screen, retrieve the data by checking the returned result:

```
onPressed: () async {
    String result = await Navigator.push(context, MaterialPageRoute(
            builder: (context) => SecondScreen(),
        ),
    );

    if (result != null) {
        ScaffoldMessenger.of(context)
        ..removeCurrentSnackBar()
        ..showSnackBar(SnackBar(content: Text('$result')));
    }
);
```

pop() vs popUntil()

- pop method is used to go back to the previous screen or route in the navigation stack. It is usually called on the Navigator object and removes the topmost route from the stack.

```
Navigator.pop(context);
```

- popUntil method, on the other hand, is used to remove all the routes from the navigation stack until a given condition is met.
- You pass a RoutePredicate to the method, which is a function that takes a Route object and returns a boolean value.

```
Navigator.popUntil(context, ModalRoute.withName('/home'));
```

- In this example, the popUntil method will remove all routes from the stack until it reaches the route with the name /home.

Named Route

Named Route

- Named routes in Flutter are a way to manage navigation between screens by defining a unique name for each screen and using that name to navigate to the desired screen.
- Unlike the default approach, which relies solely on the route name to navigate to a specific screen, named routes provide a more flexible and type-safe alternative that supports passing arguments between screens.

Named Route

- To use named routes in Flutter, you define a map of route names to screen widgets in the MaterialApp widget.
- For example, suppose you have an app with three screens: Home, Profile, and Settings. You can define named routes for each screen like this:

```
MaterialApp(
    initialRoute: '/',
    routes: {
        '/': (context) => HomeScreen(),
        '/profile': (context) => ProfileScreen(),
        '/settings': (context) => SettingsScreen(),
    },
);
```

Named Route

- In this example, the initial route is set to the Home screen, and each screen is associated with a unique route name. To navigate to a screen using a named route, you use the Navigator.pushNamed() method:

```
Navigator.pushNamed(context, '/profile');
```

- This code navigates to the Profile screen using the '/profile' route name. You can also pass arguments to the destination screen using a map of key-value pairs:

```
Navigator.pushNamed(context, '/profile', arguments: {'userId': 123});
```

- In this example, the arguments map contains a key-value pair that specifies the user ID to display on the Profile screen.

Named Route

- When the destination screen is loaded, you can access the passed arguments using the ModalRoute.of() method:

```
class ProfileScreen extends StatelessWidget {

    Widget build(BuildContext context) {
        final Map<String, dynamic> args =
                      ModalRoute.of(context)!.settings.arguments as Map<String, dynamic>;

        final userId = args['userId'];
        ...
    }
}
```

- In this example, the arguments map is retrieved using the ModalRoute.of() method, and the user ID is extracted and stored in the userId variable.

Named Route

- In summary, named routes in Flutter provide a more flexible and type-safe way to manage navigation between screens. By defining a unique route name for each screen and using that name to navigate to the desired screen, developers can create more robust and maintainable apps. Additionally, named routes support passing arguments between screens using a map of key-value pairs, which allows for greater flexibility in the app's behavior.

Router

Flutter Router

- In Flutter, the Router is a powerful tool that allows developers to manage navigation and routing within their apps. The Router widget provides a way to create and manage a custom navigation stack, which can be used to navigate between screens in the app.
- The Router widget works by creating a custom Route object for each screen in the app. A Route object represents a single screen and contains information such as the screen's widget tree, its transition animation, and any parameters or data required by the screen.

Flutter Router

- To use the Router widget in a Flutter app, you first need to create a custom RouteFactory function that returns a new Route object for each screen in the app. The RouteFactory function takes a RouteSettings object as its argument, which contains information such as the name of the route and any parameters or data required by the screen.

```
Route<dynamic>? _onGenerateRoute(RouteSettings settings) {
    if (settings.name == '/') {
        return MaterialPageRoute(
            builder: (context) => HomeScreen(),
        );
    }
    return null;
}
```

Flutter Router

- In this example, the _onGenerateRoute function checks the name of the route specified in the RouteSettings object and returns a new MaterialPageRoute object for the Home screen. If the route name does not match any known routes, the function returns null.
- Once you've defined your custom RouteFactory function, you can use it to create a new Router widget in your app's widget tree:

```
MaterialApp(
    title: 'My App',
    onGenerateRoute: _onGenerateRoute,
    // Other app settings...
)
```

- In this example, the onGenerateRoute parameter is set to the _onGenerateRoute function, which will be called whenever the app needs to create a new Route object for a screen.

Flutter Router

- To pass data to a route using the onGenerateRoute() method, you can define a route that includes a parameter for the data you want to pass. For example, you might define a route like this:

```
onGenerateRoute: (settings) {
    if (settings.name == '/details') {
        return MaterialPageRoute(
            builder: (context) => DetailsScreen(data: settings.arguments),
        );
    }
}
```

- In this example, the DetailsScreen widget is being navigated to using the /details named route. The data parameter is being passed to the DetailsScreen widget using the arguments property of the settings object.

Flutter Router

- To pass data to this route, you would navigate to the route using the Navigator.pushNamed() method and pass the data as an argument:

```
Navigator.pushNamed(context, '/details', arguments: myData);
```

- In this example, the myData variable is being passed as the argument for the /details named route.

Flutter Router

- Once the route is generated using the onGenerateRoute() method, the data can be accessed by the receiving widget using the ModalRoute.of() method. For example:

```
class DetailsScreen extends StatelessWidget {
    Widget build(BuildContext context) {
        final data = ModalRoute.of(context)!.settings.arguments;
        return Text(data);
    }
}
```

Flutter Router

- The Router widget provides a variety of features for managing navigation and routing within your app. For example, you can use the Navigator.push() and Navigator.pop() methods to add or remove Route objects from the navigation stack, or you can use the Navigator.replace() method to replace the current Route object with a new one.

- Additionally, the Router widget provides a way to handle navigation events, such as when the user taps the back button on their device. You can use the NavigatorObserver class to observe these events and take action accordingly, such as displaying a confirmation dialog before navigating back to the previous screen.

Flutter Router

- Here are some specific things that the onGenerateRoute property can do that the routes property cannot:
  - **Handle dynamic routing**: With onGenerateRoute, you can create routes at runtime based on some input, such as user actions or network requests. This is not possible with routes, which requires you to define all routes statically.
  - **Handle unknown routes**: If a user navigates to a route that is not defined in the routes map, onGenerateRoute will be called to handle the request. This allows you to display a custom error page or redirect the user to a different page, rather than simply throwing an error.
  - **Use custom route transitions**: onGenerateRoute allows you to define custom page transitions between routes, which can be more complex than the simple transitions provided by routes.

URL Strategy

URL strategy on the web

- Flutter web apps support two ways of configuring URL-based navigation on the web:
  - Hash (default): Paths are read and written to the hash fragment
    - For example, flutterexample.dev/#/path/to/screen.
  - Path: Paths are read and written without a hash.
    - For example, flutterexample.dev/path/to/screen.

URL strategy on the web

- Configuring the URL strategy
  - To configure Flutter to use the path instead, use the usePathUrlStrategy function provided by the flutter_web_plugins library in the SDK:

```
import 'package:flutter_web_plugins/url_strategy.dart';
void main() {
    usePathUrlStrategy();
    runApp(ExampleApp());
}
```

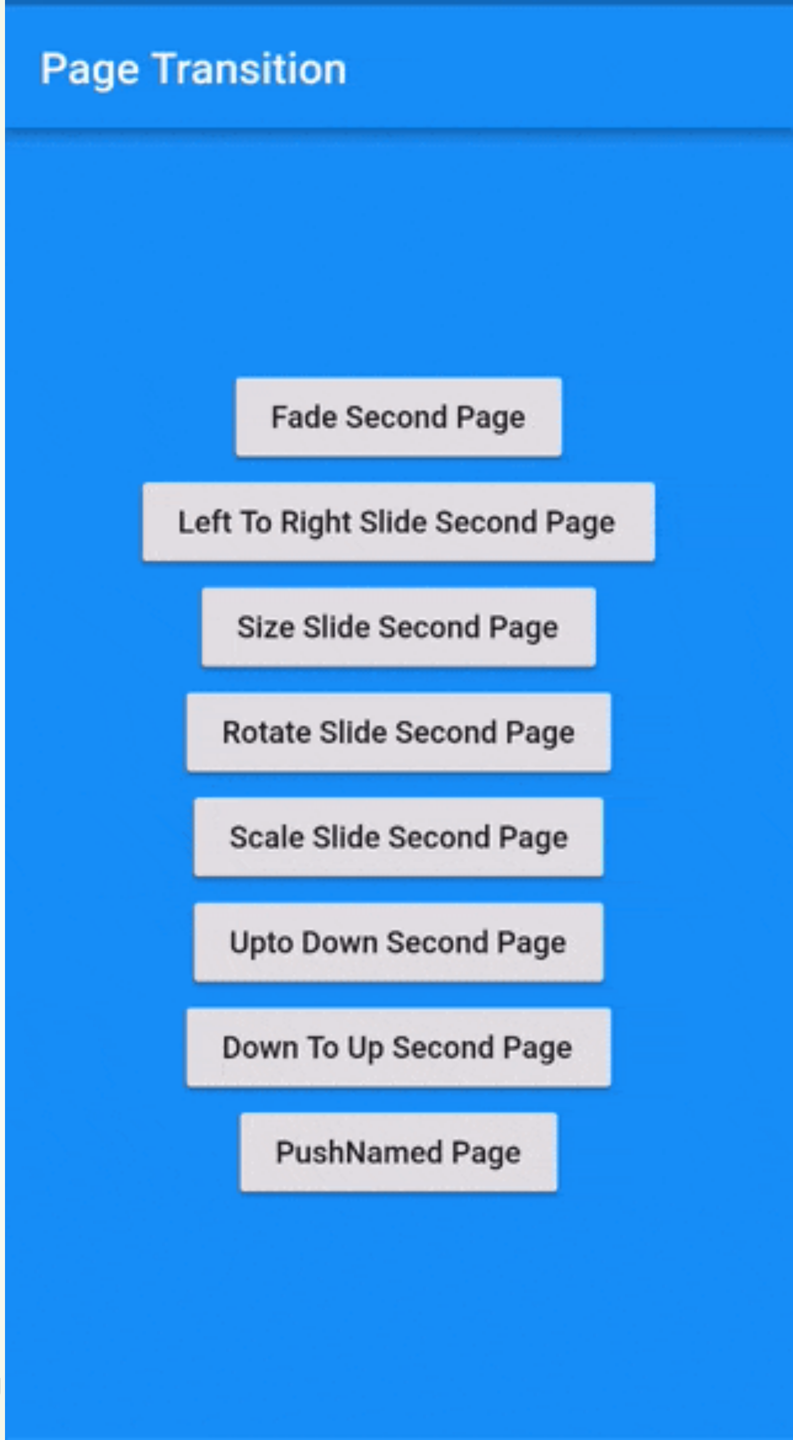
URL strategy on the web

- Configuring your web server
  - PathUrlStrategy uses the History API, which requires additional configuration for web servers.
  - To configure your web server to support PathUrlStrategy, check your web server’s documentation to rewrite requests to index.html. Check your web server’s documentation for details on how to configure single-page apps.
  - If you are using Firebase Hosting, choose the “Configure as a single-page app” option when initializing your project. For more information see Firebase’s Configure rewrites documentation.
  - The local dev server created by running flutter run -d chrome is configured to handle any path gracefully and fallback to your app’s index.html file.

Page Transition

Page Transition

- The page_transition package is a collection of animation transitions that can be used to create stunning page transitions in Flutter applications. It provides a wide range of animated transitions such as FadeTransition, ScaleTransition, SlideTransition, and many more. The package is easy to use and can be integrated seamlessly into your Flutter application.



Page Transition

- One of the key advantages of the page_transition package is that it provides a range of transitions that are not available out of the box in Flutter. This makes it an excellent choice for developers who want to create unique and visually appealing page transitions in their applications.

```
flutter pub add page_transition
```

- Once the package is added to your project, you can import it into your Flutter code and use it to create stunning page transitions.

```
RaisedButton( child: Text("Go to Next Page"), onPressed: () {
    Navigator.push(context,
        PageTransition(
            type: PageTransitionType.fade,
            child: SecondPage(),
        ),
    );
})
```

Page Transition

- The PageTransition widget in the page_transition package provides a variety of properties that can be used to customize the animation transition in a Flutter application. Here are some of the most commonly used properties:
  - type - Specifies the type of animation transition. The available types are FadeTransition, ScaleTransition, SlideTransition, and many more.
  - duration - Specifies the duration of the transition animation in milliseconds.
  - reverseDuration - Specifies the duration of the reverse transition animation in milliseconds.
  - child - Specifies the child widget that will be transitioned to the next page.
  - curve - Specifies the curve used for the animation transition. A curve defines the rate at which the animation progresses over time.
  - alignment - Specifies the alignment of the animation. This is used to determine the starting and ending position of the animation.
  - secondaryAnimation - Specifies the secondary animation that will be played in addition to the primary animation. This is useful for creating complex animations with multiple layers.
  - transitionBuilder - Specifies a custom transition builder function that can be used to create a custom animation transition.