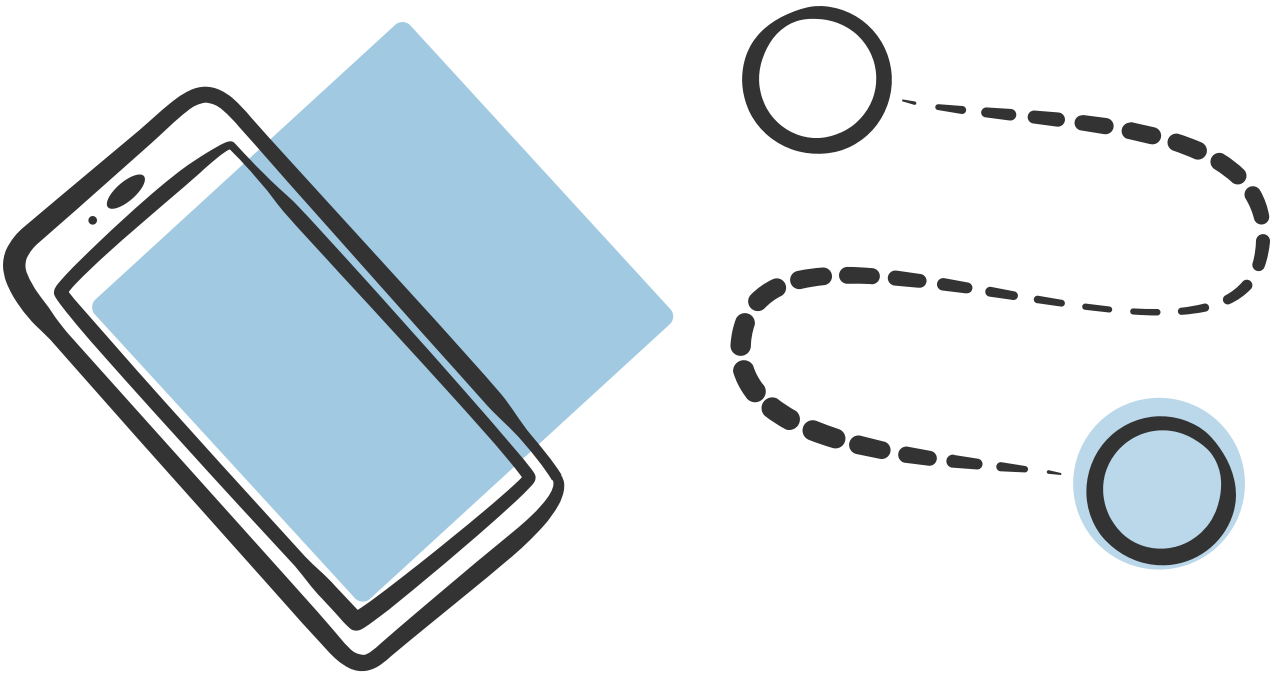
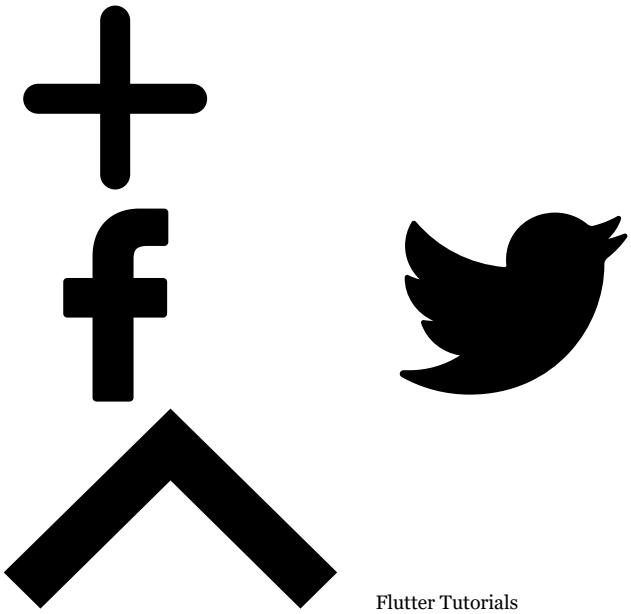




Unlocking Your Flutter Widgets With Keys

Learn how using the right keys in your Flutter widgets can help you avoid UI bugs and improve the performance of your app.

By Michael Malak Jul 6 2021 · Article (25 mins) · Beginner

4.8/5 6 Ratings

Flutter commonly uses *keys* when it needs to uniquely identify specific widgets within a collection. Using keys also helps Flutter preserve the state of `StatefulWidgets` while they're being replaced with other widgets or just moved in the *widget tree*. Almost all Flutter widgets accept keys as optional parameters in their constructors.

Have you wondered when to pass a key and what happens under the hood? In this tutorial, you'll unlock that mystery as you build a simple app to manage a TODO list and display news headlines.

By the end of this tutorial, you'll learn:

What keys are and how they work.

When to use a key.

How to work with different types of keys.

Note: This tutorial assumes that you have some experience with Flutter and Flutter widgets. If you don't, check out our Getting Started with Flutter tutorial, our Flutter UI Widgets video course or our Flutter Apprentice book.

Getting Started

Download the starter project by clicking the *Download Materials* button at the top or bottom of the tutorial.

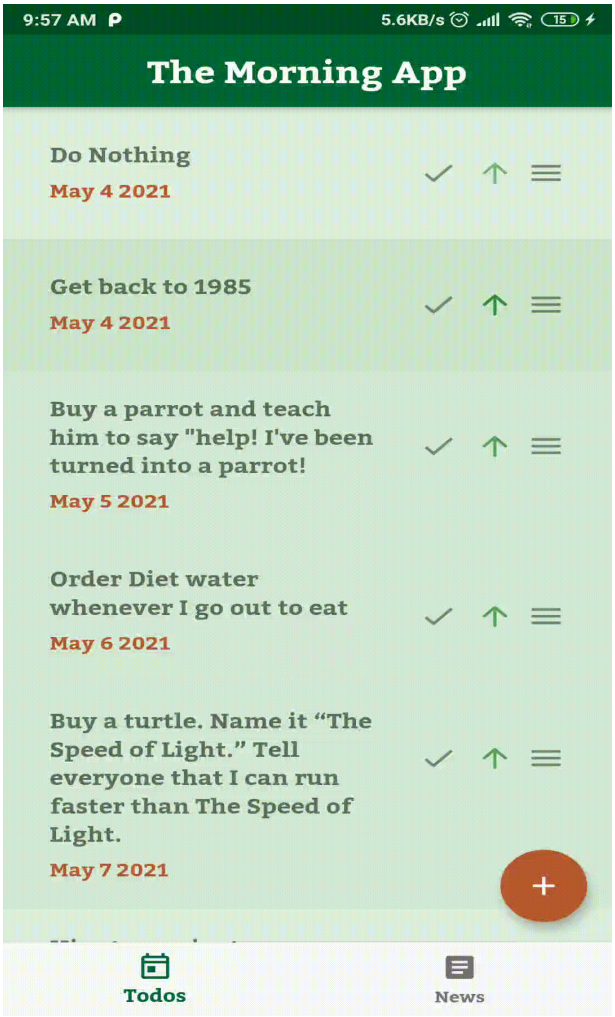
This tutorial uses Android Studio 4.1. Some of the screenshots are specific to it, but you can follow along with Visual Studio Code or IntelliJ as well.

You'll work on *The Morning App*, a single-page app that displays a TODO list on one tab and a list of news articles on another. Here's what you want to be able do with each page:

Todos: Add a new TODO, mark a TODO as done or delete a TODO.

News: View the latest news articles from *HackerNews* and tap an article to view some of its metadata.

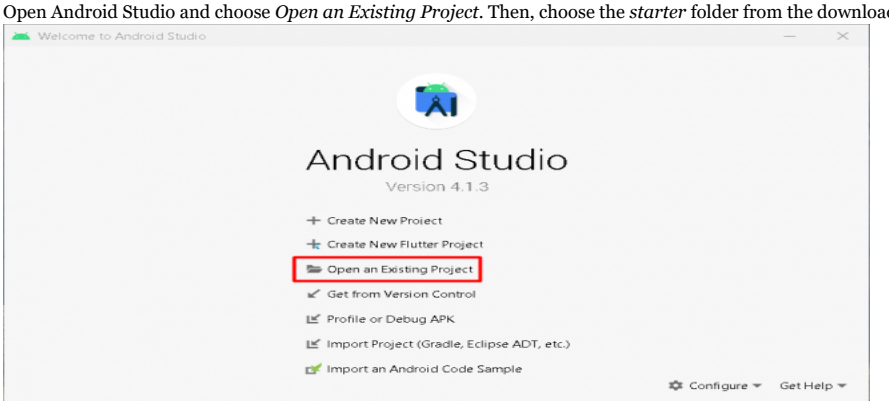
Here's how the pages will look when you're done:



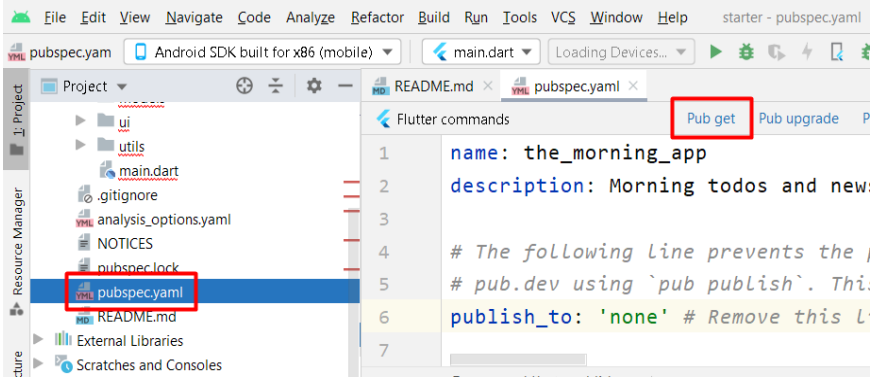
Now, it's time to take a look at the project.

Setting up the Starter Project

The starter project already contains the logic to fetch articles from HackerNews, save TODOs to the cache and read TODOs from the cache.
Open Android Studio and choose *Open an Existing Project*. Then, choose the *starter* folder from the downloaded materials.



Fetch the dependencies declared in *pubspec.yaml* by clicking *Pub get* at the top of the pane when you're in this file.



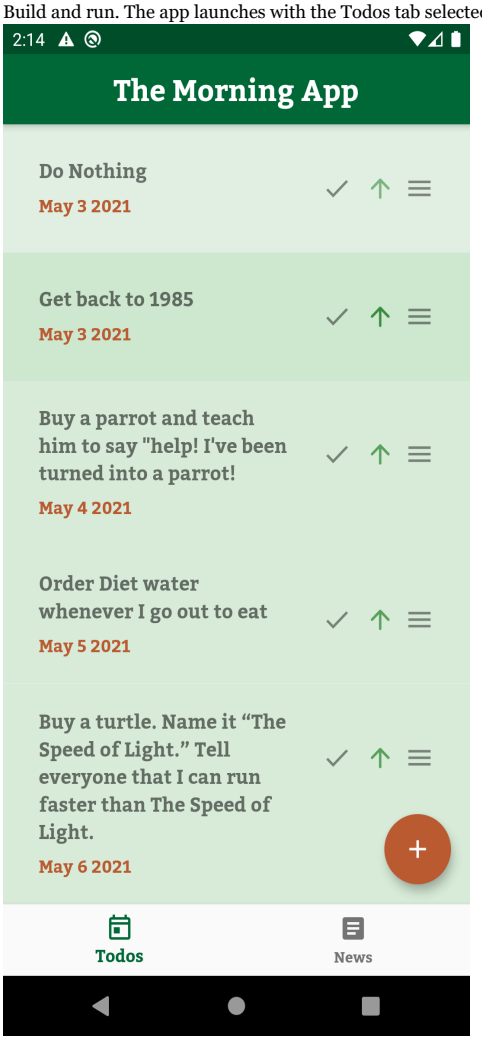
For this tutorial, the most important files in the project are:

. *lib/ui/home/home_page.dart*: The main page of the app that displays the two tabs for displaying TODOs and news articles.

. *lib/ui/todos/todos_page.dart*: The widget class of the page associated with the Todos tab.

. *lib/ui/news/news_page.dart*: The widget class of the page associated with the News tab.

. *lib/ui/todos/add_todo_widget.dart*: The widget class representing the bottom sheet, where the user can add a new TODO on the Todos page.
Build and run. The app launches with the Todos tab selected.



Now that you know what the starter project contains, you’ll take a deeper look at what keys are and why you use them.

Understanding Keys

Every Flutter widget can have a key, but adding them isn’t always useful. Here’s the key to understanding keys:

. Multiple widgets of the same type and at the same level in a widget tree may not update as expected unless they have unique keys, given that these widgets hold some state.

. Explicitly setting a key to a widget helps Flutter understand which widget it needs to update when state changes.

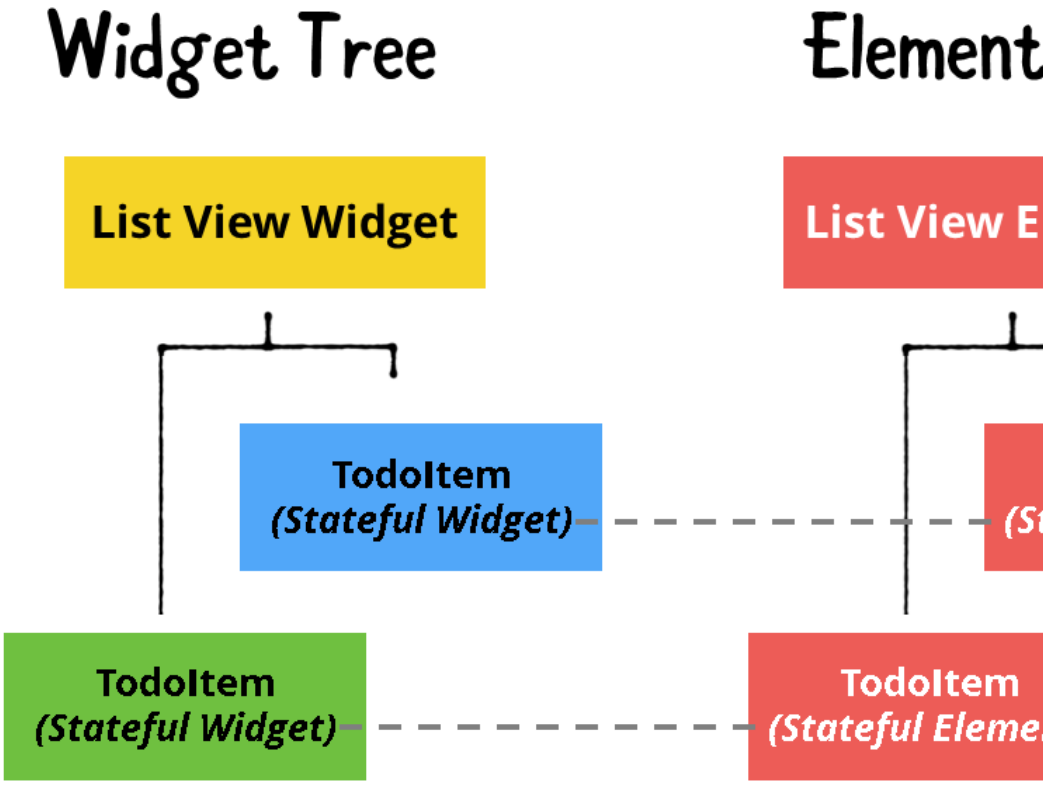
. Among other things, keys also store and restore the current scroll position in a list of widgets.

Consider an example to understand this better:

When Flutter lays out the widget tree, it builds a corresponding *element tree*. Internally, it maps each widget in the widget tree to an element in the element tree. The widget tree contains information about the UI and the element tree holds information about the structure of the app — meaning that each element holds details about:

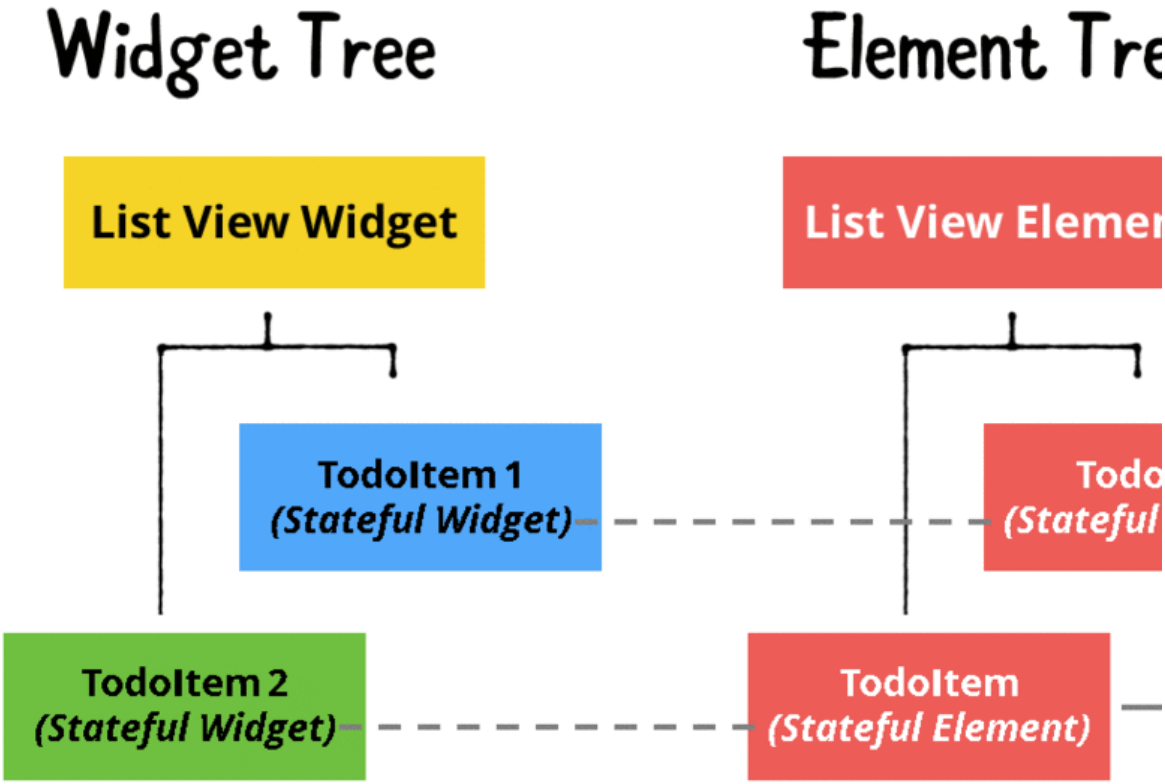
The `runtimeType` of the corresponding widget in the widget tree.
The reference to the corresponding widget in the widget tree.
The reference to its child `Element`.

You can extract the rest of the information from the reference to the widget tree that each element holds.



Making Swaps With Stateless Widgets

Every change to the UI in a Flutter app is a result of triggering the build method. During this process, Flutter checks if the element tree is the same as the corresponding widget tree. Flutter makes this comparison starting from the parent widget, then proceeding to its children widgets.
`StatelessWidgets` have no keys. Therefore, if the element has the same type as the corresponding new widget, the element updates its reference to point to the new widget and drops the reference to the old widget.

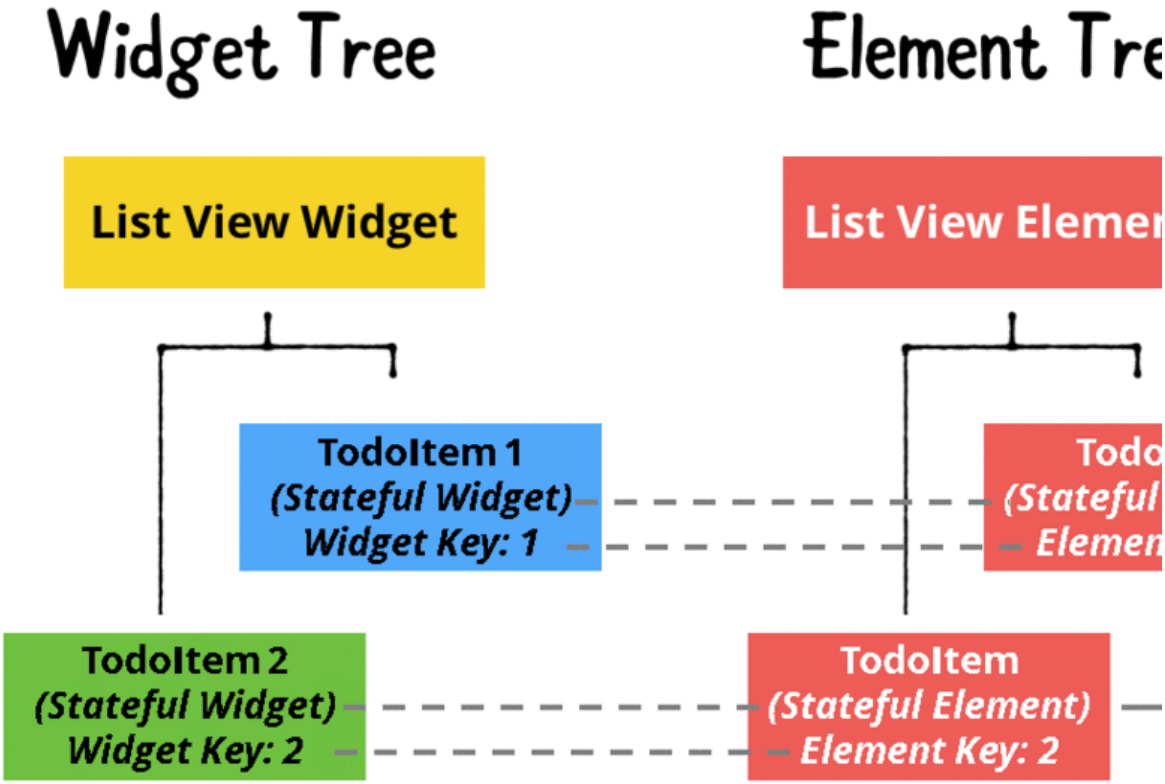


Handling Swaps in Stateful Widgets

In the case of `StatefulWidgets`, however, an element stores a reference to the state of a widget — for example, `State` — as well.
Therefore, the new widget could have the same runtime type as the old widget, but a different state. Based on the logic above, Flutter would update the reference of the widget in the element to point to the new widget but the element would still hold a reference to the state from the old widget. That's a problem.

Using Keys to Avoid Unexpected Results

Adding a key to a widget that holds a reference to the state allows Flutter to make an additional comparison beyond the type of the widget. This ensures that when the types match but the keys don't, Flutter forces the elements to drop to their widget reference and hold references to widgets where both the type and key match. This ensures that both the widget and state references update correctly.



You'll see this in action later in the tutorial.
Now that you understand some theory behind using keys, it's time to put that information to work by adding some new features to The Morning App.

Reordering TODOs

At this point, the starter code displays the TODOs. Your first goal is to give the users the ability to sort the TODO items by dragging and dropping them to new positions in the list.
In *lib/ui/todos/todos_page.dart*, replace `//TODO: Reorder To-dos` with:

```
// 1
void reorderTodos(int oldIndex, int newIndex) {
  // 2
  if (oldIndex < newIndex) {
    newIndex -= 1;
  }

  // 3
  final item = todos.removeAt(oldIndex);
  setState(() {
    todos.insert(newIndex, item);
  });
}
```

Here's what you did:
You added a function to reorder the TODOs. That function takes two indices as parameters: `oldIndex` is the index of the TODO whose position will change and `newIndex` is the new index where you'll place the TODO.
Since you're going to remove the TODO from the old index then insert it into the new one, you subtracted 1 from `newIndex` in case it's after `oldIndex`.
You removed the element at `oldIndex` and inserted it into `newIndex`. You then called `setState` so the UI reflects the changes.
Now that you've made the TODO items sortable, it's time to add the ability to drag and drop them.

Enabling Drag and Drop

Start by going to `_TodosPageState` and replacing `buildTodoList` with the following:

```
ReorderableListView buildTodoList() {
  // 1
  return ReorderableListView(
    padding: const EdgeInsets.only(bottom: 90),
    children: todos
```

```
        .map(
          (todo) => TodoItemWidget(
            todo: todo,
            isLast: todo == todos.last,
            todosRepository: widget.todosRepository,
          ),
        )
        .toList(),
    // 3
    onReorder: reorderTodos,
  );
}
```

Here's what's happening above:

. You replaced `ListView` with Flutter's `ReorderableListView` so you can drag and drop TODO items and change their positions in the list.

. This function uses `reorderTodos` for `onReorder`. You'll call this function whenever you drag and drop a TODO in the list.

Adding a Key

`TodoItemWidget` is a `StatefulWidget` that holds the state of the deletion of the TODO. Since you're working with a collection of `StatefulWidgets`, you'll add a key that identifies each TODO item.

There are two types of keys in Flutter: `GlobalKeys` and `LocalKeys`.

The different types of `LocalKeys` are:

ValueKey: A key that uses a simple value such as a String.

ObjectKey: A key that uses a more complex data format rather than a primitive data type like a String.

UniqueKey: A key that is unique only to itself. Use this type of key when you don't have any other data that makes a widget unique when using a `ValueKey` or an `ObjectKey`.

Add the `key` from the following snippet to the line where you instantiate `TodoItemWidget` in `buildTodoList` in *todos_page.dart*:

```
(todo) => TodoItemWidget(
  key: ObjectKey(todo),
  ...
),
```

In this case, there are no unique IDs for TODOs. What makes a TODO unique is the data it holds: text, priority and due date. So using an `ObjectKey` is the most suitable option here, assuming the user would never add two TODOs containing the exact same information.

Note: Before considering an object as an `ObjectKey`, make sure that it is comparable. If not, override the object's == operator and the `hashCode` getter to ensure that any two objects holding the same data are equal.

Perform a hot restart. You can now drag and drop TODOs to change their positions in the TODO list while preserving their deleted state.

Using Global Keys

The Keys you saw above are `LocalKeys`, which aren't globally unique across the entire app's widget hierarchy. You can also use some keys as unique references across the entire app. This is a job for `GlobalKeys`.

`GlobalKeys` are rarely necessary. They allow widgets to change parents without losing state. They also let you access the widget's info from a different part of the widget tree.

Note: Be cautious of the fact that reparenting an `Element` using a global key is relatively expensive and could cause massive widget rebuilds.

One of the most common ways to use a `GlobalKey` is with `Form` — as you'll see in the next section.

Note: `GlobalKeys` are like global variables: Try not to overuse them. There are almost always better ways to preserve the state globally using the right state management solution.

Adding a TODO

Even though you can view the TODOs in the app at this point, you can't add any. It's time to fix that.

Go to *lib/ui/todos/add_todo_widget.dart* and replace `TODO: Adding GlobalKey for FormState` with:

```
// 1
final formKey = GlobalKey<FormState>();

// 2
void addTodo() {
  // 3
  if (formKey.currentState!.validate()) {
    // 4
    formKey.currentState!.save();
    // 5
    return widget.onSubmitTap(todo);
  }
}
```

Here's a detailed breakdown of the code snippet above:

. You added a new `GlobalKey` to use with the form you'll create. This ensures that the focus of the input elements is unique across the app.

. `addTodo` triggers when you click the *Submit* button to add a new TODO.

. Here, you check if `currentState` is valid for all the input fields. This triggers a `validator` in each `TextFormField` in the form. If all the `validators` in `TextFormFields` return `null` instead of an error String, it means that all the fields have valid input. Therefore, `formKey.currentState!.validate()` will return `true`. Notice that you use the global `formKey` here to get the state of the form.

. You call a `save` of the form's `currentState`. This triggers `onSaved` in each `TextFormField` in the form.

. Since the parent widget holds the list of TODOs, you need to make the newly created TODO available to the list of TODOs. For this, you pass the new TODO to `onSubmitTap` so the parent widget adds it to the list.

At this point, you've already created a `formKey` in `_AddTodoWidgetState`. However, you haven't assigned this key to a `Form` yet.
Next, you'll use the following snippet in *lib/ui/todos/add_todo_widget.dart* to wrap the `Padding` in the `build` inside `_AddTodoWidgetState`:

```
@override
Widget build(BuildContext context) {
  ...
  // 1
  return Form(
    // 2
    key: formKey,
    child: Padding(
      padding: const EdgeInsets.all(15),
      child: ...
    ),
  );
}
```

In the code above, you:

. Wrap all text fields in a `Form` to link them with one another as a single entity.

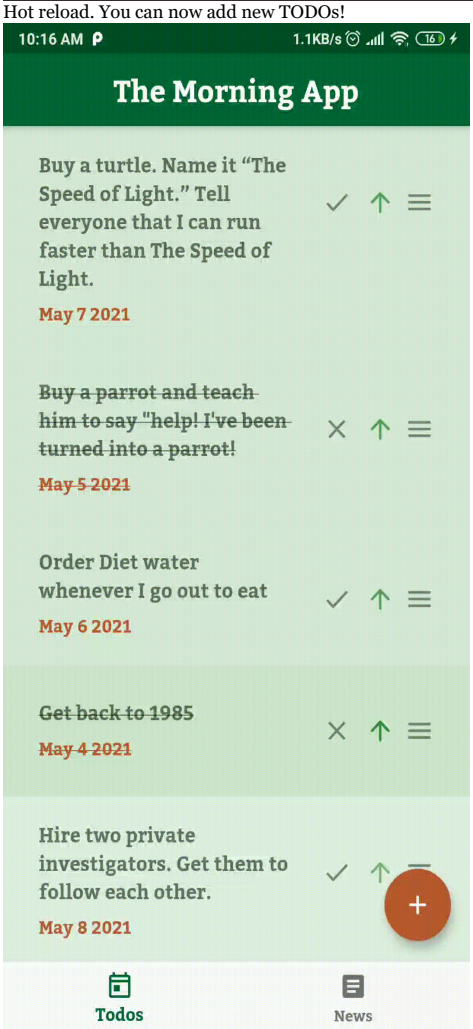
. Set the already created `formKey` as the key for this `Form`.

Adding a TODO to the List

Next, to add a TODO, you need to call `addTodo` when the user clicks the *Submit* button at the bottom of the form.
To implement this, go to `buildSubmitButton` and add `addTodo` as the `onPressed` callback function:

```
return ElevatedButton(
  style: ...,
  onPressed: addTodo,
  child: ...
);
```

Hot reload. You can now add new TODOs!



Preserving the Scroll Position

The app has two lists stacked next to each other on different tabs: a Todo list and a News list. It would be nice to preserve the scroll position in both lists when you navigate back and forth between them.

To persist the widget's state even after its destruction, Flutter suggests using `PageStorageKeys`. These keys, in combination with `PageStorageBucket`, allow you to use the key to store and restore the state.

Go to *lib/ui/home/home_page.dart* and replace `//TODO: Preserve Scroll Position on tab change` with:

```
final PageStorageBucket _bucket = PageStorageBucket();
```

`PageStorageBucket` stores state per page and persists it when the user navigates between pages.

Now that you've created the bucket, add a unique `PageStorageKey` to each page in `_HomePageState`, as follows:

```
final pages = <Widget>[
  TodosPage(
    key: const PageStorageKey('todos'),
    ...
  ),
  NewsPage(
    key: const PageStorageKey('news'),
    ...
  ),
];
```

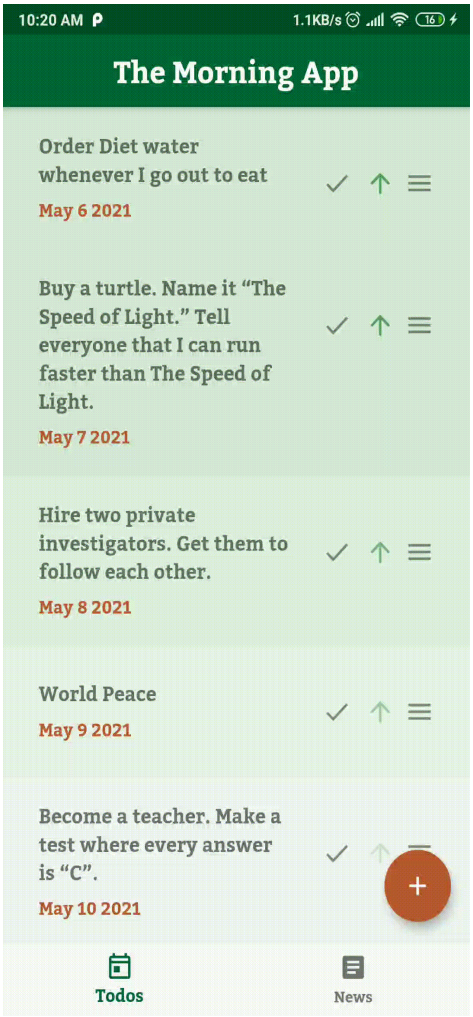
Here, you want to ensure that the Todos page and the News page have unique identifiers. To do this, provide two hard-coded values for the `PageStorageKeys`: `'todos'` and `'news'`.

Next, to link these two keys with `PageStorageBucket`, wrap the `body` of the `HomePage`'s `Scaffold` with `PageStorage`:

```
return Scaffold(
  appBar: ...,
  body: PageStorage(
    child: pages[currentTab],
    bucket: _bucket,
  ),
  bottomNavigationBar: ...,
);
```

Here, `PageStorage` links `PageStorageBucket` to `PageStorageKeys`.

Now perform a hot restart. You can see that the app now preserves the scroll position of both lists.



Preserving the News State

You've now added `PageStorage` with `PageStorageKeys` to `HomePage`. However, every time you open the news tab, the app shows a loading indicator while it fetches news articles from HackerNews. This could change the order of the news articles in the list.
To fix this, use `PageStorage` to store the entire news list, in addition to the current scroll position of the list.
To do so, go to *lib/ui/news/news_page.dart* and replace `//TODO: Preserve News State on tab change` with the following snippet:

```
// 1
void updateNewsState() {
  // 2
  final fetchedNews =
      PageStorage.of(context)!.readState(context, identifier: widget.key);

  // 3
  if (fetchedNews != null) {
    setNewsState(fetchedNews);
  } else {
    fetchNews();
  }
}

// 4
void saveToPageStorage(List<NewsModel> newNewsState) {
  PageStorage.of(context)!
      .writeState(context, newNewsState, identifier: widget.key);
}
```

Here, you:

. Add `updateNewsState`, which checks if there's a cached news list by using `PageStorage` instead of always fetching the news from the network.

. Read the state of the stored news from `PageStorage`, which is an inherited widget.

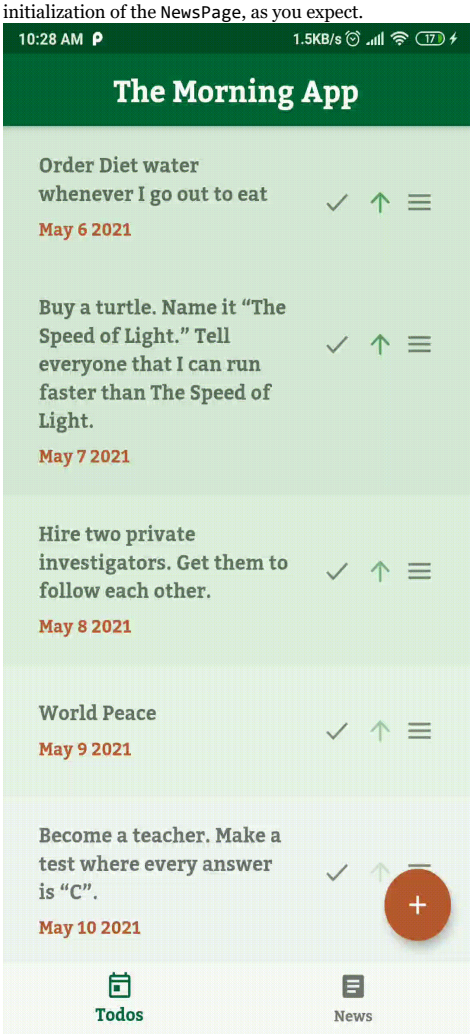
. Choose what to do, depending on whether `PageStorage` has a cached news list.

. Add `saveToPageStorage`. This caches the news list the app fetched from the network in `PageStorage`.

In `fetchNews`, add the following statement after you call `setNewsState`:

```
saveToPageStorage(shuffledNews);
```

This saves the fetched news articles list in `PageStorage`.
Finally, in the `initState` of `_NewsPageState`, replace `fetchNews();` with `updateNewsState();` so you no longer fetch news articles from the network on every initialization of `NewsPage`.
Hot restart. Now, when you switch to the News tab, the app preserves the scroll state and fetches the news articles from the network only on the first initialization of the `NewsPage`, as you expect.



But what if you've already read everything interesting? You'll handle getting new articles next.

Refetching News Articles

`PageStorage` now allows you to cache news articles, but there isn't a way to fetch the latest news. To do that, you'll implement pull-to-refresh functionality to refetch news articles. This behavior is common on mobile apps.
In `NewsPage`, replace the `Scaffold`'s `body` to the following:

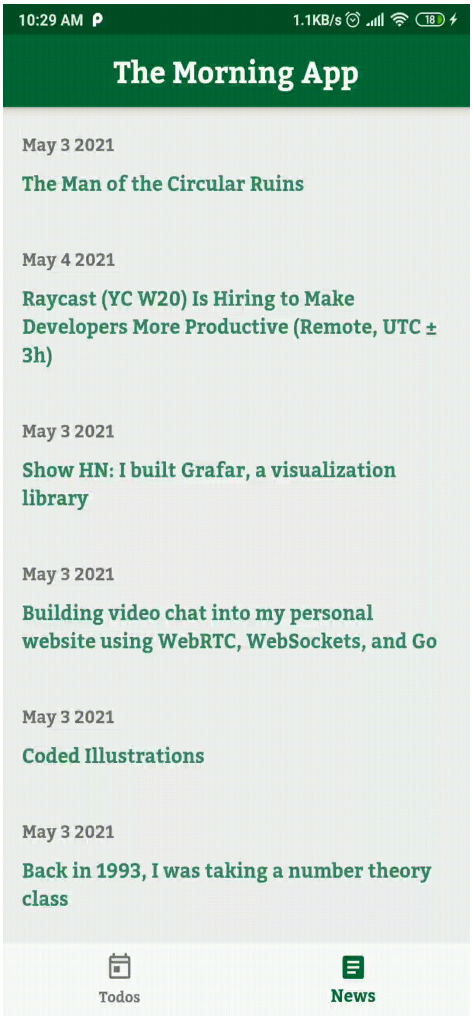
```
body: isLoading
    ? const ProgressWidget()
    :
      // 1
      RefreshIndicator(
        child: buildNewsList(),
        // 2
        onRefresh: fetchNews,
        color: AppColors.primary,
      ),
```

Here, you:

. Use Flutter's `RefreshIndicator` as a wrapper around the `ListView` containing the news articles.

. Call `fetchNews` when the user pulls on the top of the news list.

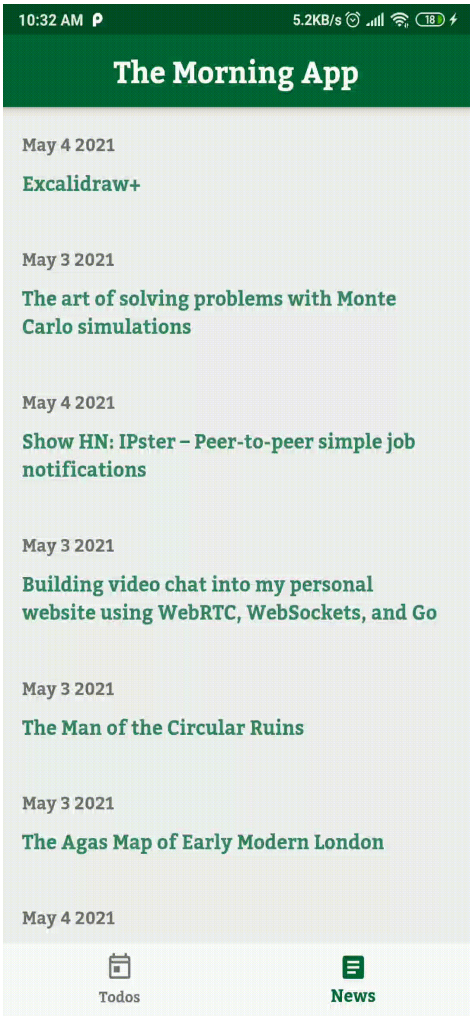
Hot reload. The app will now refetch the news articles using the refresh indicator:



Tap a news article to view metadata including the author's name and the number of votes and comments. This is the expanded state of the news article in the list view.

Fixing a Bug in the State

Notice that when you expand a news article then refresh the list, the news article at that same position expands after the refresh. This is a bit weird — you'd expect the expanded state to be tied to the news article, not to a specific position on the list.



This happens because `NewsItemWidget` is a `StatefulWidget` that holds the state of the news item — in this case, whether it's expanded or collapsed.
As you read above, you need to add a key to `NewsItemWidget` to fix this problem. So add the following key in `buildNewsList`:

```
(newsItem) => NewsItemWidget(
  key: ValueKey<int>(newsItem.id),
  ...
),
```

Since each news item has its own unique ID, you can use a `ValueKey` to help Flutter compare different `NewsItemWidgets`.
Note: When choosing a key, don't choose a random value or a value that you generate from inside the widget. For instance, don't use the index of the item in the list as the key for the widget.
Hot reload. You've fixed the issue and preserved the expanded state to match the news article instead of its position in the list of articles.

Adding Dividers

Now, you want a way to easily see where one item in the list ends and the next one begins. To do this, you'll add dividers at the bottom of each news item. To do this, add the following code in the `buildNewsList()` method inside *news_page.dart*:

```
...
.map(
  (newsItem) => Column(
    children: [
      NewsItemWidget(
        key: ValueKey<int>(newsItem.id),
        newsItem: newsItem,
      ),
      const Divider(
        color: Colors.black54,
        height: 0,
      ),
    ],
  ),
)
...
```

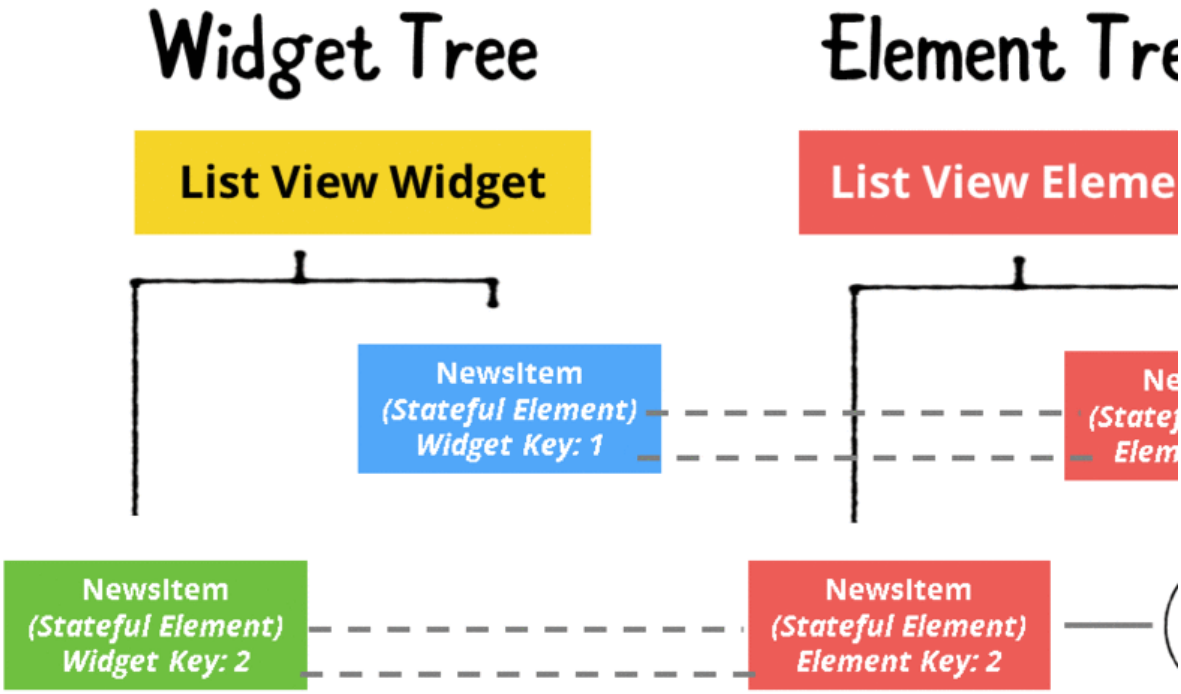
In `buildNewsList`, you currently map each `newsItem` directly to a `NewsItemWidget`. To make your change, you need to first wrap `NewsItemWidget` in a `Column`. You'll then add a `Divider` after `NewsItemWidget` in the `Column`.

Hot reload. Now, you'll see the divider. However, on every pull to refresh action, observe that the app loses the expanded state of the news articles in the list.

You'll fix that next.

Preserving the Expanded State of the News Items

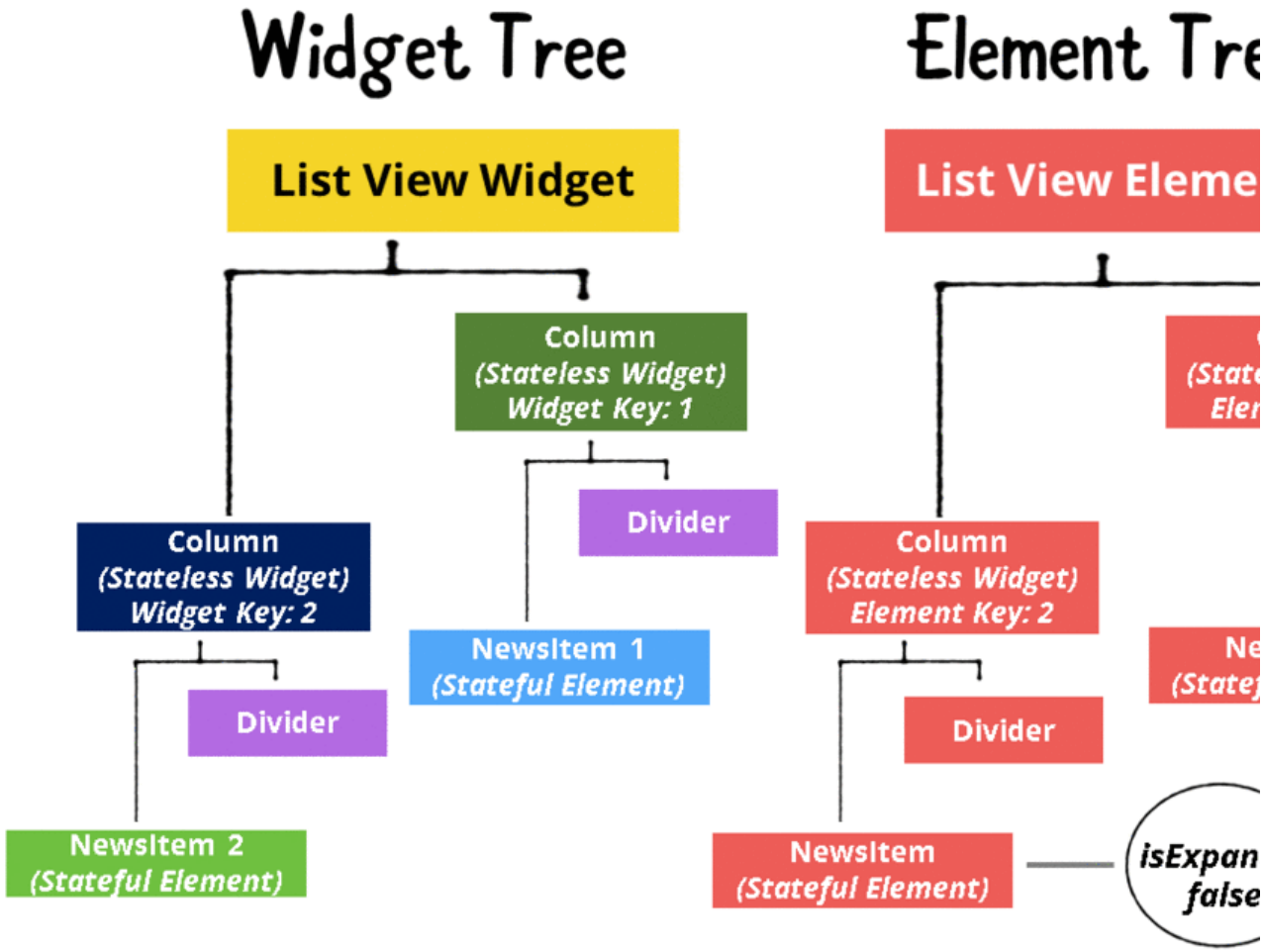
This happens because Flutter checks for keys of elements that are on the same level. In this case, Flutter compares both `NewsItemWidgets`. However, these don't contain the `Divider`.



A key should always be at the topmost level of the widget subtree you want Flutter to compare. Therefore, you want your app to compare the `Columns` wrapping both `NewsItemWidget` and `Divider`.
To do so, in `buildNewsList()`, move the key so it belongs to the `Column` instead of `NewsItemWidget`, as follows:

```
.map(
  (newsItem) => Column(
    key: ValueKey<int>(newsItem.id),
    children: [
      NewsItemWidget(
        newsItem: newsItem,
      ),
     ...
    ],
  ),
)
```

Flutter now compares the `Columns`, as you intended.



Hot reload and confirm that you once again preserve the expanded state for news articles across refreshes. Mission accomplished!

Where to Go From Here?

Download the final project files by clicking the *Download Materials* button at the top or bottom of the tutorial.
You now have a deeper understanding of widget keys, and more importantly, when and how to use them. You used `ValueKeys`, `ObjectKeys` and `PageStorageKeys` to help Flutter preserve the state of your widgets.
Check out the following links to learn more about some of the concepts in this tutorial:
When to Use Keys — Flutter Widgets 101 Ep. 4.
Flutter docs on the Element class.
We hope you enjoyed this tutorial. If you have any questions or comments, please join the forum discussion below!