Managing the data layer with InheritedWidget

How should you call the data classes in your app?

You could, in theory, set up a place in static memory where all your data classes will reside, but that won't play well with tools such as Hot Reload and could even introduce some undefined behavior down the road. The better options involve **placing your data classes in the widget tree** so they can take advantage of your application's life cycle.

The question then becomes, how can you place a model in the widget tree? Models are not widgets, after all, and there is nothing to *build* onto the screen.

A possible solution is using `InheritedWidget`. So far, we've only been using two types of widgets: `StatelessWidget` and `StatefulWidget`. Both of these widgets are concerned with rendering widgets onto the screen; the only difference is that one can change and the other cannot. `InheritedWidget` is another beast entirely. Its job is to pass data down to its children, but from a user's perspective, it's invisible. `InheritedWidget` can be used as the doorway between your **view** and **data** layers.

In this recipe, we will be updating the Master Plan app to move the storage of the to-do lists outside of the view classes.

Getting ready

You should complete the previous recipe, *Model-view separation*, before following along with this one.

How to do it...

Let's learn how to add `InheritedWidget` to our project:

1. Create a new file called `plan_provider.dart` for storing our `plans`. Place this file in the root of the project's `lib` directory. This widget extends `InheritedWidget`:

```
import 'package:flutter/material.dart';
import './models/data_layer.dart';

class PlanProvider extends InheritedWidget {
 final _plan = Plan();

 PlanProvider({Key key, Widget child}) : super(key: key, child:
   child);
```

```
 @override
 bool updateShouldNotify(InheritedWidget oldWidget) => false;
}
```

2. To make the data accessible from anywhere in the app, we need to create our first **of-context** method. Add a static `Plan of` method that takes a `BuildContext` just after `updateShouldNotify`:

```
static Plan of(BuildContext context) {
  final provider =
context.dependOnInheritedWidgetOfExactType<PlanProvider>();
  return provider._plan;
}
```

3. Now that the provider widget is ready, it needs to be placed in the tree. In the `build` method of `MasterPlanApp`, in `main.dart`, wrap `PlanScreen` with a new `PlanProvider` class. Don't forget to fix any broken imports if needed:

```
return MaterialApp(
  theme: ThemeData(primarySwatch: Colors.purple),
  home: PlanProvider(child: PlanScreen()),
);
```

4. Add two new get methods to the `plan.dart` file. These will be used to show the progress on every plan. Call the first one `completeCount` and the second `completenessMessage`:

```
int get completeCount => tasks
  .where((task) => task.complete)
  .length;

String get completenessMessage =>
    '$completeCount out of ${tasks.length} tasks';
```

5. Tweak `PlanScreen` so that it uses the PlanProvider's data instead of its own. In the `State` class, **delete the plan property** (this creates a few compile errors).

6. To fix the errors that were raised in the previous step, add `PlanProvider.of(context)` to the `_buildAddTaskButton` and `_buildList` methods:

```
Widget _buildAddTaskButton() {
  final plan = PlanProvider.of(context);
Widget _buildList() {
  final plan = PlanProvider.of(context);
```

7. Still in the `PlanScreen` class, update the `build` method so that it shows the progress message at the bottom of the screen. Wrap the `_buildList` method in an Expanded widget and wrap it in a `Column` widget.

8. Finally, add a `SafeArea` widget with `completenessMessage` at the end of `Column`. The final result is shown here:

```
@override
Widget build(BuildContext context) {
  final plan = PlanProvider.of(context);
  return Scaffold(
      appBar: AppBar(title: Text('Master Plan')),
      body: Column(children: <Widget>[
        Expanded(child: _buildList()),
        SafeArea(child: Text(plan.completenessMessage))
      ]),
      floatingActionButton: _buildAddTaskButton());
}
```

9. Change `TextField` in `_buildTaskTile` to a `TextFormField`, to make it easier to provide initial data:

```
TextFormField(
  initialValue: task.description,
  onFieldSubmitted: (text) {
    setState(() {
      task.description = text;
    });
  },
```

Finally, build and run the app. There shouldn't be any noticeable change, but by doing this, you have created a cleaner separation of concerns between your view and the models.

How it works...

InheritedWidgets are some of the most fascinating widgets in the whole Flutter framework. Their job isn't to render anything on the screen, but to **pass data down to lower widgets in the tree**. Just like any other widget in Flutter, InheritedWidgets can also have child widgets.

Let's break down the first portion of the `PlanProvider` class:

```
class PlanProvider extends InheritedWidget {
  final _plans = <Plan>[];

  PlanProvider({Key key, Widget child}) : super(key: key, child:
child);

  @override
  bool updateShouldNotify(InheritedWidget oldWidget) => false;
```

First, we define an object that will store the plans (`_plans`). Then, we define a default unnamed constructor, which takes in a `key` and a `child`, and passes them to the superclass (`super`).

`InheritedWidget` is an abstract class, so you must implement the `updateShouldNotify` method. Flutter calls this method whenever the widget is rebuilt. In the `updateShouldNotify` method, you can look at the content of the old widget and determine if the child widgets need to be notified that the data has changed. In our case, we just return `false` and opt-out of this functionality. In most cases, it is rather unlikely that you need this method to return `true`.

Then, you must create your own implementation of the **of-context** pattern:

```
static Plan of(BuildContext context) {
  final provider = context.dependOnInheritedWidgetOfExactType
   <PlanProvider>();
  return provider._plan;
}
```

Here, you are using the context's `dependOnInheritedWidgetOfExactType` method to kick off the tree traversal process. Flutter will start from the widget that owns this context and travel upward until it finds a `PlanProvider`.

An interesting side effect of this method is that after it is called, the originating widget is registered as a dependency. This now creates a hard link between the child widget and `PlanProvider`. The next time this method is called, there is no need to travel up the tree anymore; the child already knows where the data is and can retrieve it immediately. This optimization makes it extremely fast, if not almost instant, to get the data from `InheritedWidgets`, no matter how deep the tree goes.

See also

The official documentation on `InheritedWidget` can be found at https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

Making the app state visible across multiple screens

One phrase that is thrown around a lot in the Flutter community is "Lift State Up." This mantra, which originally came from React, refers to the idea that State objects should be placed higher than the widgets that need it in the widget tree. Our `InheritedWidget`, which we created in the previous recipe, works perfectly for a single screen, but it is not ideal when you add a second. The higher in the tree your state object is stored, the easier it is for your children widgets to access it.

In this recipe, you are going to add another screen to the Master Plan app so that you can create multiple plans. Accomplishing this will require our State provider to be lifted higher in the tree, closer to its root.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Let's add a second screen to the app and lift the State higher in the tree:

1. Update the `PlanProvider` class so that it can handle multiple plans. Change the storage property from a single `plan` to a list of plans:

```
final _plans = <Plan>[];
```

2. We also need to update the **of-context** method so that it returns the correct type. This will temporarily break the project, but we will fix this in the next few steps:

```
static List<Plan> of(BuildContext context) {
  final provider = context.dependOnInheritedWidgetOfExactType
   <PlanProvider>();
  return provider._plans;
}
```

3. `PlanProvider` is also going to have a new home in the widget tree. Instead of sitting ***underneath*** `MaterialApp`, we actually want this global state widget to be placed **above** it. Update the build method in `main.dart` so that it looks like this:

```
return PlanProvider(
  child: MaterialApp(
    theme: ThemeData(primarySwatch: Colors.purple),
```

4. We can now create a new screen to manage the multiple plans. This screen will depend on the `PlanProvider` to store the app's data. In the `views` folder, create a file called `plan_creator_screen.dart` and declare a new `StatefulWidget` called `PlanCreatorScreen`. Make this class the new home widget for the `MaterialApp`, replacing `PlanScreen`.

```
home: PlanCreatorScreen(),
```

5. In the `_PlanCreatorScreenState` class, we need to add a `TextEditingController` so that we can create a simple `TextField` to add new plans. Don't forget to dispose of `textController` when the widget is unmounted:

```
final textController = TextEditingController();

@override
void dispose() {
  textController.dispose();
  super.dispose();
}
```

6. Now, let's create the build method for this screen. This screen will have a `TextField` at the top and a list of plans underneath it. Add the following code before the `dispose` method to create a `Scaffold` for this screen:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Master Plans')),
    body: Column(children: <Widget>[
      _buildListCreator(),
      Expanded(child: _buildMasterPlans())
    ]),
  );
}
```

7. The `_buildListCreator` method constructs a `TextField` and calls a function to add a plan when the user taps *Enter* on their keyboard. We're going to wrap `TextField` in a `Material` widget to make the field pop out:

```
Widget _buildListCreator() {
  return Padding(
      padding: const EdgeInsets.all(20.0),
      child: Material(
        color: Theme.of(context).cardColor,
        elevation: 10,
        child: TextField(
            controller: textController,
            decoration: InputDecoration(
                labelText: 'Add a plan',
                contentPadding: EdgeInsets.all(20)),
            onEditingComplete: addPlan),
      ));
}
```

8. The `addPlan` method will check whether the user actually typed something into the field and will then reset the screen:

```
void addPlan() {
  final text = textController.text;
  if (text.isEmpty) {
    return;
  }

  final plan = Plan()..name = text;
  PlanProvider.of(context).add(plan);
  textController.clear();
  FocusScope.of(context).requestFocus(FocusNode());
  setState(() {});
}
```

9. We can create a `ListView` that will read the data from `PlanProvider` and print it onto the screen. This component will also be aware of its content and return the appropriate set of widgets:

```
Widget _buildMasterPlans() {
  final plans = PlanProvider.of(context);

  if (plans.isEmpty) {
    return Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Icon(Icons.note, size: 100, color: Colors.grey),
          Text('You do not have any plans yet.',
              style: Theme.of(context).textTheme.headline5)
        ]);
  }

  return ListView.builder(
      itemCount: plans.length,
      itemBuilder: (context, index) {
        final plan = plans[index];
        return ListTile(
            title: Text(plan.name),
            subtitle: Text(plan.completenessMessage),
            onTap: () {
              Navigator.of(context).push(
                  MaterialPageRoute(
                    builder: (_) => PlanScreen(plan: plan)));
            });
      });
}
```

10. `PlanScreen` is going to need some small tweaks as well. We need to add a constructor where the specific plan can be injected and then update the build methods to read that value. Add this property and constructor to the widget in `plan_screen.dart`:

```
final Plan plan;
const PlanScreen({Key key, this.plan}) : super(key: key);
```

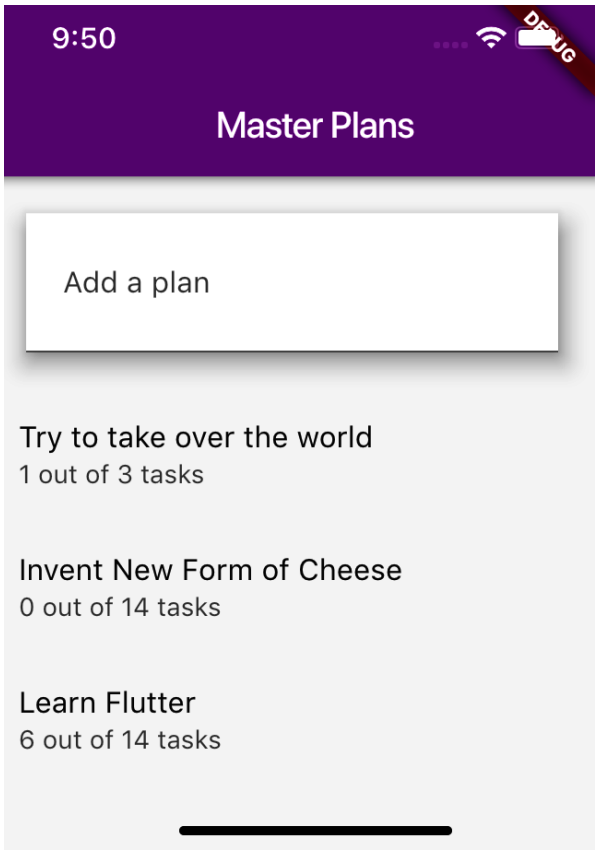
11. Finally, we just need to add an easy way to access the widget. Add this getter inside the state class:

```
Plan get plan => widget.plan;
```

12. Remove all the previous references to `PlanProvider`. You will need to skim through the class and **delete** this line everywhere it appears:

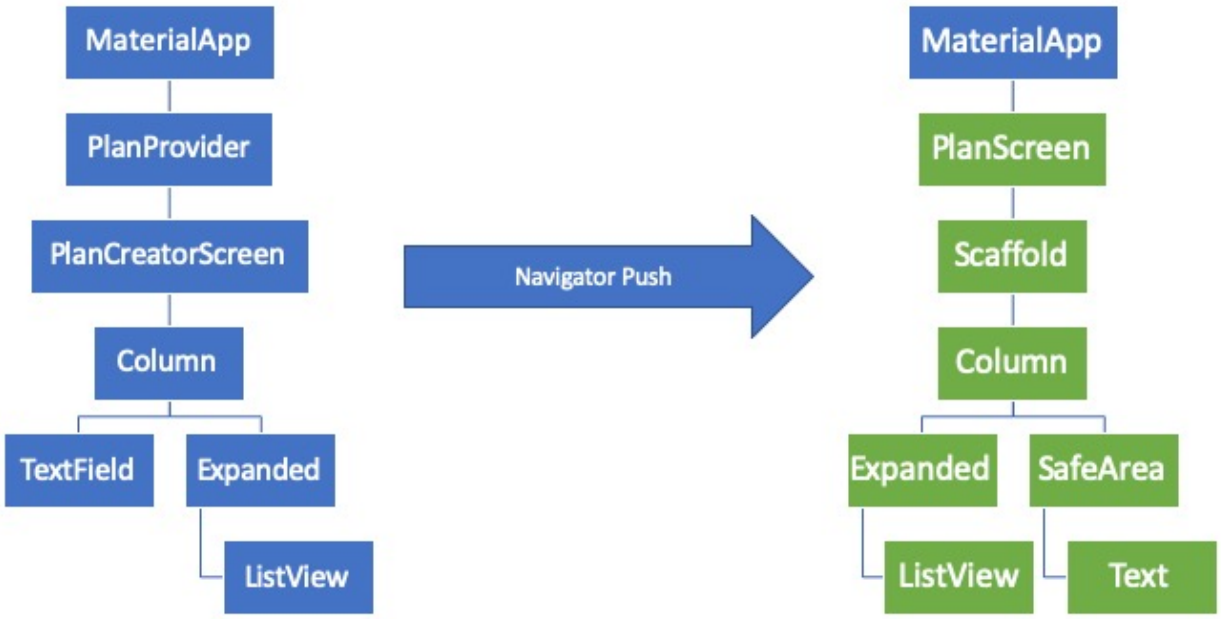
```
final plan = PlanProvider.of(context);
```

When you hot restart the app, you will be able to create multiple plans with different lists on each screen:



How it works...

The main takeaway from this recipe is the importance of proper widget tree construction. When you push a new route onto `Navigator`, you are essentially replacing every widget that lives underneath `MaterialApp`, as explained in this diagram:



If `PlanProvider` was a child of `MaterialApp`, **it would be destroyed when pushing the new route**, making all its data inaccessible to the next widget. If you have an `InheritedWidget` that only needs to provide data for a single screen, then placing it lower in the widget tree is optimal. However, if this same data needs to be accessed across multiple screens, it has to be placed above our `Navigator`.

Placing our global state widget at the root of the tree also has the added benefit of causing our app to update without any extra code. Try checking and unchecking a few tasks in your plans. You'll notice that the is data automatically updated, like magic. This is one of the primary benefits of maintaining a clean architecture in our apps.

Designing an n-tier architecture, part 1 – controllers

There are many architectural patterns that have become popular in the last couple of years – **Model View Controller** (**MVC**), **Model View ViewModel** (**MVVM**), **Model View Presenter** (**MVP**), Coordinator, and several others. These patterns have become so numerous that they are sometimes pejoratively referred to as MV* patterns. This essentially means that no matter which pattern you choose, you need some kind of intermediary object between your model and your view.

A concept that is shared by all the popular patterns is the idea of **tiers**/**layers** (we will be using the terms tier and layer interchangeably throughout this chapter). Each **tier** in your app is a

section of the MV* classes that have a single responsibility. The term *n*-tier (sometimes called a multi-tier architecture) just means that you are not limited on the number of tiers in your app. You can have as many or as few as you need.

The top-most tier is one that you are already familiar with – the view/widget tier. This tier is only interested in setting up the user interface. All the data and logic for the app should be delegated to a lower tier. The first tier that typically sits underneath the view tier is the **controller** layer. These classes are responsible for handling business logic and providing a link between the views and the lower layers in our app.

In this recipe, we'll be moving the business logic for the Master Plan app from the view to a new controller class. We will also be adding the ability to **delete** notes from the list.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

How to do it...

Let's start building an *n-tier* architecture, starting with the controller layer:

1. Since our controllers are supposed to represent a separate tier in the app, it's appropriate to put them in their own folder. Inside the `lib` directory, create a new folder called `controllers`.
2. Now, we can create a new dart file, `plan_controller.dart`. This class will be responsible for all the business logic in our app. Let's start with the class declaration:

```
import '../models/data_layer.dart';

class PlanController {
  final _plans = <Plan>[];

  // This public getter cannot be modified by any other object
  List<Plan> get plans => List.unmodifiable(_plans);
}
```

3. Now, let's add the methods that will be responsible for creating and deleting plans. This is also an appropriate location where we can apply some business logic.
First, create a private method that will check a list of items and search for duplicate names. If it finds any, a number will be appended to the end to make sure the name is unique. Add this method to the `PlanController` class:

```
String _checkForDuplicates(Iterable<String> items, String text) {
 final duplicatedCount = items
 .where((item) => item.contains(text))
```

```
 .length;
 if (duplicatedCount > 0) {
 text += ' ${duplicatedCount + 1}';
 }
 return text;
}
```

4. We can use our new business logic to check the input for new `plans`. Add the method to create a new plan, right after the public getter for the `plans` property and before the `_checkForDuplicates` method:

```
void addNewPlan(String name) {
  if (name.isEmpty) {
    return;
  }

  name = _checkForDuplicates(_plans.map((plan) => plan.name),
    name);

  final plan = Plan()..name = name;
  _plans.add(plan);
}
```

5. Under the `addNewPlan` method, add the method for deleting a plan:

```
void deletePlan(Plan plan) {
  _plans.remove(plan);
}
```

6. We can add similar methods for creating and deleting tasks inside the plan. Add the following method to add a new `Task` under the `deletePlan` method:

```
void createNewTask(Plan plan, [String description]) {
  if (description == null || description.isEmpty) {
    description = 'New Task';
  }

  description = _checkForDuplicates(
      plan.tasks.map((task) => task.description), description);

  final task = Task()..description = description;
  plan.tasks.add(task);
}
```

7. Add the method for deleting a task under the `createNewTask` method:

```
void deleteTask(Plan plan, Task task) {
  plan.tasks.remove(task);
}
```

8. With `PlanController` completed, we can now integrate it with the Flutter layer. Since we're going to follow proper separation of concerns, the only place `PlanController` is allowed to be instantiated is in the `PlanProvider` class. We need to update that class so that it can hold a `PlanController` instead of maintaining its own list. Also, update the property of the **of-context** method so that it returns the correct types (this will break the app, but don't worry – you'll fix this in the next few steps):

```
class PlanProvider extends InheritedWidget {
  final _controller = PlanController();
  /*...code elipted...*/

 static PlanController of(BuildContext context) {
 PlanProvider provider =
 context.dependOnInheritedWidgetOfExactType<PlanProvider>();
 return provider._controller;
```

9. The previous step will have created some temporary compile errors that you can now address. Open `PlanCreatorScreen` and edit the `buildMasterPlans` method, as follows:

```
Widget _buildMasterPlans() {
  final plans = PlanProvider.of(context).plans;
```

10. Edit the `addPlan` method by removing the business login from the view, as shown here:

```
void addPlan() {
  final text = textController.text;

  // All the business logic has been removed from this 'view'
  method!
  final controller = PlanProvider.of(context);
  controller.addNewPlan(text);

  textController.clear();
  FocusScope.of(context).requestFocus(FocusNode());
  setState(() {});
}
```

11. Now that the errors have been addressed, we can add a feature for deleting plans. The business logic for this feature already exists. All we need to do is add a widget that can invoke

the correct method in `PlanController`. You can wrap `ListTiles` in a `Dismissible` widget to create a nice swipe-to-delete gesture. Wrap `ListTile` in the `_buildMasterPlans()` method in a new widget and add the following code:

```
return Dismissible(
  key: ValueKey(plan),
  background: Container(color: Colors.red),
  direction: DismissDirection.endToStart,
  onDismissed: (_) {
    final controller = PlanProvider.of(context);
    controller.deletePlan(plan);
    setState(() {});
  },
  child: ListTile(...),
)
```

12. Hot reload the app. You can now create and delete plans.
13. The same updates can also be applied to the `PlanScreen` class. Make the same changes you made in the `PlanScreen` class to remove all traces of business logic from the view. In the `_buildAddTaskButton()` method, update the `onPressed` closure to create tasks via the controller:

```
onPressed: () {
  final controller = PlanProvider.of(context);
  controller.createNewTask(plan);
  setState(() {});
},
```

14. You can also use the same swipe-to-dismiss experience to delete tasks. Wrap `ListTile` in the `_buildTaskTile` method with another `Dismissible` widget:

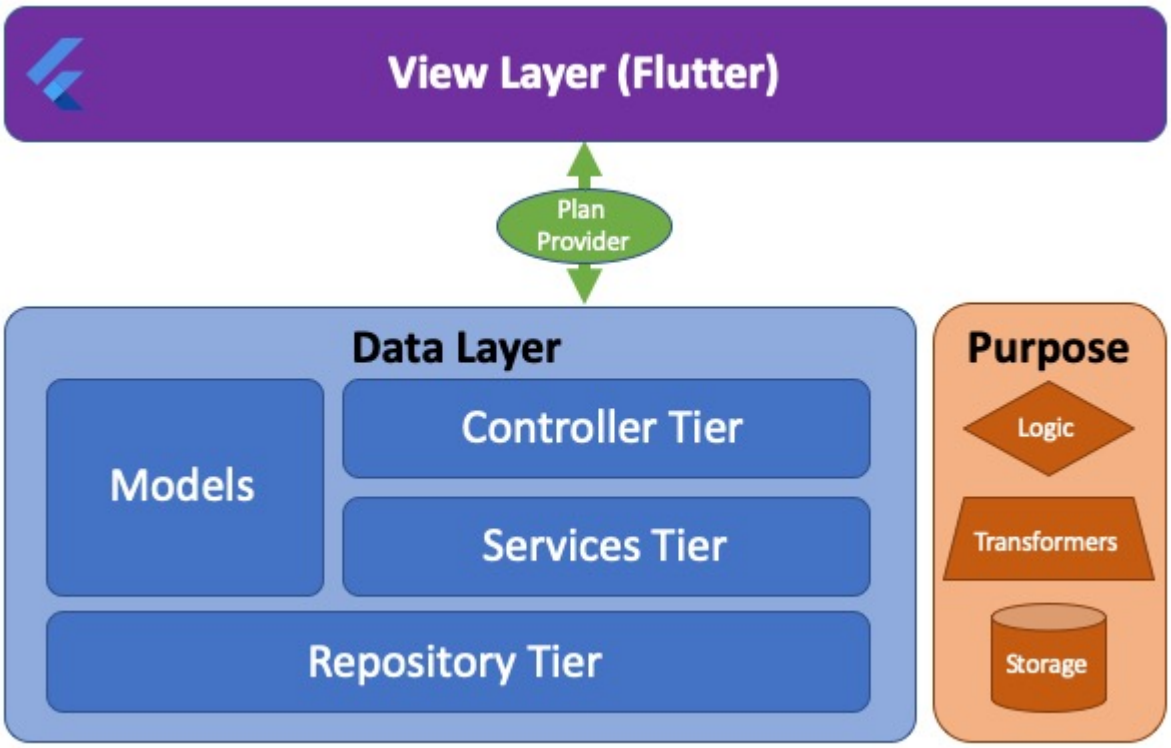
```
Widget _buildTaskTile(Task task) {
  return Dismissible(
    key: ValueKey(task),
    background: Container(color: Colors.red),
    direction: DismissDirection.endToStart,
    onDismissed: (_) {
      final controller = PlanProvider.of(context);
      controller.deleteTask(plan, task);
      setState(() {});
    },
    child: ListTile(...),
  );
}
```

15. Did you also notice that we get an annoying bug every time we hot reload? Every time we hot reload the app, all our lists disappear. This won't impact the final app, but it makes development more challenging than it should be. The solution to this bug is to simply lift the state up even higher. Remove `PlanProvider` from inside the `build` method of `MasterPlanApp` and add it as the topmost widget in the entire tree:

```
void main() => runApp(PlanProvider(child: MasterPlanApp()));
```

How it works...

The design that we will be aiming for by the end of this chapter can be summarized with this diagram:



We are beginning to divide the data layer into multiple components that will comprise the n-tier architecture – controllers, services, and repositories. While this diagram shows the full design, we will only be covering one tier at a time. In this recipe, we focused on the controller tier, which can communicate with the view layer via the `PlanProvider` interface.

To understand an n-tier architecture, it's helpful to think of your app as a cake. The topmost layer of the cake, where the icing and cherries are found, is known as the **view**. That's the first thing people see when they look at your cake.

Immediately underneath the view is where you put your **controllers**. For the purposes of this design, the job of the controllers is to **process business logic**. Business logic is defined as any rule in your app that is not related to presentation (view) or persistence (databases, web services,

and so on). The Master Plan app is pretty thin on business logic, but one key functionality that we added was a check to make sure that duplicates aren't created. This procedure has nothing to with the user interface; subsequently, it would be confusing to place it there. The correct place to store the `_checkForDuplicates` method is in the controller. You were able to reuse this exact same method for both plans and tasks. If this was placed inside the widgets, we would have to either write this code twice, once for each widget, or conjure some contrived inheritance structure. Either way, giving each class type a "job" and making sure that these roles are respected thins out the widgets and allows us to focus on one task at a time.

See also

The following resources explain the tiered architecture approach that we are striving for:

- What is Multi-Layered Software Architecture?: https://hub.packtpub.com/what-is-multi-layered-software-architecture/
- Multitiered architecture: https://en.wikipedia.org/wiki/Multitier_architecture
- Business logic: https://en.wikipedia.org/wiki/Business_logic

Designing an n-tier architecture, part 2 – repositories

The next stage of the *n*-tier architecture we are going to discuss in this recipe is the bottom-most layer: the **repositories**, or the ***data layer***. The purpose of a repository is to store and retrieve data. This layer can be implemented as a database, web service, or in the case of the Master Plan project, a simple in-memory cache. Unlike the *controller* layer, which is business logic-aware, the repository layer is only concerned with getting and storing data in its most abstract form. These classes should not even know about the **model** files that we created earlier.

The reason why repositories are so purposefully ignorant is to keep them focused entirely on their task – persistence. Communicating with a database or a web service can become complicated if you have many small requirements. These concerns are typically beneath business logic and are easier to resolve when you're only focused on abstract objects. Remember, the whole goal of an n-tier architecture is to strictly separate responsibilities; we let repositories do what they do best – store data – and let the higher layers handle the rest.

Getting ready

You should have completed the previous recipes in this chapter before following along with this one.

Get started with this tier by creating a new folder called `repositories` that will hold the code for this recipe

How to do it...

Let's define a repository interface and then implement a version of that interface as an in-memory cache:

1. In the `repositories` folder, create a new file called `repository.dart` and add the following interface:

```
import 'package:flutter/foundation.dart';

abstract class Repository {
  Model create();

  List<Model> getAll();
  Model get(int id);
  void update(Model item);

  void delete(Model item);
  void clear();
}
```

2. We also need to define a temporary storage class called `Model` that can be used in any implementation of our repository interface. Since this model is strongly coupled to the repository concept, we can add it to the same file:

```
class Model {
  final int id;
  final Map data;

  const Model({
    @required this.id,
    this.data = const {},
  });
}
```

3. The repository interface can be implemented in several ways, but for the sake of simplicity, we are just going to implement an *in-memory cache*. In the `repositories` folder, create a new file called `in_memory_cache.dart` and add the `InMemoryCache` class, which implements the `Repository` interface. This class will hold a private `Map` called `_storage` that will keep all the data:

```
import 'repository.dart';

class InMemoryCache implements Repository {
  final _storage = Map<int, Model>();
```

```
}
```

Once you've written the class declaration, you can use Android Studio/VS Code intentions dialog to automatically generate placeholders for all the required methods.

4. Now, we need to implement all the required functions from the repository interface. The most complex method is the `create` function, which needs to generate a unique identifier for every element in the storage:

```
@override
Model create() {
  final ids = _storage.keys.toList()..sort();
  final id = (ids.length == 0) ? 1 : ids.last + 1;

  final model = Model(id: id);
  _storage[id] = model;
  return model;
}
```

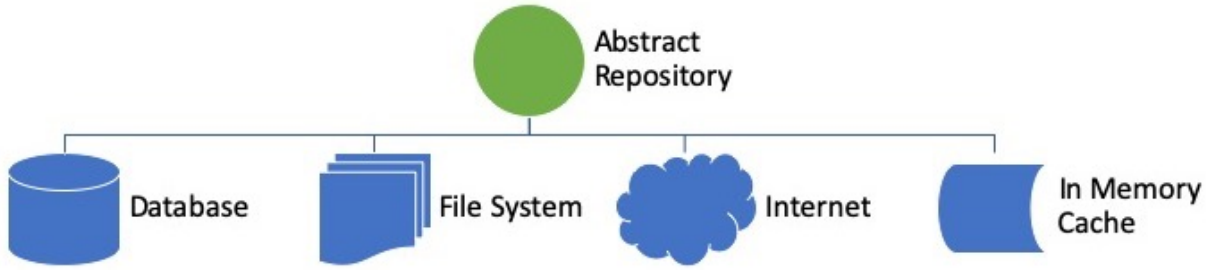
5. The remaining methods are simple wrappers of the `map` API. Write this code immediately after the `create` method:

```
@override
Model get(int id) {
  return _storage[id];
}

@override
List<Model> getAll() {
  return _storage.values.toList(growable: false);
}

@override
void update(Model item) {
  _storage[item.id] = item;
}

@override
void delete(Model item) {
  _storage.remove(item.id);
}

@override
void clear() {
  _storage.clear();
}
```

That completes the repository tier for this project. However, we won't be able to see it in action until we complete the next recipe on services.

How it works...

In this recipe, we chose to implement the repository tier as an **abstract** interface. Designing this sort of structure allows you to, in theory, have multiple implementations of the repository interface that can be swapped in and out effortlessly. The following diagram shows four of the most popular types of repositories that we could have configured:



In fact, this same abstract class could be used to store data in a database, in one or more files, or in a RESTful web service. In this recipe, we chose to implement the simplest option, `InMemoryCache`, which is nothing more than a fancy `Map`. The interesting thing about this class is that it represents a unified interface that can be used regardless of the actual destination of the data.

The higher layers in the n-tier architecture do not need to know *how* the data is stored, just that it is stored. Everything else, from their perspective, is an implementation detail.

Another requirement of the repository layer is that it cannot know anything about any domain-specific models that we have defined for our project, namely the `Plan` and `Task` models. Instead, we defined a transient `Model` class, which is just a map with an `id` property. The `id` property is used to retrieve the object from storage. It is critical that this value is unique; otherwise, your data will be overwritten.

We naively implemented a unique `id` formula in the `create` method:

```
final ids = _storage.keys.toList()..sort();
final id = (ids.length == 0) ? 1 : ids.last + 1;
```

Keys in `Maps` are not stored in any particular order. To find out what the largest value is, we need to sort all the keys in the map. Once we've retrieved the largest number, the function will just return that value incremented by one. This only works because the repository layer controls the `id` property. It should be a read-only value for any layer higher than the repository.

Designing an n-tier architecture, part 3 – services

The final tier that we will be talking about in this exploration of the n-tier architecture is the **services** tier. This tier serves as the glue between the controllers and repositories. Its primary job is to transform the data from the generic format used by the storage solution into the actual schema that is used by the controllers and the user interface.

In this recipe, we will be creating serialization and deserialization functions for our models, as well as stitching it all together with a services class.

How to do it...

This recipe will be divided into two components – serialization and integration. We're going to start with the serialization functions and then snap together all the pieces that we built over the last three recipes:

1. Open `task.dart` and add an `id` property and a default constructor. This will allow the `Task` model to be transformed into a generic `Model`:

```
import 'package:flutter/foundation.dart';
import '../repositories/repository.dart';

class Task {
  final int id;
  String description;
  bool complete;

  Task({@required this.id, this.complete = false, this.description
    = ''});
}
```

2. Now, we need to add the serialization and deserialization methods. These functions will take the data from a generic `Model` and return a more usable strongly typed object. Add the following code immediately after the constructor.

```
Task.fromModel(Model model)
 : id = model.id,
   description = model.data['description'],
   complete = model.data['complete'];


Model toModel() =>
```

```
    Model(id: id, data: {'description': description, 'complete':
complete});
```

3. Now, open `plan.dart` and add an immutable `id` property at the top of the class. We will also need to add the requisite constructor:

```
import 'package:flutter/foundation.dart';
import '../repositories/repository.dart';

final int id;
List<Task> tasks = [];
Plan({@required this.id, this.name = ''});
```

4. At the bottom of the class, add a deserialization constructor and a serialization method:

```
Plan.fromModel(Model model)
    : id = model.id,
      name = model?.data['name'],
      tasks = model?.data['task']
              ?.map<Task>((task) => Task.fromModel(task))
              ?.toList() ?? <Task>[];

Model toModel() => Model(id: id, data: {
      'name': name,
      'tasks': tasks.map((task) => task.toModel()).toList()
    });
```

5. Now, we can turn our attention to the services tier. As with all the other tiers in the n-tier architecture, we will need to create a folder. Create a new folder called `services` inside the `lib` folder.

6. Create a new file called `plan_services.dart`.
7. In the `plan_services.dart` file, instantiate a repository as a private property of the class:

```
import '../repositories/in_memory_cache.dart';
import '../repositories/repository.dart';
import '../models/data_layer.dart';

class PlanServices {
 final Repository _repository = InMemoryCache();
}
```

8. All the hard work of transforming `plans` has already been completed in the models. Now, we just need to expose the **Create, Read, Update, and Delete** (**CRUD**) methods in the services class. Add these four methods after the `repository` property:

```
Plan createPlan(String name) {
  final model = _repository.create();
  final plan = Plan.fromModel(model)..name = name;
  savePlan(plan);
  return plan;
}

void savePlan(Plan plan) {
  _repository.update(plan.toModel());
}

void delete(Plan plan) {
  _repository.delete(plan.toModel());
}

List<Plan> getAllPlans() {
  return _repository
      .getAll()
      .map<Plan>((model) => Plan.fromModel(model))
      .toList();
}
```

9. Integrating `Tasks` into the services layer is a bit easier. These objects are strongly coupled to their parent `Plan` objects. We just want to ensure that the tasks always have a unique `id`. Insert the following code after the `plan` CRUD methods:

```
void addTask(Plan plan, String description) {
  final id = plan.tasks.last?.id ?? 0 + 1;
  final task = Task(id: id, description: description);
  plan.tasks.add(task);
  savePlan(plan);
}

void deleteTask(Plan plan, Task task) {
  plan.tasks.remove(task);
  savePlan(plan);
}
```

10. Now, we need to tie the system into our existing functionality in `PlanController`. Remove the controller's private `_plan` property and replace it with an instance of the `PlanServices` class:

```
final services = PlanServices();
```

11. There will be few compile errors that need to be addressed. Go through all the red underlines and replace the code with the appropriate API in the services class. Start with the public `plan` getter and the `addNewPlan` method:

```
List<Plan> get plans => List.unmodifiable(services.getAllPlans());
void addNewPlan(String name) {
  if (name.isEmpty) {
    return;
  }

  name = _checkForDuplicates(plans.map((plan) => plan.name), name);
  services.createPlan(name);
}
```

12. The `savePlan` and `deletePlan` methods can also be updated to simply delegate their functionality to the services tier. Update this code right after the `addNewPlan` method:

```
void savePlan(Plan plan) {
  services.savePlan(plan);
}

void deletePlan(Plan plan) {
  services.delete(plan);
}
```

13. Wiring up the `Task` functionality is very similar to what we achieved for `Plans`. Simply replace the red errors that your IDE has highlighted with the appropriate method in the services class. The code that explicitly creates the task should be removed since that job is now being handled by a lower tier. To do this, update the `createNewTask` and `deleteTask` functions with the following code:

```
void createNewTask(Plan plan, [String description]) {
  if (description == null || description.isEmpty) {
    description = 'New Task';
  }

  description = _checkForDuplicates(
      plan.tasks.map((task) => task.description), description);

  services.addTask(plan, description);
}

void deleteTask(Plan plan, Task task) {
  services.deleteTask(plan, task);
}
```

14. There is one final change that needs to be made to the UI layer. When a user dismisses a `PlanScreen`, the data should be synchronized with our storage solution. Flutter has a widget called `WillPopScope` that allows you to run arbitrary code when routes are dismissed.
Open `plan_screen.dart` and wrap the screen's `Scaffold` in a new widget. Then, include the following closure:

```
return WillPopScope(
  onWillPop: () {
 final controller = PlanProvider.of(context);
 controller.savePlan(plan);
 return Future.value(true);
 },
  child: Scaffold(...)
);
```

This concludes our **MasterPlan** app. Perform a hot reload and take a moment to play with the app. This architecture will allow you to accomplish great things in the future!

How it works...

This recipe is mostly about piping. The `service` class pipes the data from the controller down to the repository. This is a relatively simple job, which means that the `service` classes should be simple to understand. There is no business logic in these classes and they are not responsible for maintaining state; it's just about unidirectionally transforming data.

When we define the class, it's also important to note that we're referencing the repository layer by its abstract interface:

```
final Repository _repository = InMemoryCache();
```

By explicitly declaring this property as a `Repository`, we are saying that it doesn't matter that we're using an `InMemoryCache`. This repository could just as easily be a database connector or an HTTP client and it wouldn't change much. This style of coding is known as coding toward an interface instead of an implementation. It is usually preferable to reference your properties in this way when you have many modular components that you want to be able to swap in and out easily.

The primary job of the services tier, as we described in the *Designing an n-tier architecture, part 1 – controllers* recipe, is to transform data. This process can be broken down into two categories:

- Serialization
- Deserialization

Serialization is defined as the process of taking your data and transforming it into a type that's more appropriate for transportation. This could be a byte stream, JSON, XML, or in the case of this recipe, a `Model`. Serialization methods are typically named by prefacing them with the

word ***to*** and then listing the type; for example, `toJson`, `toXml`, or `toModel`. We serialized the task model with the following function:

```
Model toModel() =>
    Model(id: id, data: {'description': description, 'complete':
     complete});
```

This function takes the content to `Task` and generates a `Map` with key-value pairs representing the content.

The opposite process, *deserialization*, takes the data coming **from** the transient structure and instantiates a strongly typed model. Deserialization methods are often implemented as constructors to make the API easier to work with:

```
Task.fromModel(Model model)
    : id = model.id,
      description = model.data['description'],
      complete = model.data['complete'];
```

When writing these methods, we have to make sure that the loosely-typed keys are identical in both the serialization and deserialization methods. If we make a typo or use the incorrect keys, then these functions would fail. Tasks only have two keys we need to worry about – `'description'` and `'complete'`. Even here, there is a large opportunity for errors. When writing these sort of functions, especially on more complex models, you need to carefully double-check your keys, since this is unfortunately not something the compiler can catch for you. The only way to notice that you have an error is to experience it at runtime.

There's more...

This recipe also used some Dart language features that you might be unfamiliar with – null-aware operators and null coalescing operators. Let's look more carefully at this line of code:

```
tasks = model?.data['task']
             ?.map<Task>((task) => Task.fromModel(task))
             ?.toList() ?? <Task>[];
```

When you insert null-aware operators, a `?` is inserted after any variable that might be `null`. Here, you are telling Dart that if this variable is null, then just skip everything after the question mark. In this example, the `model` property might be `null`, so if we tried to access the data property on a null, Dart would throw an exception. In this case, if a null is ever encountered, this code will be gracefully skipped.

The null coalescing operator, `??`, is used almost like a ternary statement. This operator will return the value on the right-hand side of the question marks if the value on the left-hand side is null. You can think of null coalescing operators as a short form for this kind of statement:

```
String nothing;
String something = nothing == null ? 'Something' : nothing
```

Null coalescing operators are often used as a fullback in combination with null-aware operators. This guarantees that you will have a value and not a null at the end of your statement.