Add a service class

You've created a model, a view, and a controller. Before you use the model and view in the controller, you also need to write code that will get the user's to-do items from a database.

You could write this database code directly in the controller, but it's a better practice to keep your code separate. Why? In a big, real-world application, you'll have to juggle many concerns:

- **Rendering views** and handling incoming data: this is what your controller already does.
- **Performing business logic**, or code and logic that's related to the purpose and "business" of your application. In a to-do list application, business logic means decisions like setting a default due date on new tasks, or only displaying tasks that are incomplete. Other examples of business logic include calculating a total cost based on product prices and tax rates, or checking whether a player has enough points to level up in a game.
- **Saving and retrieving** items from a database.

Again, it's possible to do all of these things in a single, massive controller, but that quickly becomes too hard to manage and test. Instead, it's common to see applications split up into two, three, or more "layers" or tiers that each handle one (and only one) concern. This helps keep the controllers as simple as possible, and makes it easier to test and change the business logic and database code later.

Separating your application this way is sometimes called a **multi-tier** or **n-tier architecture**. In some cases, the tiers (layers) are isolated in completely separate projects, but other times it just refers to how the

classes are organized and used. The important thing is thinking about how to split your application into manageable pieces, and avoid having controllers or bloated classes that try to do everything.

For this project, you'll use two application layers: a **presentation layer** made up of the controllers and views that interact with the user, and a **service layer** that contains business logic and database code. The presentation layer already exists, so the next step is to build a service that handles to-do business logic and saves to-do items to a database.

> Most larger projects use a 3-tier architecture: a presentation layer, a service logic layer, and a data repository layer. A **repository** is a class that's only focused on database code (no business logic). In this application, you'll combine these into a single service layer for simplicity, but feel free to experiment with different ways of architecting the code.

Create an interface

The C# language includes the concept of **interfaces**, where the definition of an object's methods and properties is separate from the class that actually contains the code for those methods and properties. Interfaces make it easy to keep your classes decoupled and easy to test, as you'll see here (and later in the *Automated testing* chapter). You'll use an interface to represent the service that can interact with to-do items in the database.

By convention, interfaces are prefixed with "I". Create a new file in the Services directory:

Services/ITodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;
```