As you can see, it's very, very easy to apply CQRS by using these two classes. As it turned out, I was using CQRS years before I even knew what this thing was. So, I don't push the idea of CQRS; I like working with actions and view models. Also, I don't think it's necessary for every application, but it gives you some excellent benefits:

- Your codebase becomes more understandable.
- You have small, easier-to-maintain classes.
- It gives you a perfect separation of concern.
- Each class has one well-defined responsibility.

So that's what CQRS is all about. However, you'll find some very hard-to-understand articles and tutorials if you search for them. They often show you how CQRS is used with other more complicated concepts such as event sourcing, event stores, and separate read and write databases.

In this book, I won't talk about event sourcing for three reasons:

- I'm not using it, so I'm not authentic to write about it.
- I don't think it's necessary for most business applications.
- It requires a whole different architecture and perspective.

So it's doubtful that you will start to refactor your application to event sourcing after learning about it. However, it's much simpler to adapt any other concepts we'll use in this book.

CQRS can be a bit more complex (and usually is) than I described it. For example, in the C# world, developers use a so-called mediator. It's a way to implement in-process synchronous messaging. Basically, it's a layer that handles commands and queries. If you want to learn more about it, check out this package.

Also, CQRS is a very different (and complex) animal in the microservices world. It means you have separate services and databases for reading and writing. In this context, it's almost required to use CQRS and event sourcing and event-driven async architecture together. If you're interested in this topic, check out this video (you can watch it without a master's degree in CS).