Kinds of Models

In Model-View-Controller Architectures

Share

In a Model-View-Controller (MVC) application, the Model is responsible for the application's state and non-UI specific behavior. In simple applications, there may be just one kind of model class that is used by persistence, presentation, and any business logic. However, frequently this kind of one-size-fits-all approach doesn't scale to complex applications. In that case, it may make sense to have different model types (classes) with different responsibilities. Some MVC applications will include some combination of the following types of model objects.

Domain Model

Many developers choose to encapsulate complex business logic within a *domain model*, which doesn't depend on infrastructure concerns and is easily validated with unit tests. The domain model will often include abstractions and services that allow the Controller to operate at a higher level of abstraction, and keep low-level plumbing code from cluttering the Controller and making it harder to test. The domain model will usually include interface definitions (for services, repositories, etc.) used by the app, as well as persistence-ignorant (/persistence-ignorance) entities (and some services) that represent the state and behavior of the app's business logic.

View Model

Many developers are familiar with the concept of a ViewModel, especially those who have used application frameworks that use the Model-View-ViewModel pattern. In an MVC web application, a ViewModel is a type that includes just the data a View requires for display (and perhaps sending back to the server). ViewModel types can also simplify model binding in ASP.NET MVC (http://docs.asp.net). ViewModel types are generally just data containers; any logic they may have should be specific to helping the View render data. There may be many similar ViewModel types, each tailored to the needs of a particular View.

Binding Model

Sometimes it may be worthwhile to create a type specifically for use with model binding (https://docs.asp.net/en/latest/mvc/models/model-binding.html). These types are typically just data containers with no behavior. Using a binding model can help avoid certain security issues with ASP.NET MVC model binding, by avoiding an issue where users can bind to properties of the model that are not present on the form being posted (http://codetunnel.io/aspnet-mvc-model-binding-security/).

API Model

If your application exposes an API, the format of the data you expose to clients may be separated from your app's internal domain model by defining custom API model types. This allows you to change your internal model types without impacting clients that may be using your exposed APIs. Typically these exposed API models will be used by clients both for read and write operations, so these types will act as both ViewModel and BindingModel for APIs.

Persistence Model

Some applications have separate classes that map closely to how data is stored and retrieved from persistence, often because a tool generates the classes based on a database schema. It may make sense to add business logic to these classes and use them as the domain model, but in some cases the app can benefit from a domain model that is separate from this persistence model. In such scenarios, the classes responsible for persistence (for example, repositories) would be responsible for mapping to and from the different models.

If you're using Entity Framework (https://ef.readthedocs.io/en/latest/), it can generally map your domain model entities directly to persistence without the need for a separate persistence model.

Get Started with ASP.NET Core 2 Taught by Steve Smith (@ardalis) Watch Now!

Live Search

Find help! Enter search term here. Search

Tags

agile (https://deviq.com/tag/agile/) antipattern (https://deviq.com/tag/antipattern/) code smell (https://deviq.com/tag/code-smell/) ddd (https://deviq.com/tag/ddd/) oop (https://deviq.com/tag/oop/) pattern (https://deviq.com/tag/pattern/) practice (https://deviq.com/tag/practice/) principle (https://deviq.com/tag/principle/) tool (https://deviq.com/tag/tool/) value (https://deviq.com/tag/value/) xp (https://deviq.com/tag/xp/)

Popular Posts

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- Courage (https://deviq.com/courage/)
- Feedback (https://deviq.com/feedback/)
- Communication (https://deviq.com/communication/)
- Simplicity (https://deviq.com/simplicity/)

Recent Posts

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- State Design Pattern (https://deviq.com/state-design-pattern/)
- Guard Clause (https://deviq.com/guard-clause/)
- Bounded Context (https://deviq.com/bounded-context/)
- Specification Pattern (https://deviq.com/specification-pattern/)

RECENT ARTICLES

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- State Design Pattern (https://deviq.com/state-design-pattern/)
- Guard Clause (https://deviq.com/guard-clause/)
- Bounded Context (https://deviq.com/bounded-context/)
- Specification Pattern (https://deviq.com/specification-pattern/)
- Kinds of Models (https://deviq.com/kinds-of-models/)
- Singleton (https://deviq.com/singleton/)
- Antipatterns (https://deviq.com/antipatterns/)
- Design Patterns (https://deviq.com/design-patterns/)

POPULAR ARTICLES

- Builder Design Pattern (https://deviq.com/builder-design-pattern/)
- Hollywood Principle (https://deviq.com/hollywood-principle/)
- Keep It Simple (https://deviq.com/keep-it-simple/)
- YAGNI (https://deviq.com/yagni/)
- Boy Scout Rule (https://deviq.com/boy-scout-rule/)
- Simplicity (https://deviq.com/simplicity/)
- Communication (https://deviq.com/communication/)
- Feedback (https://deviq.com/feedback/)
- Courage (https://deviq.com/courage/)

(https://www.facebook.com/DevIQPage) (https://twitter.com/deviq)