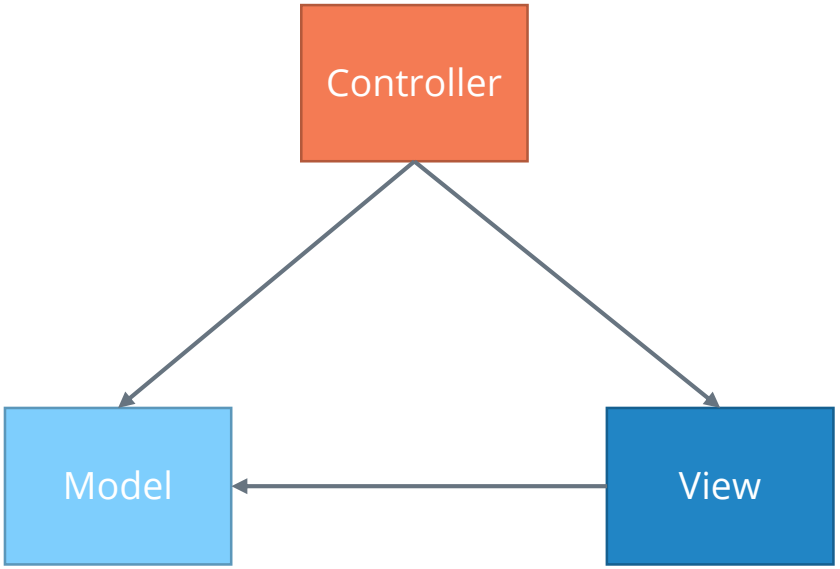

Headspring
Introduction to Models

What is a model?



Domain Model

```
namespace ContactList.Core.Domain
{
    public abstract class Entity
    {
        public Guid Id { get; set; }
    }
}
```

```
namespace ContactList.Core.Domain
{
    public class Contact : Entity
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public string PhoneNumber { get; set; }
    }
}
```

View Model

```
namespace ContactList.Features.Contacts
{
    public class ContactViewModel
    {
        public Guid Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public string PhoneNumber { get; set; }
    }
}
```

View Model

```
namespace ContactList.Features.Contacts
{
    public class AddContactForm
    {
        [Required]
        [Display(Name = "First Name")]
        public string FirstName { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Required]
        public string Email { get; set; }

        [Required]
        [Display(Name = "Phone Number")]
        public string PhoneNumber { get; set; }
    }
}
```

Building the view model

```
public ActionResult Index()
{
    var model = Database.Contacts
        .OrderBy(x => x.LastName)
        .ThenBy(x => x.FirstName)
        .Select(x => new ContactViewModel
        {
            Id = x.Id,
            FirstName = x.FirstName,
            LastName = x.LastName,
            Email = x.Email,
            PhoneNumber = x.PhoneNumber
        })
        .ToArray();

    return View(model);
}
```

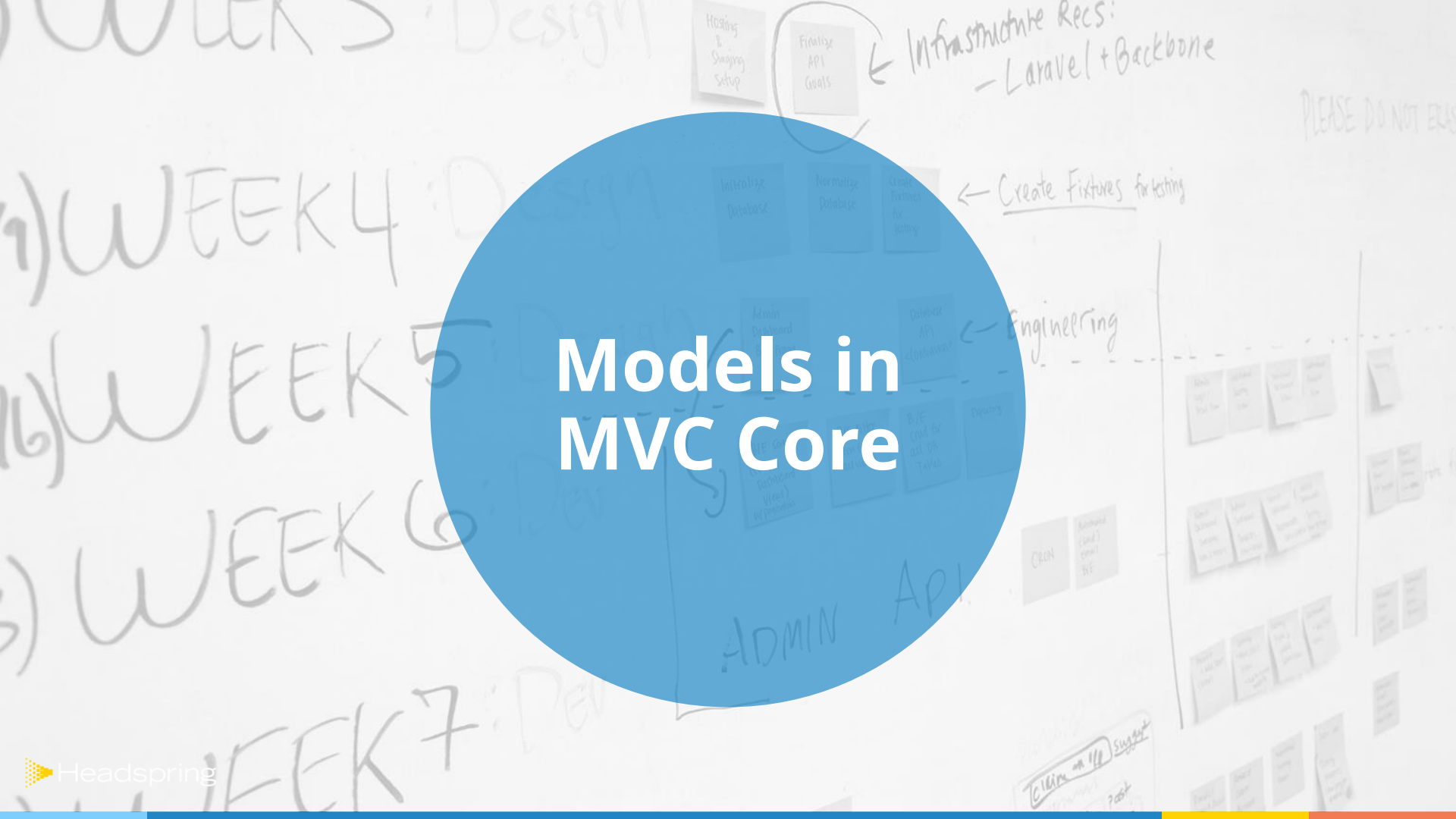
Use Dedicated View Models

- It is tempting to use your domain models as your view models, but you will quickly regret it.
  - UI concerns like validation, user-facing label text, and properties that exist only for presentation details would “pollute” your domain model.
  - Even the Add and Edit model for a single Domain class will differ from each other!

Avoid Non-Nullable Value Types

- View Model properties should generally not be non-nullable value types.
  - Prefer DateTime? over DateTime, or else you'll find your form defaulting to 1/1/0001.
  - Yet another reason to not use your Domain Model types for your View Model.

Models in MVC Core



View model in Controller: ViewDataDictionary

- Dictionary access
- Additional Model object
- Contains additional metadata/state information

```
public ActionResult Index()
{
    ViewData["Message"] = "Welcome to ASP.NET MVC!";

    ViewData.Model = "Hello world";

    return View();

}
```

ViewBag

- Dynamic data type
- Provides property-style syntax
- Still using ViewDataDictionary underneath

```
public ActionResult Index()
{
    ViewData["Message"] = "These do the same thing.";

    ViewBag.Message = "These do the same thing.";

    return View();
}
```

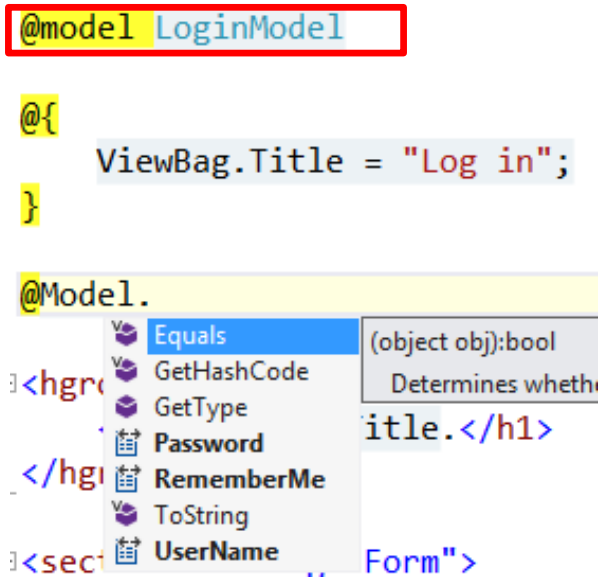
View model in view - dictionary

- Available in ViewData as dictionary
- Or ViewBag as properties

```
<hgroup class="title">
    <h1>@ViewBag.Title.</h1>
    <h2>@ViewData["Message"]</h2>
</hgroup>
```

View model in view - model

- Available in Model member (façade over ViewData.Model)



Actions and View Models

Validation

```
[HttpPost]
public ActionResult Add(AddContactForm form)
{
    if (ModelState.IsValid)
    {
        var contact = new Contact
        {
            FirstName = form.FirstName,
            LastName = form.LastName,
            Email = form.Email,
```

Typical GET Actions

- Load domain model objects from the database.
- Map domain model to view model.
- Pass that view model to the view.

Typical POST Actions

```
if (ModelState.IsValid)
{
        //Do the work (update database, etc)
        //Redirect to some other page.
}

//Rehydrate view model, if necessary.

return View();
```

GET/POST Pairs

- GET and POST actions often appear in pairs.
  - GET shows the form
  - POST handles the form submission
- Name these methods the same as each other, and place them beside each other in the controller. They are a matched set.
- Use [HttpPost] to distinguish them, or routing will get confused.

One Model In

- Technically, action methods can have multiple arguments. In this case, each parameter name gets matched to incoming form inputs.
- We strongly advise against doing so:
  - Refactor-unfriendly.
  - Dedicated types per POST action give our more advanced tools something solid to grab on to.

Keep Action Methods Small

- It’s too hard to have automated tests of the UI.
- Actions are a part of the UI.
- So, put as little in them as possible.
  - Defer to some other, more test-friendly helper classes: Command and Query objects.
- If your actions are more than 3-5 statements, something is wrong.