

Building Large, Yet Maintainable, ASP.NET Applications

Scott Sauber

Who am I?

- Lead Software Developer at Iowa Bankers Association
- Work in Mortgage and Insurance Industry
 - We're not Facebook, Netflix, Stack Overflow, etc.
- Highly Regulated
 - Lots of validation
 - Lots of changes
- Twitter: @scottsauer
- Blog (primarily ASP.NET Core): scottsauer.com

Audience

- Existing ASP.NET Developers
- Building “Large” applications
 - Large to me, means a combination of these:
 - Many Views
 - Many Controllers
 - Lots of business logic
 - High business impact
 - Frequent Change
 - Large != High Traffic
- Some of these recommendations only make sense for big apps that change frequently
 - Introduce some minor complexity, but improves overall maintainability
 - That trade off is only worth it in large apps that change a lot

Agenda

- Lightning Talk approach
- Talk about best practices for all these *in a “large” app*
 - Folder Structure
 - UI Components
 - View Models
 - Code Flow
 - Validation
 - ORM's
 - Dependency Injection
 - Unit Testing and Assertions
 - DevOps
 - Microservices

Audience Goals

- “New” perspective on a few topics
- You’re not going to agree on everything
- Take away some ideas to evaluate in your future workflow tomorrow

Overall theme

- Improve Maintainability
 - Better practices
 - Consistency
 - Enjoyable to work with
- The goal is to create systems that allow developers to make changes as easy, quick, and bug-free as possible, while allowing that trend to continue into the future.
- “Legacy software is software you have no confidence in.”



Guillermo Rauch ✓

@rauchg

 Follow



Every system tends towards complexity,
slowness and difficulty
Staying simple, fast and easy-to-use is a battle
that must be fought everyday

RETWEETS
613

LIKES
1,005



5:39 PM - 26 Dec 2016 from San Diego, CA

 16

 613

 1.0K



Michael Feathers

@mfeathers

Follow



Make something simple and add complexity reluctantly.

9:08 PM - 6 Jul 2017

86 Retweets 137 Likes



3



86



137



Folder Structure Best Practices



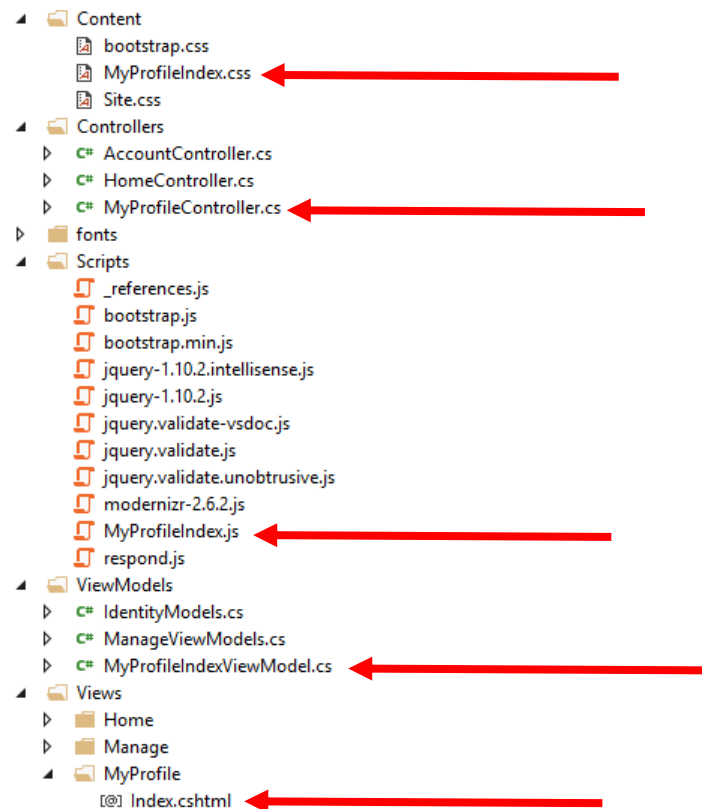
Problem: OOB MVC Folders By Responsibility

- All of these live in their own separate folders and most are required to add a new feature
 - Controllers
 - Views
 - Scripts
 - Content
 - Models
- Adds navigation friction
- Scope of a feature is scattered
- Makes it hard to add, delete or extend existing features

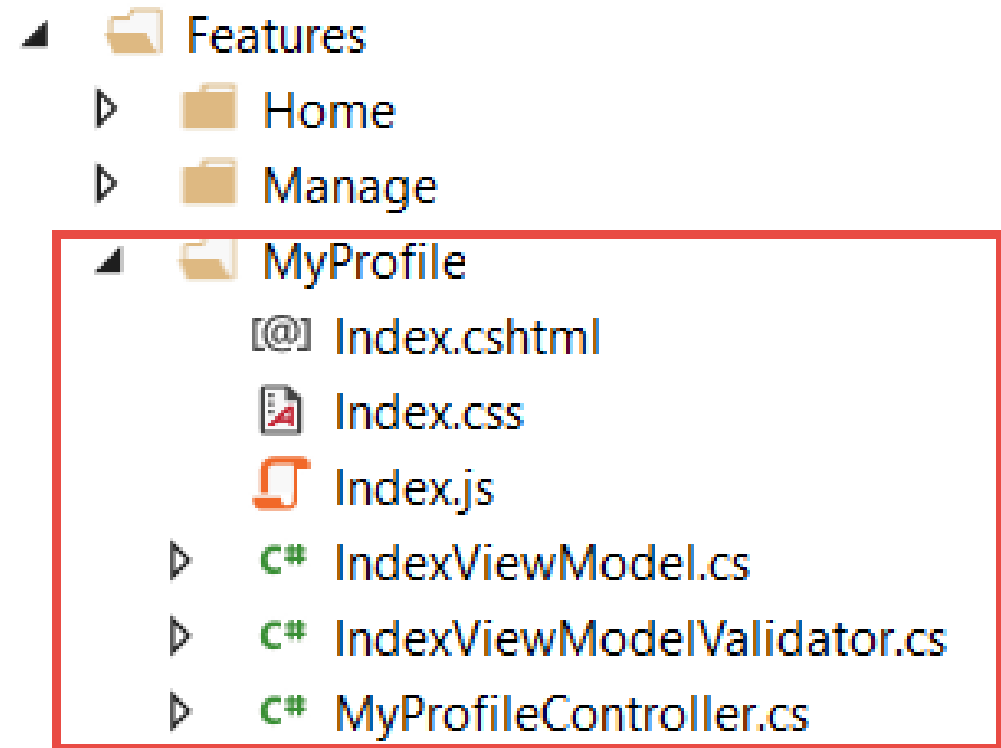
Solution: Use Feature Folders

- Grouping by Feature, not by Responsibility, results in easier maintenance
- Related things remain together (High Cohesion)

MVC out of the box:



Feature Folders:



Feature Folder Extra Resources

- How to do this in ASP.NET 4.x
 - [Tim Thomas' blog post](#)
- How to do this in ASP.NET Core
 - [My blog post](#)
 - [Steve Smith's Blog on Feature Folders vs. Areas](#)
- Also used in React, Angular, etc.
 - [John Papa](#)

The “M” in MVC



Problem: ViewBag sucks and Entities = Security Risk

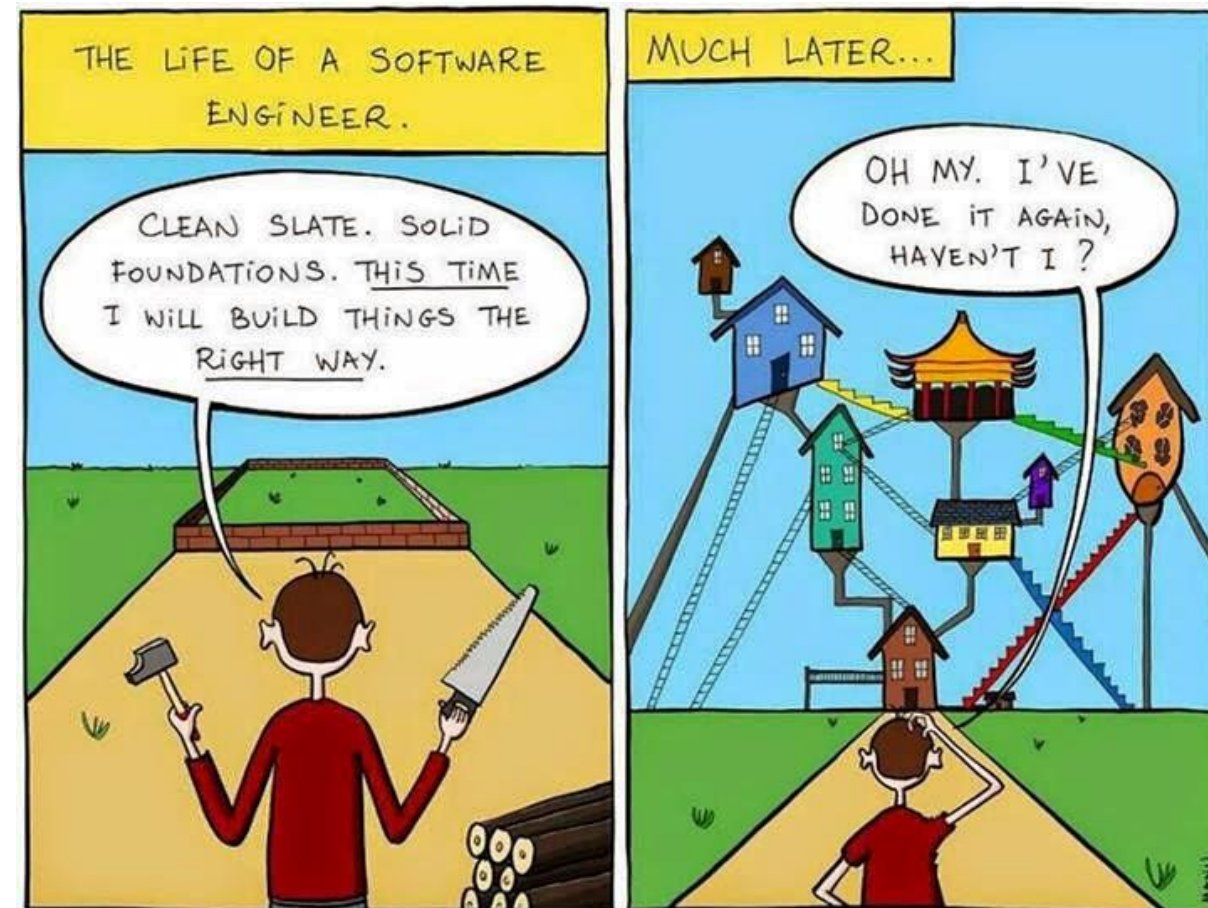
- View Bag - Nope
 - Runtime errors
 - No intellisense
- Don't use raw Entities
 - [Overposting](#) AKA Mass Assignment attacks
 - [This happens](#)
 - Also just a matter of time before “I need an Entity... and 1 more thing”

Solution: Use ViewModels

- Make a View Model for all views
 - Consistency, no surprises
- Strongly Typed



UI Composition



Make as many UI Components as you can

- Benefits
 - SRP for the UI
 - Easier to reason about
 - Reusability
 - Eliminates div soup
- Component architecture popularity is rising
 - React, Angular (2/4/whatever), AngularJS 1.5+, Knockout 3.2+
- Partial
- Child Actions (in ASP.NET 4)
- View Components (in ASP.NET Core)



50,000 BONUS POINTS

Plus 2X Points on Travel & Dining

Learn more

Welcome back

Username

Password

☐ Remember me Use token >

Sign in

Forgot username/password? >

Not enrolled? Sign up now. >

Choose what's right for you



Checking Accounts



Free credit score



Find a credit card



Home Lending



Buy a car



Making a Home Change?

Start a relationship with a home lending expert to personalize a plan that's right for you.

Learn more



Earn up to \$500 per year

Invite friends to get Chase Freedom®. You'll get \$100 cash back for each friend who gets the card – up to \$500 cash back per year.

Invite friends



See your score for free

NEW! Get your credit score for free, powered by TransUnion®. It's quick and easy.

Get your score

Code Flow and Smells



Structuring a method

- Happy Path always at the bottom of the method
 - Don't want to scan for “what happens when all goes well” and find it in the middle of a method
- Use return's instead of nested if => else

MVC 5 Template Code:

```
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);

            // Send an email with this link
            string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
            await UserManager.SendEmailAsync(user.Id, "Confirm your account", "Please confirm your account by clicking <a href=\"" + callbackUrl + "\">here</a>");

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Refactored with Happy Path at the bottom:

```
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (!ModelState.IsValid)
        return View(model);

    var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
    var result = await UserManager.CreateAsync(user, model.Password);

    if (!result.Succeeded)
    {
        AddErrors(result);
        return View(model);
    }

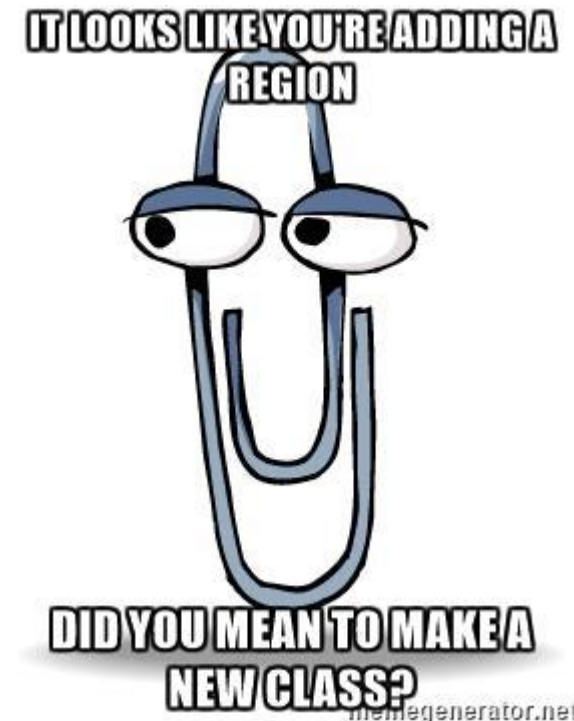
    await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);

    // Send an email with this link
    string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
    var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
    await UserManager.SendEmailAsync(user.Id, "Confirm your account", "Please confirm your account by clicking <a href=\"" + callbackUrl + "\">here</a>");

    return RedirectToAction("Index", "Home");
}
```

Code smells

- Methods > 30 lines
- Classes > 200 lines
- Anytime you scroll down, up, down to find what you're looking for
- Regions
 - You probably should've added a new class or method instead



Validation



Validation – What's wrong with OOB Options

- Data Annotations
 - Out of the box annotations only work well for simple scenarios
 - Hard to make custom ones
 - Hard to unit test
 - Separate annotations for each property
 - Heavy, lots of annotations
 - Can get “tall”
 - SRP violated
 - Model + Validation combined into one class
- Writing own Custom Validation classes
 - Lose client-side hooks Data Annotations provides
- Validation in Controller Action
 - Hard to maintain and test
 - Bloated Controllers

Solution: Use FluentValidation

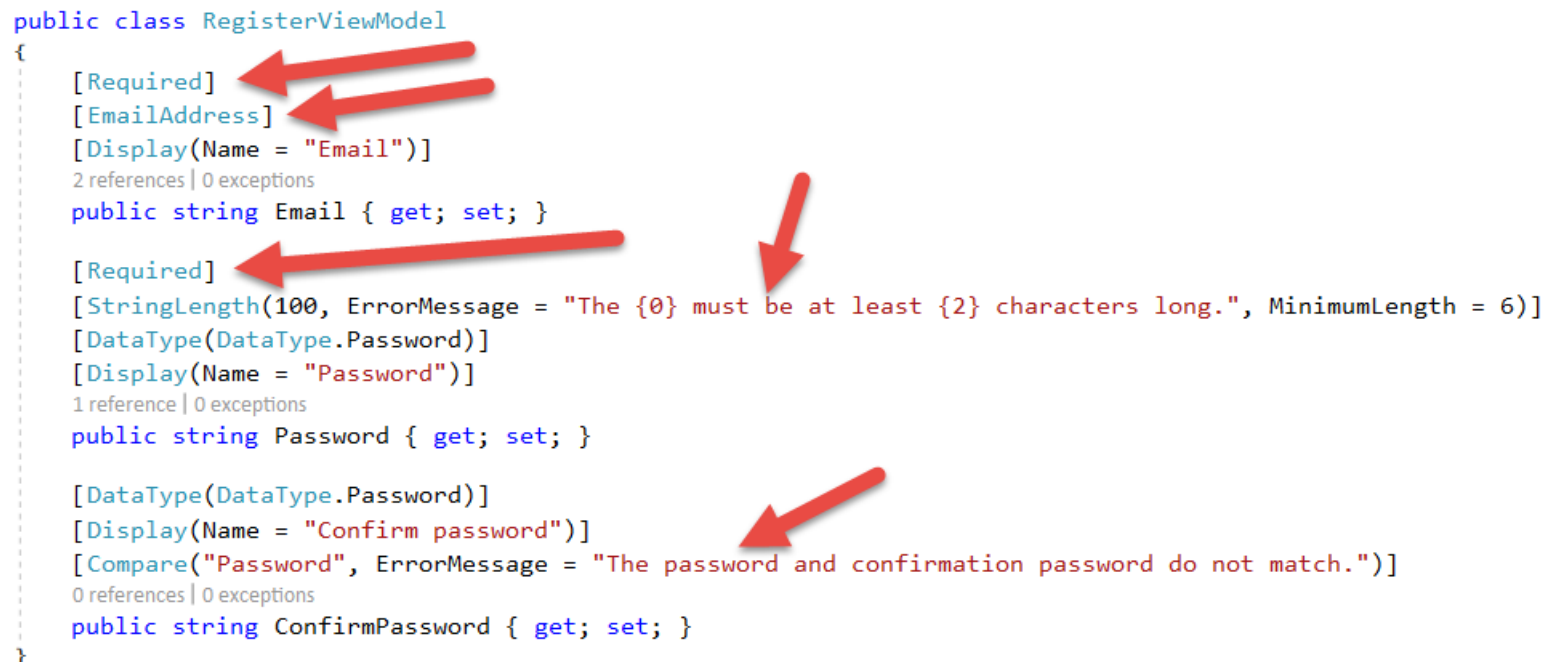
- Fluent interface
- Business rules are easy to maintain and read
- Easy to show a stakeholder
- Easy to test
- Integrates with ModelState.IsValid
- Same Client-Side validation as Data Annotations
- 3.1M downloads
- <https://github.com/JeremySkinner/FluentValidation>

MVC 5 Template Code:

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    2 references | 0 exceptions
    public string Email { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    1 reference | 0 exceptions
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    0 references | 0 exceptions
    public string ConfirmPassword { get; set; }
}
```



Refactored with Fluent Validation:

```
public class RegisterViewModel
{
    [Display(Name = "Email")]
    4 references | 0 exceptions
    public string Email { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    5 references | 0 exceptions
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    1 reference | 0 exceptions
    public string ConfirmPassword { get; set; }
}
```

```
public class RegisterViewModelValidator : AbstractValidator<RegisterViewModel>
{
    0 references | 0 exceptions
    public RegisterViewModelValidator()
    {
        RuleFor(m => m.Email).NotEmpty()
            .WithMessage("Email is required.");

        RuleFor(m => m.Email).EmailAddress()
            .WithMessage("Email must be a valid email address.");

        RuleFor(m => m.Password).NotEmpty()
            .WithMessage("Password is required.");

        RuleFor(m => m.Password).MaxLength(100)
            .WithMessage("The password cannot be longer than 100 characters.");

        RuleFor(m => m.Password).MinLength(6)
            .WithMessage("The password must be at least 6 characters long");

        RuleFor(m => m.ConfirmPassword).Equal(m => m.Password)
            .WithMessage("The password and confirmation password do not match.");
    }
}
```

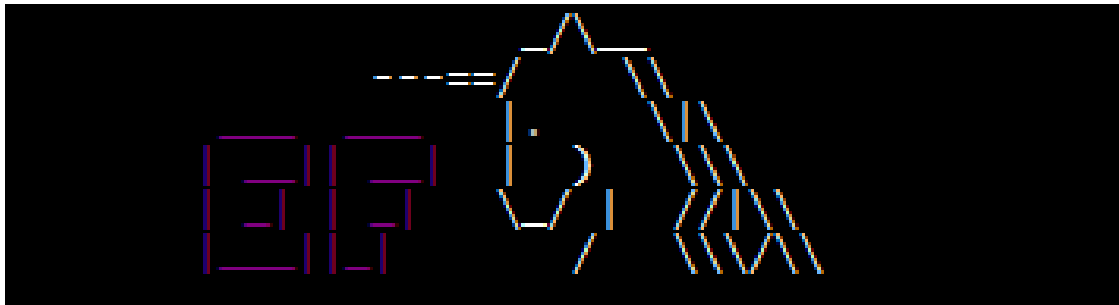
ORM's



Don't use raw ADO - Use an ORM

Entity Framework – Full ORM

- Pros
 - Developer Productivity
 - Compile-time safety with LINQ Queries
 - Extremely quick to add new CRUD operations
 - Built in Unit of Work
 - Migration support
- Cons
 - Less performant
 - Less control over the queries generated
 - Heavier



Dapper – Micro ORM

- Pros
 - Performance near ADO
 - More control over the queries
 - Extremely simple to setup
 - Stack Overflow beta tests
- Cons
 - SQL strings = Big column name refactorings are harder
 - Less features than EF



ORM usage comparison

Entity Framework:

```
public List<Customer> GetCustomersByState(string state)
{
    var context = new ApplicationDbContext();
    return context.Customers.Where(c => c.State == state).ToList();
}
```

Dapper:

```
<typeparamref name="T"/> GetCustomersByState
public List<Customer> GetCustomersByState (string state)
{
    var connection = new SqlConnection();

    string sql = @"SELECT *
                  FROM Customers
                  WHERE State = @State";

    var parameters = new
    {
        State = state
    };

    return connection.Query<Customer>(sql, parameters).ToList();
}
```

Other ORM things

- Both manage connection lifetimes
- Both give you SQL Injection protection
- We use both depending on the job
- Common for us to start with EF and supplement with Dapper
 - EF for Developer Productivity
 - Dapper for hot paths
 - Dapper for queries that are tough to write in LINQ
- DON'T WRITE YOUR OWN ORM

Performance

1st RUN				
EF 6.1.3	Dapper 1.4.2		ADO.NET	
Read	359	103		99
Write	1559	260		218
2nd RUN				
EF 6.1.3	Dapper 1.4.2		ADO.NET	
Read	359	88		79
Write	755	184		213

[Source](#)

Dependency Injection



Dependency Injection and IoC containers

- DI/IoC helps you loosely couple your “large” apps
- Lets you unit test anything if done correctly
- I use SimpleInjector
 - [Performance](#)
 - ~80x faster than Ninject, ~20x faster than Structure Map, ~30x faster than Autofac
 - Crazy good docs
 - Verify step has saved me more than once
 - 1.5M downloads



Common DI Pitfalls

- [“New Is Glue”](#)
- [“Static Cling”](#)
- `DateTime.Now` (and variants)
 - Instead inject in `IClock` or have method take in `DateTime?` and default to `Now`
- `HttpContext`
 - `IHttpContextAccessor` in ASP.NET Core
- `HttpContext.Current.User.Identity`
 - `.GetUserId()`
 - `.Name`
 - Instead, inject in your user (per request!)
- `ConfigurationManager`
 - Instead, inject a POCO Settings class

Unit Testing



Unit Testing with xUnit

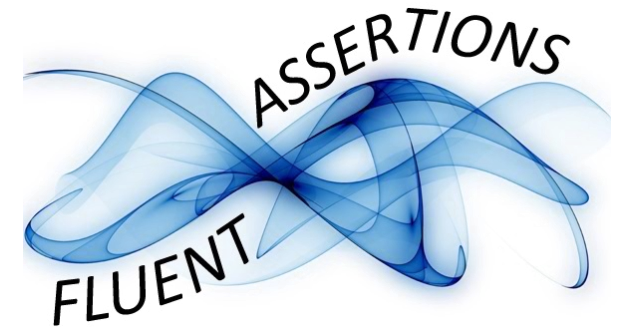
- You should be writing automated tests
 - Exposes holes in your architecture
 - Can be faster in short term than manual testing
 - Proven to be faster long-term
 - Make changes quickly and confidently because have a regression test suite
- Use xUnit or NUnit
 - Just not MSTest which is wayyy more verbose and has less features
- xUnit used by ASP.NET team
- We used to use NUnit and switched to xUnit
- NUnit more boilerplate

Problem: OOB Assertion Libraries Annoy Me

- `Assert.Equal(value, value)`
 - Hard to remember that it's `Assert.Equal(expected, actual)`
 - Can lead to funky looking assertion failures if you flip them
- No “Because” string in xUnit’s assertions

Solution: Use FluentAssertions for assertions

- Adds a Should() extension method to object
 - `result.Should().Be(0);`
 - Easier to read than `Assert.Equal(0, result);`
- `Should().Be()` is primary use, but other methods exist
 - `someBool.Should().BeTrue()` or `BeFalse()`
 - `someString.Should().BeNull()` or `NotBeNull()`
- Because string to explain why
 - `can12YearOldDriveResult.Should().BeFalse("because you must be 16 years old to drive.")`
- 4.3M downloads



DevOps



The Expert Beginner

@ExpertBeginner1

Follow



The most important thing you can do is have well-defined handoff procedures between Dev and DevOps and between DevOps and Ops.

9:24 AM - 14 Sep 2017

43 Retweets 54 Likes



5



43



54

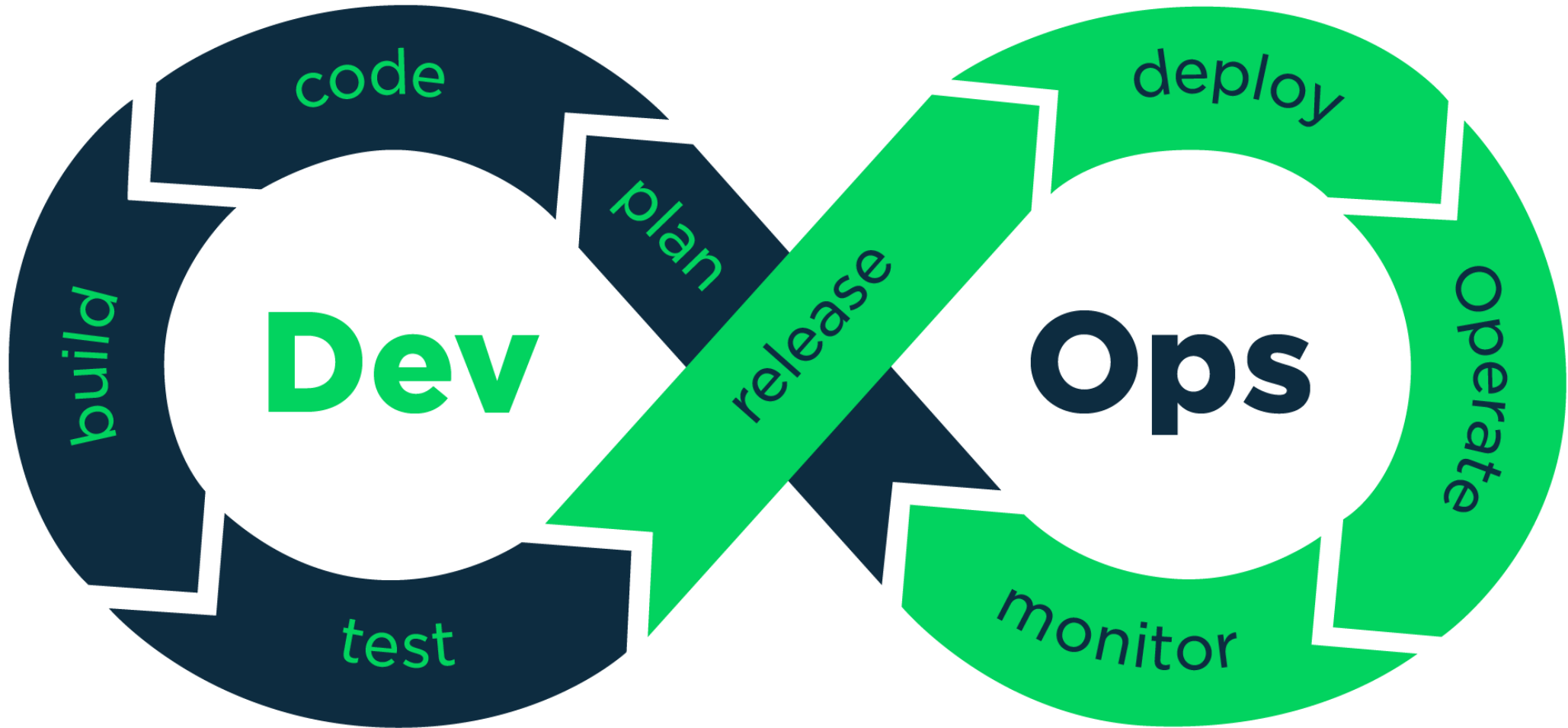


What is DevOps

“DevOps is the union of people, process, and products to enable continuous delivery of value to our end users.”

- Donovan Brown

DevOps Pipeline



Continuous Integration and Deployment

- Continuous Integration
 - Automated builds running on an independent build server
 - Automatically running automated tests
 - Creates an output
- Continuous Deployment
 - Takes the output from the build server and deploys it
 - Go through different environments
 - Automated configuration of VM, IIS, folder permissions, etc.
- These two together will be transformational for you and your company

Continuous Integration with TeamCity

- Free for up to 20 build configs (projects)
- NuGet Restores
- MSBuild
 - Don't need to install VS on your build server
- Run automated tests
- Produces an output
- Zips up and sends to Octopus Deploy



Continuous Deployment with Octopus Deploy

- Lets you promote your package through different environments
- Controls all web.config settings, scoped to each environment/role
- Auto-creates IIS Site and App Pool for you
- Deploys and rollbacks are click of a button
- Lets you run any scripts you want to as part of your deployment
- Easy to install and easy to use
- Professional – 20 users, 20 projects, 20 machines \$700 one time and 50% maintenance per year
- Full Audit Trail
 - Who/What/When changed
- My single favorite piece of software – ever



TeamCity + Octopus Deploy enables...

- Consistency
 - Machine repeats the same steps over and over again
- Consistency promotes Confidence
 - Passes build
 - Passes automated tests
 - Configuration is correct
 - Let's deploy it to Prod!
- Confidence enables agility
 - Mid-day publishes are no big deal
 - For a single app, our record is 24 publishes to Prod in 1 day with 2 developers

Feature Toggles

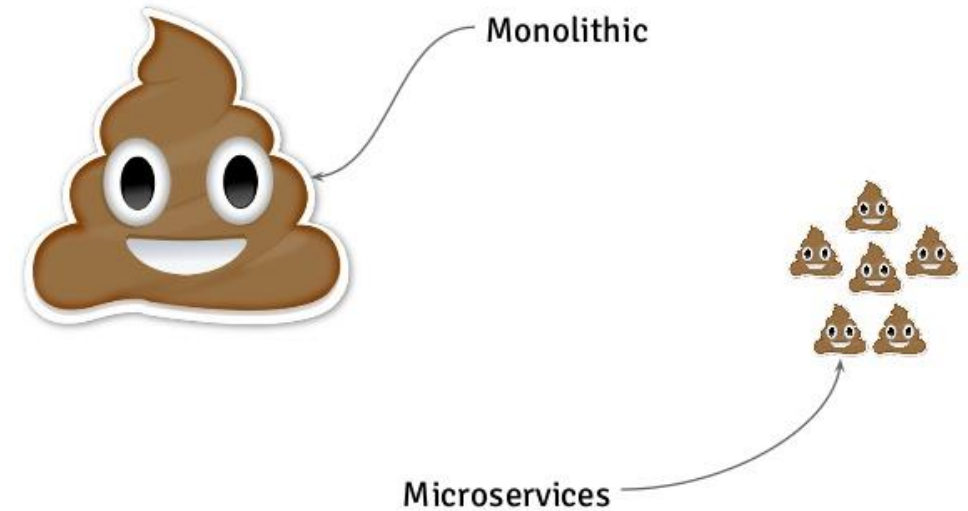


Feature Toggles

- No long lived branches
- Reduce merge conflicts
- Commit to master instead!
 - Behind an if statement
- Simplest way to toggle on and off: let Octopus Deploy handle toggling your feature on or off per Environment.
- Commit to cleaning up after it's live
- Clearly separate out your Feature Toggles to make it obvious what Feature Toggles exist
 - Have a FeatureToggles class or section in your config file

Microservices

Monolithic vs Microservices



<Something about Microservices />

- When I say Microservices I mean some out of process call of some sort (web service, queue, etc.)
- Microservices add operational complexity
 - Health Checks
 - Is it running? Are the dependencies ok?
 - Load Balancing
 - Service Discovery (via Consul or something simple like config files with primary/backup)
- Microservices add development complexity
 - HTTP call (most likely) instead of a method call
 - If calling from C#, likely want a client library wrapper around HttpClient sending JSON
 - Dev environment needs multiple projects running

<Something about Microservices />

- Benefits
 - Iterate on microservices separate from other app (SRP)
 - Reuse
 - Scale independently
- We use in moderation
 - Emails, Texts, CPU/Latency Intensive ops (Order Floods, Pull Credit), Uploads, Retrieving Files, etc.
 - I swear this is in moderation
- [You are not Google](#)/Facebook/Netflix/Amazon

Real benefits of these practices

- 1 web app went from deploying to Prod 20x a year to deploying to Prod over 500x a year.
- Faster delivery of value to users.

Summary

- Feature Folders > MVC Folders
- View Model All The Things
- SRP for UI with Components
- Happy Path at the Bottom of a Method
- FluentValidation over Data Annotations/Custom
- Use an ORM
- Avoid new, Statics, HttpContext, ConfigurationManager for testing
- Automate your Build and Release pipeline
- Resist Microservices until there's a reason not to



Closing

- Hope you got at least one idea out of this
- Remember – point is to deliver value to your end users as quickly and reliably as possible and allowing that trend to continue in the future.
- Don't get caught up in what library you're using
 - If you use NUnit instead of xUnit and have no reason to switch – who cares?
 - Important thing is you're doing automated testing.

Questions?

- Feel free to reach out on Twitter (@scottsauber) if you think of a question later
- Slides posted on my blog (scottsauber.com) and I'll tweet them out
- Don't forget to fill out evals, please

Thanks for coming!