

# Unit Testing ASP.NET Core MVC Controllers

---



**Kevin Dockx**

Architect

@KevinDockx <https://www.kevindockx.com>



# Coming Up



**Code coverage and deciding what to unit test**

**Testing controllers**

- Variety of techniques



# Test the behavior you coded yourself

**What should you unit test?**



# Code Coverage and Deciding What to Unit Test

**Steer away from generalizations like “don’t test repositories”, “don’t test constructors”, ...**

- Architectures, pattern implementations, ... often differ from project to project
- So-called best practices are sometimes diverted from, on purpose or accidentally



# Code Coverage and Deciding What to Unit Test

**Trying to achieve 100% code coverage can be counterproductive**

- ROI from writing the last 10% might not be worth it



A high code coverage percentage is not an indicator of success, or of code quality.  
A high code coverage percentage only truly represents the amount of code that is covered by unit tests.

## **Code coverage**



```
public async Task<IActionResult> Index() {  
    var internalEmployees = await _context.InternalEmployees.ToListAsync();  
  
    var internalEmployeeForOverviewViewModels =  
    internalEmployees.Select(e => new InternalEmployeeForOverviewViewModel() {  
        Id = e.Id,  
        FirstName = e.FirstName,  
        LastName = e.LastName,  
        Salary = e.Salary,  
        SuggestedBonus = e.SuggestedBonus,  
        YearsInService = e.YearsInService  
    });  
    return View(new EmployeeOverviewViewModel(internalEmployeeForOverviewViewModels));  
}
```

## Controllers Types: Thick Controllers

**Thick controllers contain logic that implements the expected behavior**  
**This is code that should be unit tested**

```
public async Task<IActionResult> Index() {  
    var internalEmployeesViewModels = await  
        _employeeService.FetchInternalEmployeesAsync();  
    return View(new EmployeeOverviewViewModel(internalEmployees)); }  
}
```

## Controllers Types: Thin Controllers

**Thin or skinny controllers delegate the actual implementation of the behavior to other components**

**These typically don't need to be unit tested**



# Introduction to Testing MVC Controllers

**A variety of reasons can lead to choosing for thin or thick controllers...**

- One isn't by definition better than the other

**Can lead to a different decision in regards to whether controllers should be unit tested**



# Introduction to Testing MVC Controllers

**You don't always have the luxury to decide**

- You may get thrown into an existing project halfway through
- You may need to improve the reliability of an existing, finished application by writing tests

**Not every application is built with the same level of quality**



# Introduction to Testing MVC Controllers

**Automated tests can improve an application's reliability (potentially on the way to refactoring)**

- Taking a pragmatic approach to unit testing can be valuable



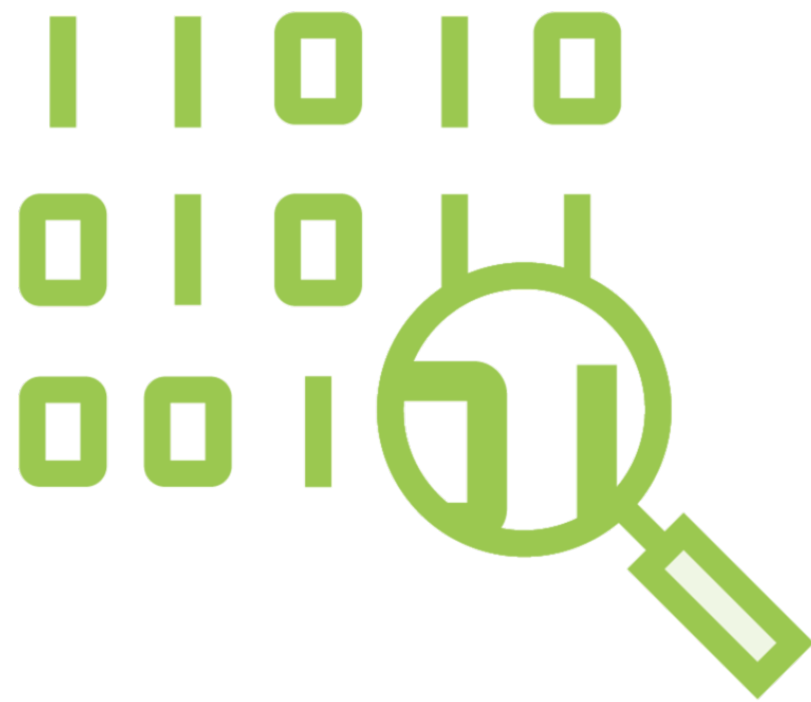
# Introduction to Testing MVC Controllers

## **Test isolation is important**

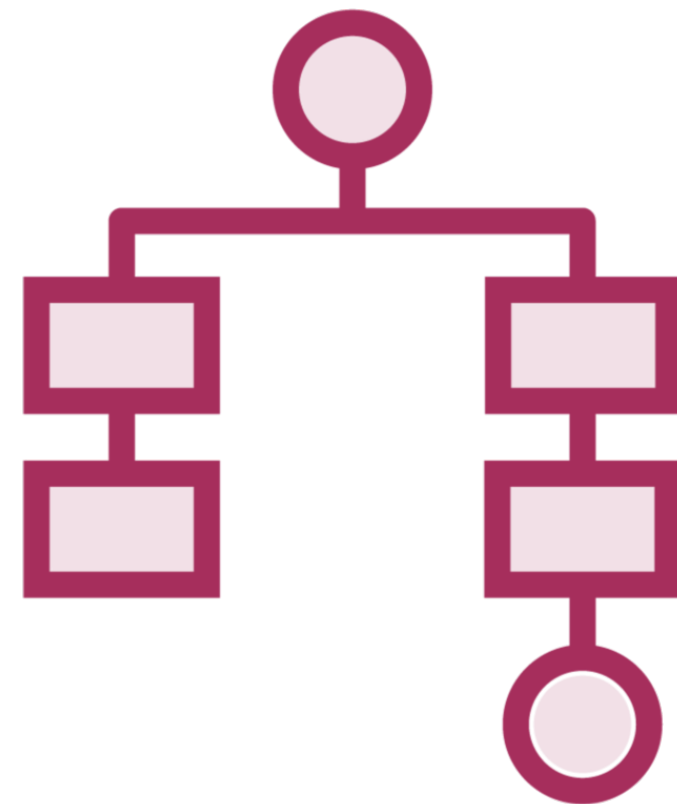
- Avoid model binding, filters, routing, ...



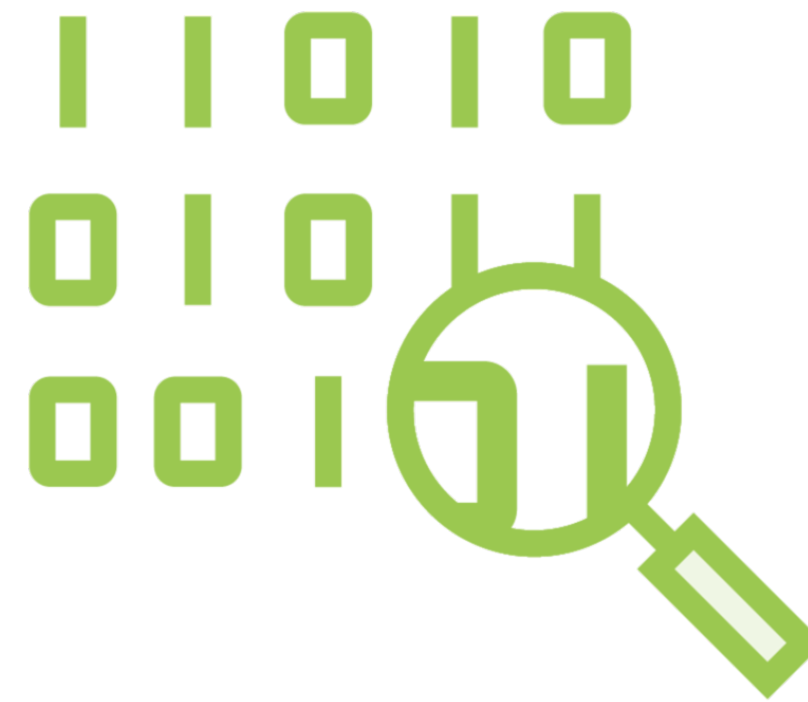
# Introduction to Testing MVC Controllers



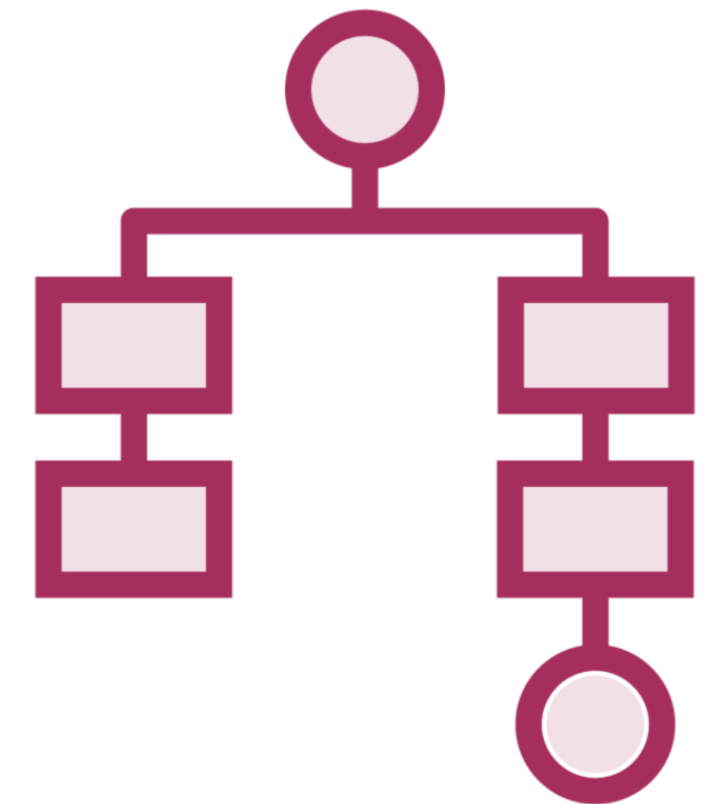
**Expected return  
type**



**Expected type  
of the returned  
data**



**Expected values  
of the returned  
data**



**Other action  
logic that's not  
framework-  
related code**



# Introduction to Testing MVC Controllers

## Concerns when unit testing controllers:

- Mocking controller dependencies
- Working with ModelState in tests
- Dealing with TempData
- Dealing with HttpContext.Session
- Working with HttpClient calls in tests
- ...



# Demo



## Verifying ViewResult when testing



# Demo



## Verifying ViewModel types when testing





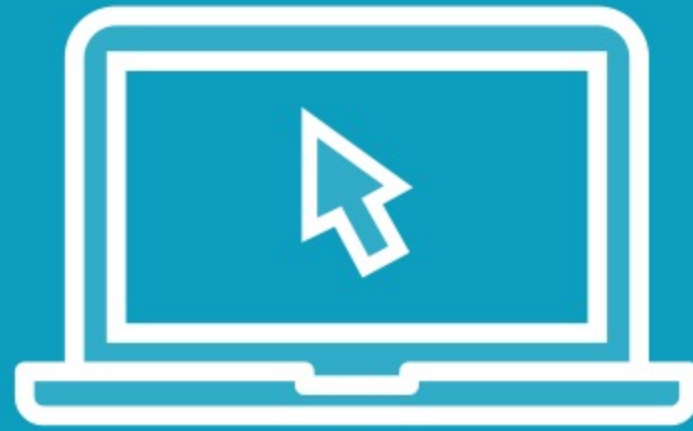
# Demo



## Verifying ViewModel content when testing



# Demo



**Combining controller action asserts in one unit test and testing mapping code**



# Demo



## Dealing with AutoMapper dependencies



# Demo



## Testing validation and ModelState



# HttpContext

**An object which encapsulates all HTTP-specific information about an individual HTTP request: a container for a single request**



# Common Information in HttpContext



**Request**



**Response**



**Features (connection, server info, ...)**



**User**



**Session**



# Testing with HttpContext



**Use the built-in default  
implementation:**  
`DefaultHttpContext`



**Use Moq for mocking:**  
`Mock<HttpContext>`

# Demo



## Testing with HttpContext and TempData





# Demo



## Testing with `HttpContext.Session`



# Demo



## Testing with `HttpContext.Features`



# Demo



## Testing with `HttpContext.User`



# Demo



## Testing with HttpClient calls



# Summary



## Testing MVC controllers

- `ViewResult`, `ViewModels`, `ModelState`,  
`TempData`, `HttpContext`, ...

**Use Moq or other test doubles**

Up Next:

Unit Testing ASP.NET Core Middleware,  
Filters and Service Registrations

---

