Model View Controller

With ASP.NET MVC

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

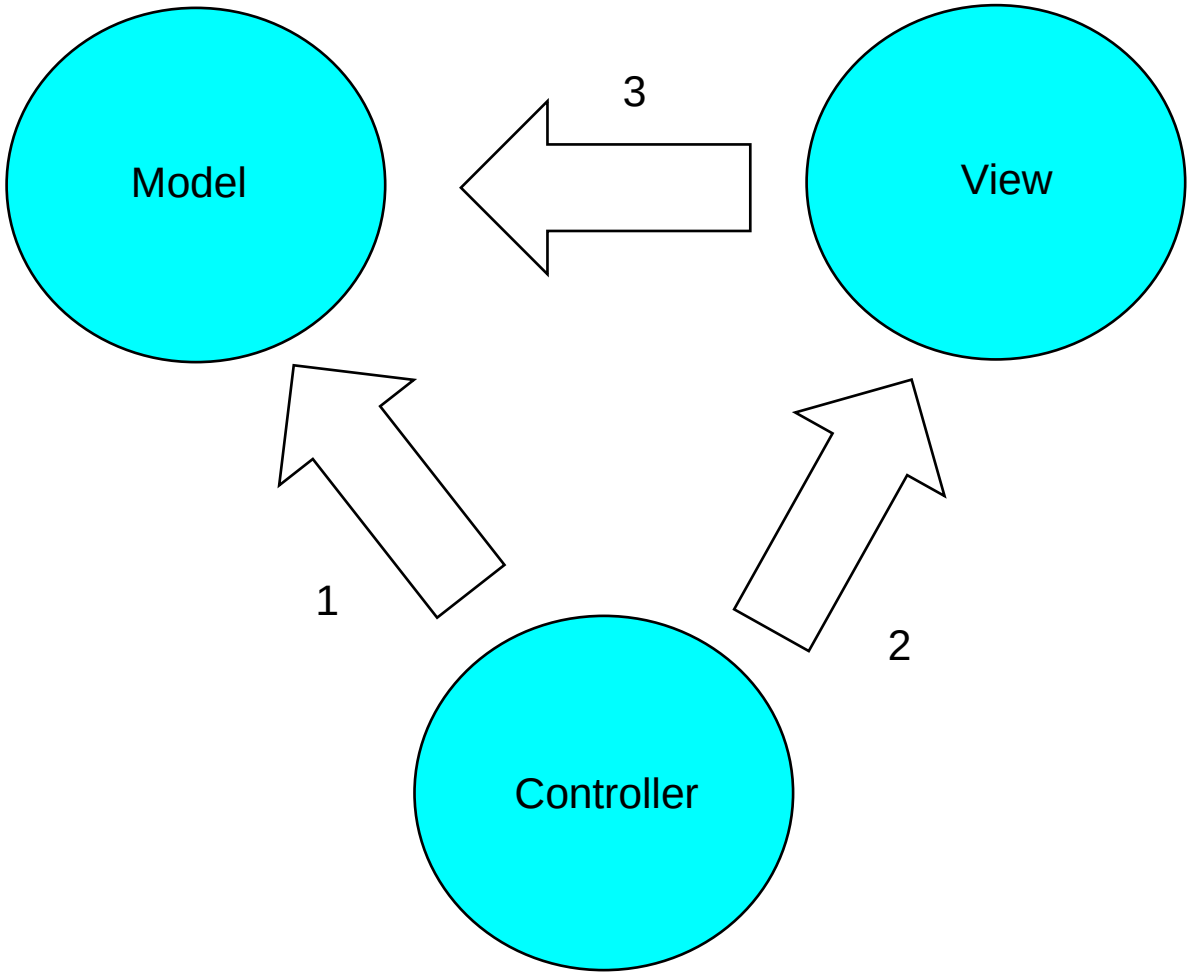
- Why the need for MVC
- Introduce Model View Controller Pattern
- Testing UI with MVC

Why the need for MVC

- Traditional UI layers
  - Code and Layout often being tightly coupled
  - Automated testing relies on UI scripting tools
- Need to separate UI behaviour from UI layout
  - Enables Designers
  - Enables unit testing of UI behaviour

Separation of concern

- Model, View, Controller each responsible for a separate concern



Model

- Model means many things to many people
- In essence represents the data consumed by the view
  - **Domain Objects**
    - Pros
      - Data is just there
    - Cons
      - Domain Objects coupled to view data binding
      - Not always in a convenient form for data binding
  - **ViewModel**
    - Data constructed for the purpose of being bound by the view
    - Pros
      - Minimise view logic
      - Increase unit test coverage
    - Cons
      - Tedious mapping between Domain Objects and ViewModel

View

- Passive
  - Consumes Model to produce output
- Only logic should be View Logic
  - Iteration
  - Conditional styling
- Output can take many forms
  - XML,HTML,JSON,Binary data...

Controller

- Responds to a request
  - Http Verb + URI maps to code
- Responsibilities
  - Construct the model
  - Act on the model
  - Select a view

Skinny Controllers

- Favour “Skinny Controllers”
- Controllers should **not** contain
  - Application logic
  - Calls to Data Access Layer
- Application logic should reside inside Domain Objects or Service Layer
- Rule of thumb :-
  - Removing the controller should not remove application logic

Skinny or Fat Controller ?

```
 public ActionResult PurchaseBasket(string ccNumber , string expiryDate,
                                    string nameOnCard) {
  Order newOrder = new Order();
  using (var ctx = new AcmeStoreDBContext()){

    Basket basket = Basket.Current;

    basket.Items.ForEach(i => newOrder.Items.Add(i));

    _cc.MakePayment(newOrder.Total,ccNumber,expiryDate,nameOnCard);

    _email.SendEmail("", "sales@acme.com", "…", "Your purchase ...");

    ctx.Orders.Add(newOrder);

    ctx.SaveChanges();
  }

  return View("ShowOrder", newOrder);
}
```

Refactoring to skinny controller

- Previous Controller taking far too much responsibility
  - Knows how to create new order
  - Knows to send email when order created
  - Knows how to save order to database
- Favour controller as below
  - Provides orchestration, by delegating to service layer

```
public ActionResult PurchaseBasket(string ccNumber,
                  string expiryDate, string nameOnCard) {
 CreditCard card = new CreditCard(ccNumber,expiryDate,nameOnCard);

 Order newOrder = _os.CreateOrderFromBasketAndPurchase(card,
                                                       Basket.Current);

 var model = newOrder.MapToOrderViewModel();

 return View("ShowOrder", model);
}
```

Avoid God Controllers

- Controllers often built around Entities
  - Operations Create,Update,Delete,List
- Operations can sometimes explode consider building controllers based around business task
  - JobApplicationsController
    - Apply
    - Reject
  - JobApplicationsReportController
    - List

Unit Testing

- Controller is a Plain Old CLR Object (POCO)
- Controller methods return ActionResult object
  - Command Pattern
- Unit Test inspects returned ActionResult to
  - Validate Model
  - Validate View Selection
- Unit Test does NOT Validate rendering

```
[TestMethod]
public void Test_OrderController_PurchaseBasket_Selects_CorrectModelAndView()
{
  IOrderService orderService = null;
  var orderController = new OrderController(orderService);

  ActionResult result = orderController.PurchaseBasket("1234546", "12/10",
                                                       "Penny Less");
  ViewResult viewResult = (ViewResult)result;

  Assert.AreEqual("ShowOrder", viewResult.ViewName);
  Assert.ArtEqual(typeof(OrderViewModel), viewResult.Model.GetType());
}
```

Decoupling controller from Http details

- Unit Test invokes controller method
  - Favour methods taking objects rather than using HttpContext
  - Use ActionFilters to bridge Http world to object world

```
public ActionResult PurchaseBasket(string ccNumber, string expiryDate,
                                   string nameOnCard) {
 Basket basket = new Basket();
 HttpCookie basketCookie = HttpContext.Request.Cookies["basket"];

 // Populate basket object from cookie values

 return View("ShowOrder", model);
}
```

```
[BasketBuilderActionFilter]
public ActionResult PurchaseBasket(Basket basket, CreditCardDetails ccd ) {
   CreditCard card = new CreditCard(ccd.Name, ccd.Number,ccd.Expiry);

   // . . .

   return View("ShowOrder", model);
}
```

Action Filter

- ActionFilter bridges gap between HTTP world and object world
- Simplifies testing

```
public class BasketBuilderActionFilterAttribute : ActionFilterAttribute
{
 public override void OnActionExecuting(ActionExecutingContext filterContext){
   HttpCookie basketCookie = filterContext.HttpContext.Request.Cookies["Basket"];

   Basket basket = basketCookie.MapToBasket();

   filterContext.ActionParameters["basket"] = basket;

 }

 public override void OnResultExecuting(ResultExecutingContext filterContext){
   string encodedBasket = Basket.Current.MapToString();
   var cookie = new HttpCookie("Basket", encodedBasket);
   filterContext.HttpContext.Response.Cookies.Add(cookie);
 }
}
```

Testing just the controller

- Controller will require Dependency Injection
  - Add additional constructors for testing
  - Wire in IoC Container, via IDependencyResolver
    - Framework utilises this interface to create required objects

```
public class MyDependencyResolver : IDependencyResolver{

 public object GetService(Type serviceType) {
      // Delegate to your chosen DI container
}

 public IEnumerable<object> GetServices(Type serviceType){
      // Delegate to your chosen DI container
 }
}

...// Inside global.asax
 protected void Application_Start(){
   DependencyResolver.SetResolver(new MyDependencyResolver());
 }
```

Testing conditional styling

- Conditional presentation in the view is considered ok, but does present issues with testing
- How to select css class based on data value ?
  - If/else logic in view
    - Hard to test
  - Method on view model
    - Can test, but less visibility for designers
  - Create Helper methods/class for designers
    - Testable
    - Visible to designers

Summary

- MVC encourages the building of testable UI
  - Clear separation of concern