# Dot.Net Mvc and Core Course Contents

*.NET is a powerful and popular development platform. With .NET's languages, editors, and libraries, you will be able to build for web, mobile, desktop, games, IoT, and more. Whether you're an experienced developer looking to expand your skills or a beginner just starting out, .NET is a powerful and versatile platform that offers a wealth of tools and features to help you achieve your goals.*

[*https://dotnetfoundation.org/learn/education*](https://dotnetfoundation.org/learn/education)

## Introduction

*Εισαγωγή, δημιουργία νέου project από την αρχή και κατανόηση των βασικών αρχών της αλυσίδας ενεργειών που Request. Κατανόηση και χρήση των μεσολαβητών (middleware), την σημασία τους, κυρίως κατά την χρήση Dependency Injection.*

*Η σημασία του project.cs αρχείου.*

* New Project and Launch Settings
* Understanding Middleware and the Request Pipeline
  + Ο σκοπός μιας asp εφαρμογής είναι να λαμβάνει http αιτήματα (Request) και να επιστρέφει απαντήσεις (Response).  
    Την προετοιμασία του response αναλαμβάνουν τα middleware components, τα οποία είναι διατεταγμένα σε αλυσίδα. Όταν φτάνει ένα αίτημα διατρέχει όλη την αλυσίδα των middleware components τα οποία το επιθεωρούν και το τροποποιούν με κάποιο τρόπο, εάν χρειάζεται, και στην συνέχεια αποστέλλεται η απάντηση.  
    Άλλα m. components παρέχουν υπηρεσίες π.χ. φορμαρίσματος για συγκεκριμένα data types ενώ άλλα παρέχουν λειτουργίες όπως να γράψουν και να διαβάσουν cookies.  
    Αν δεν παραχθεί απάντηση τότε επιστρέφεται το http 404 status code.
* Understanding Services, the Entry Point and the Project File
  + Τα Services είναι objects που παρέχουν λειτουργικότητα σε μια web εφαρμογή.
  + Κάθε κλάση μπορεί να είναι Service και δεν υπάρχουν περιορισμοί στην λειτουργικότητα που παρέχουν.  
    Αυτό που του δίνε την ιδιαιτερότητά τους είναι ότι τις διαχειρίζεται το ASP.Net Core και μια λειτουργία που ονομάζεται dependency injection, όπως θα δούμε πιο κάτω, που μας επιτρέπει να έχουμε πρόσβαση στις υπηρεσίες από οποιοδήποτε σημείο της εφαρμογής.
  + Στο αρχείο *program.cs* βρίσκεται ο κώδικας που θα εκτελεστεί μόλις ξεκινήσει η εφαρμογή και χρησιμοποιείται για να ρυθμίσουμε (configuration) την πλατφόρμα asp και τα ξεχωριστά framework που υποστηρίζει.

var builder = **WebApplication**.**CreateBuilder**(args);

Initializes a new instance of the WebApplicationBuilder class with  
preconfigured defaults.

var app = builder.Build();

Builds the [WebApplication](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication?view=aspnetcore-7.0)

app.MapGet("/", () => "Hello World!");

Defines an endpoint (a route), and a delegate that process it

app.Run();

* Project file
* NuGet Packages  
  identity entity  
  Microsoft.AspNetCore.Identity.EntityFrameworkCore και το πως εμφανίζεται στο project file.

Χρόνος 30 λεπτά

## Middleware

*Κατανόηση και χρήση των middleware, τα διάφορα είδη του και οι ρυθμίσεις που δέχεται.*

* Custom Middleware

app.Use(async (context, next) => {//midlleware logic})

* Defining Middleware Using a Class

*Middleware is a RequestDelegate, as:*

private RequestDelegate \_next;

public async Task Invoke(HttpContext context)

* Understanding the Return Pipeline Path  
  *modify response before and after, use of:*

app.Use(async (context, next) =>{

**await next();**

await context.Response.WriteAsync($"\n Status code: { context.Response.StatusCode }");});

* Short-circuiting the Request Pipeline  
  *terminates the pipeline*

app.Use(async (context, next) => {

if (context.Request.Path == "/short") {

await context.Response.WriteAsync("Request short-circuited");

}

**else** {

await next();

}

});

* Creating Pipeline Branches and Terminal Middleware

*branch object here stands as app, can use next() and a new branch.Use to create a separate sequence of middlewares.*

((IApplicationBuilder)app).Map("/branch", branch => {

branch.Use(async (HttpContext context, Func<Task> next) => {

await context.Response.WriteAsync("Branch middleware");

});

});

* Configuring Middleware

*Configure services with options, as:*

builder.Services.Configure<FruitOptions>(options => {

options.Name = "καρπούζια";});

Χρόνος 60 λεπτά

## Dependency Injection

*Εισαγωγή στην έννοια του Dependency Injection, ο default DI resolver και οι εναλλακτικές λύσεις (π.χ. Autofac)*

1. **Dependency Injection**

*In software engineering,* ***dependency injection*** *is a* ***technique*** *whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service).*

***There are basically three types of dependency injection:***

1. ***constructor injection****: the dependencies are provided through a class constructor.*
2. ***setter injection****: the client exposes a setter method that the injector uses to inject the dependency.*
3. ***interface injection****: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.*
4. **Using a Service in a Middleware Class**
   * Για να χρησιμοποιήσουμε μια κλάση με την τεχνική DI, πρέπει να υλοποιεί ένα Interface και αυτό να δηλώνεται στο program.cs, στην εκκίνηση της εφαρμογής, π.χ.
   * builder.Services.**AddSingleton**<IResponseFormatter, HtmlResponseFormatter>();
5. **Getting Services from the HttpContext Object**
   * Σε περιπτώσεις που δεν μπορούμε να χρησιμοποιήσουμε ***constructor injection*** τότε μπορούμε να χρησιμοποιήσουμε τις διαθέσιμες μεθόδους του context ως εξής:  
     IResponseFormater fomater =   
     **context**.RequestServices.GetRequirexService<IResponseFormater>();

και έτσι να έχουμε στην διάθεσή μας τις μεθόδου της κλάσης που έχει δηλωθεί.

1. **Creating Transient Services**

Αυτή η μέθοδος επίλυσης δημιουργεί ένα νέο instance της κλάσης που επιλύεται **κάθε φορά** που επιλύεται η εξάρτηση για το Interface.

1. **Using Scoped Services**

Στην πρώτη δήλωση χρησιμοποιήσαμε την AddSingleton και την AddTransient μεθόδους επίλυσης. Η μέθοδος αυτή επιλύει την εξάρτηση δημιουργώντας ένα instance της υλοποιούμενης κλάσης για όλη την ζωή (**Life Cycle**) του request.

Το Core μας δίνει τρείς μεθόδους που έχουν διαφορετικό life cycle.

*The Extension Methods for Creating Services*

**Name Description**

AddSingleton<T, U>() This method creates a single object of type U that is used to resolve all dependencies on type T.

AddTransient<T, U>() This method creates a new object of type U to resolve each dependency on type T.

AddScoped<T, U>() This method creates a new object of type U that is used to resolve dependencies on T within a single scope, such as request.

Σε αυτό το παράδειγμα θα χρησιμοποιήσουμε την AddScoped μέθοδο, που είναι και η συνηθέστερη μέθοδος επίλυσης των εξαρτήσεων της DI.

Χρόνος 60 λεπτά

## Various Features

*Κατανόηση των πιο συχνά χρησιμοποιημένων λειτουργιών όπως το αρχείο asppsetting.json, οι Sessions και τα cookies. Δημιουργία και Log files, http και https.*

* **appsettings.json**

Οι ρυθμίσεις εκκίνησης της εφαρμογής και οι ρυθμίσεις που αφορούν τον web server, όπως το URL της εφαρμογής κτλ.) ορίζονται σε τρία json αρχεία τα: Properties/launchSettings.json, appsettings.json, appsettings.Development.json.

Τα appsettings.json και appsettings.Development.json, επεκτήνουν και προσθέτουν στο Properties/launchSettings.json. Η ύπαρξη δύο appsetting.json επιτρέπει την δυνατότητα διαφορετικής παραμετροποίησης μεταξύ των **Production**, **Development**, **Staging** profiles, όπου μπορούμε να ορίζουμε π.χ. τα διαφορετικά connection strings για τις διάφορες φάσεις ανάπτυξης μιας εφαρμογής.

Για τα διάφορα περιβάλλοντα εκτέλεσης της εφαρομγής See: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-7.0>

* **Accessing Configuration Settings**

*Για να αποκτήσουμε πρόσβαση στις ρυθμίσεις χρησιμοποιούμε το interface IConfiguration και για να πάρουμε για παράδειγμα τις ρυθμίσεις για το* ***default*** *επίπεδο* ***Logging*** *της εφαρμογές χρησιμοποιούμε την σύνταξη,  
config["Logging:LogLevel:Default"]*

* **Using Configuration Data with the Options Pattern**

*Μπορούμε να προσθέσουμε στο appsettings.json οποιαδήποτε αρχική ρύθμιση αφορά την εφαρμογή μας και δεν μπορούμε να την έχουμε διαθέσιμη με άλλο πιο δυναμικό τρόπο π.χ. Database.*

*Ένα τέτοιο παράδειγμα παρουσιάζουμε εδώ, αλλάζοντας τις τιμές των properties της κλάσης FruitOptions που χρησιμοποιήσαμε σε προηγούμενο παράδειγμα. Μπορούμε να ρυθμίσουμε όποια αξία θέλουμε από τα properties της κλάσης με την προσθήκη των αντίστοιχων ρυθμίσεων στο appsettings.json.*

builder.Services.Configure<FruitOptions>(servicesConfig.GetSection("Fruit"));

*Και να χρησιμοποιήσουμε το* FruitMiddleware.cs *για να το δούμε στο path* /fruit*.*

* **Getting the Environment**

*Σε προηγούμενο παράδειγμα είδαμε πως μπορούμε να αλλάξουμε το περιβάλλον εργασίας σε: Production, Development, Staging. Σε αυτό το παράδειγμα θα δούμε πως μπορούμε να ελέγξουμε στον κώδικα σε ποιο περιβάλλον λειτουργεί η εφαρμογή και να διαφοροποιήσουμε τυχόν τον κώδικα με Debug Logs όπως θα δούμε στο επόμενο παράδειγμα.*

app.MapGet("/config", async (HttpContext context, IConfiguration config) =>{

*// Πρώτος τρόπος*

string environment = config["ASPNETCORE\_ENVIRONMENT"];

*// Χρησιμοποιώντας μέθοδο IsDevelopment()* *του app*

if (app.Environment.IsDevelopment()){*//some code*}

}

* **Using the Logging Service and Logging Messages in the Program File**

*Το Dot.Net Core διαθέτει έτοιμο μηχανισμό καταγραφών (Logging), με την χρήση του ILogger<T>, ενός generic interface όπου* ***T****, από σύμβαση ή κλάση στην οποία κάνουμε τις καταγραφές. Ο Logger έχει μεθόδους όπως: LogCritical, LogDebug, LogError, LogInformation, LogTrace, LogWarning.*

*Παράδειγμα χρήσης στην κλάση* FruitMiddleware*.*

…

public async Task Invoke(HttpContext context, ILogger<FruitMiddleware> logger)

και πιο κάτω

logger.LogDebug($"Started processing for {context.Request.Path}");

…

* **Using Static Content and Client-Side Packages**

*Το Dot.Net Core άλλαξε το default directory των στατικών αρχείων που στις προηγούμενες εκδόσεις μοιραζόταν ανάμεσα στα Content, Lib, Script directories, σε ένα με συμβατικό όνομα* **wwwroot***, με ειδική οπτική σήμανση, όπου συγκεντρώνονται όλα τα στατικά αρχεία (html, css, js κτλ.). Επιπλέον πρόσθεσε στο context menu με δεξί κλικ στο συγκεκριμένο directory, την εντολή Add/Client-Side Library.. αντλώντας βιβλιοθήκες* css *ή* js *από διάφορους providers (π.χ. bootstrap).*

* **Using Cookies**

*Ο πιο συνήθης τρόπος αποθήκευσης εξατομικευμένων πληροφοριών των επισκεπτών μιας web εφαρμογής γίνεται με την χρήση των cookies. Τα cookies είναι μικρά αρχεία text που αποθηκεύονται στον client και περιέχουν ένα key value pare (π.χ. “page”: 1). Δημιουργούνται, ρυθμίζονται και διαγράφονται μέσω της συλλογής Request.Cookies*

*Η δημιουργία ενός cookie γίνεται με την εντολή*

context.Response.Cookies.Append

*Η διαθέσιμες ρυθμίσεις του cookie (π.χ. MaxAge) γίνεται μέσω της κλάσης CookieOptions*

context.Response.Cookies.Append(

"counter",

counter.ToString(),

new CookieOptions {

MaxAge = TimeSpan.FromMinutes(30)

}

);

*Η διαγραφή ενός cookie γίνεται με την εντολή*

context.Response.Cookies.Delete("counter");

*Server Side μπορούμε να διαβάσουμε ένα cookie (σημειώστε ότι ο client browser αποστέλλει σε κάθε request και όλα τα cookies που έχει θέσει η εφαρμογή), με την εντολή:*

context.Request.Cookies[*cookieNameString*]

* **Using Sessions**

*Η Session είναι ένα τμήμα της μνήμης του server που μπορεί να εκχωρηθεί σε κάθε αρχική επίσκεψη στην εφαρμογή. Ο χώρος είναι ξεχωριστός για κάθε επισκέπτη και τακτοποιείται μέσω του Session Id, ενός Guid string, που δημιουργείται με την χρήση της Session.*

*Στην Session μπορούμε να αποθηκεύσουμε (πάντα στην μνήμη του server), οποιαδήποτε πληροφορία, όμως αυτό επιβαρύνει την διαθέσιμη μνήμη που εκχωρεί ο web server στην εφαρμογή μας και εξαρτάται από την ρύθμιση του Application Pool της εφαρμογής.*

*Για να χρησιμοποιήσουμε την Session πρέπει αμέσως μετά την δημιουργία του builder να δώσουμε την εντολή:*

builder.Services.**AddDistributedMemoryCache**();

*Με αυτή την ρύθμιση η χρησιμοποιούμενη μνήμη είναι αυτή που αναφέραμε. Υπάρχουν ωστόσο βιβλιοθήκες που χρησιμοποιούν δική τους database όπου καταχωρούν τις πληροφορίες που θέλουμε να διατηρούμε για τον επισκέπτη και τις ανασύρουν με πολύ εύκολο και γρήγορο τρόπο, διαθέτοντας ταυτόχρονα και πολύ ευέλικτες μεθόδους για την παροχή πάντα επίκαιρων τιμών. Μια τέτοια βιβλιοθήκη είναι η Redis.*

*Η χρήση της Session απαιτεί την δημιουργία ενός ιδιαίτερου cookie, παράδειγμα:*

builder.Services.AddSession(options => {

options.IdleTimeout = TimeSpan.FromMinutes(30);

options.Cookie.IsEssential = true;

});

*Για να χρησιμοποιηθεί η Session πρέπει να δώσουμε την εντολή*

app.UseSession();

*και στην συνέχεια μπορούμε να αποθηκεύσουμε τιμές, π.χ.*

context.Session.SetInt32("counter", counter);

*και να την διαβάσουμε π.χ.*

context.Session.GetInt32("counter")

* **Detecting HTTPS**

*Ο εντοπισμός του secure πρωτοκόλλου https είναι κρίσιμη λειτουργία για την ασφάλεια μιας εφαρμογής. Μπορούμε να ρυθμίσουμε το Url της εφαρμογής μας και για τα δύο πρωτόκολλα με την χρήση του "applicationUrl" τμήματος στα profile του launchSetting.json. Ο εντοπισμός γίνεται με την χρήση της Boolean μεταβλητής context.Request.IsHttps.*

* **Enforcing HTTPS Requests**

Μπορούμε να εξαναγκάσουμε την αυτόματη μετάβαση σε https πρωτόκολλο ασχέτως με το πρωτόκολλο που αιτείται ο client browser με την εντολή:

app.UseHttpsRedirection();

* Enabling HSTS Strict Transport Security (HSTS)

Η χρήση του HSTS (διαβάστε λεπτομέρειες εδώ:  
<https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-6.0&tabs=visual-studio%2Clinux-ubuntu#http-strict-transport-security-protocol-hsts>

Γίνεται με την εντολή builder.Services.AddHsts,π.χ.

builder.Services.AddHsts(options => {

options.MaxAge = TimeSpan.FromDays(1);

options.IncludeSubDomains = true;

});

**Χρόνος 60 λεπτά**

## Using Entity Framework Core

*Εγκατάσταση και ρύθμιση του Entity Framework, ενναλακτικές (π.χ. dupper) υβριδική συνύπαρξη.*

* **Installing EF Core**

*Για να εγκαταστήσουμε τις αναγκαίες βιβλιοθήκες για την επικοινωνία με διάφορες βάσεις δεδομένων χρησιμοποιούμε την δέσμη βιβλιοθηκών του EntityFrameWorkCore*

*Πρώτα, σε όλες τις περιπτώσεις, εγκαθιστούμε την βασική βιβλιοθήκη και μετά την Design για την υποστήριξη του Code First, και τέλος την υποστήριξη του sql server provider που θέλουμε, φυσικά από τους διαθέσιμους providers του EntityFrameWorkCore, στην περίπτωσή μας του SqlServer της Microsoft.*

* + *Test dotnet ef*
  + *Dotnet tool install –global dotnet-ef*
  + *Add reference, Microsoft.AspNetCore.Identity.EntityFrameworkCore*
  + *Add reference, Microsoft.EntityFrameworkCore.Design*
  + *Add reference, Microsoft.EntityFrameworkCore.SqlServer*
* **Creating Data Models**

*Για την δημιουργία κλάσεων που αντιπροσωπεύουν πίνακες στην βάση δεδομένων, χρησιμοποιούμε κλάσεις με το όνομα του πίνακα που θέλουμε και ακολουθούμε μια σειρά συμβάσεις που καθιστούν πιο εύκολη την δημιουργία των πινάκων καθώς και των relations μεταξύ τους. Μερικές από αυτές τις συμβάσεις:*

*Οι κλάσεις αυτές αναφέρονται συνήθως ως entities, ενώ οι κλάσεις που χρησιμοποιούμε για να δείξουμε τα δεδομένα τους αναφέρονται ως view models.*

*Το πρωτεύον κλειδί είναι πάντα το Id*

*Ένα foreign key έχει το όνομα της κλάσης/πίνακα με την προσθήκη Id (π.χ. CategoryId)*

*Παραδείγματα για τις κλάσεις Product και Category που θα χρησιμοποιήσουμε στην συνέχεια.*

namespace Core.Models {

public class Category {

public long Id { get; set; }

public string Name { get; set; }

//Navigation Property

public IEnumerable<Product> Products { get; set; }

}

public class Product {

public long Id { get; set; }

public string Name { get; set; }

[Column(TypeName = "decimal(8, 2)")]

public decimal Price { get; set; }

//Foreign key

public long CategoryId { get; set; }

//Navigation Property

public Category Category { get; set; }

}

}

* **DbContext Class**

*Η βασική κλάση που παρέχει την λειτουργικότητα της επικοινωνία με την βάση δεδομένων είναι η DbContext, την οποία και επεκτείνουμε με μια δική μας για να την ρυθμίσουμε με τα δικά μας δεδομένα (π.χ. connection string). Από σύμβαση την αποθηκεύουμε σε ένα νέο directory με το όνομα Infrastructure, παράδειγμα η κλάση μας DataContext:*

*using Core.Models;*

*using Microsoft.EntityFrameworkCore;*

*namespace Core.Infrastructure {*

*public class DataContext : DbContext {*

*public DataContext(DbContextOptions<DataContext> options) : base(options) { }*

*public DbSet<Product> Products { get; set; }*

*public DbSet<Category> Categories { get; set; }*

*}*

*}*

*Για να χρησιμοποιήσουμε την σύνδεση με την βάση στο program.cs πρέπει να ρυθμίσουμε το αντίστοιχο service ως εξής:*

*builder.Services.AddDbContext<DataContext>(options => {*

*options.UseSqlServer(builder.Configuration["ConnectionStrings:DbConnection"]);*

*});*

*και στο appsettings.json θέτουμε το κατάλληλο connection string για την db, π.χ.*

*"ConnectionStrings": {*

*"DbConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CoreDB;MultipleActiveResultSets=True"*

*}*

* **Configuring the Database Service and Using Migrations**

*Για να δημιουργήσουμε την βάση δεδομένων το* EntityFrameworkCore *μας παρέχει μια σειρά εντολών που μπορούμε να τις εκτελέσουμε μέσα από το power shell terminal. Πρώτα πρέπει να έχουμε εγκαταστήσει το* ef *στα εργαλεία του dotnet με την εντολή:*

*dotnet tool install --global dotnet-ef*

*και στην συνέχεια να δώσουμε την εντολή να δημιουργηθούν οι πίνακες στην βάση:*

*dotnet ef migrations add Initial*

*dotnet ef database update*

*μπορούμε να ανακαλέσουμε το τελευταίο migration με την εντολή:*

*dotnet ef migrations remove*

*αν έχουμε ήδη κάνει update την βάση τότε θα μας προειδοποιήσει ότι έχει ήδη γίνει update το migration που προσπαθούμε να ανακαλέσουμε.*

*Μπορούμε επίσης να δημιουργήσουμε script με όλα τα migrations (π.χ. σε περίπτωση που κάνουμε update μια ολόκληρη version με πολλές αλλαγές).*

*dotnet ef migrations script --output udate.sql*

*Μπορούμε επίσης και να δημιουργήσουμε script από κάποιο migration και μετά.*

*dotnet ef migrations script 20240311182621\_Initial --output udate.sql*

*τέλος αν θέλουμε να ξαναδημιουργήσουμε την βάση μπορούμε με την εντολή:*

*dotnet ef database drop --force*

* **Adding Seed Data**

*Μπορούμε να εγκαινιάσουμε την βάση με αρχικές εγγραφές χρησιμοποιώντας την ακόλουθη μέθοδο.*

*1 Δημιουργούμε μια κλάση, με ονομασία π.χ. SeedData*

*public class SeedData*

*2 Δημιουργούμε μια static μέθοδο με ονομασία π.χ. SeedDatabase και ως όρισμα την DataContext κλάση μας.*

*public static void SeedDatabase(DataContext context)*

*3 Καλούμε την Database.Migrate() μέθοδο του context*

context.Database.Migrate();

*4 Ελέγχουμε αν υπάρχουν εγγραφές π.χ.*

if (context.Products.Count() == 0 && context.Categories.Count() == 0)

*5 Προσθέτουμε τις εγγραφές που θέλουμε π.χ.*

Category fruits = new Category { Name = "fruits" };

Category shirts = new Category { Name = "shirts" };

context.Products.AddRange(

new Product {

Name = "Apples",

Price = 1.50M,

Category = fruits

},

new Product {

Name = "Lemons",

Price = 2M,

Category = fruits

},

…

)

*6 Αποθηκεύουμε τις αλλαγές με την μέθοδο SaveChanges() του context.*

context.SaveChanges();

*7 και τέλος αν θέλουμε να «τρέχει» στην εκκίνηση της εφαρμογής, στο* program.cs *δίνουμε την εντολή:*

var context = app.Services.CreateScope().ServiceProvider

.GetRequiredService<DataContext>();

SeedData.SeedDatabase(context);

Χρόνος 60 λεπτά

## Web Services

*Εισαγωγή στα Web Service στο κλασσικό mvc.*

* **Creating a Simple Web Service**

*Ένα Web Service δέχεται HTTP Requests και παράγει αποκρίσεις που περιέχουν δεδομένα.*

*Ο πλέον απλός τρόπος χρήσης των back end web εφαρμογών σήμερα είναι να λειτουργούν ως web service, δηλαδή να παρέχουν δομημένα δεδομένα σε απλές μορφές, συνήθως σε μορφή* json *ή* xml*. Με αυτόν τον τρόπο, μπορούν αν χρησιμοποιούν τις πιο βασικές κλάσεις του Dot.Net Core και να παρέχουν δεδομένα χωρίς την ανάγκη πολύπλοκων μορφοποιήσεων, όπως χρειάζεται να κάνουν οι front end εφαρμογές, με αποτέλεσμα να είναι εξαιρετικά πιο γρήγορες εφαρμογές, μεταφέροντας τα χρονοβόρα καθήκοντα όπως η μορφοποίηση με την χρήση π.χ.* html, css, javascripts *στην client εφαρμογή, δηλαδή στον browser. Επιπλέον με βάση τα δεδομένα που παρέχει στους front end clients μπορεί να χρησιμοποιούνται από διαφορετικές εφαρμογές.*

*Στο παράδειγμα θα δημιουργήσουμε μια απλούστατη εκδοχή, την μορφή των middleware που γνωρίζουμε ήδη. Σημειώστε ότι και οι σύνθετη εργασία που εκτελείται με την χρήση των Controllers δεν είναι τίποτε άλλο από μια ειδική μορφή middleware, δηλαδή παραλαμβάνει requests και παρέχει responses*.

**const** string BASEURL = "/api/products";  
**app**.MapGet($"{BASEURL}/{{id}}", async (HttpContext context, DataContext **data**) => {  
 **string** **id** = **context**.Request.RouteValues["**id**"] **as** **string**;  
  
 **if** (**id** != **null**) {  
 Product **product** = **data**.Products.Find(**long**.Parse(id));  
   
 **if** (**product** == **null**) {  
 **context**.Response.StatusCode = StatusCodes.Status404NotFound;  
 } else {  
 **context**.Response.ContentType = "**application**/**json**";  
 **await** **context**.Response.WriteAsync(JsonSerializer.Serialize<Product>(**product**));  
 }  
 }  
});

*εκτός από την λίστα των προϊόντων μπορούμε να χρησιμοποιήσουμε την ίδια τεχνική για να πάρουμε τις λεπτομέρειες ενός προϊόντος ή και να εισάγουμε ένα νέο.*

*Η δημιουργία web services συνδέεται άρρηκτα με την έννοια των endpoints. Η ανεύρεση του τμήματος του κώδικα που θα αναλάβει να παράγει ένα αποτέλεσμα για κάποιο αίτημα, είναι μια διαδικασία που αναζητά ανάμεσα στα endpoints της εφαρμογής, το κατάλληλο για το συγκεκριμένο αίτημα. Η διαδικασία αυτή ονομάζεται Routing, δρομολόγηση. Το Dot.Net Core παρέχει διάφορους, ευέλικτους τρόπους για τον καθορισμό routing.*

*Διαβάστε σχετικά στο συνοδευτικό έγγραφο «****Routing in ASP.docx****»*

* **Creating a Controller**

*Οι Controllers είναι οι βασικοί μηχανισμοί που χρησιμοποιούμε για την επεξεργασία των αιτημάτων. Από σύμβαση στην ονομασία τους προστίθεται το επίθεμα Controller (π.χ. ProductController) και βρίσκονται στο directory Controllers. Για να τους χρησιμοποιήσουμε δίνομε τις ακόλουθες εντολές στο program.cs*

builder.Services.**AddControllers**();  
var app = builder.Build();  
app.**MapControllers**();

*Οι Controllers είναι κλάσεις, που επεκτείνουν την κλάση Controller του Dot.Net MVC, οι μέθοδοι τους είναι γνωστοί ως actions και επεξεργάζονται τα httpRequests.*

*Κατ’ αρχή προσθέτουμε την υπηρεσία builder.Services.****AddControllers****(); που προσφέρει την απαιτούμενη λειτουργικότητα στους Controllers. Εγγράφει στο μητρώο της εφαρμογής (Registers) οτιδήποτε χρειαζόμαστε για την ανάπτυξη Web εφαρμογών, όπως: , Model Binding, API Explorer, Authorization, CORS, Validations, Formatter Mapping, κτλ.*

*Με την χρήση της μεθόδου app.MapControllers();, ρυθμίζουμε την εφαρμογής μας να βασίζεται στα* ***attributes*** *που βάζουμε πάνω από τα actions π.χ.* ***[Route("api/Color/GetColors/")]****,ώστε να έχουμε πιο εύκολα και κατανοητά routes αντίθετα με τον παραδοσιακό τρόπο με την χρήση του RouteCollection και τις μεθόδους του.*

*Περισσότερα για το routing:  
<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-6.0> , υπάρχει και σε μορφή docx μαζί με το παρόν documentation (στο συνοδευτικό έγγραφο «****Routing in ASP.docx****»).*

*Μπορούμε να επεκτείνουμε την κλάση ControllerBase που μας δίνει την δυνατότητα να χρησιμοποιούμε Actions χωρίς Views, π.χ. στα Api ή για χάρη απλούστευσης.*

*Παράδειγμα 1*

[Route("api/[controller]")]  
 public **class** **ProductsController** : **ControllerBase** {  
 // api/products  
 [HttpGet]  
 public IEnumerable<Product> **GetProducts**() {  
 return **new** Product[] {  
 **new** Product() { Name = "Product #1" },  
 **new** Product() { Name = "Product #2" }  
 };  
 }  
  
 // api/products/1  
 [HttpGet("{id}")]  
 public Product **GetProduct**() {  
 return **new** Product() { Id = 1, Name = "Product #1" };  
 }  
 }

* **Using DI in Controllers**

*Χρησιμοποιώντας Dependency Injection, θα πάρουμε τα δεδομένα από την βάση δεδομένων με την κλάση DataContext. Η τεχνική που χρησιμοποιείται είναι η constructor injection ως εξής:*

**private** DataContext \_context;  
  
public **ProductsController**(DataContext context){  
 \_context = context;  
}  
  
// api/products  
[HttpGet]  
public IEnumerable<Product> **GetProducts**(){  
 return \_context.Products;  
}

* **Using Model Binding to Access Route Data**

*Μια από τις βασικότερες δυνατότητες που μας δίνει η κλάση Controller και η* ControllerBase*, είναι το Model Binding.*

*Στο παράδειγμα, προσθέτουμε μια long παράμετρο με το όνομα id στη μέθοδο GetProducts. Όταν γίνεται η κλήση της μεθόδου (Action), το* MVC Framework *εισάγει την τιμή με το ίδιο όνομα από τα δεδομένα δρομολόγησης, μετατρέποντάς την αυτόματα σε μια long τιμή, η οποία χρησιμοποιείται από την GetProducts για την αναζήτηση στη βάση δεδομένων χρησιμοποιώντας τη μέθοδο Find της LINQ.*

*Ταυτόχρονα χρησιμοποιούμε*

// api/products/1  
[HttpGet("{id}")]  
public Product GetProduct(long id, [FromServices] ILogger<ProductsController> logger){  
 logger.LogDebug("GetProduct Action Invoked");  
 return \_context.Products.Find(id);  
}

*Το FromServicesAttribute [FromServices] επιτρέπει την παροχή μιας υπηρεσίας, μέσω DI, απευθείας σε μια μέθοδο χωρίς τη χρήση της τεχνικής* ***constractor injection****:*

* **Model Binding from the Request Body**

*Με την χρήση του Model Binding στην πιο κάτω POST μέθοδο η [FromBody] attribute ελέγχει τις τιμές που θα έρθουν στο body του αιτήματος με αυτές της κλάσης Product και παρέχει ένα νέο Product object.*

// api/products  
 [HttpPost]  
 public void SaveProduct(**[FromBody] Product** product) {  
 \_context.Products.Add(product);  
 \_context.SaveChanges();  
 }

* **Adding Update and Delete Actions**

*Με αντίστοιχο τρόπο δημιουργούμε τις μεθόδους UpdateProduct και DeleteProduct για να ολοκληρώσουμε όλες τις CRUD (Create, Update, Delete) διαδικασίες.*

// api/products  
[HttpPut]  
public **void** **UpdateProduct**([FromBody] Product product) {  
 \_context.Update(product);  
 \_context.SaveChanges();  
}  
  
// api/products/1  
[HttpDelete("{id}")]  
public **void** **DeleteProduct**(long id) {  
 \_context.Products.Remove(new Product { Id = id });  
 \_context.SaveChanges();  
}

* **Using Asynchronous Actions**

*Ήδη από τα πρώτα middleware που φτιάξαμε, χρησιμοποιήσαμε Asynchronous Programming με τα keywords async και await δείτε περισσότερα εδώ:  
<https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>  
Κυρίως στις λειτουργίες που απαιτούν ανάσυρση δεδομένων από την βάση δεδομένων εκμεταλλευόμαστε τον ασύγχρονο προγραμματισμό. Με αυτόν τον τρόπο ένα thread δεν μπλοκάρετε από το να εκτελέσει άλλα αιτήματα όσο περιμένει μια χρονοβόρα διαδικασία.*

*Στο παράδειγμα μετατρέπουμε όλες τις CRUD μεθόδους στις αντίστοιχες ασύγχρονες.*

[HttpGet]  
public IAsyncEnumerable<Product> **GetProducts**() {  
 return \_context.Products.AsAsyncEnumerable();  
}  
  
// api/products/1  
[HttpGet("{id}")]  
public **async** Task<Product> **GetProduct**(**long** id, [FromServices] ILogger<ProductsController> logger) {  
 logger.LogDebug("GetProduct Action Invoked");  
 return **await** \_context.Products.FindAsync(id);  
}  
  
// api/products  
[HttpPost]  
public **async** Task **SaveProduct**([FromBody] Product product) {  
 **await** \_context.Products.AddAsync(product);  
 **await** \_context.SaveChangesAsync();  
}

*Το Entity Framework Core παρέχει ασύγχρονες εκδόσεις ορισμένων μεθόδων, όπως το FindAsync, AddAsync και SaveChangesAsync, και τα χρησιμοποιούμε με τη λέξη-κλειδί await. Δεν μπορούν όλες οι λειτουργίες να εκτελούνται ασύγχρονα, γι' αυτό οι μέθοδοι Update και Delete παραμένουν αμετάβλητες εντός των μεθόδων UpdateProduct και DeleteProduct.*

* **Using Action Results**

*Λόγω του ότι το DotNet προσθέτει http status codes με μια δική του λογική, που δεν είναι σίγουρο ότι ταιριάζει με τα status codes που περιμένει η front end εφαρμογή μας, ιδιαίτερα με την χρήση του Ajax, χρησιμοποιούμε το interface IActionResult αυτό επιτρέπει στην μέθοδο να καθορίζει αυτή το τι status code θα στείλει και δεν το αφήνει στις αυτόματες μεθόδους του DotNet.*

// api/products/1  
[HttpGet("{id}")]  
public **async** Task<IActionResult> **GetProduct**(**long** id) {  
 Product product = **await** \_context.Products.FindAsync(id);  
 **if** (product == null) {  
return NotFound();  
 }  
 return Ok(product);  
}

* **Performing Redirections**

Είναι σύνηθες να ανακατευθύνουμε μια μέθοδο σε μια άλλη μέθοδο. Η πιο συνηθισμένη χρήση του είναι όταν εκτελείται μια Update μέθοδος η οποία κάνει update το object που θέλουμε στην βάση και στην συνέχεια μεταφέρει τον χρήστη στην Edit μέθοδο και δείχνει τις ενημερωμένες τιμές στον χρήστη, έτσι μάλιστα αποφεύγουμε το repost όταν ο χρήστη ανανεώνει την σελίδα. Υπάρχουν διάφοροι τρόποι Redirection, όπως βλέπουμε και στα σχόλια το κώδικα.

// api/products/redirect  
[HttpGet("redirect")]  
public IActionResult **Redirect**() {  
 //return Redirect("/api/products/1");  
 //return RedirectToAction(nameof(GetProduct), new { Id = 1 });  
 return **RedirectToRoute**(**new** {  
 controller = "Products",  
 action = "GetProduct",  
 Id = 1  
 });  
}

* **Validating Data**

Η αξιολόγηση (validating) των δεδομένων που παρέχει ο χρήστης σε μια εφαρμογή **πρέπει να αξιολογείται οπωσδήποτε και server side**. Το Framework μας παρέχει διάφορους τρόπους να καθορίσουμε ποια properties είναι απαραίτητα, τι τιμές και σε ποιο εύρος επιτρέπονται κτλ. Η πιο απλή μορφή είναι με την χρήση attributes στις κλάσεις, όπως φαίνεται στο πιο κάτω παράδειγμα.

public **class** **Product**  
{  
 public **long** Id { **get**; **set**; }  
   
 [Required]  
 public **string** Name { **get**; **set**; }  
  
 [Required]  
 [Range(0.01, double.MaxValue)]  
 [Column(TypeName = "decimal(8, 2)"**)]**  
 public **decimal** Price { **get**; **set**; }  
  
 [Required]  
 [Range(1, long.MaxValue)]  
 public **long** CategoryId { **get**; **set**; }  
 public Category Category { **get**; **set**; }  
}

και να αξιολογήσουμε τα δεδομένα στην SaveProduct μέθοδο.

// api/products  
[HttpPost]  
public **async** Task<IActionResult> **SaveProduct**([FromBody] Product product) {  
 **if** (**ModelState.IsValid**) {  
 **await** \_context.Products.AddAsync(product);  
 **await** \_context.SaveChangesAsync();  
  
 return Ok(product);  
 }  
 return **BadRequest**(ModelState);  
}

* **Applying the ApiController Attributes**

*Χρησιμοποιώντας το [ApiController] attribute, μπορούμε να αποφύγουμε τα attributes [FromBody] και τον έλεγχο ModelState.IsValid, το binding και το validation το κάνει αυτόματα ο Controller. Παρ’ όλο ότι είναι βοήθεια δεν μας αφήνει να στείλουμε άλλες μορφές απόκρισης που να περιλαμβάνουν μηνύματα κτλ., έτσι το αποφεύγουμε.*

*Δείτε στο παράδειγμα, αν αποτύχει η αξιολόγηση ο Controller θα ανταποκριθεί με BadRequest Status.*

**[ApiController]**  
[Route("api/[controller]")]  
public **class** ProductsController : ControllerBase  
  
[HttpPut]  
public void **UpdateProduct**(Product product) {  
 \_context.Update(product);  
 \_context.SaveChanges();  
}

* **Omitting Null Properties**

*Όταν ζητήσαμε τις λεπτομέρειες ενός προϊόντος, θα παρατηρήσουμε ότι στην απόκριση συμπεριλαμβάνεται και το property* Category *του προϊόντος με τιμή null, πρόκειται για navigation property, και που πολλές φορές δεν θέλουμε, δημιουργείται στις json κυρίως αποκρίσεις. Αυτό μπορεί να γίνει ανά περίπτωση με attribute ή να δηλωθεί στο program.cs και να ισχύει για όλες τις αποκρίσεις.*

//[JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]  
public Category Category { **get**; **set**; }  
  
builder.Services.Configure<JsonOptions>(**options** => {  
 **options**.**JsonSerializerOptions**.DefaultIgnoreCondition =   
 **System**.Text.Json.Serialization.**JsonIgnoreCondition**.**WhenWritingNull**;  
});

*δείτε και εδώ:  
<https://learn.microsoft.com/en-us/dotnet/api/system.text.json.serialization.jsonignorecondition?view=net-6.0>*

* **Dealing with Related Data**

*Στην περίπτωση, που θέλουμε να συμπεριλάβουμε και τα navigation properties τότε χρησιμοποιούμε την μέθοδο Includes όπως φαίνεται πιο κάτω.*

[ApiController]  
[Route("api/[controller]")]  
public **class** **CategoriesController** : **ControllerBase** {  
 **private** DataContext \_context;  
  
public **CategoriesController**(DataContext context) {  
 \_context = context;  
 }  
  
 // api/categories/1  
 [HttpGet("{id}")]  
public **async** Task<Category> **GetCategory**(**long** id) {  
 Category category = **await** \_context.Categories  
 .Include(c => c.Products).FirstAsync(c => c.Id == id);  
 **if** (category.Products != null) {  
 **foreach** (Product product **in** category.Products) {  
 product.Category = null;  
 }  
 }  
 return category;  
 }  
}

* **Supporting the HTTP PATCH Method**

*Αν θέλουμε να αλλάξουμε μόνο ένα property/field σε μια κλάση που αντιπροσωπεύει πίνακα, δηλαδή σε ένα entity class, τότε μπορούμε να χρησιμοποιήσουμε την μέθοδο PATCH.*

*Θα χρειαστεί να εγκαταστήσουμε, αν δεν την έχουμε εγκατεστημένη το Package*

Microsoft.AspNetCore.Mvc.NewtonsoftJson 6.0.5

Και να προσθέσουμε στα service της εφαρμογής, στο program.cs

builder.Services.AddControllers().AddNewtonsoftJson();

builder.Services.Configure<MvcNewtonsoftJsonOptions>(opts => {

opts.SerializerSettings.NullValueHandling

= Newtonsoft.Json.NullValueHandling.Ignore;

});

*Και ένα παράδειγμα patch action*

[HttpPatch("{id}")]

public async Task<Category?> PatchSupplier(long id,

JsonPatchDocument< Category > patchDoc)

{

Category? c = await context. Categories.FindAsync(id);

if (c != null) {

patchDoc.ApplyTo(c);

await context.SaveChangesAsync();

}

return c;

}

*Kai to patch json που στέλνουμε*

[{"op":"replace","path": "Name","value": "Φρούτα"}]

* **Specifying an Action Result Format**

*Εκτός από json μπορούμε να επιστρέψουμε και άλλες μορφοποιήσεις, όπως π.χ. xml*

*Χρειάζεται να προσθέσουμε στους controllers την δυνατότητα αυτή στο* program.cs

builder.Services.AddControllers()

.AddNewtonsoftJson()

.AddXmlDataContractSerializerFormatters();

*Και να προσθέσουμε την επιλογή content headers στα actions ως εξής:*

[HttpGet("{id}")]  
**[Produces("application/json", "application/xml")]**  
public **async** Task<Category> **GetCategory**(**long** id)

*Και για να δεχτούμε κάποια μορφοποίηση*

// api/products  
[HttpPost]  
**[Consumes("application/xml")]**  
public **async** Task<IActionResult> **SaveProduct**([FromBody] Product product)

Χρόνος 60 λεπτά

## MVC

*Εισαγωγή στην αρχιτεκτονική του MVC (Model View Controller), routing, view models, razor syntax, view data, temp data (αποφυγή καταχρήσεων), αυτόματη επίλυση των namespaces για τις κλάσεις που χρησιμοποιούνται στα views, εισαγωγή στην έννοια και την χρήση των layouts (όχι την μορφοποίηση, αλλά την χρήση τους, την μορφοποίηση θα την δούμε σε ιδιαίτερο course για το bootstrap css framework και του scss preprocessor.*

* **Getting Started with Views**

*Σε αυτό το κεφάλαιο, θα ασχοληθούμε τη μηχανή προβολής Razor, η οποία είναι υπεύθυνη για τη δημιουργία αποκρίσεων HTML που μπορούν να εμφανιστούν απευθείας στον χρήστη (σε αντίθεση με τις αποκρίσεις JSON και XML, οι οποίες συνήθως χρησιμοποιούνται από άλλες εφαρμογές). Οι προβολές (Views) είναι αρχεία που περιέχουν εκφράσεις C# και κομμάτια HTML που υποβάλλονται σε επεξεργασία από τη μηχανή προβολής για τη δημιουργία αποκρίσεων HTML.*

*Οι αποκρίσεις των Web Services δεν διαφέρουν από τις αποκρίσεις των Controllers που γίνονται μέσω των Actions και των Views, και οι δύο τρόποι μοιράζονται κοινά χαρακτηριστικά. Θα ρυθμίσουμε την εφαρμογή ώστε να χρησιμοποιεί Controllers και Views.*

*builder.Services.AddControllersWithViews();*

*και επίσης θα πρέπει να προσθέσουμε και το ιδιαίτερο routing που χρειάζονται οι Controllers, προσθέτοντας μετά την εντολή app.MapControllers(); την εντολή:*

*app.****MapControllerRoute****("Default",  
"{controller=Home}/{action=Index}/{id?}");*

*Στην συνέχεια δημιουργούμε τον HomeController που κληρονομεί από την κλάση Controller που με την σειρά της κληρονομεί από την κλάση ControllerBase που χρησιμοποιούσαμε στους Controllers των Web Services.*

*Τελευταία δημιουργούμε το directory Views, στο οποίο από σύμβαση αποθηκεύονται όλα τα αρχεία .cshtml που είναι αρχεία που περιέχουν html και C# κώδικα. Το Mvc περιμένει να βρει μέσα στο dir View ένα dir με το όνομα του Controller και εκεί ένα αρχείο .cshtml με την ονομασία της Action μεθόδου. Στο παράδειγμά μας περιμένει το αρχείο Views/Home/Index.cshtml.*

<!DOCTYPE html>  
<**html**>  
<**head**>  
 <**link** href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />  
</**head**>  
<**body**>  
 <**h6** class="bg-primary text-white text-center m-2 p-2">Product Table</**h6**>  
 <**div** class="m-2">  
 <**table** class="table table-sm table-striped table-bordered">  
 <**tbody**>  
 <**tr**><**th**>Name</**th**><**td**>@**Model?.Name**</**td**></**tr**>  
 <**tr**><**th**>Price</**th**><**td**>@**Model?.Price**.ToString("c")</**td**></**tr**>  
 </**tbody**>  
 </**table**>  
 </**div**>  
</**body**>  
</**html**>

*Το δημιουργούμε όπως φαίνεται στον κώδικα του commit.*

* **Understanding Convention Routing**

*Οι Controllers βασίζονται στη συμβατική δρομολόγηση αντί για το routing μέσω Route attributes. Η σύμβαση σε αυτόν τον όρο αναφέρεται στη χρήση του ονόματος του Controller και του ονόματος της Action μεθόδου για τη διαμόρφωση του συστήματος δρομολόγησης, που ρυθμίσαμε με την εντολή app.MapControllerRoute("Default", "{controller=Home}/{action=Index}/{id?}");. Μπορούμε να την αντικαταστήσουμε με την εντολή:*

*app.MapDefaultControllerRoute();*

*που κάνει ακριβώς την ίδια δήλωση.*

* **Displaying Product Info**

*Η Index μέθοδος του HomeController παρέχει στο αντίστοιχο View ένα αποτέλεσμα τύπου Product. Στην σελίδα έχουμε τις δηλώσεις @****Model?.Name*** *και @****Model?.Price.*** *Το σύμβολο @ δηλώνει στην razor σύνταξη, ότι ακολουθεί C# κώδικας. Μια Action μέθοδος παρέχει στο View πάντα ένα μόνο Object και συνήθως είναι, αντίθετα από εδώ, όχι ένα entity object αλλά ένα ViewModel, που συμπεριλαμβάνει όλα τα πιθανά properties που έχουν τα δεδομένα που θέλουμε να στείλουμε στο View. Εδώ όμως χρησιμοποιούμε για λόγους απλότητας το ίδιο το entity class.*

*Το object που περνά στο View πάντα, ασχέτως τον τύπο του (π.χ. List<T>), αναφέρεται πάντα ως Model και περιέχει όλα τα properties του. Για αυτό τον λόγο τα @****Model?.Name*** *και @****Model?.Price*** *μπορούν να μας δώσουν το όνομα και την τιμή του προϊόντος.*

* **Selecting a View by Name**

*Από σύμβαση η ονομασία του View file είναι ίδιο με αυτό της Action μεθόδου. Όμως έχουμε την δυνατότητα να δηλώσουμε άλλο όνομα Vies file. Δείτε το παράδειγμα:*

public async Task<IActionResult> Index(long id) {  
 **return View("Fruit", await \_context.Products.FindAsync(id));**  
}

*που παρέχει τα δεδομένα στο αρχείο*Views/Home/Fruit.cshtml αντί του Views/Home/Index.cshtml

* **Using Shared Views**

*Όταν η Razor view engine (γιατί η Razor είναι ένα View Engine), αναζητά να εντοπίσει ένα View, κοιτά στο directory View/[controller] και μετά στο Views/Shared. Το μοτίβο (pattern) αναζήτησης σημαίνει ότι τα views που περιέχουν κοινό περιεχόμενο μπορούν να μοιράζονται (shared) μεταξύ των Controllers αποφεύγοντας τις επαναλήψεις.*

*Δημιουργούμε το view Common.cshtml στο directory Views/Shared.*

<!DOCTYPE html>  
<**html**>  
<**head**>  
 <**link** href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />  
</**head**>  
<**body**>  
 <**h6** class="bg-secondary text-white text-center m-2 p-2">**Shared View**</**h6**>  
</**body**>  
</**html**>

Και προσθέτουμε μια Action μέθοδο στον Home Controller.

**public** IActionResult **Common**() {  
 **return View("/Views/Shared/Common.cshtml");**  
}

* **Setting the View Model Type**

*Η κλάση που παράγεται στην Fruit.cshtml προέρχεται από την Razor<****T****> generic method, αλλά η Razor δεν «γνωρίζει» ποια κλάση αντιπροσωπεύει και έτσι αντιμετωπίζεται σαν dynamic κλάση generic type. Αυτό σημαίνει ότι η έκφραση @Model μπορεί να χρησιμοποιηθεί με οποιοδήποτε property ή μέθοδο και αυτό θα οδηγήσει σε σφάλμα, χωρίς όμως ο compiler να σημάνει σφάλμα.*

*Για το λόγο αυτό προσδιορίζουμε στο view ποιο data type αντιπροσωπεύει το generic object που παραλαμβάνει από τον controller συμπεριλαμβάνοντας στην αρχή του αρχείου την έκφραση.*

@model Core.Models.Product

* **Using the \_ViewImports File**

*Όταν δηλώσαμε τον τύπο στο view έπρεπε να δώσουμε και το name space όπου βρίσκεται. Αυτό δεν αποτελεί πρόβλημα όσο έχουμε μια απλή δομή, για τον λόγο χρησιμοποιούμε ένα κοινό αρχείο που βρίσκεται απευθείας μέσα στο directory Views/.*

*Το αρχείο αυτό έχει συγκεκριμένο όνομα: \_ViewImports.cshtml το οποίο κατ’ αρχή περιέχει την δήλωση για το directory που περιέχει τις κλάσεις που θα δηλώνονται στα διάφορα views.*

*@using Core.Models*

*Φυσικά μπορούν να δηλωθούν πολλαπλά namespace*

* **Understanding Razor Syntax**

*Στο πιο κάτω πίνακα είναι οι πιο συνηθισμένες χρήσεις της razor σύνταξης.*

##### **Name Description**

@<expression> This is the basic Razor expression, which is evaluated, and the result it produces is inserted into the response.

@if This expression is used to select regions of content based on the result of an expression.

@switch This expression is used to select regions of content based on the result of an expression.

@foreach This expression generates the same region of content for each element in a sequence. See the “Enumerating Sequences” for examples.

@{ ... } This expression defines a code block. See the “Using Razor Code Blocks” section for an example.

@: This expression denotes a section of content that is not enclosed in HTML elements. See the “Using Conditional Expressions” section for an example.

@try This expression is used to catch exceptions.

@await This expression is used to perform an asynchronous operation, the result of which is inserted into the response.

* **Enumerating in Razor and Using Code Blocks**

*Εδώ βλέπουμε ένα παράδειγμα για την σύνταξη @foreach έκφραση. Προσθέτουμε ένα νέο Action στον Home controller.*

**public** IActionResult **List**() {  
 **return** **View**(context.Products);  
}

*που επιστρέφει μια λίστα από προϊόντα, και για να τα δείξουμε χρησιμοποιούμε την @foreach στο view, ως εξής:*

@model IEnumerable<**Product**>  
  
@{  
 decimal average = Model.Average(p => p.Price);  
}  
  
<!DOCTYPE html>  
<**html** lang="en">  
<**head**>  
 <**title**></**title**>  
 <**meta** http-equiv="content-type" content="text/html;charset=UTF-8" />  
 <**link** href="/lib/bootstrap//css/bootstrap.css" rel="stylesheet" />  
</**head**>  
<**body**>  
 <**div** class="container">  
 <**div** class="row m-2">  
 <**h2** class="text-white bg-info p-2">All Product</**h2**>  
 <**table** class="table table-striped table-bordered">  
 <**thead**>  
 <**tr**>  
 <**th**>Name</**th**>

<**th**>Price</**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 **@foreach (Product p in Model) {**  
 <**tr**>  
 <**td**>@p.Name</**td**>  
 <**td**>@p.Price</**td**>  
 <**td**>@((p.Price / average \* 100).ToString("F1")) % of average</**td**>  
 </**tr**>  
 **}**  
 </**tbody**>  
 </**table**>  
 </**div**>  
 </**div**>  
</**body**>  
</**html**>

* **Using the ViewBag and TempData**

*Οι Action μέθοδοι παρέχουν δεδομένα για επισκόπηση στην razor σελίδα μέσω μιας κλάσης στην οποία συγκεντρώνονται όλα τα απαραίτητα δεδομένα. Αυτές τις κλάσεις, από σύμβαση, τα ονομάζουμε View Models. Όμως μερικές φορές χρειαζόμαστε επιπλέον δεδομένα. Οι Action μέθοδοι μπορούν να χρησιμοποιήσουν ένα dynamic object με την ονομασία ViewBag δημιουργώντας properties απλώς δίνοντας τιμές σε αυτές, όπως με τα objects της JavaScript. Π.χ.*

*Στην Action μέθοδο:*

ViewBag.AveragePrice = await context.Products.AverageAsync(p => p.Price);

*Και στην chtml σελίδα παίρνουμε την τιμή με την χρήση του κώδικα ViewBag.AveragePrice.*

<td>

@Model?.Price.ToString("c") (@(((Model?.Price / ViewBag.AveragePrice) \* 100).ToString("F2"))% of average price)

</td>

*Που μας δίνει ένα αποτέλεσμα π.χ.*

€15.67 (3.05% of average price)

*Άλλος ένα τρόπος είναι η χρήση του Dictionary TempData που παρέχει δεδομένα από ένα request στο άλλο, πράγμα χρήσιμο όταν κάνουμε redirections. Τα δεδομένα του TempData αποθηκεύονται ως cookie εκτός αν έχουμε ενεργοποιήσει την Session οπότε αποθηκεύονται εκεί και «σημειώνονται» για διαγραφεί έχουν διαβαστεί και διαγράφονται όταν το request έχει επεξεργαστεί.*

*Παραδείγματα στον κώδικα.*

***Πότε Χρησιμοποιούμε τo ViewBag***

*Το ViewBag λειτουργεί καλύτερα όταν χρησιμοποιείται για να παρέχει στις View σελίδες μικρές ποσότητες συμπληρωματικών δεδομένων χωρίς να χρειάζεται να δημιουργηθούν νέα View Models για κάθε Action. Το πρόβλημα με τα ViewBag δεδομένα είναι ότι ο compiler δεν μπορεί να ελέγξει τη χρήση των properties σε δυναμικά αντικείμενα, όπως και οι View σελίδες που δεν χρησιμοποιούν την έκφραση @model. Μπορεί να είναι δύσκολο να κρίνουμε πότε πρέπει να δημιουργηθεί μια νέα κλάση τύπου View Model και ο εμπειρικός κανόνας είναι να δημιουργούμε μια νέα κλάση τύπου View Model όταν σε μια Action μέθοδο προσθέτουμε περισσότερες από δύο ή τρία properties στο ViewBag.*

* **Using Layouts**

Οι View σελίδες στα παραδείγματα μέχρι τώρα περιέχουν διπλότυπα στοιχεία που αφορούν τη ρύθμιση του HTML document, τον ορισμό του head section, τη φόρτωση του αρχείου CSS Bootstrap κ.λπ. Η Razor υποστηρίζει **Layouts** templates, που ενοποιούν το κοινό περιεχόμενο σε ένα μόνο αρχείο που μπορεί να χρησιμοποιηθεί από οποιαδήποτε View σελίδα. Τα **Layouts** αποθηκεύονται συνήθως στον φάκελο Views/Shared, επειδή συνήθως χρησιμοποιούνται από τις Action μεθόδους περισσότερων του ενός Controller. Από σύμβαση το όνομα του αρχείου είναι \_Layout.cshtml. Παράδειγμα απλού Layout.

<!DOCTYPE html>

<html>

<head>

<link href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />

</head>

<body>

<h6 class="bg-primary text-white text-center m-2 p-2">Shared View</h6>

**@RenderBody()**

</body>

</html>

Οι View σελίδες που χρησιμοποιούν το Layout θα περιέχουν πλέον μόνο το τμήμα html και razor κώδικα που θα συμπεριληφθεί στην θέση που θα καλέσουμε την μέθοδο @RenderBody() στο Layout.

Στις View σελίδες θα δηλώσουμε την χρήση του Layout με την έκφραση:  
@{ Layout = "\_Layout"; }, στην αρχή της σελίδας.

* **Using the \_ViewStart File**

*Για να αποφύγουμε να δηλώνουμε το layout σε κάθε σελίδα View, χρησιμοποιούμε το αρχείο \_ViewStart.cshtml που τοποθετείται απευθείας στο φάκελο Views/, και συνήθως περιέχει μια μόνο δήλωση:*

@{ Layout = "\_Layout"; }

* **Overriding and Disabling Layouts**

*Όποια View σελίδα δεν ορίζει κάποιο άλλο layout, θα χρησιμοποιείται το συγκεκριμένο. Μπορούμε επίσης σε κάποιες View σελίδες που απαιτούν άλλο Layout, αλλά αυτό δεν θα χρησιμοποιηθεί αλλού, να ορίσουμε σε αυτή την σελίδα ότι δεν θέλουμε layout με την δήλωση:*

@{ Layout = null; }

*Μεγάλη προσοχή χρειάζεται η θέση στον κώδικα του layout όπου θα εισέλθουν τα δεδομένα από τις View σελίδες, εάν είναι σε μικρό «βάθος» του κώδικα θα χρειαζόμαστε να ξαναγράφουμε τον ίδιο κώδικα σε πολλές σελίδες View, αν είναι σε μεγάλο «βάθος», τότε θα βρεθούμε στην ανάγκη να χρησιμοποιήσουμε πολλαπλά layout.*

* **Using Layout Sections**

Η Razor View engine υποστηρίζει την έννοια των ενοτήτων (section), που μας επιτρέπουν να ορίσουμε περιοχές περιεχομένου σε ένα layout. Τα section παρέχουν μεγαλύτερο έλεγχο ως προς το ποια μέρη της View σελίδας εισάγονται στο layout και πού τοποθετούνται.

Τα section δηλώνονται σε μια σελίδα View με την έκφραση @section ακολουθούμενη από το όνομα του section, π.χ.

**@section Header** {

Product Information

}

Και στο Layout χρησιμοποιούμε την έκφραση:

@RenderSection("Header", **false**)

To flag false δηλώνει ότι το section δεν είναι υποχρεωτικό να δηλώνεται σε κάθε σελίδα, και θα χρησιμοποιείται μόνο εφ’ όσον ορίζεται.

* **Using Partial Views**

Συχνά θα χρειαστεί να χρησιμοποιήσουμε το ίδιο σύνολο στοιχείων και εκφράσεων HTML σε πολλά διαφορετικά μέρη. Τα Partial Views περιέχουν τμήματα περιεχομένου που θα συμπεριληφθούν σε άλλες View σελίδες για να παράγουν σύνθετα response χωρίς διπλότυπο κώδικα.

<https://learn.microsoft.com/en-us/aspnet/core/mvc/views/partial?view=aspnetcore-6.0>

Για να τα χρησιμοποιήσουμε πρέπει να το δηλώσουμε στο \_ViewImports.cshtml ως εξής:

@using WebApp.Models

**@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers**

Στην πραγματικότητα αυτό που δηλώνουμε, είναι ότι επιθυμούμε την χρήση όλων των Tag Helpers που διαθέτει η Razor Engine, και όχι μόνο των partials.

Στα partials μπορούμε να «περάσουμε» από την View σελίδα την κλάση του View Model της σελίδας ή ένα μόνο property ή ακόμη και δεδομένα από το ViewBag, ViewData κτλ object, π.χ.

@foreach (Product p in Model){

<partial **name**="\_RowPartial" **model**="p" **view-data**="ViewData" />

}

* **HTML and JSON Encoding**

*Εάν στα δεδομένα του View Model περιλαμβάνονται δεδομένα σε μορφή string που όμως είναι ήδη μορφοποιημένα ως html τότε θα πρέπει, αν θέλουμε να εμφανίζονται ως html, να χρησιμοποιήσουμε την έκφραση:*

@Html.Raw(Model)

*Και αν θέλουμε μια κλάση να εμφανιστεί ως Json, τότε μπορούμε να χρησιμοποιήσουμε την έκφραση:*

@Json.Serialize(Model)

Χρόνος 120 λεπτά

## Razor Pages

* **Using Razor Pages**

*Οι Razor Pages είναι απλουστευμένοι Controllers με μία συνήθως Action που περιέχουν και τον C# κώδικα και τον κώδικα Razor και Html στο ίδιο αρχείο ή σε δύο εμφωλευμένα αρχεία π.χ.* Index.cshtml *και* Index.cshtml.cs*, τεχνική που συνήθως ονομάζεται code behind.*

*Δίνουν την δυνατότητα να έχουμε γρήγορα αποτελέσματα χωρίς την πολυπλοκότητα του MVC, όταν έχουμε ένα μικρό και απλό project.*

*Για να χρησιμοποιήσουμε τις Razor Pages πρέπει να προσθέσουμε τις ακόλουθες δηλώσεις στο Program.cs*

builder.Services.AddControllersWithViews();

**builder.Services.AddRazorPages();**

var app = builder.Build();

app.MapControllers();

app.MapDefaultControllerRoute();

**app.MapRazorPages();**

app.UseStaticFiles();

* **Understanding Razor Pages Routing**

*Όλες οι Razor Pages αποθηκεύονται, από σύμβαση στο φάκελο* Pages

*Το routing γίνεται ακολουθώντας την δομή στο Pages στο file system*

*Δείτε την σελίδα Pages/Index.cshtml ως Razor Page*

*Οι παράμετροι λαμβάνονται από το query string*

*Στο routing έχουν προτεραιότητα έναντι του ίδιου path με τους controllers.  
Μπορεί να αλλάζει με την εντολή****app.MapRazorPages().Add(b => ((RouteEndpointBuilder)b).Order = 2);***

*Οι Razor Pages εκθέτουν μια δική τους κλάση με την ονομασία* ***[Name]Model*** *που κληρονομεί από την* ***PageModel*** *και μία μέθοδο* ***public void OnGet()*** *ή την* ***public async Task OnGetAsync(long id = 1)****.*

* **Specifying a Routing Pattern in a Razor Page**

*Μπορούμε να ορίσουμε routing παραμέτρους, όπως και στο routing των controllers με την έκφραση π.χ. @page "{id:long?}" στην αρχή της σελίδας.*

*Επίσης μπορούμε να αλλάξουμε το εμφανιζόμενο routing path με την έκφραση π.χ.  
@page "/list/categories" στην αρχή της σελίδας και να κάνουμε overwrite την δομή του αρχείου που θα χρησιμοποιούνταν ως routing path (αντί /Categories/List σε /list/Categories)*

* **Understanding Action Results in Razor Pages**

*Το γεγονός ότι οι Razor pages κληρονομούν από την κλάση PageModel που τους επιτρέπει να χρησιμοποιούν ως return data type το γνωστό μας IActionResult που παρέχει τα δεδομένα με την μέθοδο return Page(), αντίστοιχη με την return View() των controllers.*

* **PageModel ActionResult Methods**

*Έτσι μπορούμε να χρησιμοποιήσουμε μεθόδους όπως οι NotFound(), BadRequest(), Redirect(), RedirectToPage(), RedirectToAction(), κτλ.*

* Razor Pages POST

Οι Razor Pages μπορούν να ορίσουν Action handlers για να ανταποκριθούν σε διαφορετικές HTTP μεθόδους. Η πιο κοινή χρήση είναι ο συνδυασμός Get και Post μεθόδων που μας επιτρέπουν να επεξεργαστούμε τα δεδομένα. Δείτε τα σχετικά παραδείγματα στο αντίστοιχο commit.

* **Selecting a Handler Method**

Επίσης μπορούμε να ορίσουμε και ιδιαίτερες Action μεθόδους που ονομάζονται handler με την εξής σύνταξη OnGet[**Το ιδιαίτερο όνομα**]Async, όπου τα τμήματα του ονόματος OnGet και Async είναι υποχρεωτικά και το τμήμα μεταξύ τους το ιδιαίτερο όνομα που θέλουμε.

Για να καλέσουμε την μέθοδο στο url το query string με την σύνταξη ?handler=το ιδιαίτερο όνομα της μεθόδου

* **Using Partial Views in Razor Pages**

*Στα Razor pages μπορούμε να χρησιμοποιήσουμε Layouts και Partial views με τον ίδιο τρόπο που τους χρησιμοποιούμε στα Razor Views. Για να το κάνουμε θα πρέπει να προσθέσουμε απευθείας στο* dirPage/ *τα αρχεία* \_ViewStart.cshtml *και* \_ViewImports.cshtml *όπως ακριβώς και στα Views.*

* Creating Razor Pages Without Page Models

*Αν ο λόγος που θέλουμε να χρησιμοποιήσουμε ένα Razor Page είναι απλά για να δείξουμε εύκολα δεδομένα* ***που δεν χρειάζονται επεξεργασία****, τότε μπορούμε να χρησιμοποιήσουμε ένα απλουστευμένο τρόπο κωδικοποίησης π.χ.*

*Χρησιμοποιώντας ένα Service στην σελίδα Data.cshtml στο dir Pages/*

@page

**@inject DataContext context;**

<h5 class="bg-primary text-white text-center m-2 p-2">Categories</h5>

<ul class="list-group m-2">

**@foreach (Category c in context.Categories) {**

<li class="list-group-item">@c.Name</li>

}

</ul>

*Δείτε τον τρόπο που γίνεται το Dependency Injection με την έκφραση @inject για να χρησιμοποιήσουμε τον context κλάση.*

Χρόνος 60 λεπτά

## View Components

* **View Components**

*Τα View Components είναι κλάσεις που κληρονομεί από την ViewComponent, και παρέχουν business logic με το στυλ των Actions, χρησιμοποιώντας μία μόνο μέθοδο την Invoke ή IViewComponentResult, για να υποστηρίξουν partial views.*

[*https://learn.microsoft.com/en-us/aspnet/core/mvc/views/view-components?view=aspnetcore-6.0*](https://learn.microsoft.com/en-us/aspnet/core/mvc/views/view-components?view=aspnetcore-6.0)

*και δείτε μια συζήτηση εδώ*

[*https://www.davepaquette.com/archive/2016/01/02/goodbye-child-actions-hello-view-components.aspx*](https://www.davepaquette.com/archive/2016/01/02/goodbye-child-actions-hello-view-components.aspx)

*Μια κλάση* view component*:*

* *Υποστηρίζει την τεχνική* constructor dependency injection
* *Δεν παίρνει μέρος στον κύκλο ζωής του* controller*, και συνεπώς δεν μπορεί να κάνει χρήση των φίλτρων (*[*filters*](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-8.0)*)*

*Ένα* view component *καθορίζει την λογική του ως εξής:*

* *Καλεί την InvokeAsync μέθοδο που επιστρέφει ένα Task<IViewComponentResullt>.*
* *Ή καλεί την* Invcoke *σύγχρονη μέθοδο που επιστρέφει ένα* IViewComponentResullt*.*
* *Τυπικά εγκαινιάζει ένα model και το περνάει σε μια view καλώντας την* [*ViewComponent.View*](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.viewcomponent.view?view=aspnetcore-8.0#microsoft-aspnetcore-mvc-viewcomponent-view) *μέθοδο.*
* *Οι παράμετροι προέρχονται από την καλούσα μέθοδο, όχι από το HTTP. Δεν υπάρχει model binding.  
  (Καλούμε την* Invoke() *ή την* @Component.InvokeAsync() *μέσα σε μια view. Δεν υπάρχει ανάγκη για model binding σε αυτό το σημείο καθώς οι παράμετροι δεν έρχονται από το HTTP Request. Το view component καλείται απευθείας από την C#. No model binding σημαίνει ότι μπορούμε να έχουμε overloaded invoke μεθόδους με διαφορετικές παραμέτρους).*
* *Δεν είναι προσεγγίσιμο απευθείας όπως ένα endpoint. Τυπικά καλείται μέσα σε μια view. Ένα view component δεν χειρίζεται ποτέ ένα request.*
* *Μπορεί να έχει overloaded invoke μεθόδους με διαφορετικές παραμέτρους, παρά να βασίζεται σε λεπτομέρειες του τρέχοντος request.*

*Για να τα υποστηρίξει η εφαρμογή μας, χρειάζεται να έχουμε δηλώσει στο* \_ViewImports *την χρήση των* Tag Helpers *(γιατί με την μορφή των html tag elements θα τα εισάγουμε στις view σελίδες), με τις πιο κάτω δηλώσεις:*

@using Core.Models

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

**@addTagHelper \*, [ProjectName]**

*Π.χ.* CoreTest*π.χ.**ως όνομα του project μας.*

*Στην συνέχεια μπορούμε να δημιουργήσουμε ένα παράδειγμα View Component ως εξής:*

//[ViewComponent(Name = "JuicyFruit")]

public class **ProductListingViewComponent** : **ViewComponent**

{

private DataContext \_context;

public IEnumerable<Product> Products;

public ProductListingViewComponent(DataContext context) {

\_context = context;

}

public string Invoke() {

return $"There are {\_context.Products.Count()} products";

}

}

* **Returning Views**

*Αντί της Invoke μεθόδου μπορούμε να χρησιμοποιήσουμε την IViewComponentResult που μας παρέχει ότι και η IActionResult, που επιστρέφει Views τα οποία από σύμβαση βρίσκονται στο Components φάκελο είτε στον φάκελο που βρίσκεται το View που τα καλεί είτε, το σύνηθες, στο Shared/Componets.*

*Κάθε Componet έχει το δικό του φάκελο, με το όνομά του, και μία Default.cshtml*

* **Returning HTML Fragments**

*Μπορούμε να επιστρέψουμε απευθείας content χωρίς την χρήση View με την μέθοδο*

return **Content**(“this is a string”)

*ή να επιστρέψουμε html με την μέθοδο*

return new **HtmlContentViewComponentResult**(new **HtmlString**("This is a <h3><i>string</i></h3>"));

* **ViewComponent Properties**

*Δείτε και συγκρίνεται τις μεθόδους που διαθέτει η βασική κλάση ViewComponent και η Controller και ControllerBase.*

* **Providing Context From the Parent View**

*Οι γονικές cshtml σελίδες μπορούν να παρέχουν επιπλέον περιεχόμενο σε ένα view component, είτε παρέχοντας απευθείας δεδομένα ή δίνοντας οδηγίες για τα δεδομένα που θα παραχθούν από αυτό. Τα δεδομένα παρέχουν μέσω των μεθόδων Invoke ή InvokeAsync.*

* **Creating Asynchronous View Components**

*Όλα τα παραδείγματα μέχρι τώρα γινόταν με στοιχεία σύγχρονης εκτέλεσης, τα οποία μπορούμε να τα αναγνωρίσουμε επειδή ορίζουν τη μέθοδο Invoke. Εάν το ViewComponent βασίζεται σε ασύγχρονα API, τότε μπορείτε να χρησιμοποιήσετε την μέθοδο InvokeAsync που επιστρέφει ένα Task. Τότε η Razor σελίδα θα περιμένει να ολοκληρωθούν οι ασύγχρονες εργασίες και στη συνέχεια θα εισαγάγει το αποτέλεσμα στην View σελίδα.*

Χρόνος 60 λεπτά

## Tag Helpers

<https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-6.0>

Οι Tag Helpers επιτρέπουν στον server-side κώδικα να συμμετέχει στη δημιουργία και απόδοση στοιχείων HTML σε αρχεία Razor. Για παράδειγμα, ο ενσωματωμένος ImageTagHelper μπορεί να προσαρτήσει έναν αριθμό έκδοσης στο όνομα της εικόνας. Κάθε φορά που αλλάζει η εικόνα, ο server δημιουργεί ένα νέο μοναδικό αριθμό έκδοσης, έτσι οι clients είναι εγγυημένο ότι θα λάβουν την τρέχουσα εικόνα (αντί για μια αποθηκευμένη στην προσωρινή μνήμη). Υπάρχουν πολλοί ενσωματωμένοι Tag Helpers για κοινές εργασίες - όπως δημιουργία φορμών, συνδέσμων, φόρτωση στοιχείων και άλλα - και ακόμη περισσότεροι είναι διαθέσιμοι δημόσια στον GitHub και ως πακέτα NuGet. Οι Tag Helpers έχουν συνταχθεί σε C# και στοχεύουν στοιχεία HTML με βάση το όνομα στοιχείου, το όνομα ενός attribute ή του γονικού tag. Για παράδειγμα, ο ενσωματωμένος LabelTagHelper μπορεί να στοχεύσει το στοιχείο HTML <label>.

* **Using Anchor Tag Helpers**

<a **asp-controller**="Speaker" **asp-action**="Index">All Speakers</a>

* **Using JavaScript Tag Helpers**

[*https://stackoverflow.com/questions/51363291/how-do-asp-net-cores-asp-fallback-cdn-tag-helpers-work*](https://stackoverflow.com/questions/51363291/how-do-asp-net-cores-asp-fallback-cdn-tag-helpers-work)

<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.js"

**asp-fallback-src**="~/lib/jquery/dist/jquery.js"

**asp-fallback-test**="window.jQuery"

crossorigin="anonymous"

integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">

</script>

* **Using CSS Tag Helpers**

<link rel="stylesheet"

href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/css/bootstrap.css"

**asp-fallback-href**="~/lib/bootstrap/dist/css/bootstrap.css"

**asp-fallback-test-class**="sr-only" asp-fallback-test-property="position"

**asp-fallback-test-value**="absolute"

crossorigin="anonymous"

integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE=" />

* **Working with Image Elements**

<img src="~/images/asplogo.png" asp-append-version="true">

* **Using the Cache Tag Helper**
* **Using the Environment Tag Helper**

Χρόνος 30 λεπτά

## Custom Tag Helpers

Custom Tag Helpers

Narrowing the Scope of a Tag Helper

Widening the Scope of a Tag Helper

Creating Shorthand Elements

Inserting Content Around the Output Element

Inserting Content Inside the Output Element

Χρόνος 60 λεπτά

## Form Tag Helpers

FormController

Index and Result Views

Using Form Tag Helpers

Working with Input Elements

Using the Anti-Forgery Feature

Χρόνος 30 λεπτά

## Model Binding

Understanding Model Binding

Binding Data Types

Specifying a Model Binding Source

Χρόνος 30 λεπτά

## CRUD

CrudController and Index

ViewModel and View

Details

Create

Create POST

Edit

Edit POST

Delete

Χρόνος 30 λεπτά

## Identity ( and Areas )

Adding Identity

Using Areas

Listing Users

User Model

Create User

Create User POST

Edit User

Edit User POST

Delete User

Roles Controller

Role Create

Role Create POST

Role Edit

Role Edit POST

Delete Role

Login

Login POST, Details and Logout

Getting User Details

Authorization

Χρόνος 90 λεπτά

## Solution Architecture

Create multi project web application

Core project

Data project

Service project

Web project

Web Framework project

Χρόνος 90 λεπτά

## Basic Git configuration and commands

Create repository

Commits

.gitignore

Creation and uses of Branches

Versioning

Χρόνος 60 λεπτά

## Security

* JavaScript
* Same-Origin Policy
* Same-Site Concept
* Cookies!!!
* Transport Security (TLS)
* Wrap Up

Χρόνος 60 λεπτά

## Jira

* Intro
* Project Board
* Scrum
* Kanban
* Technicality
* Automation
* Features
* Boards
* Tools
* Settings

Χρόνος 60 λεπτά

## Refactoring

* Use of Visual Studio tools for Refactoring
* Use of ReSharper
* Clean Code Concepts

Χρόνος 30 λεπτά