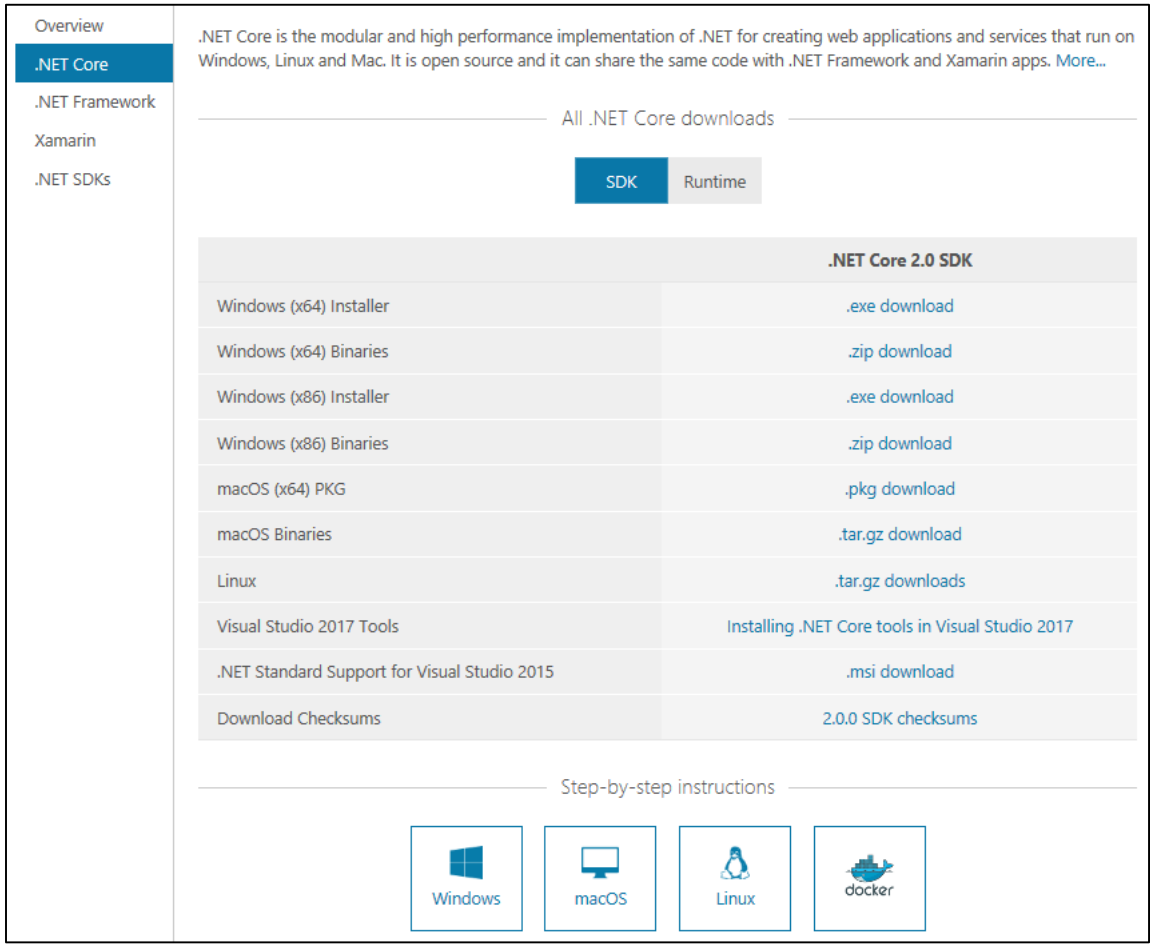
ASP.Net Core MVC

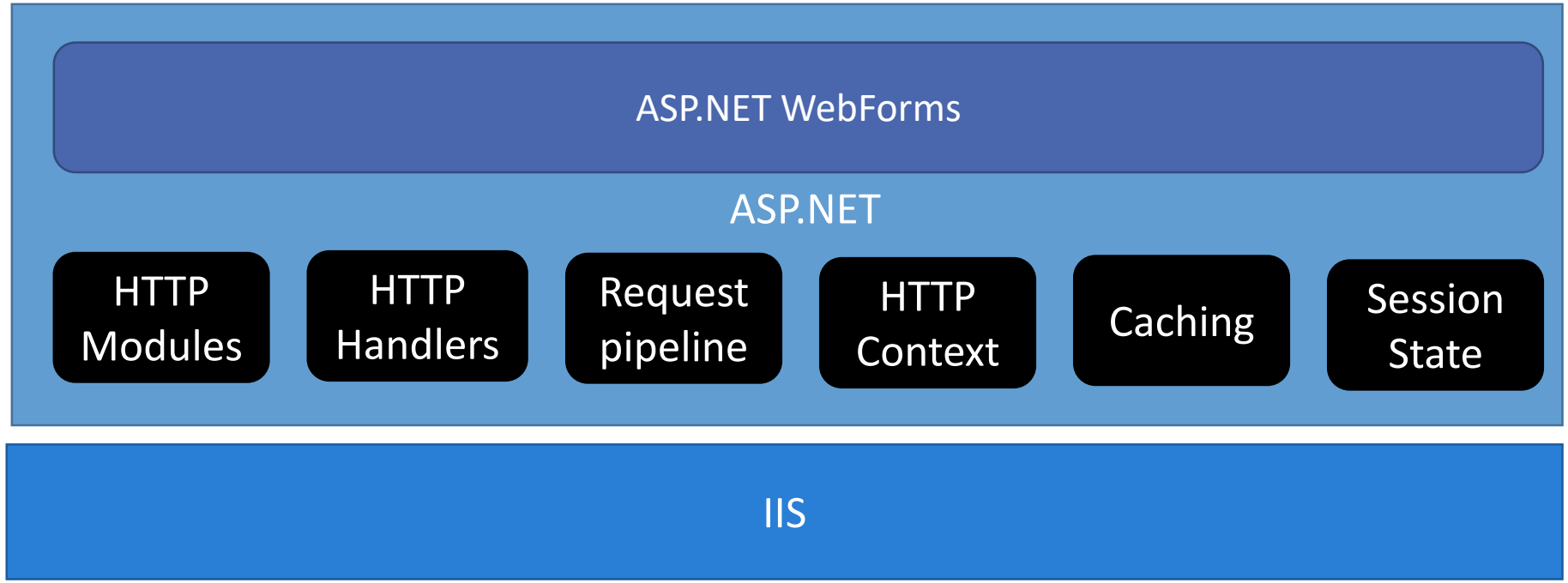
AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Cross-platform server apps

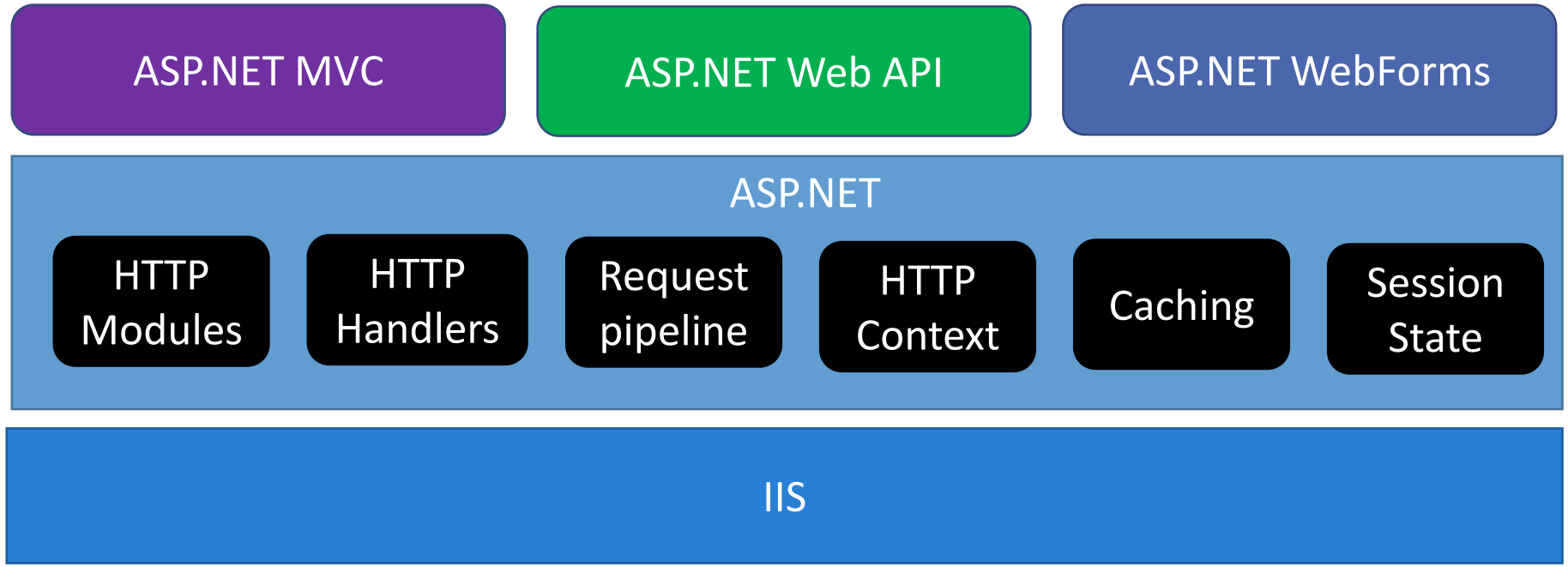
- Develop web sites and services that run on Linux, Windows and macOS with the blazing fast and modular platform provided by .NET Core and ASP.NET Core
  - Open-source
    - On GitHub
  - Cross-platform
  - Optimized
  - Modular



ASP.NET v1

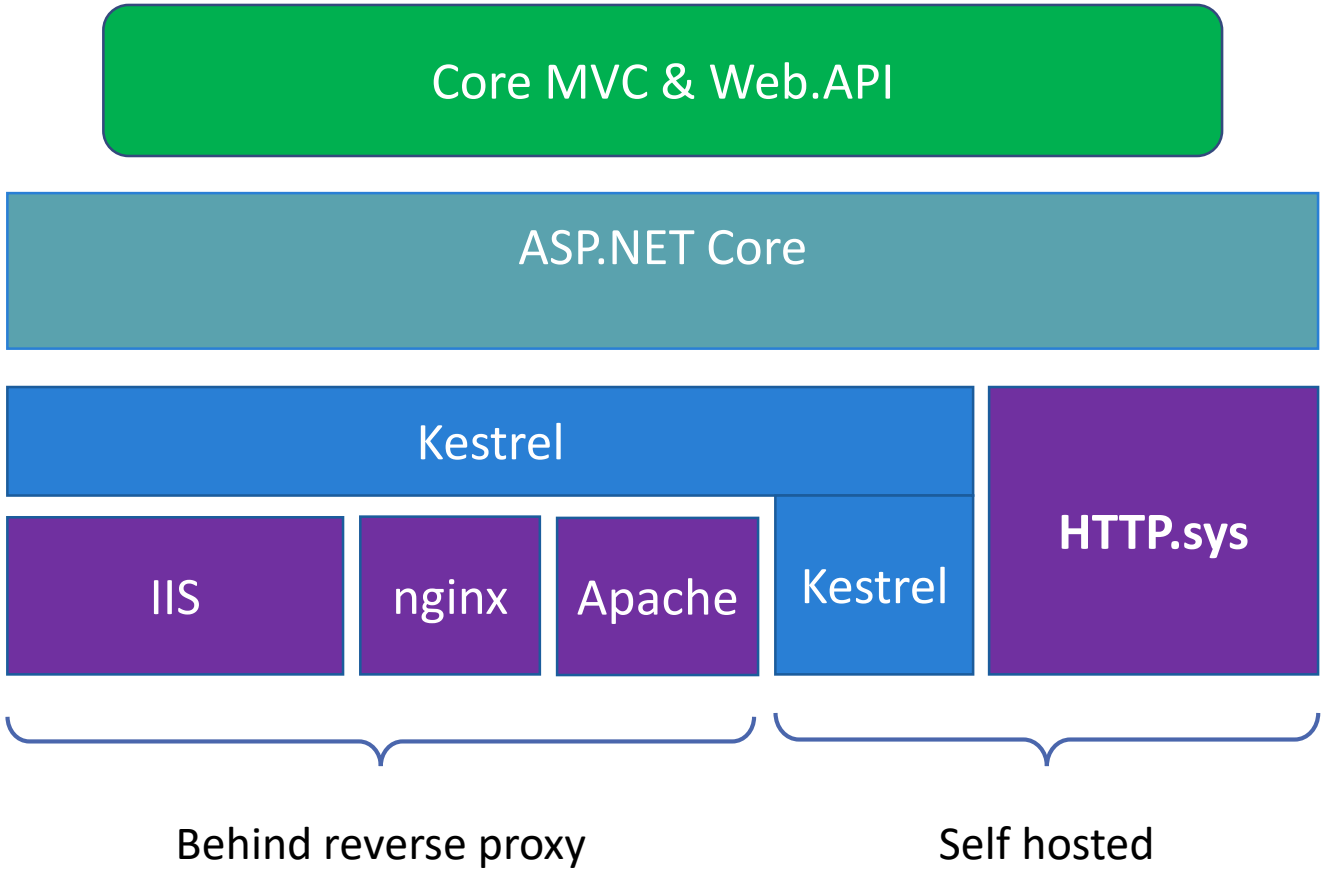

ASP.NET WebForms
ASP.NET
HTTP Modules
HTTP Handlers
Request pipeline
HTTP Context
Caching
Session State
IIS

ASP.NET v4

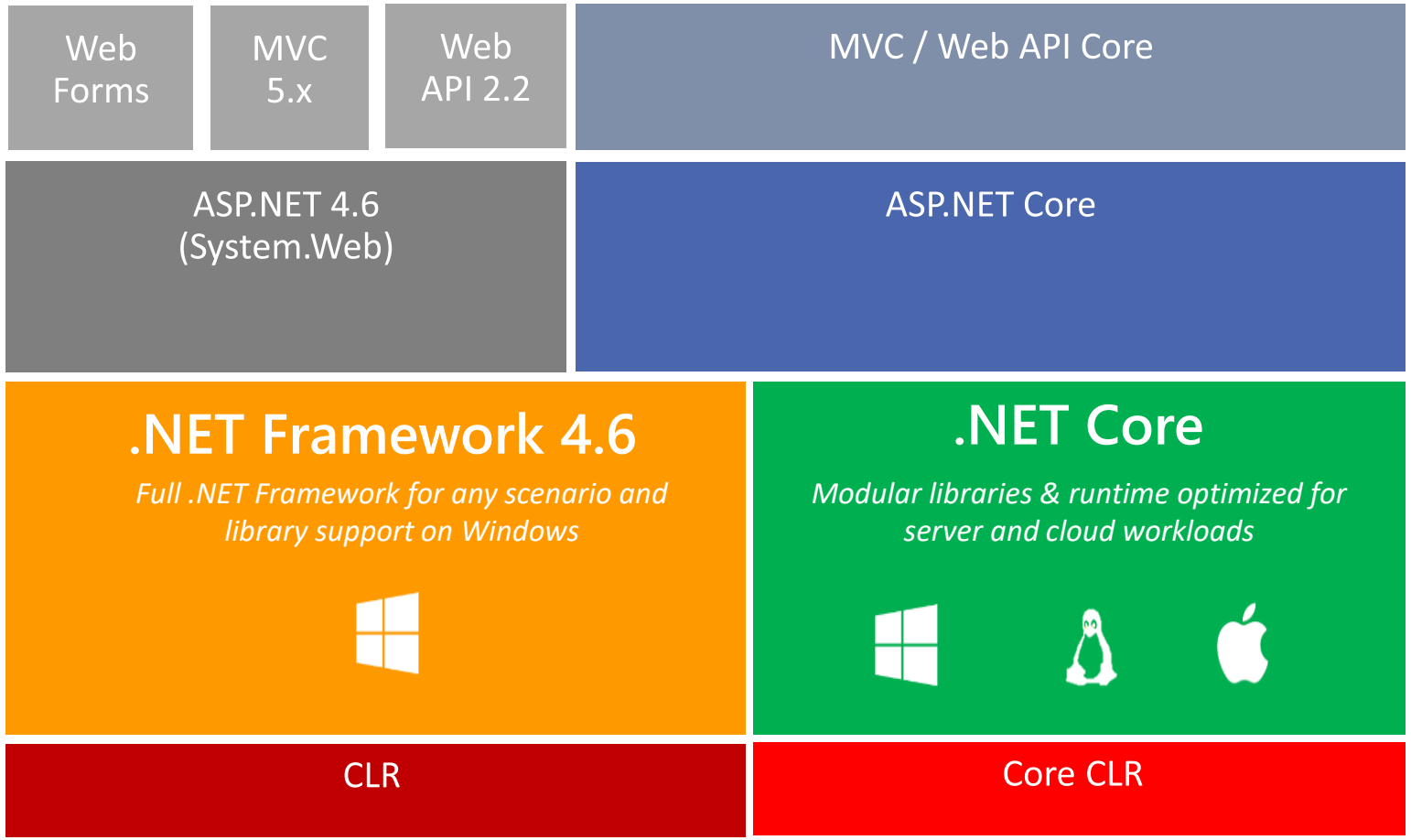

ASP.NET MVC
ASP.NET Web API
ASP.NET WebForms
ASP.NET
HTTP Modules
HTTP Handlers
Request pipeline
HTTP Context
Caching
Session State
IIS

ASP.NET Core

- System.Web is no longer used
- MVC and WebAPI is unified and WebForms is gone
- ASP.NET Core is completely decoupled from the web server environment that hosts the application
- IIS is Windows-only
- Kestrel is a cross-platform web server based on libuv
  - Libuv is a cross-platform asynchronous I/O library build for Node.js

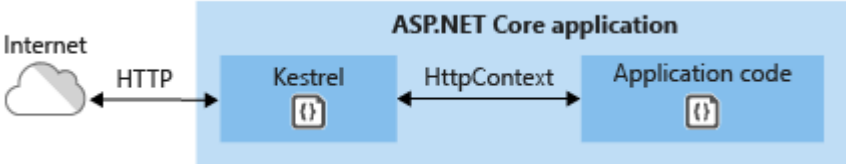


ASP.NET 4.6 and Core in a Nutshell

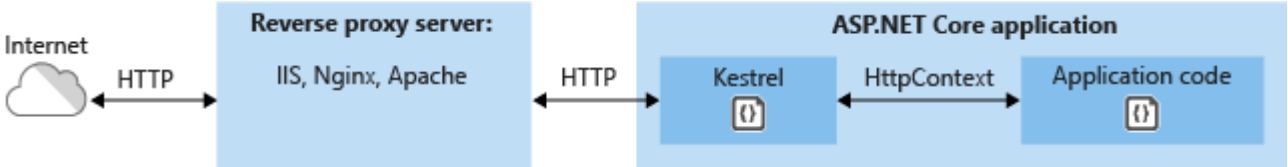


Kestrel - ASP.NET Core Server

- Kestrel is a cross-platform web server (Windows, Mac, and Linux) for ASP.NET Core based on **libuv**, a cross-platform asynchronous I/O library
- Kestrel is the web server that is included by default in ASP.NET Core project templates.



- You can use Kestrel by itself or with a *reverse proxy server*, such as IIS, Nginx, or Apache
- A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling



Application anatomy

- An ASP.NET Core app is simply a console app that creates a web server in its Main method
- The Startup class is where you define the request handling pipeline and where any services needed by the app are configured

```
public class Startup {
  public void ConfigureServices(IServiceCollec
  }

  public void Configure(IApplicationBuilder ap
    if (env.IsDevelopment()) {
      app.UseDeveloperExceptionPage();
    }
    app.Run(async (context) => {
      await context.Response.WriteAsync("Hello
    });
  }
}
```

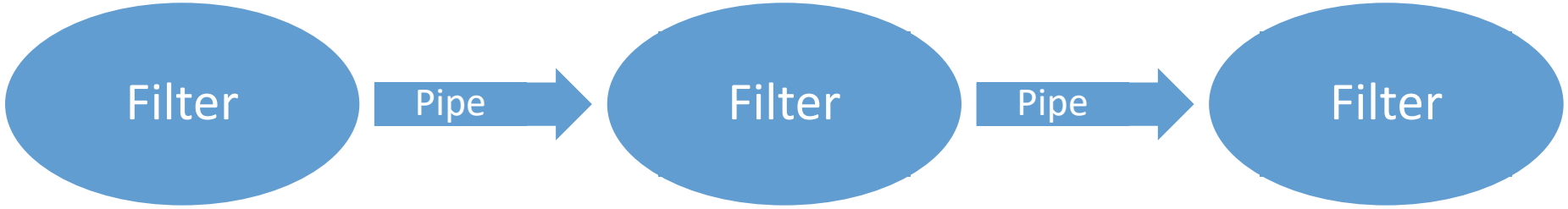
```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebAppDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost
            BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

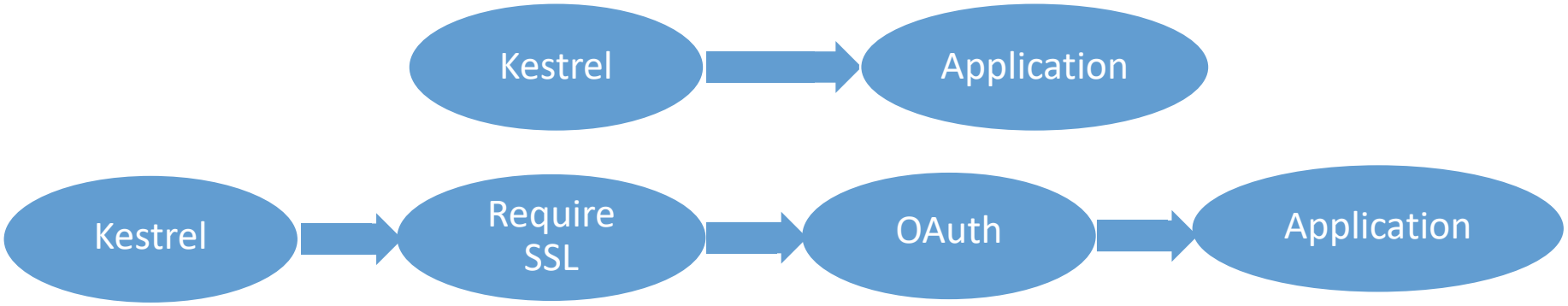
Pipe and Filters

In software engineering, a pipeline consists of a chain of processing elements (processes, threads, coroutines, functions, etc.), arranged so that the output of each element is the input of the next; the name is by analogy to a physical pipeline.
Usually some amount of buffering is provided between consecutive elements.
The information that flows in these pipelines is often a stream of records, bytes or bits, and the elements of a pipeline may be called filters; this is also called **the pipes and filters design pattern**.
From Wikipedia



ASP.NET Core: Pipeline

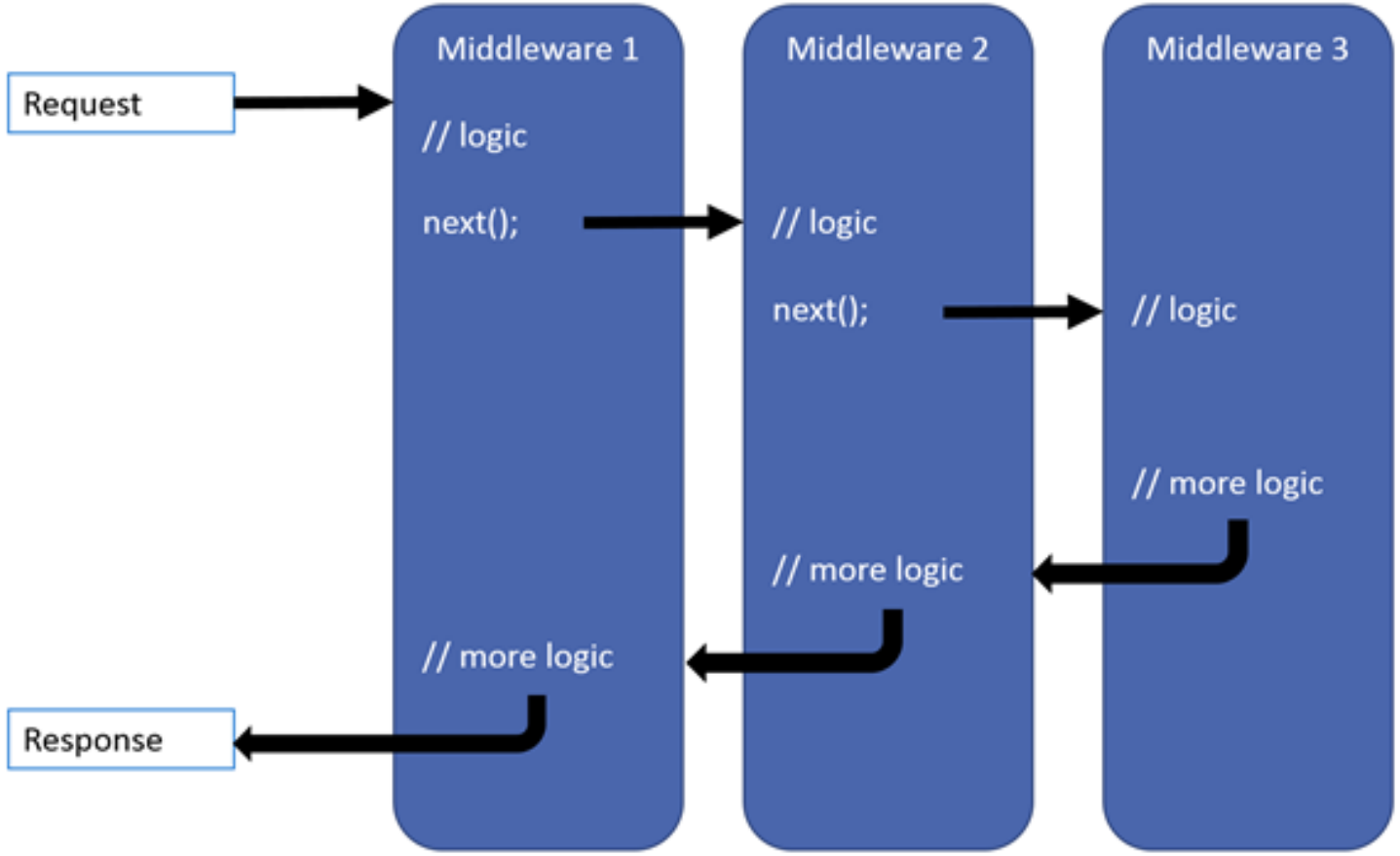
- A context is piped through middleware



- Middleware transforms the request and response
  - Takes a context in
  - Can call next step in pipeline
    - Or not
  - Uniform interface

Creating a Middleware Pipeline

- The ASP.NET request pipeline consists of a sequence of request delegates, called one after the next



The Configure Method

- Request delegates are configured using Run, Map, and Use extension methods on the **IApplicationBuilder** type that is passed into the **Configure** method in the **Startup class**
- An individual request delegate can be specified in-line as an anonymous method, or it can be defined in a reusable class
- These reusable classes are *middleware*, or *middleware components*
- Each middleware component in the request pipeline is responsible for invoking the next component in the chain, or choosing to short-circuit the chain if appropriate

Configuring the Pipeline

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (contex, next) => {
    if (contex.Request.Path == "/middleware" ) {
      contex.Response.ContentType = "text/html";
      contex.Response.StatusCode = 200;
      await contex.Response.WriteAsync("Hello Middleware");
      return;
    }
    await next.Invoke();
  });

  app.Run(async (context) => {
    await context.Response.WriteAsync("Hello World!");
  });
}
```

NuGet

- ASP.NET Core is based on a set of granular and well factored NuGet packages
  - allows you to optimize your app to have just what you need :
    - Reduce the surface area of your application to improve security
    - Reduce your servicing burden
    - Improve performance
- A true pay-for-what-you-use model
- Some Middleware packages (extract):
  - Authentication
  - CORS
  - Response Caching
  - Session
  - StaticFiles
  - OAuth

Not Dependent on Visual Studio

- You can develop an ASP.NET application efficiently by use of command line tools like:
  - **Dotnet**
  - Npm
  - Bower
  - Grunt
  - Gulp
- And cross-platform editors like:
  - Visual Studio Code
  - Atom
- But Visual Studio 2017 with service pack 3 or newer has superior support for ASP .Net Core v. 2.0

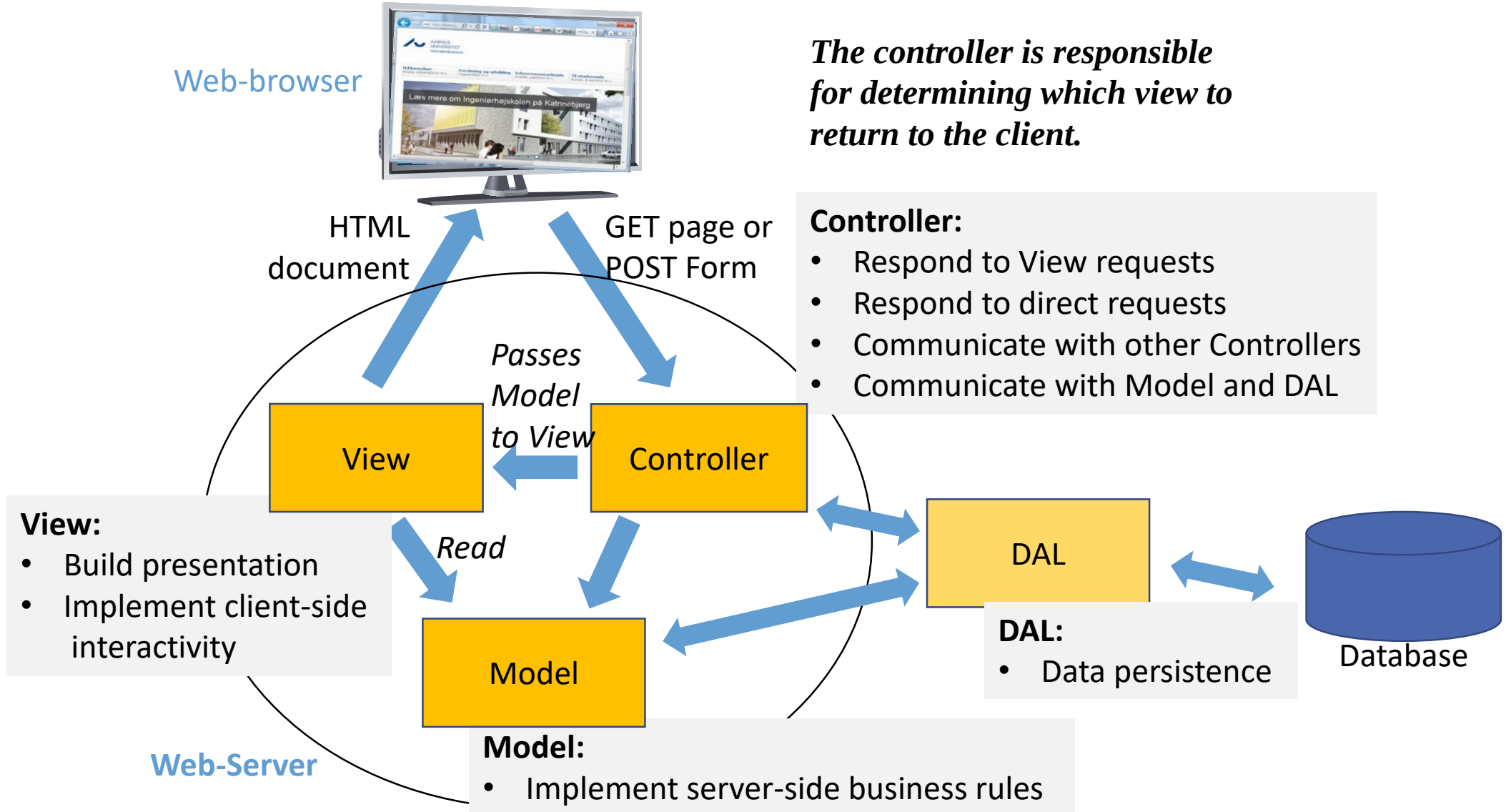
ASP Core MVC

There are several project templates to chose from.

The main types are:

- Razor pages
- ASP.MVC
- ASP Web.API

ASP.NET – MVC - Architecture



The controller is responsible for determining which view to return to the client.

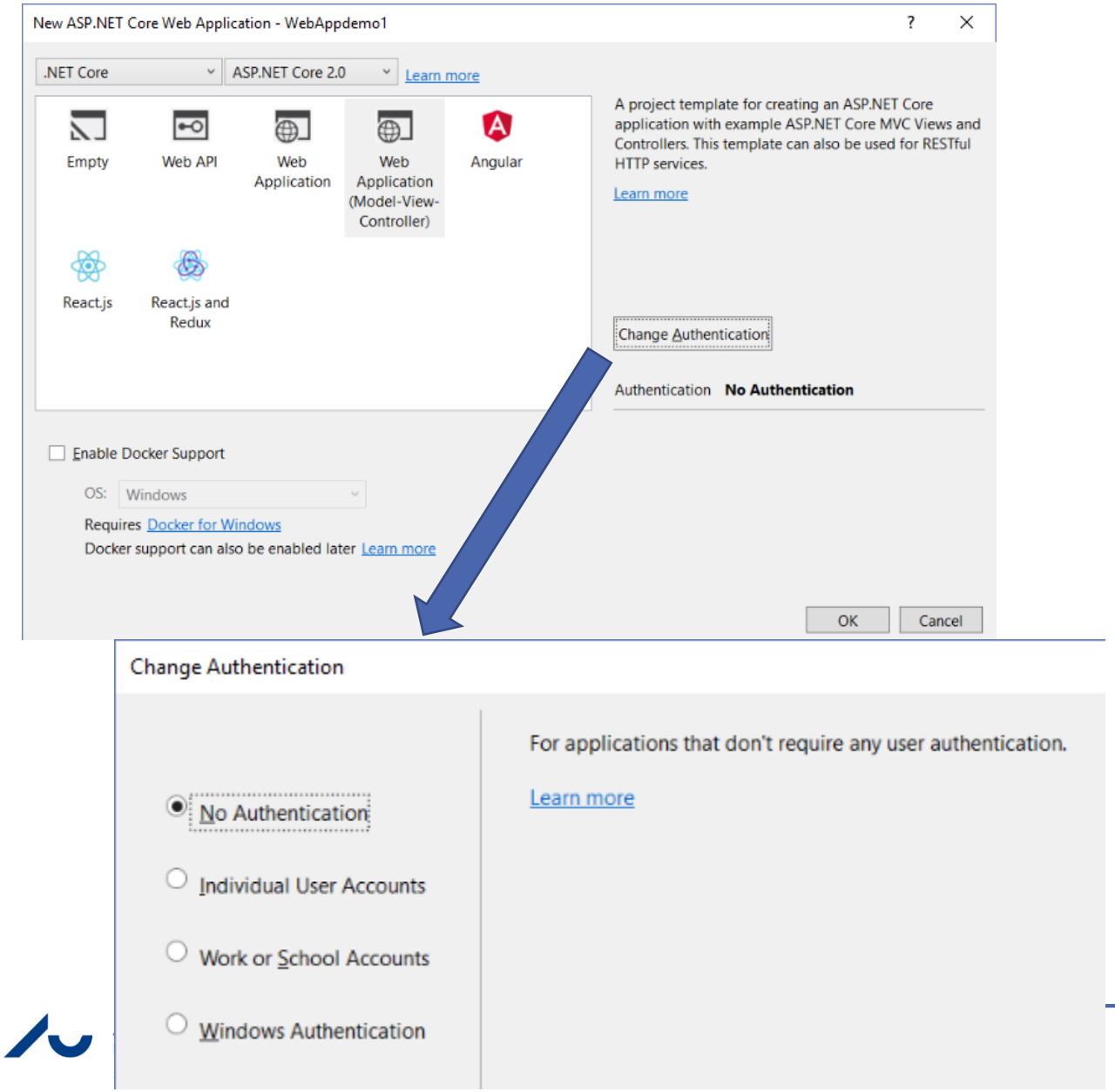
Controller:
- Respond to View requests
- Respond to direct requests
- Communicate with other Controllers
- Communicate with Model and DAL

View:
- Build presentation
- Implement client-side interactivity

DAL:
- Data persistence

Model:
- Implement server-side business rules
- Handle structured data
- Data manipulation

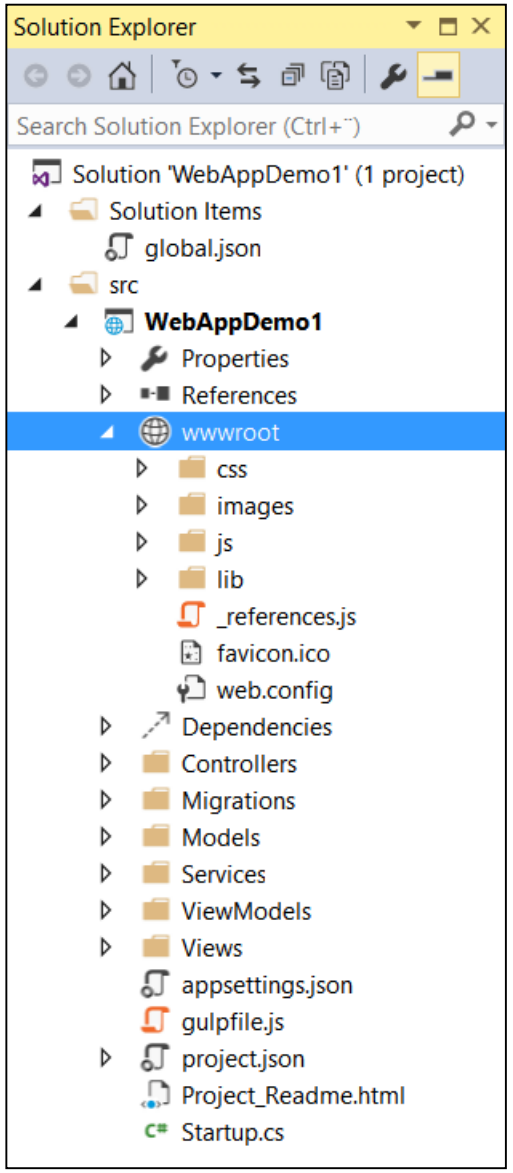
MVC Project Templates in MSVS



- Empty
  - An empty project template
- Web API
  - Generates a project with an example controller for a restfull API
- Web Application
  - Generates a project with Razor Pages without use of controllers
- **Web Application (M – V – C)**
  - Generates a project with controllers and Razor Views
- Angular
- React.js
- *You may install additional templates*

Folder Structure

- WWWroot:
  - Static files are placed in sub folders under the wwwroot folder
- Controllers
  - A controller is the coordinator that is responsible for processing input and then deciding which actions should be performed
- Models
  - Contain classes that represent the core concepts of your application
- ViewModels
  - Contains classes that hold data in a format that is specific to a particular view
- Views
  - Contains the templates used to render your user interface. Each of these templates is represented as a Razor view (a .cshtml file) within a subdirectory named after the controller responsible for rendering that view



The Generated Start Page

Home Page - WebAppdemo

localhost:8664

WebAppdemo1 Home About Contact

ASP.NET Core | Windows Linux OSX

Learn how to build ASP.NET apps that can run anywhere.

Learn More

Application uses

- Sample pages using ASP.NET Core MVC
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

- Add a Controller and View
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet.
- Add client packages using Bower.
- Target development, staging or production environment.

Overview

- Conceptual overview of what is ASP.NET Core
- Fundamentals of ASP.NET Core such as Startup and middleware.
- Working with Data
- Security
- Client side development
- Develop on different platforms
- Read more on the documentation site

Run & Deploy

- Run your app
- Run tools such as EF migrations and more
- Publish to Microsoft Azure Web Apps

© 2017 - WebAppdemo1

Convention-over-Configuration

- Means that the application will execute tasks based on the names of objects
- When you call **http://mysite.com/Account**, MVC will look for an **`AccountController`** object with an **`Index`** method
  - No custom code needs to be written to make this happen
- It's simply the convention that the name of the HTTP URL action corresponds to the name of the controller, and the default view will always be named Index

Controllers

- Controllers are represented as classes that inherit from the Controller base class, where individual methods (known as ***actions***) correspond to individual URLs

```
public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult About()
        {
            ViewData["Message"] = "Your application description page.";

            return View();
        }
}
```

THE VIEW

- About.cshtml

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

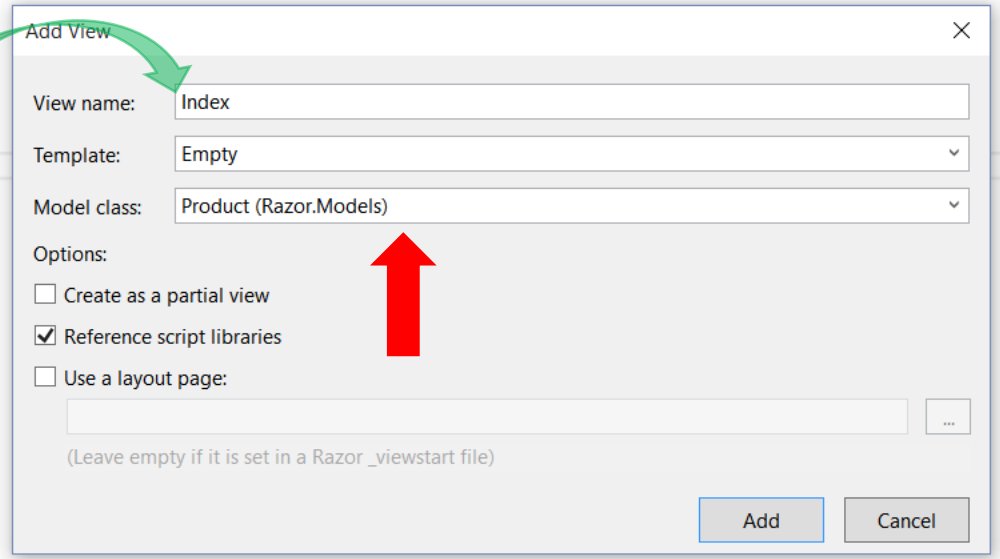
From Controller to View

- Right-click on an action and select Add-View

```
public class HomeController : Controller
{
    Product myProduct = new Product
    {
        ProductID = 1,
        Name = "Kayak",
        Description = "A boat for one person",
        Category = "Watersports",
        Price = 275M
    };

    // GET: Home
    0 references
    public ActionResult Index()
    {
        return View(myProduct);
    }
}
```

Product is a Model class



You can also use the ViewData object (or ViewBag) to transfer data to the view, but properties of the ViewBag are weakly typed (ViewBag is of type dynamic)

MVC - Separation of Concerns

- The responsibilities of Controller and View

Component	Does Do	Doesn't Do
Controller (Action Method)	Pass a (view-) model object to the view	Pass formatted data to the view
View	Use the view model object to present content to the user	Change any aspect of the model object
Model	Contains data and business logic	View logic
VievModel (not always needed)	A composite object holding all the data needed for a view	business logic
DAL	Data persistence	View logic

ASP Routing

Controlling URLs with routing

Controlling Urls with Routing

- ASP.NET MVC decouples the URL from a physical file by making use of URL routing to provide a way to map **URLs without extensions** to controller actions

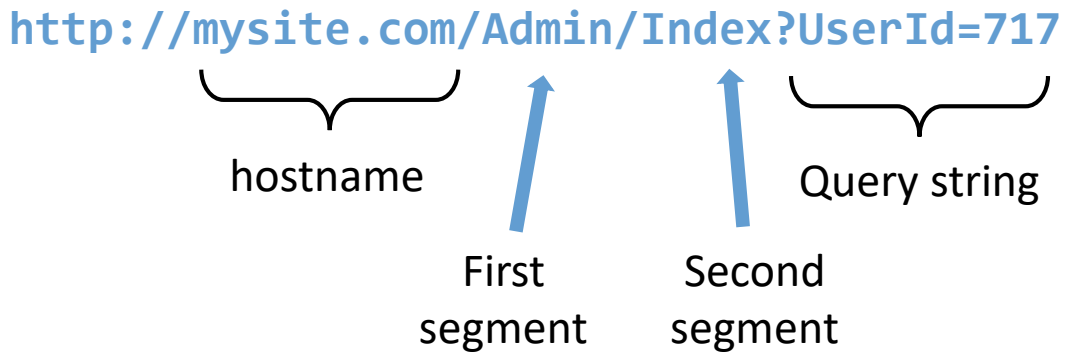
Duties of The Routing System

The routing system has two functions:

- Examine an ***incoming*** *URL* and figure out for which controller and action the request is intended
- Generate ***outgoing*** *URLs*
  - These are the URLs that appear in the HTML rendered from views so that a specific action will be invoked when the user clicks the link
    - at which point, it has become an incoming URL again

URL Patterns

- Each route contains a ***URL pattern***, which is compared to incoming URLs
- If a URL matches the pattern, then it is used by the routing system to process that URL



- URL pattern that maps segments to controllers and actions:
  - The segment variables are expressed using braces

```
{controller}/{action}
```

The Default Route

conventional routing

Extract from Startup.Configure

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

URL	Method called
http://example.com/Users/Edit/5	UsersController.Edit(5)
http://example.com/Users/Edit	UsersController.Edit()
http://example.com/Users	UsersController.Index()
http://example.com	HomeController.Index()

To change the name of the default controller or action simply enter your names

URL Pattern Matching

- By default, a URL pattern will match any URL that has the correct number of segments

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}"
);
```

Request URL	Segment Variables	
	Controller	Action
http://mysite.com/Admin/Index	Admin	Index
http://mysite.com/Index/Admin	Index	Admin
http://mysite.com/Apples/Oranges	Apples	Oranges
http://mysite.com/Admin	*No match—too few segments*	
http://mysite.com/Admin/Index/Soccer	*No match—too many segments*	

Multiple routes

- You can add multiple routes inside UseMvc by adding more calls to MapRoute
- Doing so allows you to define multiple conventions,
  - or to add conventional routes that are dedicated to a specific action

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

References & Links

- Welcome to ASP.NET Core
  https://docs.microsoft.com/da-dk/aspnet/core/
- Getting started with ASP.NET Core MVC and Visual Studio
  https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?tabs=aspnetcore2x
  - On Mac
    https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app-mac/
- **Routing to Controller Actions**
  https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing
- .NET Core command-line interface (CLI) tools
  https://docs.microsoft.com/en-us/dotnet/core/tools/?tabs=netcore2x