

# Designing Data Models & Accessing Data



# Agenda

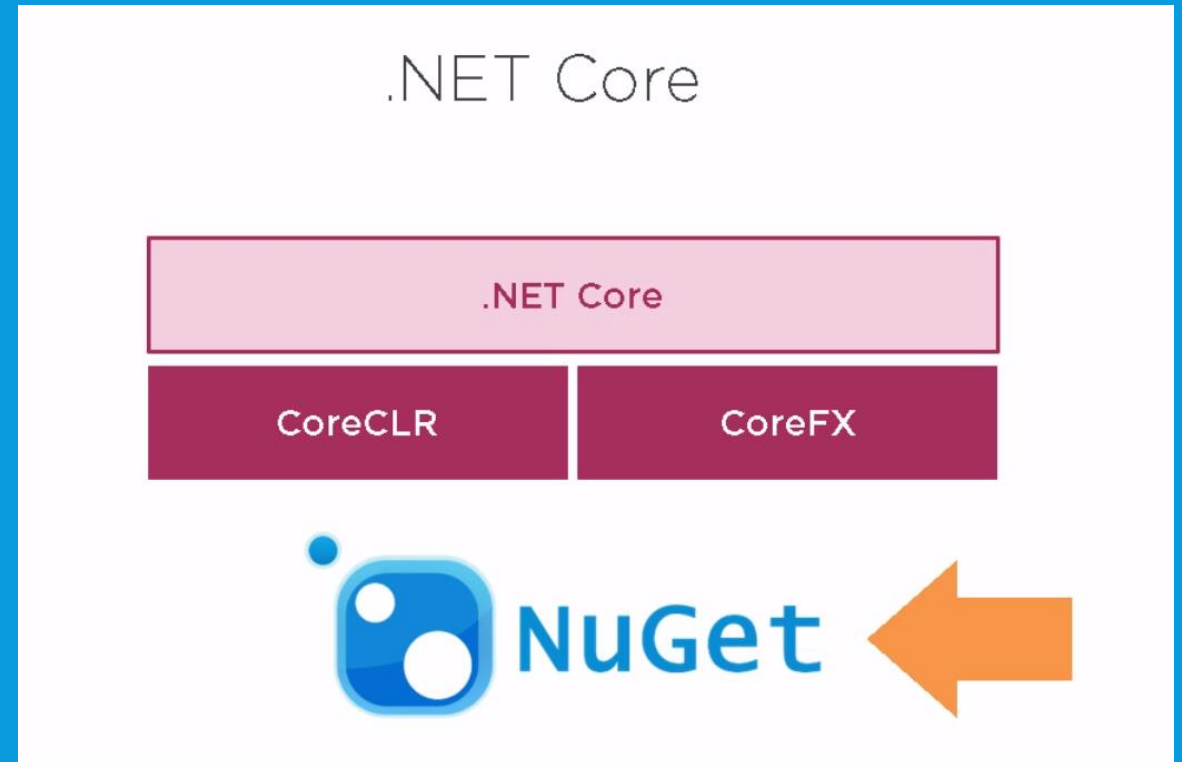
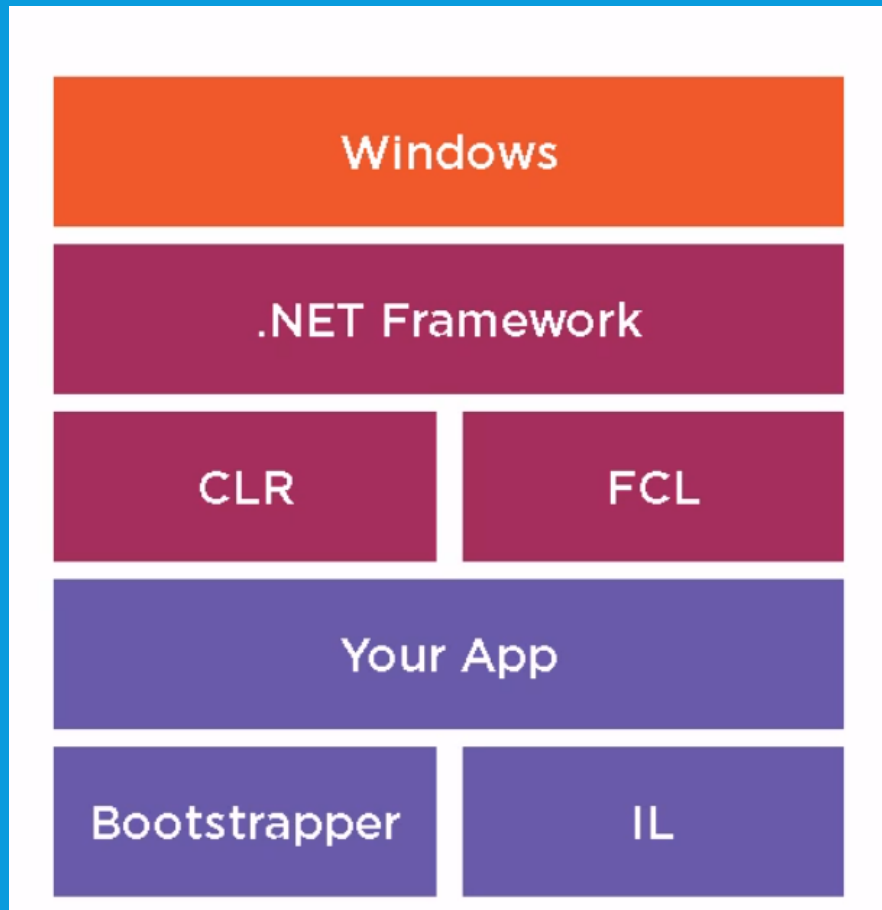
- .NET Core Quickintro
- Implementing Data Models using .NET Core and Entity Framework 2.0
- Angular Project Configuration for Integration with ASP.NET Core
- Implementing Client Side Data Models using Angular & TypeScript
- Comparing Http & HttpClient
- Consuming .NET Core RESTful API using Angulars HttpClient
- Consuming NoSQL DBs using Angulars HttpClient

# .NET Core Quickintro

# Why .NET Core?

- One .NET for all platforms
- Reduce overall footprint
- Run across operating Systems

# Comparing .NET / Core 1.0



# DOTNET CLI

- Command line tool to manage ASP.NET Core
- Can be extended with several add-ins that have to be configured using "tools"-section of project.json
- Used to:
  - Run Projects – dotnet run
  - Publish – dotnet publish
  - Manage EF – dotnet ef ...
  - ....

# \*.proj.cs

- Configuration File for NuGet Packages (actually ms build)
- Contains:
  - .NET Core Target Framework
  - NuGet Runtime Packages
  - DOTNET CLI Extensions

```
<PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
<PackageReference Include="Microsoft.AspNetCore.Diagnostics" Version="2.0.0" />
<PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.0" />
<PackageReference Include="Microsoft.AspNetCore.Server.IISIntegration" Version="2.0.0" />
<PackageReference Include="Microsoft.AspNetCore.Server.Kestrel" Version="2.0.0" />
<PackageReference Include="Microsoft.Extensions.Logging.Console" Version="2.0.0" />
<PackageReference Include="Microsoft.Extensions.Configuration" Version="2.0.0" />
```

# startup.cs

- Constructor – Startup(IHostingEnvironment env)
  - Add JSON config
  - Environment variables
- ConfigureServices using Dependency Injection
  - Add Services like MVC - services.AddMvc();
  - Entity Framework
- Configure – control the ASP.NET pipeline
  - Add Logger
  - FileHandler
  - MVC Routes



# dotnet watch

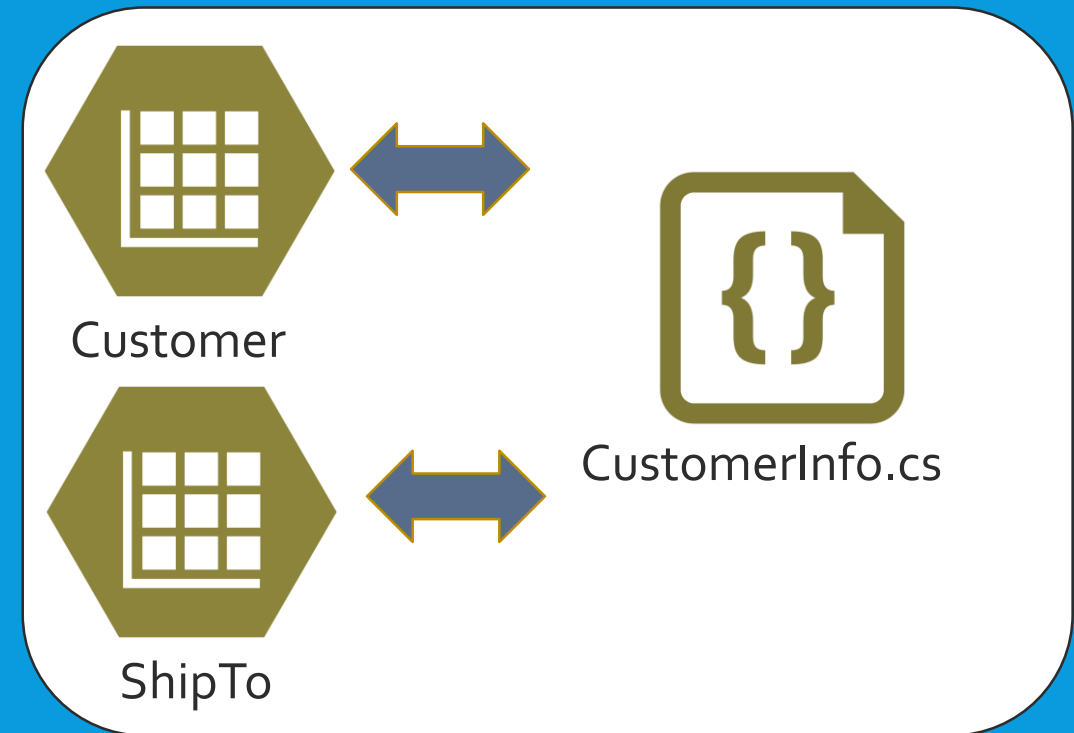
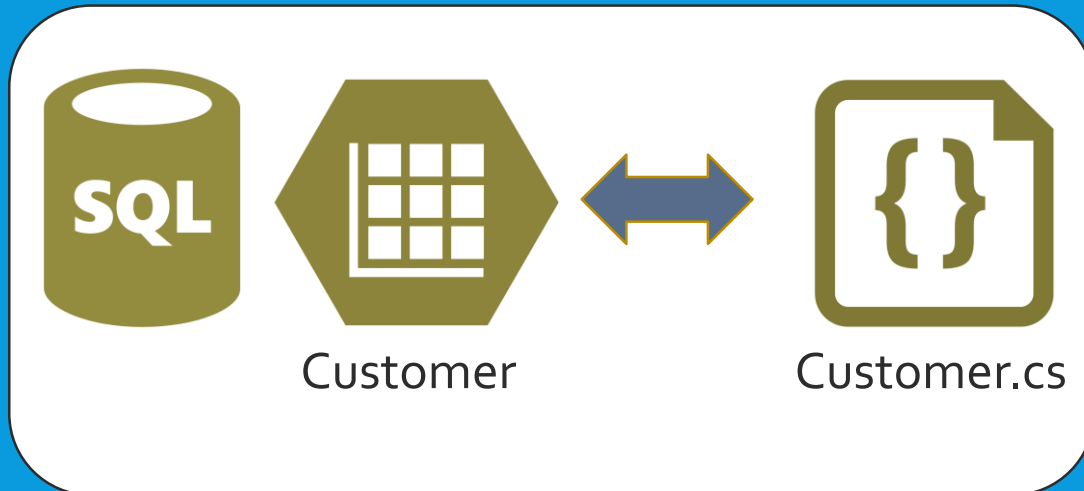
- A development time tool that runs a dotnet command when source files change
- Enables easy F5 debugging in your browser
- Requires Microsoft.DotNet.Watcher.Tools in project.json & dotnet restore afterwards
- Enabled using dotnet watch run | dotnet watch test

```
"tools": {  
  "Microsoft.DotNet.Watcher.Tools": {  
    "version": "1.0.0-preview2-final",  
    "imports": "portable-net451+win8"  
  },  
}
```

# Implementing Data Models using .NET Core & EF 2.0

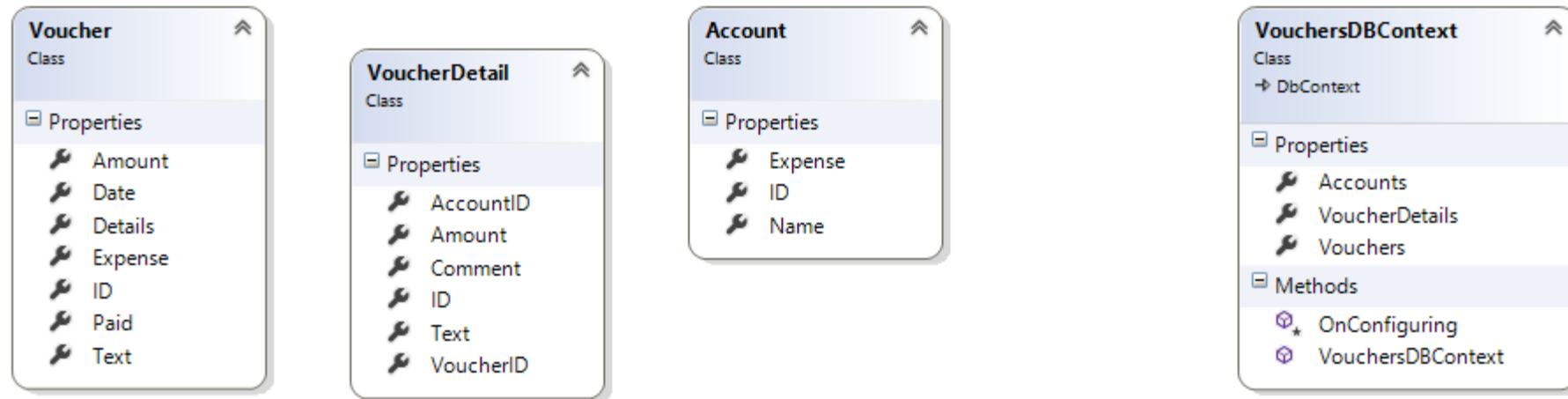
# Entity Framework Core 2.0

- What's an ORM?
- Maps your database types to your code types
- Avoids repetitive data access code



# Model

- Voucher – Our "VoucherHeader"
- VoucherDetail – The individual Voucher Rows
- Account – The accounts used to classify income and expenses



# Build Model

- Build classes representing your data in folder Models
- No special inheritance required
- Related entities should be created using ICollection

```
public class Voucher
{
    public int ID { get; set; }
    public string Text { get; set; }
    public DateTime Date { get; set; }
    public ICollection<VoucherDetail> Details { get; set; }
}
```

# Attributes

- [NotMapped] - Exclude Types / Properties from Data Model
- [Required] – Used for required Properties
- Optional Properties must be implemented using a nullable type (string, int?, ...)
- [MaxLength(500)] – Specifies a max length
- [ConcurrencyCheck] – Checks for concurrency

# DatabaseGeneratedOption

- Represents the pattern used to generate values for a property in the database
  - Computed
  - Identity
  - None

```
public class Account
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int ID { get; set; }

    public string Name { get; set; }

    public bool Expense { get; set; }
}
```

# Foreign Key with DataAnnotations

- ForeignKey – Specifies a ForeignKey
- InverseProperty – Used when there is more than one reference to the same parent entity

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int ID { get; set; }

[Required]
public int VoucherID { get; set; }

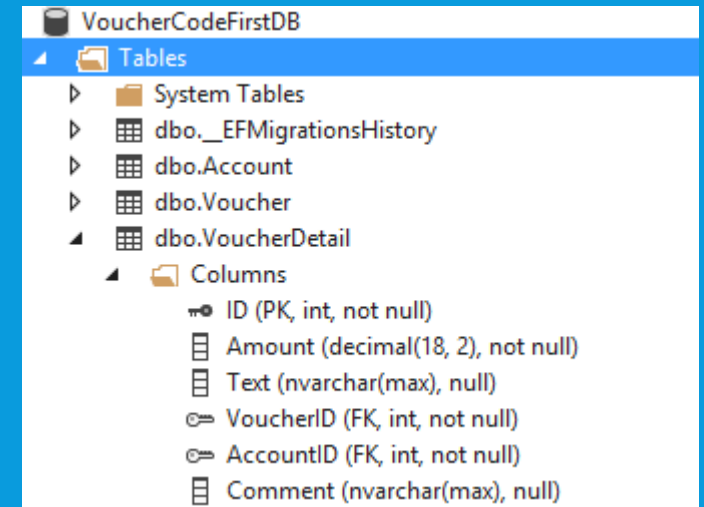
public int AccountID { get; set; }

[ForeignKey("AccountID")]
public virtual Account Account { get; set; }

public string DetailText { get; set; }

public int DetailAmount { get; set; }

public string Comment { get; set; }
```





# Related Entities

- Related entities can be included using INCLUDE Keyword

```
public Voucher Get(Guid id)
{
    return ctx.Vouchers.Include(f => f.Entries).FirstOrDefault(v => v.Id == id);
}
```

- Subqueries on related entities possible

```
var incomeAccts = kpi.GetIncomeAccts();

var items = ctx.Vouchers.Include(v=>v.Entries).Where(
v => v.Entries.Count(e=>incomeAccts.Contains(e.AccountId))>0 &&
v.PaymentDate == null && v.Date.Date <= state.focusDate.Date &&
vgs.Contains(v.VoucherGroupId));
```

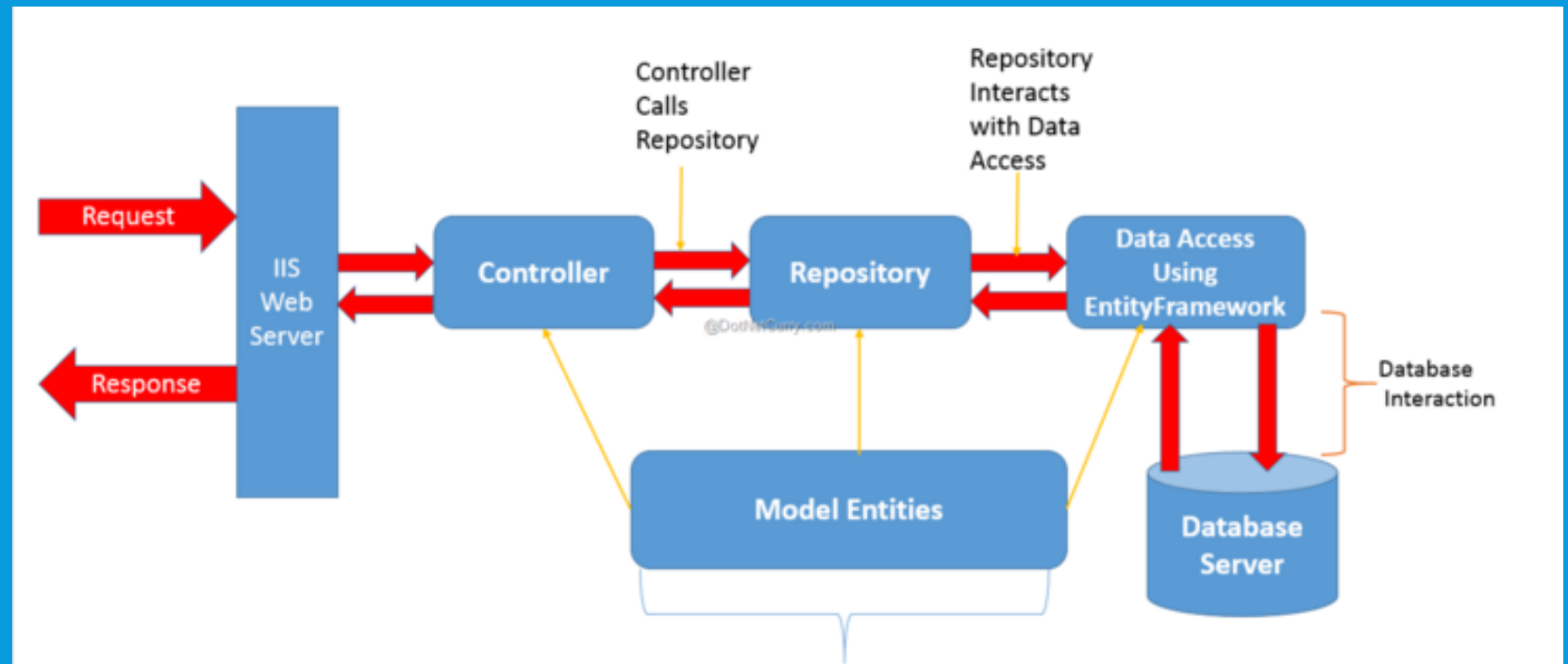
Query on entries



# Repository

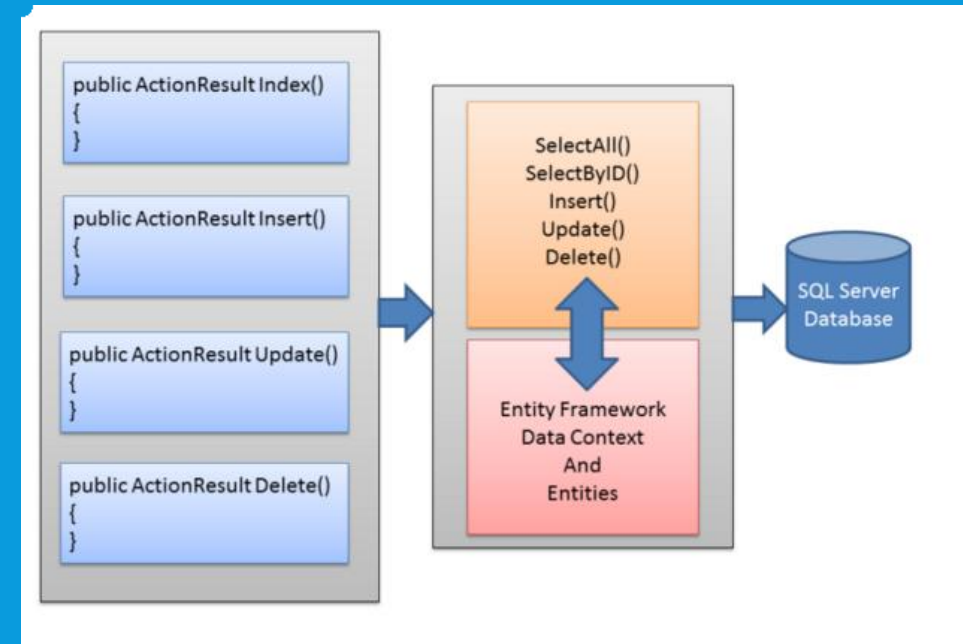
# Repository Pattern

- A Repository acts like a middleman between the application and the data access logic.
- It isolates the data access code from the rest of the application



# Repository Pattern – Im Detail

- CRUD operations in the controller point to corresponding methods in the Repository
- Repositories can include Bussiness Logic



# VoucherRepository

- Consists of
  - IVouchersRepository
  - VouchersRepository
- Must be registered as a service in Startup.cs as Singleton in order to be used in DI

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IVouchersRepository, VouchersRepository>();
}
```

# Database & Data Context

# Build Context

- Implement your DataContext using a class inheriting from DbContext
- Implement entity collections using DbSet<T>

```
public class VouchersDbContext : DbContext
{
    private VouchersConfig config;

    public VouchersDbContext(DbContextOptions<VouchersDbContext> options) : base(options)
    {
    }

    public DbSet<Voucher> Vouchers { get; set; }
    public DbSet<VoucherDetail> VoucherDetails { get; set; }
    public DbSet<BalanceAccount> BalanceAccounts { get; set; }
}
```

# DbContext – Startup.cs

- DbContext must be provided using Startup.cs in ConfigureServices

```
string conStr = Configuration["ConnectionStrings:LocalDBConnection"];
services.AddSingleton(typeof(IConfigurationRoot), Configuration);
services.AddEntityFrameworkSqlServer()
    .AddDbContext<VouchersDbContext>(options => options.UseSqlServer(conStr));
services.AddScoped<IVouchersRepository, VouchersRepository>();
```

- DB Seeding is done in Program.cs

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<VouchersDbContext>();
            DbInitializer.Initialize(context);
        } catch (Exception ex) {}
    }
    host.Run();
}
```

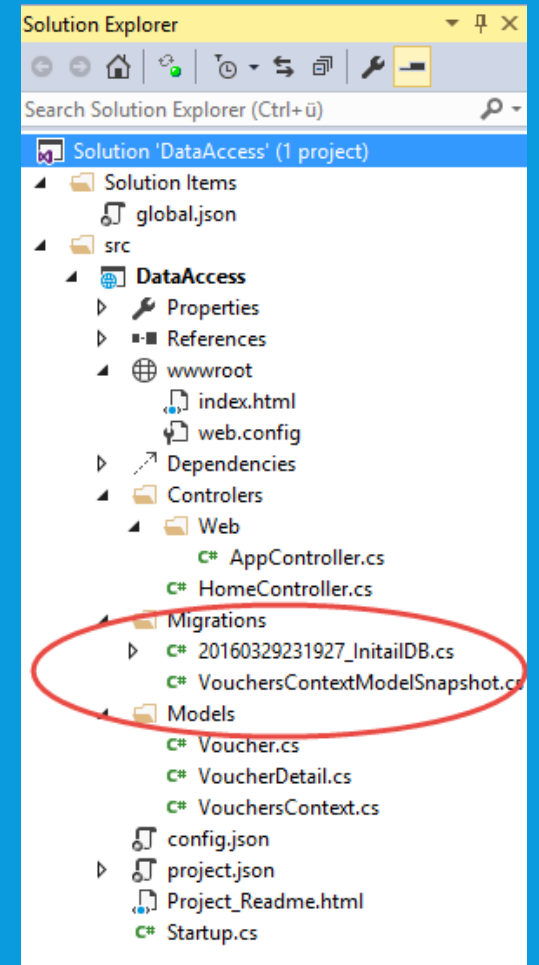


# Migrations

- Migrations are "Versions" of your database
- Reference EFCore.Tools.DotNet in \*.csproj to extend dotnet cli

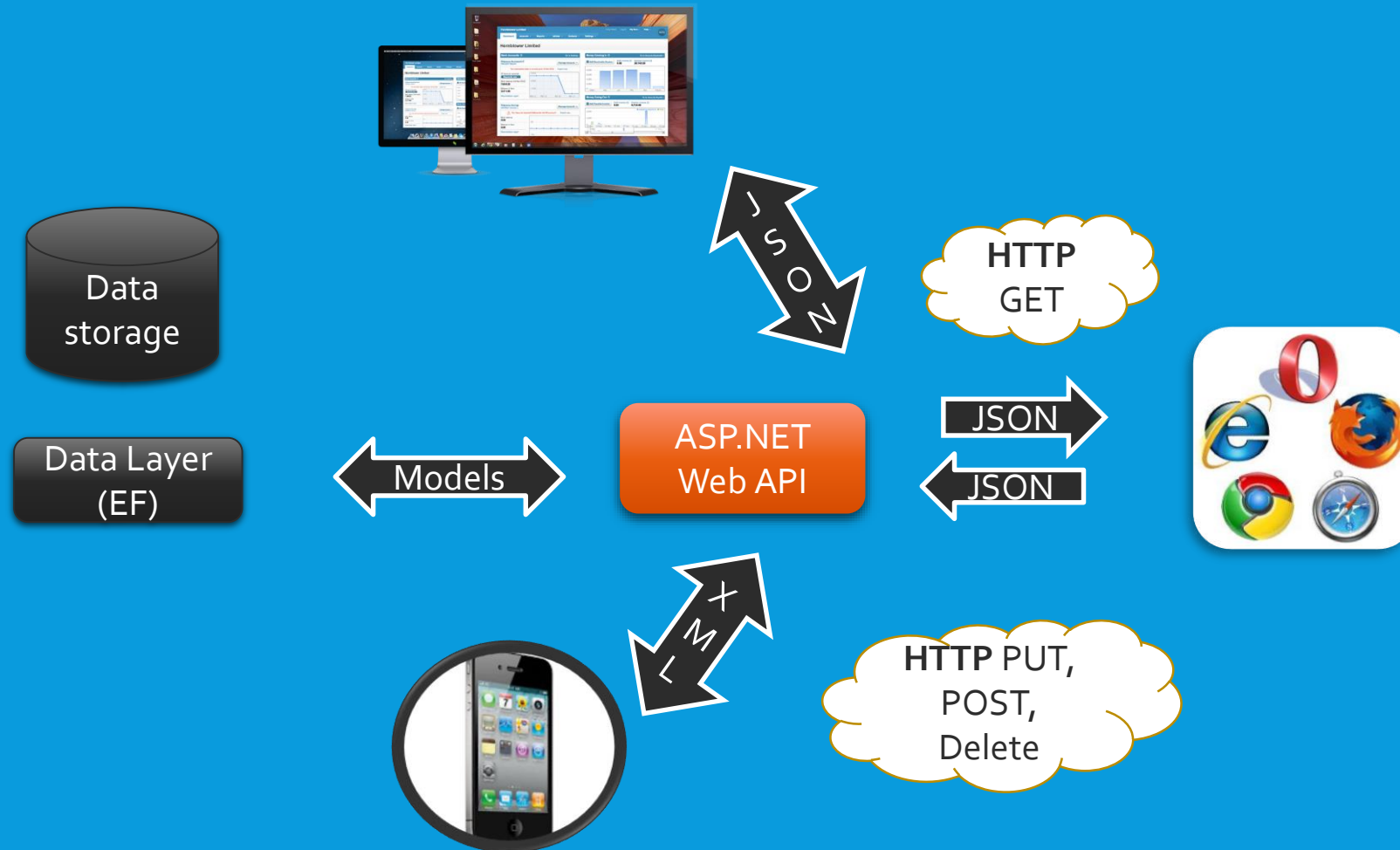
```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools"
    Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
    Version="2.0.0" />
</ItemGroup>
```

```
//To manage Migrations & create the DB go to console:
//[dotnet restore]
//dotnet ef migrations add MIGRATION-NAME
//dotnet ef database update
```



# Web API

# Web API



# Controller

- A *controller* is an object that handles HTTP requests
  - All API controllers derive from Controller
- By default ASP.NET Web API will map HTTP requests to specific methods called actions

```
[Route("api/[controller]")]
public class AppController : Controller
{
    private VouchersContext ctx;

    public AppController(VouchersContext context)
    {
        ctx = context;
    }

    [HttpGet]
    public IEnumerable<Voucher> Get()
    {
        var vouchers = ctx.Vouchers.OrderByDescending(v=>v.Date).ToList();
        return vouchers;
    }
}
```

# Controller Return Types

- A Controller may return various responses:
  - View - `return View(voucher);`
  - Http Status Code - `return BadRequest();`
  - Formatted Response - `return JsonResult(voucher);`
  - Redirect – `return RedirectToAction("Complete", new {id = 123});` | `return RedirectToRoute`

# HTTP Verbs

- RESTful (Representational State Transfer) web services use HTTP verbs to map CRUD operations to HTTP methods.
- RESTful web services expose either a collection resource (representational of a list) or an element resource (representational of a single item in the list)
- HTTP verbs are used as follows;
  - Create (POST) > create a new resource.
  - Read (GET) > retrieve one or many resources.
  - Update (PUT) > update an existing resource.
  - Delete (DELETE) > delete an existing resource.

# EF Core Change-Tracking

An entity can be in one of five states as defined by the EntityState enumeration.

- Added: the entity is being tracked by the context but does not yet exist in the database
- Unchanged: the entity is being tracked by the context and exists in the database, and its property values have not changed from the values in the database
- Modified: the entity is being tracked by the context and exists in the database, and some or all of its property values have been modified
- Deleted: the entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called
- Detached: the entity is not being tracked by the context

# Update Strategies

## Using Mapping / POCO

```
public void Put(int id, [FromBody]Voucher value)
{
    var v = ctx.Vouchers.FirstOrDefault
        (f => f.ID == id);

    if (v != null)
    {
        Mapper.CopyData(value, v);
        ctx.SaveChanges();
    }
}
```

## Using Entity State

```
public int Save([FromBody] Voucher value)
{
    if (value.ID == 0)
    {
        ctx.Vouchers.Add(value);
    }
    else
    {
        //Update using attach and entity state pattern
        ctx.Vouchers.Attach(value);
        ctx.Entry(value).State = EntityState.Modified;
    }
    ctx.SaveChanges();
    return value.ID;
}
```



# Web API Routing

# Routing

- Routing is how ASP.NET Web API matches a URI to a controller and an action
- Web APIs support the full set of routing capabilities from ASP.NET (MVC)
  - Route parameters
  - Constraints (using regular expressions)
  - Extensible with own conventions

# Convention based routing

- Bring your routes closer to your resources

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
);
```



ControllerSelector



ActionSelector



```
public IEnumerable<Resource> GetResource () { ... }
```

# Attribute routing

- Use Attributes to define your custom routing
- Attribute routes INCLUDE the base route of the Controller!

```
//route: http://localhost:8082/api/vouchers/getconstant
[Route("getconstant")]
public string GetConstant()
{
    return "constant string";
}

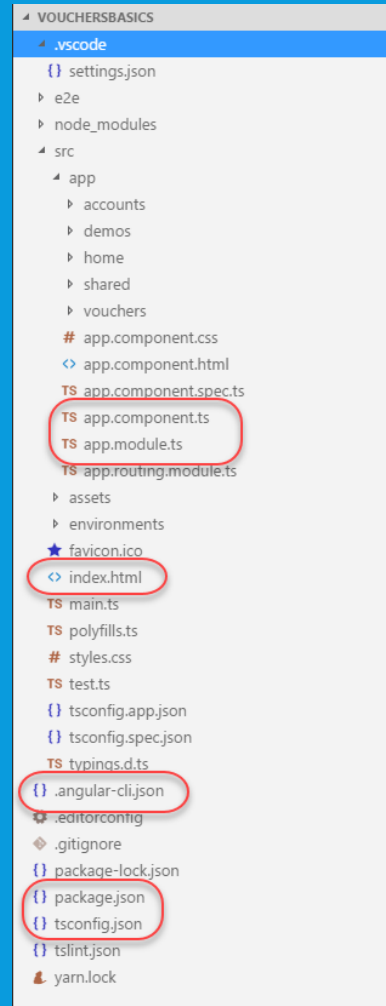
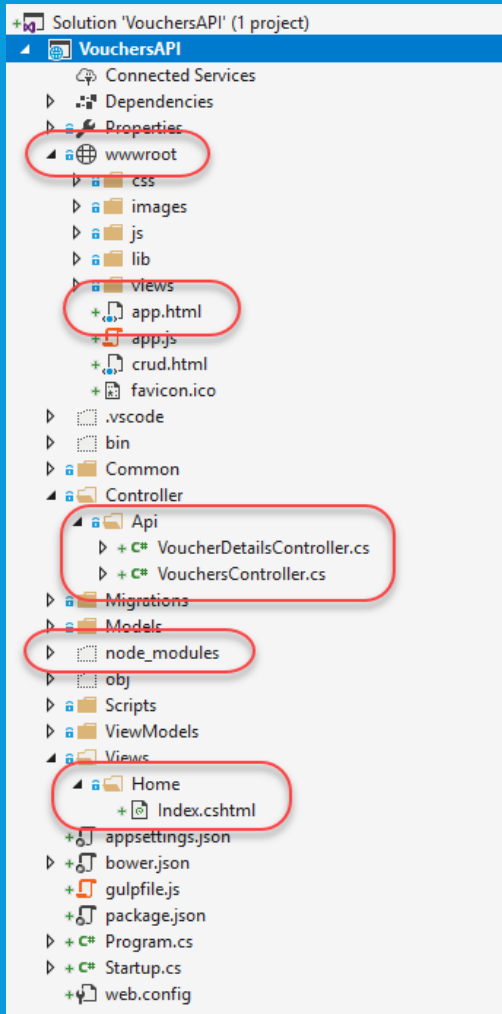
//route: http://localhost:8082/api/vouchers/getpaid/false
[Route("GetPaid/{paid}")]
public IEnumerable<Voucher> GetForVoucher(bool paid)
{
    var vs = ctx.Vouchers.Where(v => v.Paid == paid).ToList();
    return vs;
}
```

# Project Configuration for integration with ASP.NET Core

# Why Combine Angular & .NET Core

- Maybe you want to write your APIs using .NET Core & Entity Framework Core
- Benefits
  - Use a well known language (C#) for business layer
  - Data Layer is easy to implement using EF Core
  - Reuse existing API written in C#

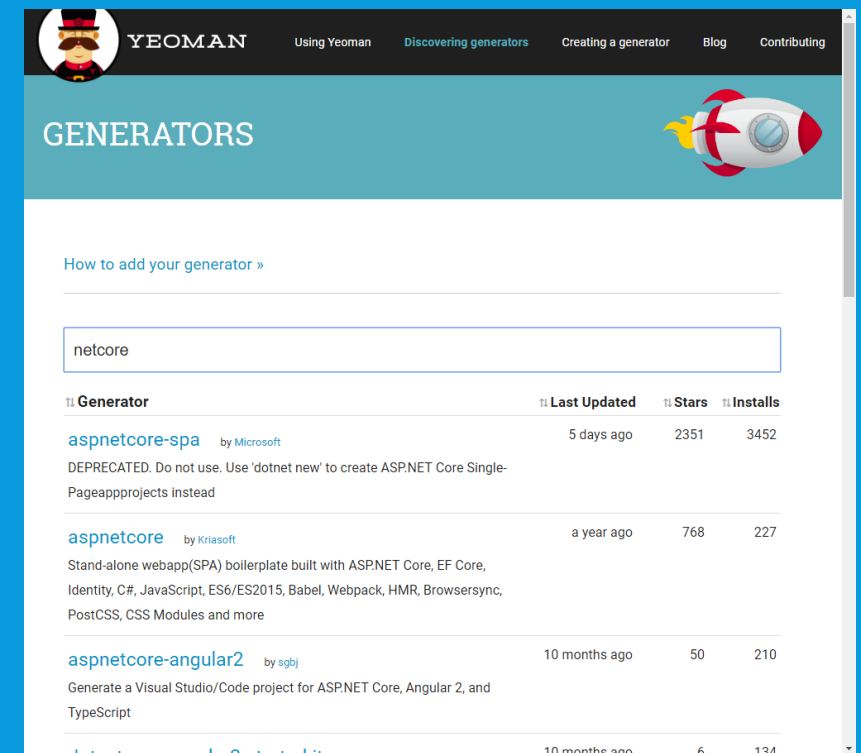
# .NET Core vs Angular Project Structure



- Angular CLI Structure vs .NET Core Structure
- Root Page?
  - .NET index.html (under wwwroot)
  - MVC 6 index.cs.html
  - Angular index.html (project root)
- One Project vs Two Projects
  - (Angular) Web Project
  - (.NET Core) Web APIs + Business Layer

# One Project for Angular & .NET Core

- Lots of Generators, Guides, Github samples on the internet available
- Disadvantages:
  - Maybe need to update versions
  - Maybe lots of npm package copy
  - Works now – does it work for future versions?





# Yeoman

- Yeoman helps you to kickstart new projects, prescribing best practices and tools
- Based on Node.js, Yeoman uses Generators, to set up different technologies
- You can add your own generators
- Using Yeoman & ASP.NET Core: <https://docs.asp.net/en/latest/client-side/yeoman.html>
- Documentation at <http://yeoman.io/>



# Gulp

Client Side Task Runner

Configured in gulpfile.js

Lots of Gulp plugins available @ <https://gulpjs.com/plugins/>

Gulp can be used to automate

- Copy files (from node\_modules to wwwroot)
- Other tasks
- Installation:
  - `npm install gulp -g`



```
{  
  "name": "ASP.NET",  
  "version": "0.0.0",  
  "devDependencies": {  
    "gulp": "3.8.11",  
    "gulp-concat": "2.5.2",  
    "gulp-cssmin": "0.1.7",  
    "gulp-uglify": "1.2.0",  
    "rimraf": "2.2.8"  
  }  
}
```

# gulpfile.js

- gulpfile.js is the configuration file for Gulp
- Used to Implement Tasks
  - e.g. compile TypeScript

```
"devDependencies": {  
  "gulp": "3.9.1",  
  "gulp-sourcemaps": "^2.6.0",  
  "gulp-tsc": "^1.3.2",  
  "gulp-typescript": "^x.x.x",  
  "gulp-cached": "1.1.0"  
}
```

```
var gulp = require('gulp');  
var typescript = require('gulp-tsc');  
var sourcemaps = require('gulp-sourcemaps');  
  
var paths = {  
  webroot: "./wwwroot/",  
  scriptSource: "./wwwroot/demos/*.js",  
  scriptDest: "./wwwroot/js/",  
  demos: "./wwwroot/demos/",  
  scss: "./wwwroot/sass/**/*.scss",  
  scssDest: "./wwwroot/css/"  
}  
  
gulp.task('compile:ts', function() {  
  gulp.src(['wwwroot/**/*.ts'])  
    .pipe(typescript())  
    .pipe(gulp.dest('dest/'));  
});
```

# Enable Cors

- Must be done before .AddMvc()
- Configure Services

```
services.AddCors(options =>
{
    options.AddPolicy("AllowAll",
        builder => builder.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials());
});
```

- Configure

```
app.UseCors("AllowAll");
app.UseMvcWithDefaultRoute();
```



smart Angular Single Page Application Development

Demo: Using Promises CRUD - Component:

Voucher

Result

Get Vouchers  
Get Voucher  
Insert Voucher  
Update Voucher  
Delete Voucher

```
[ { "ID": 3, "Text": "Amazon", "Date": "2017-08-25T17:51:27.9323052", "Amount": 56, "Paid": false, "Expense": true, "Remark": false, "Details": null }, { "ID": 2, "Text": "BP Tankstelle", "Date": "2017-08-25T17:51:27.9323013", "Amount": 65, "Paid": false, "Expense": true, "Remark": true, "Details": null }, { "ID": 1, "Text": "Bogus AG", "Date": "2017-08-25T17:51:27.9286173", "Amount": 800, "Paid": false, "Expense": false, "Remark": true, "Details": null }, { "ID": 4, "Text": "Media Markt", "Date": "2017-08-24T17:51:27.9323057", "Amount": 100, "Paid": true, "Expense": true, "Remark": false, "Details": null } ]
```

# Debugging using launch.json

- The configuration file for VS Code F5 debugging, Located in .vscode
- Uses a preLaunchTask to compile the project using dotnet build

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (web)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      // If you have changed target frameworks, make sure to update the program path.
      "program": "${workspaceRoot}/bin/Debug/netcoreapp1.1/VouchersTypeScript.dll",
      "args": [],
      "cwd": "${workspaceRoot}",
      "stopAtEntry": false,
      "internalConsoleOptions": "openOnSessionStart",
      "launchBrowser": {
        "enabled": true,
        "args": "${auto-detect-url}",
        "windows": {
          "command": "cmd.exe",
          "args": "/C start ${auto-detect-url}"
        }
      }
    }
  ]
}
```

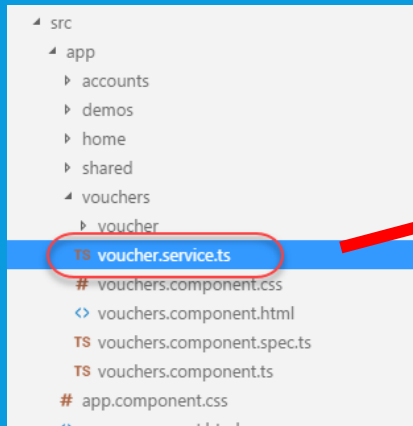
# Compilation

- Task allow automatic compilation
  - .NET compilation the project based on \*.csproj (.NET Core 1.1+)
  - Typescript transpilation using tsc in watch mode

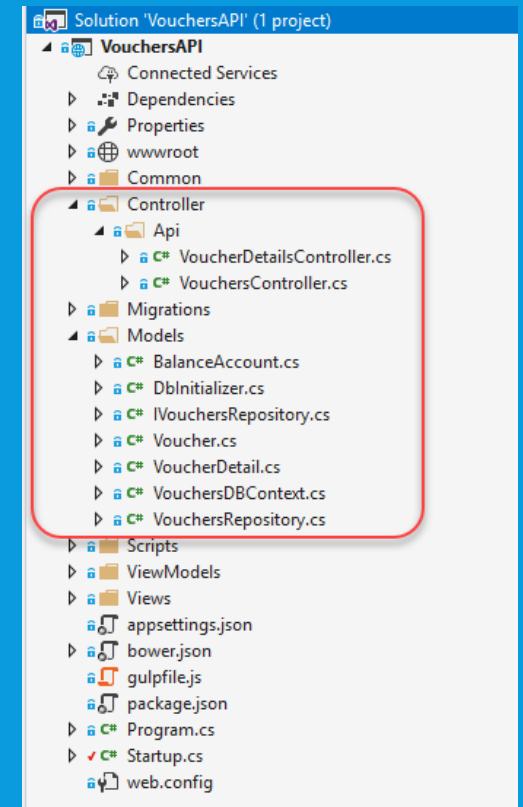
```
{
  "version": "2.0.0",
  "args": [],
  "tasks": [
    {
      "taskName": "tsc",
      "command": "tsc",
      "args": ["-w"],
      "isBackground": true,
      "problemMatcher": "$tsc-watch"
    },
    {
      "taskName": "build",
      "command": "dotnet build",
      "args": ["${workspaceRoot}/VouchersTypeScript.csproj"],
      "problemMatcher": "$msCompile"
    }
  ]
}
```

# Two Projects for Angular & .NET Core

- Separation is State-of-art
- (Angular) Web Project
- (.NET Core) Web APIs + Business Layer
- Need to enable CORS in .NET Core Startup.cs



```
getVouchers(){  
  this.httpClient.get('http://localhost:5000/api/vouchers')  
    .toPromise()  
    .then((response)=>this.result = response);  
}
```



# Implementing Client Side Data Models using Angular & TypeScript

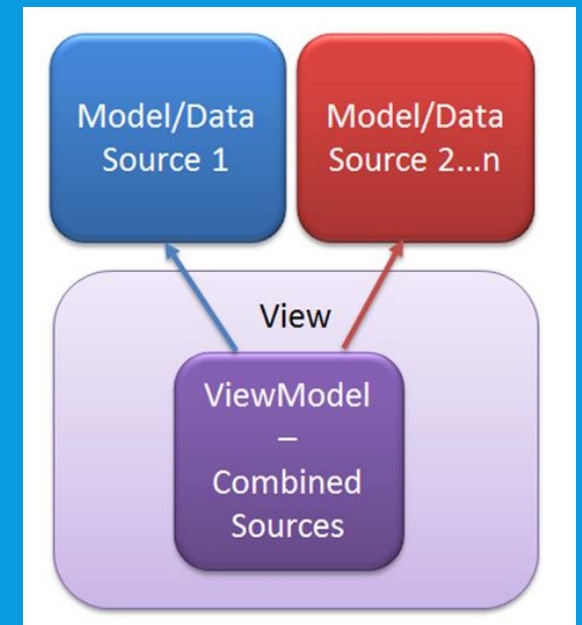


# Client Side Data Models

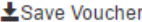
- In most cases Client Side Data Models correspond to Server side Data Models
- Angular uses Data Models implemented in TypeScript as Interfaces (Classes possible)
- Data Models can be made available in Modules or Barrels

# ViewModel

- ViewModels allow you to shape multiple entities from one or more data models or sources into a single object, optimized for consumption and rendering by the view
- Many business reasons :
  - Incorporating dropdown lists of lookup data into a related entity
  - Master-detail records view
  - Components like a shopping cart or user profile widget






# ViewModel Example

 Save Voucher

**Voucher**

Amazon 26-05-2016 56 ☒ Expense: ☐ Income: ☐ Paid:

**All Details**

 New  Save  Delete

Text	Amount	Account
USB Stick	11	undefined
DVI Kabel	45	undefined

**Current Detail**

USB Stick 11

Comment

**Accounts**

- Allgemeiner Aufwand
- Internetgebühren
- Fahrzeugkosten

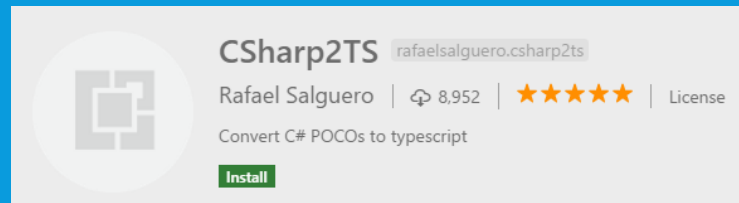
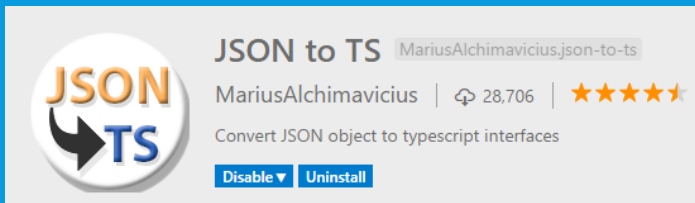
# C# Classes -> TS Interfaces

- In TypeScript the Data Model is implemented using Interfaces
- Can be done manually or using VS Code Extension

```
public class Voucher
{
    public int ID { get; set; }
    public string Text { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
    public bool Paid { get; set; }
    public bool Expense { get; set; }
    public bool Remark { get; set; }
}
```



```
export interface Voucher {
    ID: number;
    Text: string;
    Date: Date;
    Amount: number;
    Paid: boolean;
    Expense: boolean;
    Remark: boolean;
}
```



# Consuming .NET Core RESTful API using Angulars HttpClient & Promises

# CRUD

- Angular provides a HttpClient in order to implement Http Operations (Can use custom)
- HttpClient returns Observables by default – can return Promises
- CRUD Operations correspond to Http-Verbs
  - Create (POST) > create a new resource.
  - Read (GET) > retrieve one or many resources.
  - Update (PUT) > update an existing resource.
  - Delete (DELETE) > delete an existing resource.
- A Smart API can implement insert and update with one method

# HttpClient

- Angular 4.2+ provides an HttpClient for Async Http Operations
- Import '@angular/common/http'

```
export class VouchersService {  
  constructor(private http: HttpClient) { }  
  
  getVouchers() : Promise<any> {  
    return this.http.get('/api/vouchers').toPromise();  
  }  
  
  getVoucher(id: number) : Promise<any> {  
    return this.http.get('/api/vouchers/' + id).toPromise();  
  }  
  saveVoucher(v: Voucher){  
    return this.http.post('/api/vouchers', v).toPromise();  
  }  
  
  deleteVoucher(v: Voucher){  
    return this.http.delete('/api/vouchers/1').toPromise();  
  }  
}
```

# Comparing Http & HttpClient



# [Old ] Http vs HttpClient

- Utility classes to make http calls
- For simple calls take the one you like more
- HttpClient allows
  - Fine tuning Headers
  - Listening to Events
  - Use Interceptor (Add Bearer Token, Params to every Request)
  - Both return Observables by default
- HttpClient returns Body of Request – Http returns Request

# Http vs HttpClient Code

- Http -> import { Http, Response } from '@angular/http';
- HttpClient -> import { HttpClient } from '@angular/common/http';

```
getVouchers() : Observable<Voucher[]> {  
  return this.httpClient.get<Voucher[]>('/api/vouchers');  
}  
  
getVouchersHttp() : Observable<Voucher[]> {  
  return this.http.get('/api/vouchers').map((response: Response) => {  
    return <Voucher[]>response.json();  
  })  
}
```

# Fetching Response using HttpClient

- HttpClient returns Body of Request by default
- Use options to change Response Type
- Use Options to set Headers

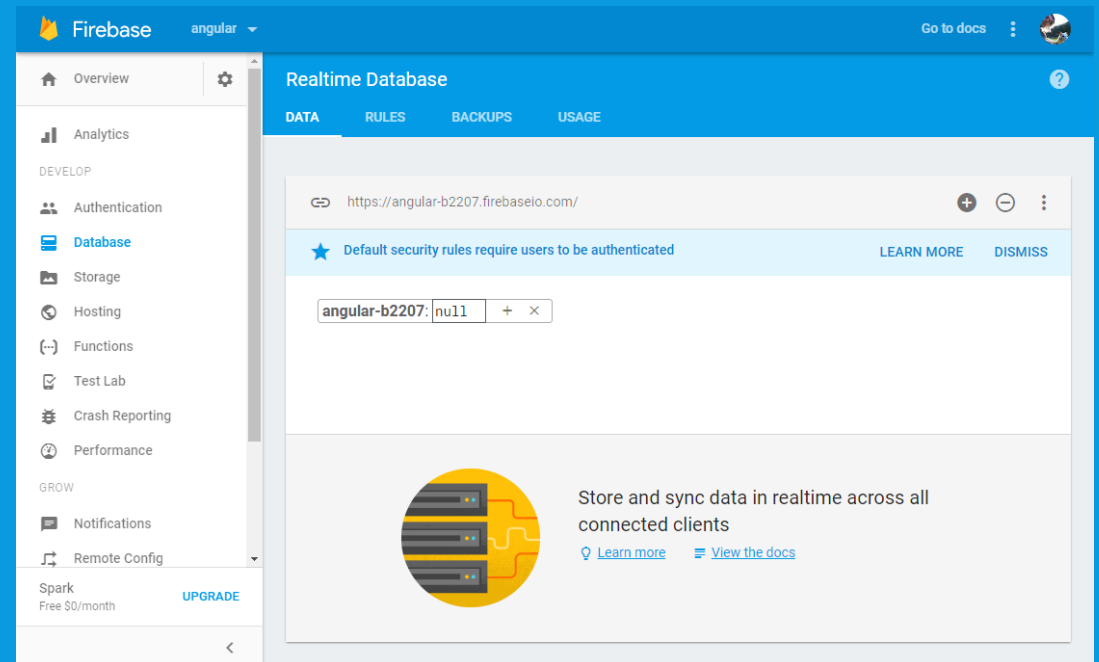
```
getVouchers(){  
  this.httpClient.get('http://localhost:5000/api/vouchers',  
    {responseType: 'text', observe: 'response'})  
    .toPromise()  
    .then((response)=>{  
      console.log(response);  
      this.resultB = response});  
}
```

```
▼ HttpResponse {headers: HttpHeaders, status: 200, statusText: "OK", url: "http://  
  ► body: (6) [{...}, {...}, {...}, {...}, {...}, {...}]  
  ► headers: HttpHeaders {normalizedNames: Map(0), lazyUpdate: null, lazyInit: f}  
    ok: true  
    status: 200  
    statusText: "OK"  
    type: 4  
    url: "http://localhost:5000/api/vouchers"  
  ► __proto__: HttpResponseBase
```

# Consuming NoSQL DBs using Angulars HttpClient & Interceptor

# Firebase

- Firebase is a NoSQL DB in the cloud provided by Google
- Available @ <https://firebase.google.com>
- Google Account required



# Using Interceptors

- Interceptors are used to "modify" Request before it is sent
- Can be used to include reusable headers or add Tokens (automate Auth)
- Provided by HttpClient – NOT Http

```
providers: [  
  VouchersService,  
  FirebaseService,  
  {provide: LOCAL_ID, useValue: "de-DE"},  
  {provide: HTTP_INTERCEPTORS, useClass: FirebaseInterceptor, multi: true},  
  RouteGuard  
,  
],  
bootstrap: [AppComponent]
```

```
public intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
  console.log('Interceptor used for request', req);  
  let clonedRequest = req.clone({headers: req.headers.append('Authorization', 'Bearer' + environment.firebaseToken)});  
  return next.handle(clonedRequest);  
}
```