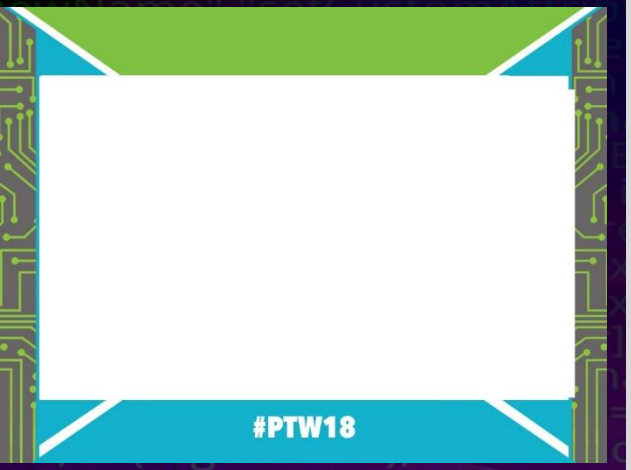
Building a Sortable/Filterable/Paginated Grid in C# .NET Core & Angular

Agenda (1 pm – 3 pm)

- What's up w/ .NET Core, Entity Fmwk Core and ASP .NET Core?
- Build a REST API server with C# Web API
- Simple Angular example
- Theme with Bootstrap
- Tie Angular front end to C# REST API
- Use ag-grid to display data via Dynamic LINQs



Brian Woelfel, Eigen X

- Salesforce Admin, Force.com, Lightning
- Tableau, Datawarehouse/BI, ETL
- Custom Dev- C#/Java/Angular
- Project Mgmt, Training

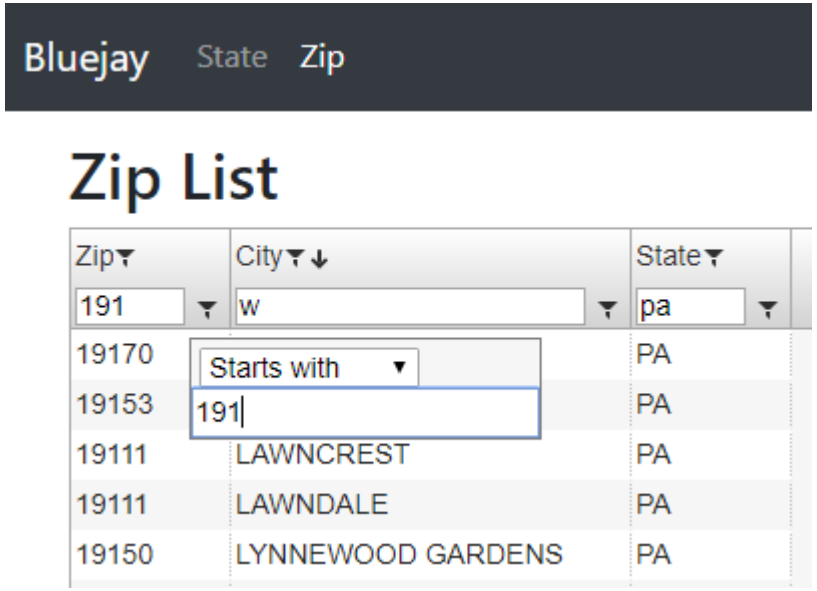
https://github.com/brianwoelfel/bluejaySrc

http://linkedin.com/in/brianwoelfel

bwoelfel@eigenx.com

Show Final Product

- Web page with table of data
- User can sort and filter
- Supports very large data sets from pagination and infinite scroll and server side data feed
- C# MVC controllers providing data
- Angular 5 handling frontend



Tech Stack

- DB: MSSQL 2017
- Backend
  - Visual Studio 2017 C#
  - .NET Core 2.0
  - ASP .NET Core
  - Web API
  - Entity Framework Core
- Frontend
  - Angular 5
  - Bootstrap

Database

Data for this demo

- 50 states
- 80,000 zip codes

```
select * from state;
select * from zip;
```

100 %

Results | Messages

	Name	Code
1	Alaska	AK
2	Alabama	AL
3	Arkansas	AR
4	Arizona	AZ
5	California	CA

	Id	Zipcode	City	State
1	1	19503	BALLY	PA
2	2	19504	BARTO	PA
3	3	19505	BECH...	PA

.NET Core Summary

- Modern ground up rewrite from Microsoft
- Platform independent, lightweight
- You can still layer DotNetFramework on top of it if you want
- Opensource on github so easier for people to make addins
- Tightly integrated with nuget
- Better command line support
- Can serve as a foundation for all of Microsoft's client heavy stacks, e.g. windows app development or xamarin

ASP .NET Core

- Complete rewrite of all the ASP stuff we're used to in .NET Framework
- It's more light weight
- Same functionality, but everything is configured very differently, and it's not obvious or common sense how to change stuff or where to find it
- Feels much more linux-y and component-y and less like big old Microsoft behemoth servers
- Doesn't feel fully baked yet, occasionally stuff is missing or poorly documented

ASP .NET Core – Program.cs

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

namespace BluejayWeb
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }
        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

ASP .NET Core – Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) { // set up EF connection string from env specific config file
        string connection = Configuration.GetSection("ConnectionStrings").GetValue<string>("ZipDb");
        services.AddDbContext <BluejayContext> (options => options.UseSqlServer(connection));
        services.AddMvc(); // tell it to use MVC
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        app.Use(async (context, next) => { // all incoming requests to www/index.html unless /api/*
            await next();
            if (context.Response.StatusCode == 404 &&
               !System.IO.Path.HasExtension(context.Request.Path.Value) &&
               !context.Request.Path.Value.StartsWith("/api/")) {
                context.Request.Path = "/index.html";
                await next();
            }
        });
        app.UseMvcWithDefaultRoute();
        app.UseDefaultFiles();
        app.UseStaticFiles(); // allow site to serve static angular content
    }
}
```

REST API

Informal API that leverages original HTTP verbs to transfer data between systems.

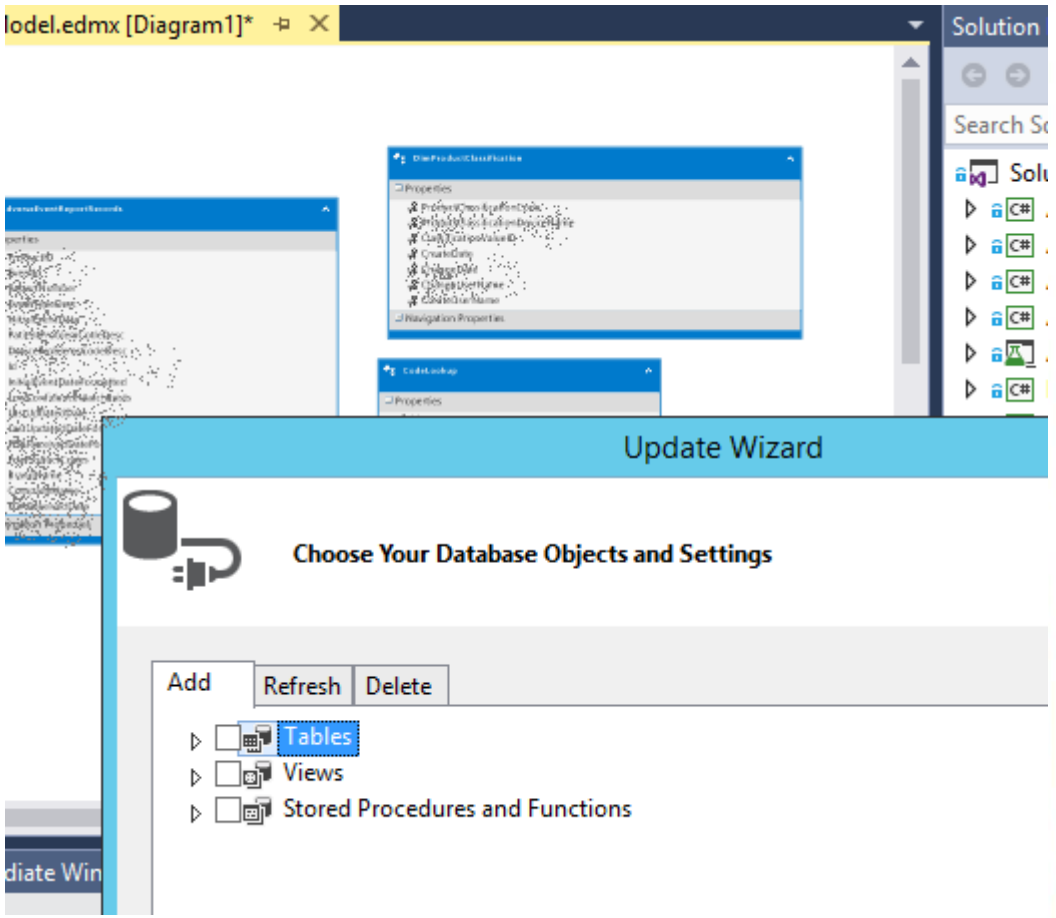
- **GET**— with a key on the url, get data entity
- **GET**— with no key on the url, get a list of records
- **PUT**— change data (usually one entity)
- **POST**— create data (usually one entity)
- **DELETE** — delete data (usually one entity)

Notes:

- Data payloads are simple JSON
- Much simpler than SOAP or WCF. No WSDL files or strict XML contracts
- You can use a browser to debug **GET** commands, or a tool like Postman or SOAPUI to test other verbs
- Can build with Visual Studio C#, Web API and MVC Controllers

Entity Framework .NET Framework (OLD)

- Used to be that in Visual Studio you'd make an EDMX model with a visual canvas, drag-and-drop tables around, and use a wizard to keep changes up to date in your Entity Framework Model, and it generated POCOs for you
- Lots of clicking, but configurable

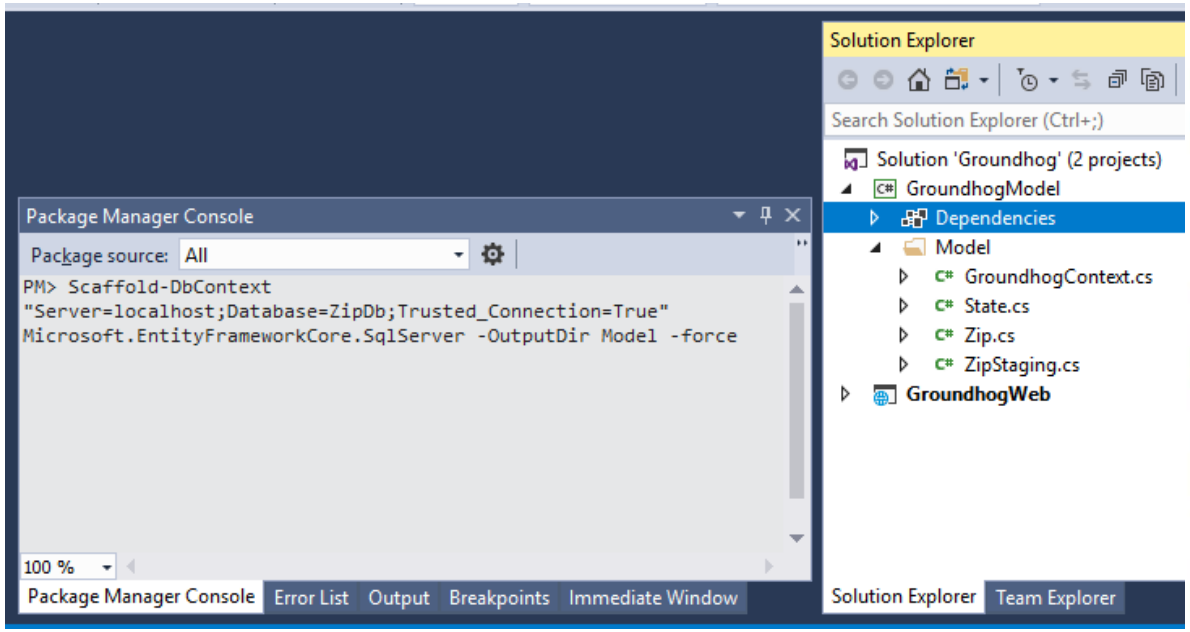


Entity Framework Core (NEW)

- No UI or wizards
- Have to run command line from Package Manager Console

```
Scaffold-DbContext
"Server=localhost;Database=Foundat
ion;Trusted_Connection=True"
Microsoft.EntityFrameworkCore.SqlS
erver -OutputDir Model
```

- Doesn't support database views



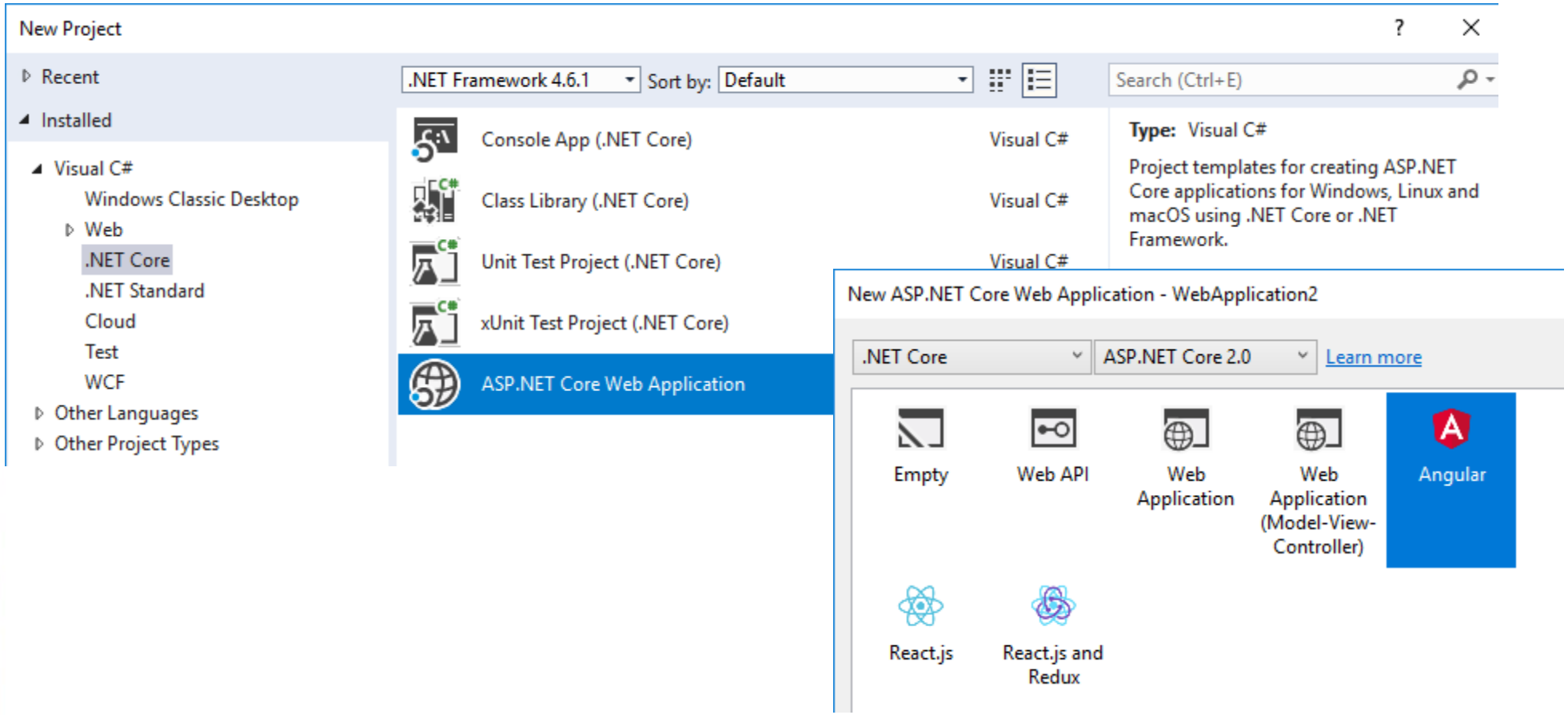
dotnet CLI

```
C:\temp>dotnet --help
.NET Command Line Tools (2.0.2)
Usage: dotnet [runtime-options] [path-to-application]
Usage: dotnet [sdk-options] [command] [arguments] [command-options]

path-to-application:
  The path to an application .dll file to execute.

SDK commands:
  new                 Initialize .NET projects.
  restore             Restore dependencies specified in the .NET project.
  run                 Compiles and immediately executes a .NET project.
  build               Builds a .NET project.
  publish             Publishes a .NET project for deployment (including the
runtime).
```

dotnet CLI - Commands

- **`dotnet new sln -n [name] -o [outdir]`** Build a new empty Visual Studio solution
- **`dotnet new classlib --framework netcoreapp2.0`** Build a new class library, in this DotNetCore 2.0.
- **`dotnet sln [*.sln] add [*.csproj]`** Add a project to a solution
- **`dotnet new angular`** Create a new MVC project with angular support built in
- **`dotnet new webapi`** Create a new WebAPI project with no fancy frontend
- **`dotnet ef dbcontext scaffold [connstr] Microsoft.EntityFrameworkCore.SqlServer -c [contextname] -o [outdir]`** Create entity framework context and C# models from database
- **`dotnet add [*.csproj] package [nuget package]`** Add a nuget package to a Visual Studio project.
- **`dotnet restore`** Find all the nuget references in .csproj and download them
- **`dotnet run`** Run a project, either a console app or a DotNetCore ASP site
- **`dotnet build`** Perform a build of the solution to produce .exe/.dll, just like in Visual Studio

New C# WebAPI Project



Bluejay1: C# WebAPI + Fake Data

- Make a new ASP .NET Core Web Application, Web API
- Configure Startup.cs and Program.cs to serve up a simple website with REST controllers
- Add a controller that serves some fake zip codes and states

Bluejay2: C# WebAPI + EF

- Make a new ASP .NET Core Web Application, Web API
- Configure Startup.cs and Program.cs to serve up a simple website with REST controllers
- Pull connection string from appSettings.json with special developer version controlled by environment variables
- Add an Entity Framework project as .NET Core 2.0 class library
- Add all the nuget packages to support scaffolding
- Scaffold database to create entity POCOs and db context
- Add a controller that serves real zipcodes and states from database

Bluejay2: C# WebAPI + EF: Controller LINQ

```
public class ZipController : Controller {
    private readonly BluejayContext _context;

    public ZipController(BluejayContext context) { // Get DB from DI and appSettings.json
        this._context = context;
    }

    // REST API to serve a list of zip records as JSON
    [HttpGet("/api/[controller]")]
    public IActionResult List() => Json(_context.Zip.ToList());

    // REST API to retrieve one zip record as JSON by Id
    [HttpGet("/api/[controller]/{id}")]
    public IActionResult Get(int id) {
        Zip zip = _context.Zip.Find(id);
        if (zip != null) {
            return Json(zip);
        } else {
            return NotFound(id);
        }
    }
}
```

Angular Overview - Taxonomy

- **component**— a web page or part of a web page with HTML with fancy angular tags and typescript code behind. Similar to *.aspx and *.aspx.cs code behind
- **service**— behind the scenes typescript code that transfers data with the backend or does business logic
- **directive**— allows you to make custom HTML attributes that have custom behaviors
- **pipe**— text formatting functions that you can embed in Angular tags inside HTML
- **module**— collection of angular code packaged inside of a module to expose some functionality

TypeScript – Strongly Typed JavaScript

```
class Student {
    fullName: string;
    constructor(public firstName: string,
        public middleInitial: string, public lastName: string) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}
interface Person {
    firstName: string;
    lastName: string;
}
function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}
let user = new Student("Jane", "M.", "User");
console.log(greeter(user));
```

Angular Controller

- Backend code for a web page
- Calls services to get data
- Prepares data for frontend scope in JavaScript
- Supplies code for buttons and events

```
@Component({
  selector: 'state-list',
  templateUrl: './state-list.component.html',
  styleUrls: ['./state-list.component.css']
})
export class StateListComponent {
    states: State[];
    constructor(private stateService: StateService, private router: Router) {}
    ngOnInit(): void {
        this.stateService.list().then(states => this.states = states);
    }
}
```

Angular Service

- Angular TypeScript code
- Backend server calls to async get/update data
- Perform other nonUI calculations

```
// Angular service to read list of Zip entities from C# backend
import { Injectable } from '@angular/core';
import { Http } from '@angular/http'; // Standard libraries for http calls
import { Zip } from '../model/Zip'; // our custom entity POJO
@Injectable() // make available for dependency injection
export class ZipService  {
    constructor(private http: Http) {} // get HTTP library via dependency injection
    // Retrieve a list of Zips from C# backend as a promise
    public list(): Promise<Zip[]> {
        return this.http.get(`/api/Zip`) // do the HTTP call
            .toPromise() // wrap results into a promise
            .then(response => response.json() as Zip[]) // if successful, convert payload to our desired type
    }
}
```

Angular Entity POJOs

- Make an entity model with POJO's to match C# Entity Framework

```
export class Zip {
    id: number;
    zipcode: string;
    city: string;
    state: string;
}
```

Node.js and npm

- Node.js is a highly scalable asynchronous backend server built in Chrome's V8 JavaScript engine
- Uses open source repository (http://npmjs.org) and npm command line tool to manage installing your packages. Node.js used during development, not deployed to web servers
- package.json config file keeps track of installed toolkits
- node_modules folder stores all the javascript libraries used, at compile time only a subset is used
- Angular 5 CLI runs within NodeJS environment
- Example (install bootstrap)

```
npm install @ng-bootstrap/ng-bootstrap --save
```

Angular CLI

- **`ng new [project]`** Make a new angular app with basic scaffolding
- **`ng build`** Build the angular app in the current directory
- **`ng serve`** Build angular from current dir and serve via NodeJs at http://localhost:4200
- **`ng generate`** Generate new scaffolded components and directives
- **`npm install`** Install third party JavaScript toolkits into your project

Angular CLI - Example

- Install latest version of Angular (ng) globally

```
npm install -g @angular/cli
```

- Create new angular project myapp, add all scaffolding files

```
ng new myapp
```

- Make Node.js serve the angular website

```
ng serve --open
```

- Example

```
cd \code
ng new myapp
cd myapp
ng serve --open
```

Angular Routing

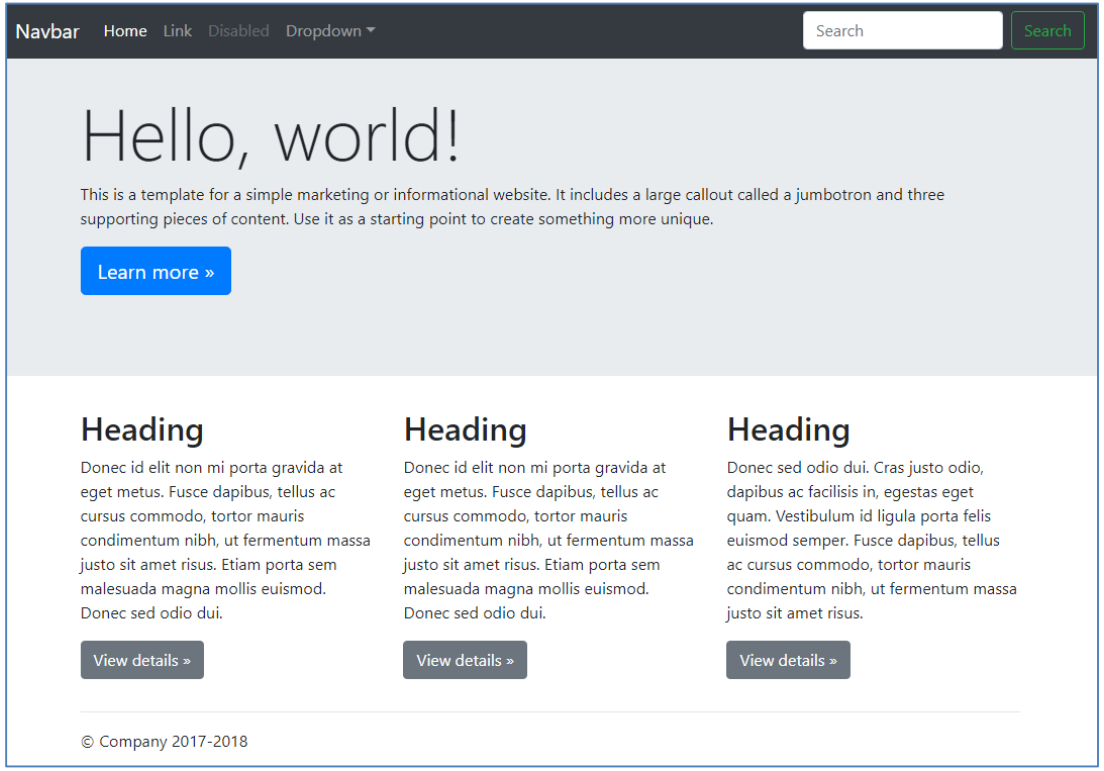
app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './component/app.component'; // components are like web pages
import { ZipListComponent } from './component/zip-list.component';
import { StateListComponent } from './component/state-list.component';

const routes: Routes = [ // these are all the routes.
     { path: '', component: StateListComponent }, // we could be pattern to match parameters and keys
     { path: 'zip', component: ZipListComponent },
     { path: 'state', component: StateListComponent },
];

@NgModule({
    imports: [ RouterModule.forRoot(routes) ], // I have no idea what this means
    exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

app-module.ts needs a reference added to this routing table

Bootstrap Theming

- Makes clean, easy to ready, mobile friendly pages without complex HTML/CSS
- Use npm to install bootstrap and angular support
- Feeds into SCSS built into Angular 5
- Can use bootstrap HTML styling throughout



Bluejay3: Angular Tables with Fake Data

- Build a simple angular app
- Split out single app component into:
  - Home page app component
  - Zip list component to show list of zip codes
  - State list component to show list of states
- Add a service for each data type
- Use “promise” async calls for data lookups, even though we’re just using fake data for this version
- Tie in routing for different components
- Add Bootstrap theming

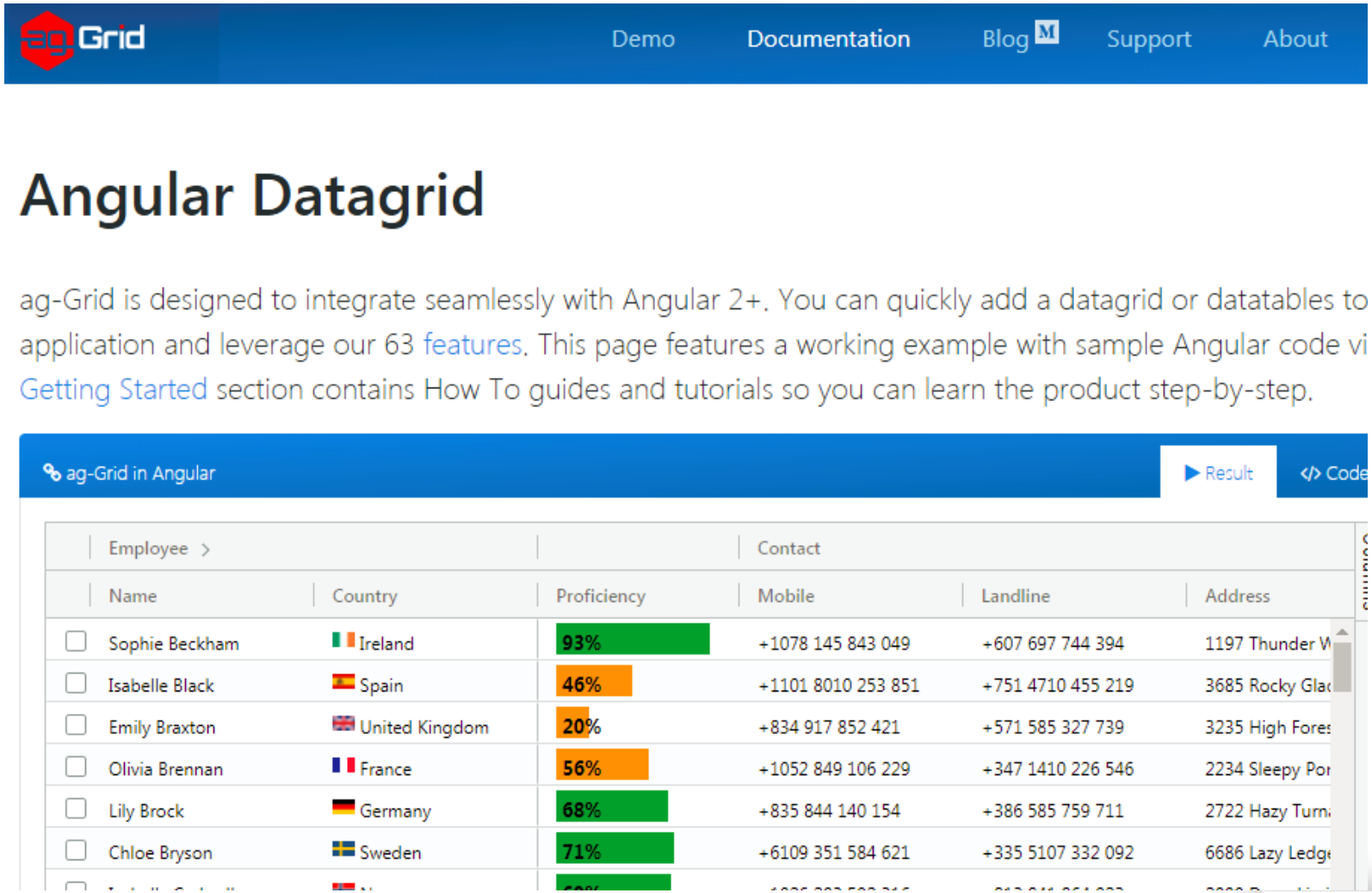
Angular File Dissection

- **`.angular-cli.json`**— Primary config file for setting up angular. Tie together test packages, list output folder.
- **`src\index.html`** — The main HTML that serves this app. It's really just a shell with some HTML doctype and meta tags. Normally we don't mess with this. There's an "approot" tag which gets the runtime pages.
- **`src\styles.css`** — Global CSS styles. It's actually SCSS so we can use imports
- **`src\app`** — After site is configured and we figure out which modules we want, this folder is where developers spend most of their time implementing features
- **`src\app\app.component.css`** — Custom CSS for just this component
- **`src\app\app.component.html`** — HTML snippets for just this component. Angular tags are scoped to the local variables in the *.ts file
- **`src\app\app.component.ts`** — TypeScript for this component. Set up local variables for HTML to use. Load data. Implement button behaviors.
- **`src\app\app.module.ts`** — List of all modules installed, and how they interact

Bluejay4: Angular/C# Integration

- Make a new ASP .NET Core Web Application, Web API
- Configure Startup.cs and Program.cs to serve up a simple website with REST controllers, and serve static HTML content from wwwroot folder
- Add an Entity Framework project as .NET Core 2.0 class library
- Add all the nuget packages to support scaffolding
- Scaffold database to create entity POCOs and db context
- Add a controller that serves real zipcodes and states from database
- Create a new angular app in sibling folder, output compiled javascript to wwwroot
- Add components for web pages for home page, states, and zips
- Add services for states and zips that do real async data calls
- Display data as HTML tables

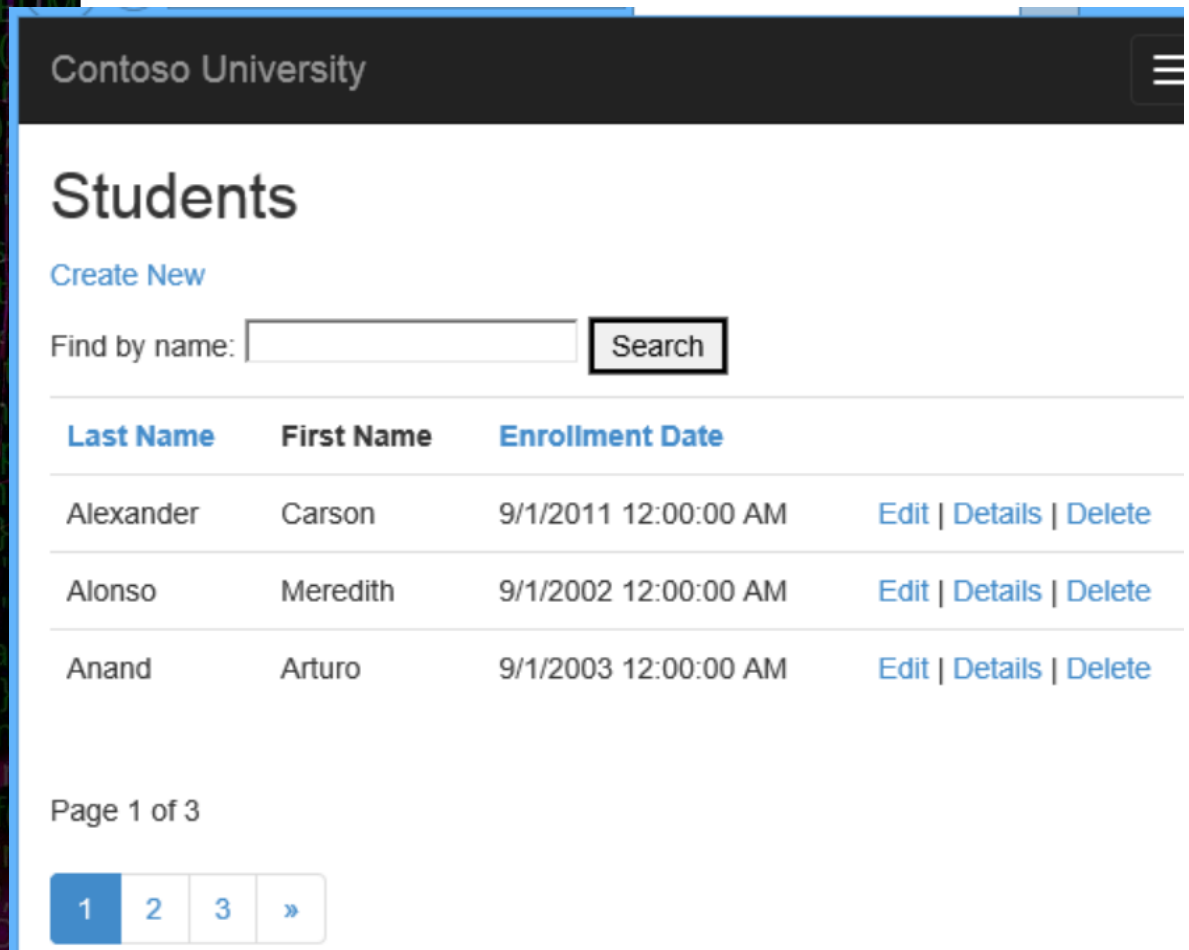
ag-grid


ag Grid
Demo
Documentation
Blog
Support
About
Angular Datagrid
ag-Grid is designed to integrate seamlessly with Angular 2+. You can quickly add a datagrid or datatables to
application and leverage our 63 features. This page features a working example with sample Angular code vi
Getting Started section contains How To guides and tutorials so you can learn the product step-by-step.
ag-Grid in Angular
Result
Code
Employee
Contact
Name
Country
Proficiency
Mobile
Landline
Address
Sophie Beckham
Ireland
93%
+1078 145 843 049
+607 697 744 394
1197 Thunder W
Isabelle Black
Spain
46%
+1101 8010 253 851
+751 4710 455 219
3685 Rocky Gla
Emily Braxton
United Kingdom
20%
+834 917 852 421
+571 585 327 739
3235 High Fores
Olivia Brennan
France
56%
+1052 849 106 229
+347 1410 226 546
2234 Sleepy Por
Lily Brock
Germany
68%
+835 844 140 154
+386 585 759 711
2722 Hazy Turn
Chloe Bryson
Sweden
71%
+6109 351 584 621
+335 5107 332 092
6686 Lazy Ledge

Bluejay5: C# WebAPI + EF + NG + Client Grid

- Make a new ASP .NET Core Web Application, Web API
- Configure Startup.cs and Program.cs to serve up a simple website with REST controllers, and serve static HTML content from wwwroot folder
- Add an Entity Framework project as .NET Core 2.0 class library
- Scaffold database to create entity POCOs and db context
- Add a controller that serves real zipcodes and states from database
- Create a new angular app in sibling folder, output compiled javascript to wwwroot
- **Add npm packages for ag-grid and CSS styles**
- **Add components for web pages for home page, states, and zips**
- **Component HTML is just a DIV for grid**
- **Component JavaScript calls ag-grid objects and configures settings about grid layout and feeds initial data**
- Add services for states and zips that do real async data calls

Server Side Grids (The Old Way)




```
public ViewResult Index(string sortOrder, string searchString)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : ""
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
```

Server Side Grids (The New Way)

1. Tell ag-grid to do server side data callbacks via configuration
2. Add ag-grid callback method to get data
3. Create new POJOs to store ag-grid filter/sort model
4. Update angular service to pass along user's filter/sort selections as filter/sort POJOs
5. Create C# filter/sort POCOs to match ag-grid filter/sort POJOs
6. Install nuget package System.Linq.Dynamic.Core
7. Make C# utility functions to glue everything together
8. Create C# MVC controller methods to accept filter sort requests

Bluejay6: C# WebAPI + EF + NG + Server Grid

- Starts same as Bluejay5
- Added custom C# folder to web project AgGridFilterSort
  - C# POCOs to match ag-grid
  - Utility method to convert ag-grid filter request to list of entities as JSON via Dynamic LINQ
  - Added hook into Zip controller
- Updated javascript to do server calls to get data

Swagger and Swashbuckle

- Swagger – instead of complex XML WSDL for web services, lightweight realtime documentation for REST based APIs
- Swashbuckle is a C# implementation that leverages native C# documentation to expose information quickly and easily about your methods with minimal configuration
- Need to add Nuget package a change a few lines of Startup.cs and change a few project properties
- Don’t need to use an API tester like SoapUI or PostMan

Swashbuckle Example Startup.cs

```
using Swashbuckle.AspNetCore.Swagger;
//...
    public class Startup  {
        // public Startup(), public Iconfiguration …
        public void ConfigureServices(IServiceCollection services)  {
            // Set up EF context from connection string …
            // Register the Swagger generator, defining one or more Swagger documents
            services.AddSwaggerGen(c => {
                c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
                var xmlFile = $"{Assembly.GetEntryAssembly().GetName().Name}.xml";
                var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
                c.IncludeXmlComments(xmlPath);
        }); }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            // app.Use(async (context, next) => ... Then setup MVC, default files, static files
            app.UseSwagger();
            app.UseSwaggerUI(c => {
                c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
            });
        }
```

Dynamic LINQ

- Instead of normal LINQ

```
return _dbContext.Zip.Where(x => x.State == "PA")
```

- Do this. We can feed all parameters at runtime, not compile time

```
using System.Linq.Dynamic.Core;
public static IQueryable<T> AddWhereClauseChunk<T>(IQueryable<T> item, string
colId, FilterItem filterItem)  {
    item = item.Where(colId + ".Contains(@0)", filterItem.Filter.Trim());
    return item;
}
```

Credits

- Some TypeScript samples from **TypeScript in 5 Minutes** *https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html*
- .NET Core advantages and disadvantages pulls from https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/ *
- Zip code list from http://federalgovernmentzipcodes.us/
- Blue Jay picture from Jim Ridley https://commons.wikimedia.org/wiki/File:Blue_Jay_(Jim_Ridley_2010.).jpg