# Developing large scale applications using Web API, Dojo, and ArcGIS API for JavaScript

# Executive Summary

There are many samples using ArcGIS API for JavaScript on the ESRI website. As useful as they are, I wish there is one comprehensive example that would walk through the steps of setting up a large scale application using ArcGIS API for JavaScript. This blog is intended for just that, showing how to architect such an application, how to implement each component, and how to build the JavaScript portion of the application. You could skip the entire *Server side* section if you don’t intend to expose data on the server side.

# Architecture

Our example application is divided into two parts: server side and client side. On the server side, we utilize ASP.NET Web API to expose data over HTTP. On the client side, we utilize Dojo and ArcGIS API for JavaScript to consume the data.

REST

Endpoints

ArcGIS API, Dojo

Web API

ASP.NET Web API is a framework that can be used to build HTTP services. On a high level, there could be multiple controllers in a Web API application; each controller is mapped to an URI (a REST endpoint) which is used to deliver data over HTTP.

Controller 1

Controller N

Rest Endpoint 1

Rest Endpoint N

On the client side, we use ArcGIS API for JavaScript and Dojo to consume the data being exposed by the controllers via REST endpoints. For better performance, we will use dojo modules locally. Studies of library CDN usage shows that using CDNs usually offers worse performance than locally hosted scripts, especially local scripts can be built into layers to significantly reduce HTTP round trips[[1]](#footnote-1). On the other hand, we want to load ESRI modules via a content delivery network (CDN) to simplify our upgrade process in the future. In other words, we load dojo modules locally and ESRI modules via CDN.

# Server side

## Prerequisites

ASP.NET MVC 4 is a framework for building Web applications using the MVC (Model-View- Controller) pattern. ASP.NET Web API is part of the MVC 4 framework. If you are running VS 2010 SP1, you will need to install ASP.NET MVC 4 for VS 2010 SP1 [here](http://www.microsoft.com/en-ca/download/details.aspx?id=30683). FOR VS2012 or higher, the MVC4 framework is integrated into Visual Studio.

## Web API boilerplate

As with many Visual Studio templates, the default Web API template comes with gobs of features. Often it makes more sense to start with a barebones template, adding features as required as opposed to using the default template. The following website has instructions on creating a barebones Web API application

<http://typecastexception.com/post/2013/07/01/Creating-a-Clean-Minimal-Footprint-ASPNET-WebAPI-Project-with-VS-2012-and-ASPNET-MVC-4.aspx>

After creating a barebones Web API application, you need to create Models and Controllers. It’s important to name the controller methods in accordance with the convention. The Model is responsible for querying data and the Controller is responsible for exposing the acquired data as a REST service endpoint.

## Data access overview

The Model is responsible for querying data from the database. It defines the data model and queries data in accordance with the data model. The model thus contains two types of object: the data definition object and the data access object. Lastly, the data access object is injected into the controller’s constructor using Unity.

Inject Data Access Object into Controller’s constructor

Controller1

Data Access Object1

Database1

The Unity Application Block is a lightweight, extensible dependency injection container. There are a few advantages to using Unity as opposed to instantiating the data access object directly within the controller. First, it forces you to use Interface, allowing a cleaner and more consistent coding style. Second, it groups the object creation code within one place, providing an intuitive way to organize code. Lastly, it simplifies the client code as you do not need to worry about instantiating the data access object in the controller; Unity does that for you.

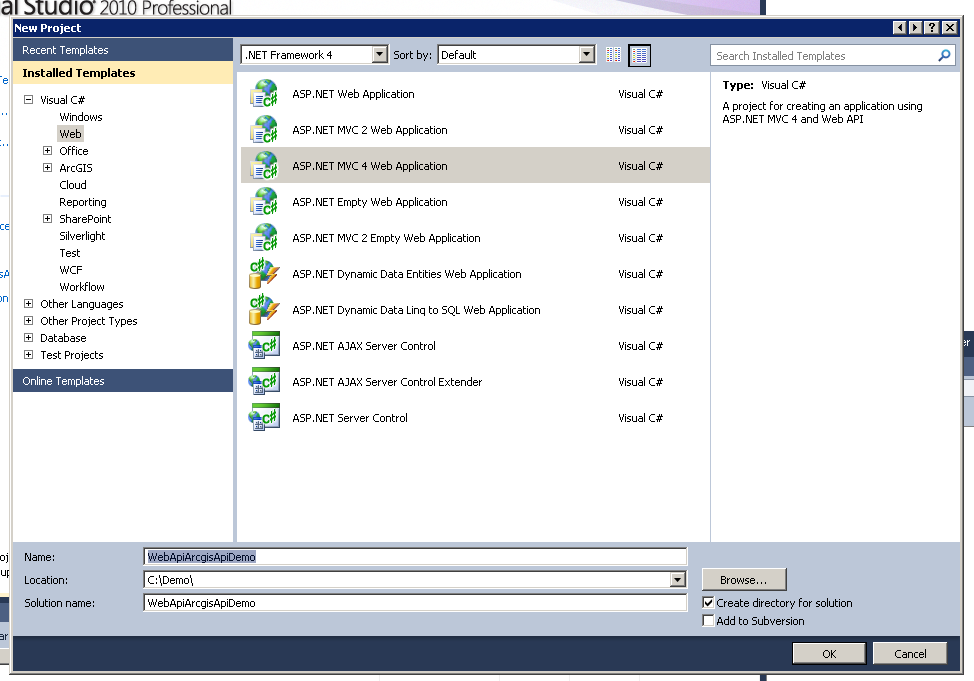
The following walkthrough example demonstrates the above concepts, creating a barebones WebAPI, creating a model and injecting the model into a controller using Unity.

## Server side walkthrough

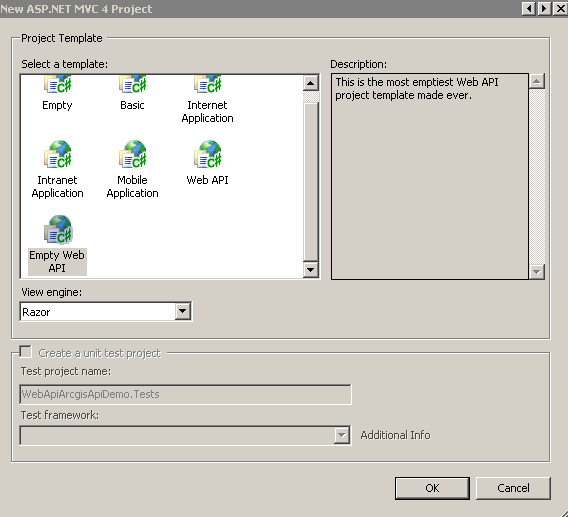
We are using VS2010 SP1 for this walkthrough. You need to download [the APS.NET MVC 4 framework](http://www.asp.net/mvc/mvc4). First, let’s create a barebones WebAPI application.

### Creating an empty WebAPI template

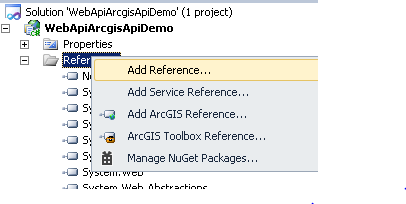
1. Download the Empty WebApi Project Template Installer [HERE](http://visualstudiogallery.msdn.microsoft.com/a989a149-4bc3-4292-ac8a-5101ee1722d7)
2. If you want to get the installer to work with VS2010, you will need to change the vsix extension to zip. Double click on the zip file to view the content. Open extension.vsixmanifest in a text editor; change <VisualStudio Version="11.0"> to <VisualStudio Version="10.0">. Save and close extension.vsixmanifest. Lastly, you need to rename the zip extension to vsix. Please note the template installer works with Visual Studio 2012 as is; no changes to vsixmanifest required
3. Double click the template installer, i.e., the vsix file to install the template into Visual studio
4. Create a WebApi application using the Empty WebApi Project Template. *New* -> *Project*, choose *ASP.NETMVC4 Web Application*



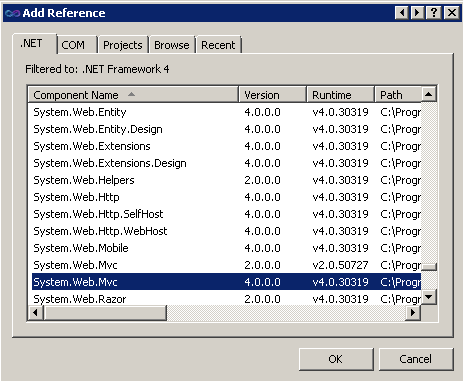
Choose *Empty Web API* from Project Template



1. The template comes with WebApi.Core 4.0.20710.0. Update WebApi.Core to 4.0.30506.0 from Nuget. Run update-package -id microsoft.aspnet.webapi.core -version 4.0.30506.0 from Package Manager Console in VS
2. Install the tracing package to 4.0.30506.0 from Nuget for troubleshooting purposes. Run install-package -id microsoft.aspnet.webapi.tracing -version 4.0.30506.0 from Package Manager Console in VS
3. If you are running VS 2012 or higher, you could simply get the latest versions of those packages by omitting the version parameter
4. Add System.Web.Mvc to References. The template forgot to add this dll



Choose run time version v4.0.30319 of the dll



1. Add App\_Start folder. Add WebApiConfig.cs and FilterConfig.cs to App\_Start

##### Code for the WebApiConfig Class:

|  |
| --- |
| using System.Web.Http;  namespace WebApiArcgisApiDemo.App\_Start  {  public static class WebApiConfig  {  public const string DEFAULT\_ROUTE\_NAME = "MyDefaultRoute";  public static void Register(HttpConfiguration config)  {  config.Routes.MapHttpRoute(  name: DEFAULT\_ROUTE\_NAME,  routeTemplate: "api/{controller}/{id}",  defaults: new { id = RouteParameter.Optional }  );  config.EnableSystemDiagnosticsTracing();  }  }  } |

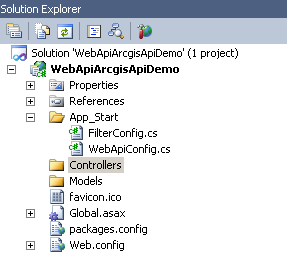
##### Code for the FilterConfig Class:

|  |
| --- |
| using System.Web.Mvc;  namespace WebApiArcgisApiDemo.App\_Start  {  public class FilterConfig  {  public static void RegisterGlobalFilters(GlobalFilterCollection filters)  {  filters.Add(new HandleErrorAttribute());  }  }  } |

1. Update Global.asax to initializes Routes and Filters

|  |
| --- |
| using System.Web;  using System.Web.Http;  using System.Web.Mvc;  using WebApiArcgisApiDemo.App\_Start;  namespace WebApiArcgisApiDemo  {  // Note: For instructions on enabling IIS6 or IIS7 classic mode,  // visit http://go.microsoft.com/?LinkId=9394801  public class WebApiApplication : HttpApplication  {  protected void Application\_Start()  {  // Add these two lines to initialize Routes and Filters:  WebApiConfig.Register(GlobalConfiguration.Configuration);  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);  }  }  } |

1. Add a Models folder and a Controllers folder to maintain convention. You should be able to build the project without any errors.



### Creating Models

A model is comprised of the data access class and the data definition class. The data access class should implement the data access interface. Each subfolder within *Models* represents a database instance. In our example, we have a database instance *DB1*, the data access and the data definition classes are stored in the subfolder *DB1*.



Code for the data definition class, DB1DataModel.

|  |
| --- |
| using System.Collections.Generic;  namespace WebApiArcgisApiDemo.Models.DB1  {  public class ESRIService  {  public int id { get; set; }  public string URL { get; set; }  public string Name { get; set; }  public List<ESRILayer> Layers { get; set; }  }  public class ESRILayer  {  public int id { get; set; }  public string Name { get; set; }  public int LayerPosition { get; set; }  public string ImageURL { get; set; }  }  } |

Code for the data access interface, IDAO

|  |
| --- |
| using System.Collections.Generic;  namespace WebApiArcgisApiDemo.Models  {  public interface IDAO  {  List<object> GetTable(string queryKey, List<string> parameterList = null);  }  } |

The data access class implements the data access interface and is responsible for querying the database to get a list of ESRI services to be consumed by the client code later. In our example, we hardcode a list of ESRIService objects.

Code for the data access class, DB1DAO

|  |
| --- |
| using System.Collections.Generic;  using System.Linq;  namespace WebApiArcgisApiDemo.Models.DB1  {  public class DB1DAO: IDAO  {  public List<object> GetTable(string queryKey, List<string> parameterList = null)  {  List<object> returnList = new List<object>();  if (queryKey.ToLower() == "getservices")  returnList = GetServices();  return returnList;  }  private List<object> GetServices()  {  List<ESRIService> serviceList = new List<ESRIService> ();  ESRIService service0 = new ESRIService();  service0.id = 0;  service0.Name = "Sample data of the US";  service0.URL = @"http://sampleserver6.arcgisonline.com/ArcGIS/rest/services/USA/MapServer";    List<ESRILayer> layerList = new List<ESRILayer> ();  ESRILayer layer0 = new ESRILayer();  layer0.id = 0;  layer0.LayerPosition = 0;  layer0.Name = "Cities";  layerList.Add(layer0);  ESRILayer layer1 = new ESRILayer();  layer1.id = 1;  layer1.LayerPosition = 1;  layer1.Name = "Highways";  layerList.Add(layer1);  ESRILayer layer2 = new ESRILayer();  layer2.id = 2;  layer2.LayerPosition = 2;  layer2.Name = "States";  layerList.Add(layer2);  ESRILayer layer3 = new ESRILayer();  layer3.id = 3;  layer3.LayerPosition = 3;  layer3.Name = "Counties";  layerList.Add(layer3);  service0.Layers = layerList;  serviceList.Add(service0);    return serviceList.OfType<object>().ToList();  }  }  } |

### Using Unity

1. Install Unity.WebAPI version 0.10 in your project. Run install-package unity.webapi -version 0.10 from Package Manager Console.
2. The package updates References and creates Bootstrapper.cs and Unity.WebAPI.README.txt in your project.
3. We register the data access object within the container in Bootstrapper.cs so that we can inject the data access object into the controller later on.

Code for Bootstrapper.cs

|  |
| --- |
| using System.Web.Http;  using Microsoft.Practices.Unity;  using WebApiArcgisApiDemo.Models;  using WebApiArcgisApiDemo.Models.DB1;  namespace WebApiArcgisApiDemo  {  public static class Bootstrapper  {  public static void Initialise()  {  var container = BuildUnityContainer();  GlobalConfiguration.Configuration.DependencyResolver = new Unity.WebApi.UnityDependencyResolver(container);  }  private static IUnityContainer BuildUnityContainer()  {  var container = new UnityContainer();  // register all your components with the container here  // The default concrete implementation of IDAO is DB1DAO and  // it is created as a singleton object  container.RegisterType<IDAO, DB1DAO>(new ContainerControlledLifetimeManager());  return container;  }  }  } |

1. Add Bootstrapper.Initialise() method to Application\_Start in Global.asax.cs

|  |
| --- |
| protected void Application\_Start()  {  // Add these two lines to initialize Routes and Filters:  WebApiConfig.Register(GlobalConfiguration.Configuration);  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);  // Add unity bootstrapper initialise code  Bootstrapper.Initialise();  } |

### Creating Controllers

1. Right click on the Controllers folder in your project -> Add -> Controller. Change the controller name to DB1ServicesController. Select Empty API controller template. Click Add. This will create DB1ServicesController.cs in the folder.
2. Add using WebApiArcgisApiDemo.Models; to the Using statements of DB1ServicesController.cs
3. We are ready to inject our data access object to the controller’s constructor. The constructor parameter db1DAO will be resolved by Unity. In BootStrapper.cs, we have specified the default concrete implementation of IDAO being DB1DAO and it is a singleton object.

public DB1ServicesController(IDAO db1DAO)

{

\_db1DAO = db1DAO;

}

1. Now we are ready to use the data access object in the controller.

public List<object> GetServices()

{

return \_db1DAO.GetTable("getservices", null);

}

Code for DB1ServicesController.cs

|  |
| --- |
| using System.Collections.Generic;  using System.Web.Http;  using WebApiArcgisApiDemo.Models;  namespace WebApiArcgisApiDemo.Controllers  {  public class DB1ServicesController : ApiController  {  private readonly IDAO \_db1DAO;  public DB1ServicesController(IDAO db1DAO)  {  \_db1DAO = db1DAO;  }  public List<object> GetServices()  {  return \_db1DAO.GetTable("getservices", null);  }  }  } |

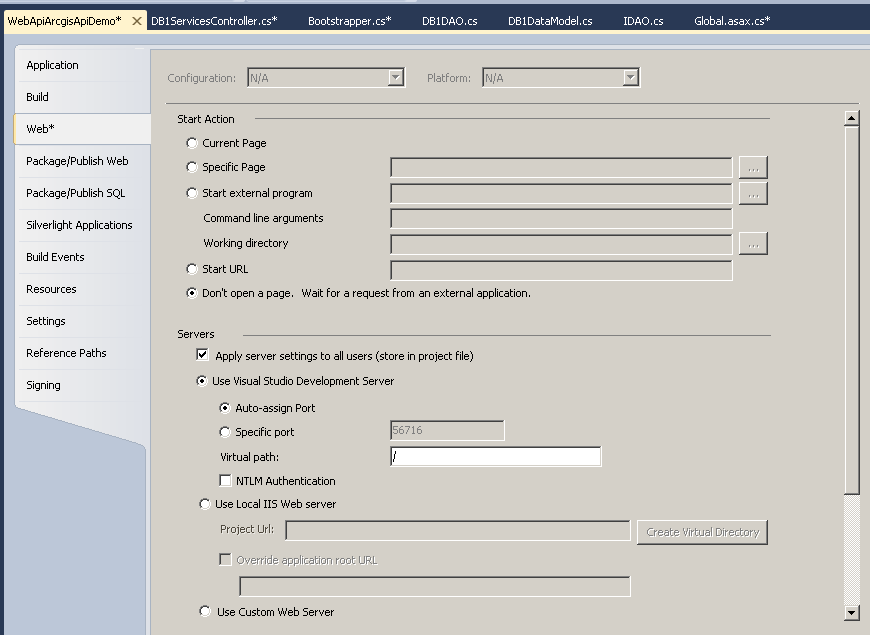
### Testing REST Service

1. Add the following line to Application\_Start () method in Global.asax.cs to make sure the service returns a JSON response, so you can view it in your browser

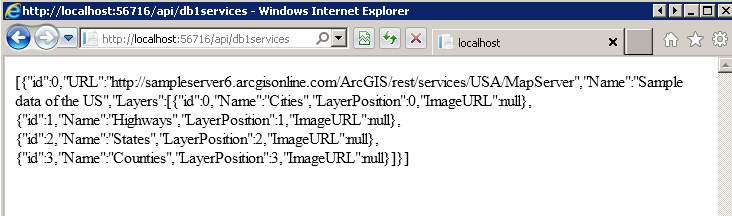
GlobalConfiguration.Configuration.Formatters.JsonFormatter.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));

|  |
| --- |
| protected void Application\_Start()  {  // Add these two lines to initialize Routes and Filters:  WebApiConfig.Register(GlobalConfiguration.Configuration);  FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);  // Add unity bootstrapper initialise code  Bootstrapper.Initialise();  // Return JSON responses  GlobalConfiguration.Configuration.Formatters.JsonFormatter.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));  } |

1. Go to the properties of the project and change Start Action to ‘Don’t open a page. Wait for a request from an external application.” We want to trigger the service from a browser.



1. Run the project
2. Start an Internet Browser, navigate to localhost:{port\_number}/api/db1services. You should see the ESRIService list, there is only one service record on the list; the service has four layers. The routeTemplate defined in the static Register method in WebApiConfig.cs dictates the format of the URI. In our example, we have routeTemplate: "api/{controller}/{id}"



In the following section, we will talk about how to consume the exposed data using Dojo and ArcGIS API for JavaScript.

# Client Side

## Version Compatibility

The Dojo version used in ArcGIS API for JavaScript should be the same as the local Dojo version. For example, if you use ArcGIS API for JavaScript 3.7, which is implemented using Dojo version 1.9.1, then you should use Dojo version 1.9.1 for local Dojo modules. In addition, we use a custom Dojo dijit, cbTree, to represent our table of contents. We use v0.9.4 which is compatible with Dojo version 1.9.1. Here is a summary of the versions we use in this example.

|  |  |
| --- | --- |
| ArcGIS API for Javascript | 3.7 |
| Dojo | 1.9.1 |
| cbTree | 0.9.4 |

## Dojo boilerplate

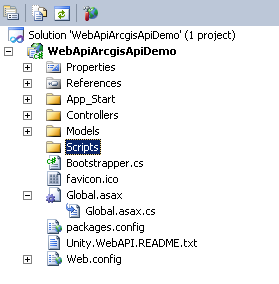
If you don’t want to start from scratch, the Dojo Boilerplate includes everything you need to start a barebones yet scalable project that is ready to build. The boilerplate ensures that project has a *pro forma* directory layout that is ready to build. You can download a copy of the boilerplate from <https://github.com/csnover/dojo-boilerplate>.

After getting a copy of the boilerplate, we will make the following changes:

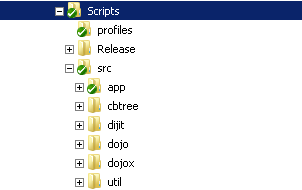
* Use the ESRI CDN to load the require module and the ESRI modules. We then use the require module to load the local Dojo modules.
* Refactor main.js. Instead of loading a dialog box before, we will call mainDesktop.js which will in turn load the map and the toc modules.
* The boilerplate has empty dojo, dijit, dojox and util directories. You need to copy the Dojo source code to those directories. You may remove the packages that you do not use. For example, if you are not using dojox, you can delete that directory. Then you need to remove the packages reference of dojox in run.js as well as in app.profile.js

## Dojo directory structure

First we need to create a *Scripts* directory within the WebAPI project to house all the client code.



Within the *Scripts* directory, there should be three sub directories: a *profiles* directory that stores Dojo build scripts; a *release* directory that stores production ready files; lastly, a *src* directory that contains all of your packages, including your custom code in the app subdirectory. The Dojo build command assumes that your project is in such a format.



## Custom code

The custom code starts in mainDesktop.js; we do three things there. First we consume the REST service and convert the JSON response to a JavaScript object. Second, we pass the JavaScript object to our custom map object to render the map. Lastly, we pass the map object to the toc object to render the table of contents.

The map object is quite simple; it has two methods: *processDynamicLayers* for adding the dynamic layers to the ESRI map object; *ToggleLayer* for toggling the visibility of dynamic layers depending on the checkbox status in the table of contents.

In the constructor of the toc object, we set up the datasource for the table of contents tree. We use the cbTree dijit which requires the data to be arranged in a specific way. Once the table of contents is set up, we will simply call the ToggleLayer method of the map object to toggle the visibility of the layer.

## Dojo build

In order to build Dojo projects, they should have the correct directory layout. You need to have Java installed on your workstation and the executable path should also be in your path environment. Here is an example of a build command:

java -Xms256m -Xmx256m -cp ../shrinksafe/js.jar;../closureCompiler/compiler.jar;../shrinksafe/shrinksafe.jar org.mozilla.javascript.tools.shell.Main ../../dojo/dojo.js baseUrl=../../dojo load=build --require ../../app/run.js --profile ../../../profiles/app.profile.js --releaseDir ../release

There are a few things to be noted:

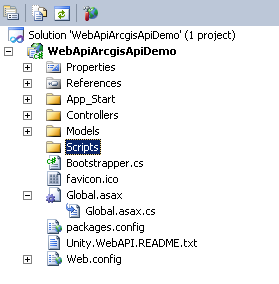
* You need Java or Node to build
* --require should be ahead of –profile. Otherwise, the settings of –require overwrite the settings of –profile and it will not build successfully.
* With the exception of –releaseDir, all other parameters are relative to the *buildscripts* directory. –releaseDir is relative to the location of the build profile

I created a build batch file and placed in the profiles directory. The batch file changes the directory to *buildscripts* and uses Java to build the project into the *release* directory. Lastly, it copies Index.html from *src* to *release*. The content of the *release* directory can be deployed to a Web server.

## Client code walkthrough

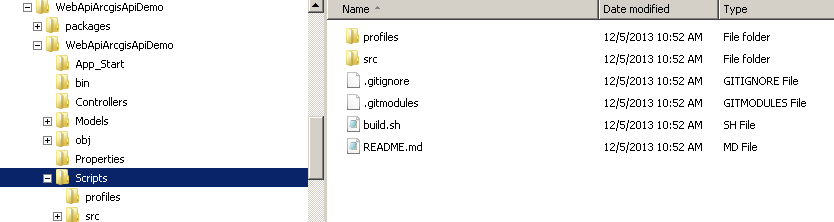
### Downloading the Dojo boilerplate

Create the Scripts folder within the Visual Studio project



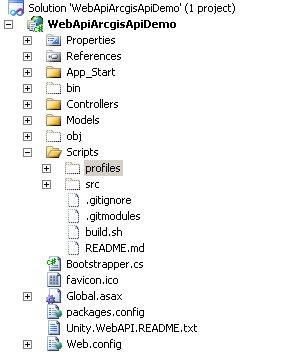
Download a copy of Dojo boilerplate [here](https://github.com/csnover/dojo-boilerplate)

Copy the content of the downloaded zip file to the *Scripts* directory using Windows Explorer



### Including files and directories in Visual Studio project

Click Show All Files button in Solution Explorer in VS



Right click on profiles -> Include in project

Right click on src -> Include in project

Right click on dijit (under src directory) -> Exclude From Project

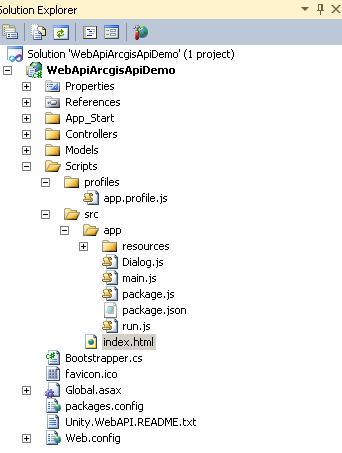
Right click on dojo(under src directory) -> Exclude From Project

Right click on dojox(under src directory) -> Exclude From Project

Right click on util (under src directory) -> Exclude From Project

Click on server.sh file -> Exclude From Project

Turn off Show All Files button, your project should look like this



### Copying dojo, dijit and util folders of Dojo 1.9.1 to VS project

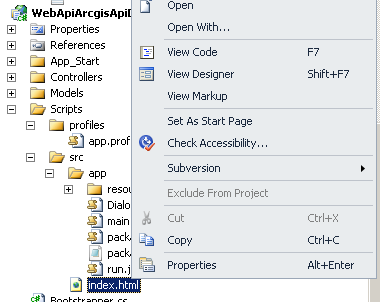
Download a full, uncompressed source release of the Dojo Toolkit 1.9.1 in the zip format [here](http://dojotoolkit.org/download/)

Copy the dojo, dijit, and util folders within the zip file to the src directory of the VS project using Windows Explorer, overwriting the existing empty dojo, dijit and util folders

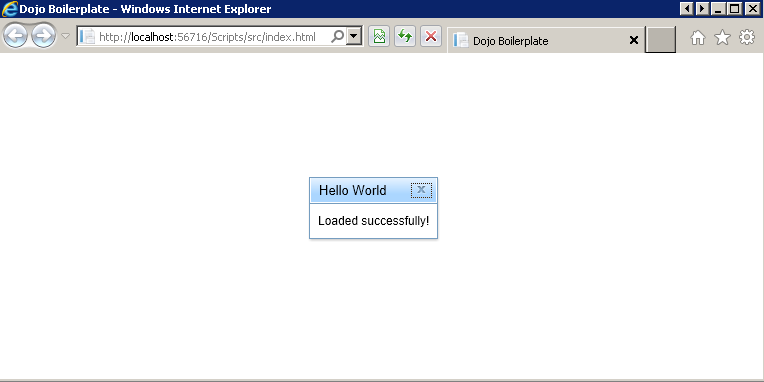
Delete the empty dojox folder

### Testing Dojo boilerplate

Set Index.html as Start Page



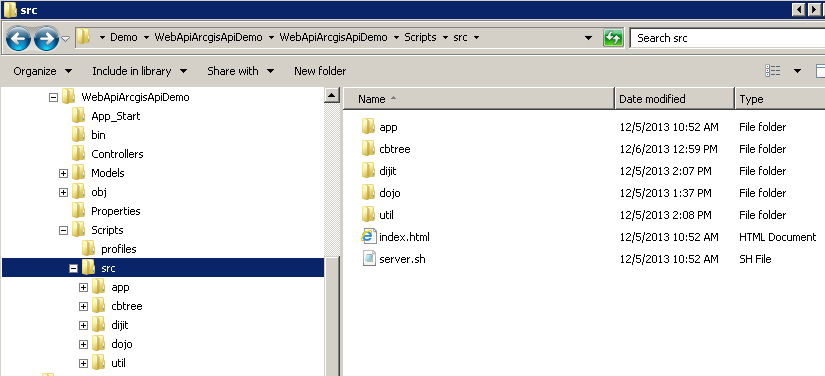
Run the project. You should see the Hello World dialog box



### Dijit Tree

We use a tree Dijit to render the table of contents. The instructions for using the tree Dijit are as following:

* Download a zip copy of the source code [here](https://github.com/pjekel/cbtree)
* Open up the zip file in Windows Explorer. Copy the cbtree-master folder to the src directory in Windows Explorer
* Rename cbtree-master to cbtree in Windows Explorer



### run.js

We remove dojox from the packages. We load dojo, dijit and cbtree locally.

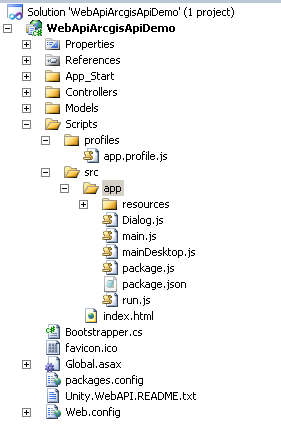
|  |
| --- |
| packages: [  'dojo',  'dijit',  'cbtree',  { name: 'app', location: 'app', map: {} }  ] |

### main.js

The boilerplate loads a dialog box in main.js. We will modify main.js to load mainDesktop.js.

Code for main.js

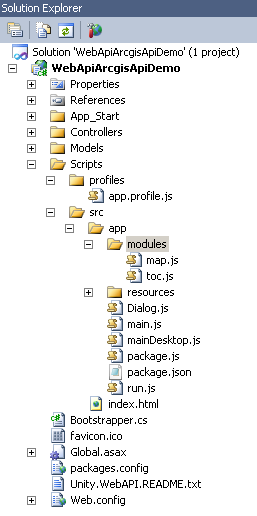
|  |
| --- |
| if (has('host-browser')) {  require(['./mainDesktop'], function () { });  }  else {  console.log('Hello from the server!');  } |

Add mainDesktop.js to the app directory in the Visual Studio project

Run the project again, you will see a blank page. Obviously, mainDesktop.js has no code!

### modules

Add a modules folder within the app folder. Add two blank js files, map and toc



### Index.html

So far, we almost finish with scaffolding the client side code. Index.html calls run.js. Run.js loads dojo, dijit and cbtree locally, then calls main.js, which in turn calls mainDesktop.js, which creates the map and the toc objects. There is only one thing left to do, loading the ESRI modules via the CDN. We will do that here in index.html.

Instead of using the local dojo.js to load the require module (which is used to load all other packages), we will use the ESRI CDN to load the module. We change the following line in Index.html from

<script data-dojo-config="async: 1, tlmSiblingOfDojo: 0, isDebug: 1" src="dojo/dojo.js"></script>

to

<script data-dojo-config="async: 1, tlmSiblingOfDojo: 0, isDebug: 1" src="\_http://js.arcgis.com/3.7/"></script>

We then build our UI in Index.html. It’s simple enough, a border container that has three panes.

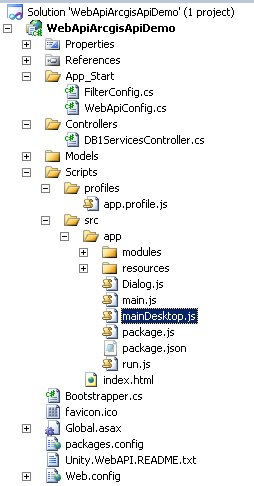
|  |
| --- |
| <div id="mainDiv" data-dojo-type="dijit/layout/BorderContainer">  <div id="headerDiv" data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'top'">Map</div>  <div id="tocDiv" data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'leading'"></div>  <div id="mapDiv" data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'center'"></div>  </div> |

Now, we are done with scaffolding the client code. We just need to write application specific code for mainDesktop.js, map.js and toc.js

### mainDesktop.js

This class does three things. First, it parses the Dojo containers defined in Index.html. Second, it consumes the Web API service. Lastly, it creates a map and a toc object using the WebAPI JSON response.

Please note the Scripts directory is a sibling of the Controllers directory in our example. To access our WebAPI service, we can simply go up two directories from mainDesktop.js to get to the Controllers level and call the Web API service using the relative path.



Code for mainDesktop.js

|  |
| --- |
| define(['app/modules/map', 'app/modules/toc', 'dojo/parser', 'dojo/store/JsonRest', 'dojo/ready', 'dijit/layout/BorderContainer', 'dijit/layout/ContentPane'],  function (Map, TOC, Parser, JsonRest, READY) {  READY(function () {  Parser.parse();  var basemapURL = 'http://services.arcgisonline.com/ArcGIS/rest/services/World\_Terrain\_Base/MapServer';  this.\_dynamicServicesStore = new JsonRest({  target: "../../api/DB1Services/"  });  // Query the entire aimt dynamic services  this.\_dynamicServicesStore.query().then(function (dynamicServices) {  var map = new Map('mapDiv', basemapURL, dynamicServices);  var toc = new TOC(dynamicServices, map);  });  });  }); //closing require brackets |

### map.js

The map class has a constructor which takes on three parameters: mapDivName, baseMapURL and dynamicServices. mapDivName is the map div id in Index.html. baseMapURL is the ESRI service url for the basemap,. And dynamicServices is the JSON response from our WebAPI service.

The class has two methods. ProcessDynamicLayers iterates the JSON response from the WebAPI service and adds each ESRI service. In our case, we have only one ESRI service endpoint containing four layers: Cities, Highways, States, and Counties. ToggleLayer toggles the visibility of the layer; this method is called whenever a user toggles the layer list in the table of contents. Please note we are able to reference the ESRI packages here because we loaded the require module via the ESRI CDN.

Code for map.js

|  |
| --- |
| define(['dojo/\_base/declare', 'dojo/\_base/array', 'esri/map', 'esri/layers/ArcGISDynamicMapServiceLayer'],  function (declare, Array, Map, ArcGISDynamicMapServiceLayer) {  return declare(null, {  \_mapDivName: null,  \_baseMapURL: null,  \_dynamicServices: null,  \_map: null,  constructor: function (mapDivName, baseMapURL, dynamicServices) {  this.\_mapDivName = mapDivName;  this.\_baseMapURL = baseMapURL;  this.\_dynamicServices = dynamicServices;  //Create map  this.\_map = new Map(this.\_mapDivName, { center: [-86.636, 40.882], zoom: 10 });  this.processDynamicLayers(this.\_dynamicServices);  },  ToggleLayer: function (serviceID, layerPosition, checked) {  var selectedService = this.\_map.getLayer(serviceID);  if (checked) {  if (selectedService.visibleLayers[0] == -1)  selectedService.setVisibleLayers([layerPosition]);  else {  var visibleLayers = selectedService.visibleLayers;  visibleLayers.push(layerPosition);  selectedService.setVisibleLayers(visibleLayers);  }  }  else {  var visibleLayers = selectedService.visibleLayers;  if (visibleLayers.length == 1)  selectedService.setVisibleLayers([-1]);  else {  var index = visibleLayers.indexOf(layerPosition);  visibleLayers.splice(index, 1);  selectedService.setVisibleLayers(visibleLayers);  }  }  },  processDynamicLayers: function (dynamicServices) {  var layers = [];  //Add base layer  var baseLayer = new ArcGISDynamicMapServiceLayer(this.\_baseMapURL);  layers.push(baseLayer);  //Add dynamic layer  Array.forEach(dynamicServices, function (dynamicService, i) {  var layer = new ArcGISDynamicMapServiceLayer(dynamicService.URL, { id: dynamicService.id });  layer.setVisibleLayers([-1]);  layers.push(layer);  });  this.\_map.addLayers(layers);  }  });  }); |

### toc.js

In order to create the data source for cbtree, we need to transform the JSON response to a format that is acceptable to cbtree. For a more detailed discussion, you should consult [cbtree documentation](https://github.com/pjekel/cbtree/wiki). Once we transform the JSON response, we simple create a model using the data and pass the model to the tree. And Voilà! We have our table of contents! When a user clicks on a layer item, the code will trigger the ToggleLayer method in the map object.

Code for toc.js

|  |
| --- |
| define(['dojo/\_base/declare', 'dojo/\_base/array', 'dojo/\_base/lang', 'cbtree/store/ObjectStore', 'cbtree/Tree', 'cbtree/model/ForestStoreModel', 'cbtree/extensions/TreeStyling', 'dojo/domReady!'],  function (declare, Array, Lang, ObjectStore, Tree, ForestStoreModel) {  return declare(null, {  \_mapObj: null,  constructor: function (dynamicServices, mapObj) {  this.\_mapObj = mapObj;  //Set tocdata for cbtree  var tocData = [];  var parentEntry = { id: "tableHeader", Name: "TOC" };  tocData.push(parentEntry);  Array.forEach(dynamicServices, function (dynamicService, i) {  var serviceEntry = Lang.clone(dynamicService);  serviceEntry.type = "service";  serviceEntry.parent = "tableHeader";  delete serviceEntry.Layers;  tocData.push(serviceEntry);  Array.forEach(dynamicService.Layers, function (dynamicLayer, i) {  var layerEntry = Lang.clone(dynamicLayer);  layerEntry.type = "layer";  layerEntry.parent = serviceEntry.id;  tocData.push(layerEntry);  });  });  //Set up Store, Model, Tree  var store = new ObjectStore({ data: tocData });  var model = new ForestStoreModel({ store: store,  labelAttr: 'Name',  query: { id: "tableHeader" }  });  var tree = new Tree({ model: model, showRoot: false, id: "serviceTree" });  tree.placeAt("tocDiv");  tree.on("checkBoxClick", Lang.hitch(this, this.LayerClicked));  tree.startup();  },  LayerClicked: function (item, treeNode, event) {  if (item.type == "layer") {  this.\_mapObj.ToggleLayer(item.parent, item.LayerPosition, item.checked);  }  }  }); //Declare closing brackets  }); |

### Dialog.js

The boilerplate comes with Dialog.js, you can simply delete it from the boilerplate. We do not need it for our example.

### CSS

In Index.html, we should add the ESRI CDN CSS file

<link rel="stylesheet" href="http://js.arcgis.com/3.7/js/esri/css/esri.css">

In app.css under the resources directory, add the styles for the Index page elements.

|  |
| --- |
| /\*\*  \* If you are developing a project that makes use of Dijit widgets, you will probably want to include the two files  \* below; otherwise, you can remove them. When building for release, Dojo will automatically combine all of your  \* @imported CSS files into a single file.  \*  \* It is highly, highly recommended that you develop your CSS using Stylus: http://learnboost.github.com/stylus/.  \*/  @import '../../dojo/resources/dojo.css';  @import '../../dijit/themes/claro/claro.css';  @import "../../cbtree/themes/claro/claro.css";  html, body {  height: 100%; width: 100%;  margin: 0; padding: 0;  }  body{  background-color:white; overflow:hidden;  font-family: "Kimberley", sans-serif  }  #mainDiv {  gutters:false;  width:100%;  height:100%;  }  #headerDiv {  -moz-border-radius: 5px;  margin:2px;  padding-top: 6px;  padding-right:20px;  background-color:#929761;  color:#421b14;  border: solid 2px #79663b;  height:55px;  text-align:right;  color:#000;  font-size:18pt;  font-weight:bold;  }  #mapDiv {  margin:3px;  padding:1px;  border:solid 2px #79663b;  width: 80%;  }  #tocDiv {  margin:3px;  padding:1px;  border:solid 2px #79663b;  width: 20%;  } |

## Limitations and Issues

1. This example is tested in Internet Explorer 9. It will not run in Mozilla Firefox.
2. It seems that there is a timing glitch. Often, the page does not load correctly and will have the following error message generated from the ArcGIS API.

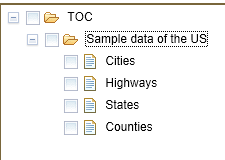
|  |
| --- |
| SCRIPT5007: Unable to get value of the property 'toLowerCase': object is null or undefined  [3.7, line 32 character 82](file:///C:\Users\devyshi\Desktop\0) |

This problem does not occur in ArcGIS API for Javascript 3.6. The following is a more stable platform to run this example. To downgrade the example to using 3.6, you need to change the ESRI CDN to 3.6 in Index.html and copy 1.8.3 dojo, dijit, and util directories to the src directory in the project as well as copying cbtree 0.9.3 to the src directory.

|  |  |
| --- | --- |
| ArcGIS API for Javascript | 3.6 |
| Dojo | 1.8.3 |
| cbTree | 0.9.3 |

Lastly, when you do run into this problem in 3.7, you could open the URL in a new browser page. It often works.

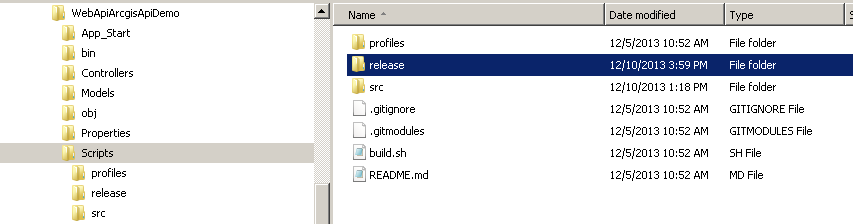
1. For the table of contents, toggling the service (Sample Data of the US) or the header (TOC) has no effect on the map. You need to toggle the layer checkboxes themselves (i.e. Cities).



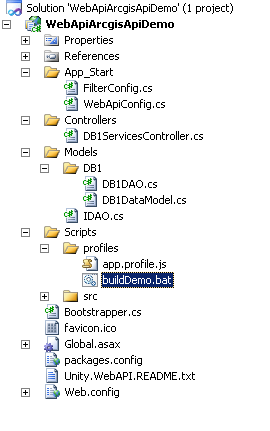
## Building the client code

The boilerplate is ready to build. We just need to do a few tweaks before we can build the entire project to be production ready. Java should be installed on the dev workstation before we can run the build script.

1. create a release directory in Windows Explorer



1. Create an empty buildDemo.bat to the profiles directory in VS2010.



First, we set up the base directory for the build; it’s the location of the *Scripts* directory. Then we clear the *release* directory. Now we are ready to build. After the build, we copy the index.html from *src* to *release* and change the home directory back to *profiles*. Please note that we need to be in *buildscripts* when we build. The batch file takes care of that for us.

Code for buildDemo.bat

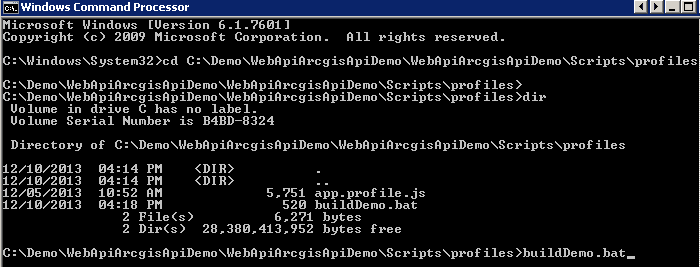
|  |
| --- |
| SET BASE\_DIR=C:\Demo\WebApiArcgisApiDemo\WebApiArcgisApiDemo\Scripts  cd "%BASE\_DIR%\src\util\buildscripts"  rd /s /q "%BASE\_DIR%\release"  java -Xms256m -Xmx256m -cp ../shrinksafe/js.jar;../closureCompiler/compiler.jar;../shrinksafe/shrinksafe.jar org.mozilla.javascript.tools.shell.Main ../../dojo/dojo.js baseUrl=../../dojo load=build --require ../../app/run.js --profile ../../../profiles/app.profile.js --releaseDir ../release  copy "%BASE\_DIR%\src\index.html" "%BASE\_DIR%\release"  cd "%BASE\_DIR%\profiles" |

1. Modify app.profile.js to remove app/dialogue and to include the new modules (map and toc) and mainDesktop in the build.

Partial code for app.profile.js

|  |
| --- |
| layers: {  'dojo/dojo': {  include: [ 'dojo/i18n', 'dojo/domReady', 'app/main', 'app/mainDesktop', 'app/run' ],  boot: true,  customBase: true  },  'app/modules/map': {},  'app/modules/toc': {}  }, |

1. To build the Dojo code, open a cmd; change directory to *profiles*; run buildDemo.bat



1. https://dojotoolkit.org/documentation/tutorials/1.7/cdn/ [↑](#footnote-ref-1)