* Course Overview
  + Design your api with associations
* Introduction
  + Build api with asp.net web api
* How HTTP Works
  + Request: verb, headers, content
  + Response: status code, headers, content
  + Server is stateless
    - Connectionless
  + Verbs
    - Get: retrieve a resource
    - Post: add a new resource
    - Put: update an existing resource
    - Patch: update an existing resource with set of changes
    - Delete: removing the existing resource
  + REST
    - REpresentational State Transfer
      * Separation of client and server
      * Server Requests are Stateless
      * Cacheable Requests
      * Uniform Interface
    - Problems
      * Too difficult to be qualified as “REST”
      * Dogma of Rest vs Pragmatism
      * Structure architectural style
      * The need to be productive
* What are Resources
  + People, invoices, payments, products
  + Are resources just entities?
    - Resources has context
    - A person or an order in the order details
  + URIs
    - Uniform resource identifiers
    - Paths to resources
    - ex) ap.yourserver.com/people
  + Query strings for non-data elements
    - Format, sorting, searching
* Where We’re Headed
  + Camp -> location -> talks -> speaker
* Design the API
  + <http://.../api/camps>
  + <https://../api/camps/moniker>
  + <https://../api/camps/moniker/talks>
  + <https://../api/camps/moniker/talks>?topic=database
* Starting the Project
  + Get the starting code from ‘<https://github.com/psauthor/webapifundamentals>’
  + Global.asax
    - Set up that runs the configs files in App\_start
  + AutofacConfig
    - Dependency injection
* Using Postman
  + Download postman
  + Right click on project
    - Click properties
    - Click web
    - Click ‘Don’t open a page’
* Getting the Project Ready
  + In web.config
    - There is a connection string to the database
  + The database will be generated the first we need to access data
  + Click on view
    - Click on ‘SQL Server Object Explorer’
    - Check out the server
* What We’ve Learned
* Introduction
  + Request comes in with route in the form of a url
  + Search routes that match
  + Matches to action on a controller
  + Execution action
  + Return results as a response
* Creating an Action
  + Create a new class filer in the controller file
    - CampsController.cs
    - Have it inherit from ApiController
    - Ex) public class CampsController : ApiController
  + The routing will find a [name] + Controller and is an api controller
* Status Codes
  + Codes
    - 200: ok
    - 302: found
    - 400: bad request
    - 500: internal error
    - Etc
* Using Status Codes
  + To return payload and status codes return IHttpActionResult
  + ex)
    - public IHttpActionResult Get()
    - {
    - return Ok(new { Name = “Shawn”, Occupation = “Teacher });
    - }
* Using GET for Collections
  + Update database
  + Use migrations to add some initial data
  + Open pack manager console
    - Type ‘ Update-Database’
    - Make sure migrations have been created an applied
  + Create a constructor to be able to use dependency injection
    - private readonly ICampRepository \_repository;
    - public CampsController(IcampRepository repository)
    - {
    - \_repository = repository;
    - }
  + To set up dependency injection
    - Have a AutofacConfig file in App\_Start
    - Then register the services you want to inject
  + Then in action use the service
    - public async Task<IhttpActionResult> Get()
    - {
    - try{
    - var result = await \_repository.GetAllCampsAsync();
    - return Ok(result);
    - }
    - catch
    - {
    - return InternalServerError();
    - }
    - }
* Why Models instead of Entities
  + Payload is a contract with your users
  + Filter data for security
  + Return models instead of entities
  + Create a models folder
  + Create a new file ‘CampModel’
    - public class CampModel
    - {
    - public string Name { get; set;}
    - public string Moniker { get; set;}
    - ….
    - }
  + Map from model to entity and back using a mapping tool
  + Click on project
    - Click on manage nuget packages
    - Search for automapper and install
  + Then in Data folder create a profile for mapping and have it inherit from profile
    - public class CampMappingProfile : Profile
  + in the constructor for the class create maps
    - public CampMappingProfile()
    - {
    - CreateMap<Camp, CampModel>().ReverseMap();
    - }
  + Unless you have exceptions the mapping will use conventional ways to map
    - Based on variables spelling, etc
  + Then in AutofacConfig, register the mapping profile so we can use dependency injection
    - private static void RegisterServices(ContainerBuilder bldr)
    - {
    - var config = new MapperConfiguration(cfg =>
    - {
    - cfg.AddProfile(new CampMappingProfile());
    - });
    - bldr.RegisterInstance(config.CreateMapper())
    - .As<IMapper>()
    - .SingleInstance();
    - …
    - }
  + Then in the controller add the service into the constructor
    - private readonly ICampRepository \_repository;
    - private readonly IMapper \_mapper;
    - public CampsController(ICampRepository repository, Imapper mapper)
    - {
    - \_repository = repository;
    - \_mapper = mapper;
    - }
  + Then in action, use the mapper to map between the model and entity
    - …
    - var result = await \_repository.GetAllCampsAysnc();
    - var mappedResult = \_mapper.Map<IEnumerable<CampModel>>(result);
    - return Ok(mappedResult);
  + output is default to json
* Contract Serialization in Web…
  + If the request specifies a return format, web api can infer can return in the same format
  + In WebApiConfig you can set up serializers
    - public static void Register(HttpConfiguration config)
    - {
    - AutofacConfig.Register();
    - config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
    - …
    - }
* Getting an Individual Item
  + Using a route
    - [Route(“{moniker}”)]
    - public async Task<IHttpActionResult> Get(string moniker)
    - {
    - try
    - {
    - var result = \_repository.GetCampAsync(moniker);
    - If(result == null) return NotFound();
    - return Ok(\_mapper.Map<CampModel>(result));
    - }
    - catch(Exception ex)
    - {
    - return InternalServerError(ex);
    - }
    - }
    - The url to access this action is ‘http://localhost:6600/api/camps/{moniker}’
  + The moniker in the route will map to the moniker in the parameters
    - The url will map it
  + In the WebApiConfig file
    - The MapHttpRoute() creates default routes for controllers
    - May need to comment out if it interferes with the desired routing
  + Can add route prefix to class to ensure correct routing
    - [RoutePrefix(“api/camps”)]
    - public class CampsController : ApiController
  + use on methods
    - [Route()]
    - public async Task<IHttpActionResult> Get() {}
    - [Route(“{moniker}”)]
    - public async Task<IHttpActionResult> Get(string moniker) {}
* Returning Related Data
  + Can include related complex data into the model(flattening complex related data)
  + In auto mapper you can prefix those data with the Entity name and automapper will map it
    - public class CampModel
    - {
    - public string Name { get; set;}
    - ….
    - public string LocationVenueName { get; set; }
    - ….
    - }
  + Can create custom mapping rules to map as well
    - CreateMap<Camp, CampModel>()
    - .ForMember(c => c.Venue, opt => opt.MapFrom( m => m.Location.VenueName)).ReverseMap();
  + Add one for each scenario that doesn’t match automapper convention
* Using Query Strings
  + Query strings should never be required
  + They only change behavior of what is happening
  + Ex)
    - public async Task<IHttpActionResult> Get(bool includeTalks = false) {}
  + update model to include talks
  + create a talk model
  + update mapping profile to take of the new models
* Implementing Searching
  + Use Http verbs to specify which verb the action is if the action is named something different then a verb
    - ex)[HttpGet]
  + Also use a route constraint to specify what type of data should be in the route(ex: age:int)
    - [Route(“searchByDate/{eventDate:datetime}”)]
    - [HttpGet]
    - public async Task<IHttpActionResult> SearchByEventDate( bool includeTalks = false) {}
* What We’ve Learned
* Introduction
  + URI Desgin
* Model Binding
  + To post data from post man
    - Select verb post
    - Set data to raw
    - Select JSON
  + Model binding takes request and bind it to properties in the method
    - [Route()]
    - public async Task<IHttpActionResult> Post(CampModel model){}
* Implementing POST
  + Should validate the model
  + [Route()]
  + public async Task<IHttpActionResult> Post(CampModel model)
  + {
    - try
    - {
    - if(await \_repository.GetCampAsync(model.Moniker) != null)
    - ModelState.AddModelError(“Moniker”, “Moniker in use”);
    - if(ModelState.IsValid)
    - {
    - var camp = \_mapper.Map<Camp>(model);
    - \_repository.AddCamp(camp);
    - if(await \_repository.SaveChangesAsync())
    - {
    - var newModel = \_mapper.Map<CampModel>(camp);
    - return Created(“GetCamp”, new { moniker = newModel.Moniker }, newModel);
    - }
    - }
    - }
    - ….
    - return BadRequest();
  + New to add the Name attribute to route that will get the camp
    - [Route(“{moniker}”, Name = “GetCamp”]
* Adding Model Validation
  + Add validation to the model by using attributes
    - public class CampModel
    - {
    - [Required]
    - public string Name { get; set; }
    - [Required]
    - public string Moniker { get; set; }
    - …
    - [Required]
    - [Range(1,30)]
    - public int Length { get; set; } = 1;
    - }
  + These validations is checked with “ModelState.IsValid”
* Implementing PUT
  + To map model in place
    - Map(source, destination)
    - Ex) \_mapper.Map(model, camp)
    - Takes model and put it into camp
* Implementing DELETE
* Where Are We?
  + Model Binding allows us to map bodies to .NET Objects
  + Use models to include validation
  + Using verbs to match to the operations is key to an API
* Introduction
  + /api/camps/atl2016/talks
  + Create a separate controller that is responsible for associated resources
* Create an Association Control…
  + Create a new controller with association route
    - [RoutePrefix(“api/camps/{moniker}/talks”)]
    - public class TalksController : ApiController {..}
  + then include the {moniker} as a parameter in all actions of this contorller
    - [Route()]
    - public async Task<IHttpActionResult> Get(string moniker)
    - {
    - }
* GET an Individual Talk
* POST a New Talk
  + ex)
    - …..
    - var camp = await \_repository.GetCampAsync(moniker);
    - if(camp != null)
    - {
    - var talk = \_mapper.Map<Talk>(model);
    - talk.Camp = camp;
    - }
    - if(model.Speaker != null)
    - {
    - var speaker = await \_repository.GetSpeakerAsync(model.Speaker.SpeakerId);
    - if(speaker != null) talk.Speaker = speaker;
    - }
    - ..
* Add Validation
* PUT to Update a Talk
  + When doing data mapping you can ignore the mapping of certain attributes
    - CreateMap<Talk, TalkModel>
    - .ReverseMap()
    - .ForMember(t => t.Speaker, opt => opt.Ignore())
    - .ForMember(t => t.Camp, opt => opt.Ignore())
* DELETE a Talk
* Where Are We?
  + API design should imply the structure to your users
  + Breaking up your API into Individual controllers is good
  + The association hierarchy can be deep or shallow
* Introduction
  + REST defines URIs as resources
    - But exceptions exist
  + Dont be afraid of functional apis
  + But avoid Remote Procedure Calls(RPC) style APIs at all costs
  + Use function api for
    - Operational needs
    - Avoid for reporting
    - Returning data != functional API
* Creating a Functional API
  + Ex)
    - public classOperationsController : ApiController
    - {
    - [HttpOptions]
    - [Route(“api/refreshconfig”)]
    - public IHttpActionResult RefreshAppSettings()
    - {
    - try
    - {
    - ConfigurationManager.RefreshSection(“AppSettings”);
    - return Ok();
    - }
    - catch(Exception ex)
    - {
    - return InternalServerError(ex);
    - }
    - }
    - }
  + Choose a verb that is less discoverable
* What Have We Learned?
  + Rest is important but be pragmatic
  + Create functional APIs if necessary
  + Don’t fall into RPC trap
    - Make it the exception not the rule
* Introduction
  + Once you publish an API, it’s set in stone
    - User/customer rely on the API
    - But requirements change
    - Need a way to evolve API
      * Don’t break old clients
  + Done with package versions
  + Needs to support new and old users
* API Versioning Schemes
  + There are lots of ways to version an API
  + Versioning in the URI
    - <https://foo.org/api/v2/customers>
    - <https://foo.org/api/Custoemr?v=2.0>
  + Versioning with headers
    - GET /api/camps HTTP/1.1
    - …
    - X-Version: 2.0
  + Versioning with Accept Header
    - GET /api/camps HTTP/1.1
    - …
    - Accept: application/json;version=2.0
  + Versioning with Content Type
    - GET /api/camps HTTP/1.1
    - …
    - Content-Type: application/vnd.yourapp.camp.v1+json
* Introducing Versioning
  + Install nuget package ‘Microsoft.AspNet.WebApi.Versioning’
  + In WebApiConfig add versioning support
    - public static void Register(HttpConfiguration confg)
    - {
    - ….
    - config.AddApiVersioning(cfg = >
    - cfg.DefaultApiVersion = new ApiVersion(1,1);
    - cfg.AssumeDefaultVersionWhenUnspecified = true;
    - cfg.ReportApiVersions = true;
    - );
    - ….
    - }
  + Without configuration settings, ApiVersioning uses query string to specify versioning by default
    - The default version is 1.0 for all
    - <http://localhost:6600/api/camps?api-version-1.0>
* Versioning Actions