**Web APIs**

**FORMATS**

1. JSON
2. XML

For XML, we can use:

1. XmlSerializer: Older, but opt out, meaning we set which properties to ignore
2. DataContractSerializer: newer, faster, but opt in, meaning we set which properties to use

To set a different serialization, go to:

1. Startup.cs > ConfigureWebApi > add to the top of the body:

Config.Formatters.XmlFormatter.<FormatterType> = true;

1. We need to set the api to send and accept xml data.

**JSON SETTINGS**

1. DateTimeKind
2. DateTimeZoneHandling : configures how the DateTimes will be formatted
3. DateFormatHandling : leave it in default to use ISO8601. Change if support for older Microsoft systems is needed
4. ContractResolver: Set serialization to use CamelCase or else.

e.g. var json = config.Formatters.JsonFormatter.SerializerSettings;

json.ContractResolver =new CamelCasePropertyNamesContractResolver;

1. Formatting: Use indentation for readability

e.g. json.Formatting = Formatting.Indented;

1. ReferenceLoopHandling: Fix problem with endless lazy loading loop
   1. Error
   2. Ignore e.g. json.ReferenceLoopHandling = ReferenceLoopHandlingIgnore;
   3. Serialize

**Exception handling**

1. Try Catch: for each controller, not so reusable
2. Exception Filters
3. Exception Loggers
4. Global Exception Handler

**Exception filters**

1. Create Filters folder
2. New class having the exception name
3. Inherit from *ExceptionFilterAttribute*
4. Override *OnException method*
5. Add logic for the specific exception that needs handling

Example:

public override void OnException (HttpActionExecutedContext context)

{

if (!(context.Exception is DbUpdateException)) return;

var sqlException = context.Exception?.InnerException?,InnerException as SqlException;

If (sqlException?.Number == 2627)

context.Response = new HttpResponseMessage(HttpStatusCode.Conflict);

context.Response = new HttpResponseMessage (HttpStatusCode.Conflict);

}

1. Add attribute before each controller or
2. Before the class name, if all controllers need to use it or
3. Add it to the *Startup.cs* for the entire project to use it

e.g. ConfigureWebApi > config.Filters.Add(new <ExceptionFilterName>)

**Exception Logger**

1. New class in the Loggers folder
2. Inherit from *ExceptionLogger*
3. Override *Log* method

e.g. var log = context.Exception.Message;

Debug.WriteLine(log);

1. Replace logger to the Startup.cs

ConfigureWebApi>

config.Services.Replace(typeof(IExceptionLogger), new <MyExceptionLoggerName>());

**Exception Handler**

1. New class in the Exception Handlers folder
2. Inherit from *ExceptionHandler*
3. Override the *Handle* method
4. Replace logger to the Startup.cs

ConfigureWebApi>

config.Services.Replace(typeof(IExceptionHandler), new <MyExceptionHandlerName>());

Example:

*#compiler directives: see documentation*

public override void Handle (ExceptionHandlerContext context)

{

#if DEBUG

var content = JsonConver.SerializeObject(context.Exception);

#else

var content = @”{ “”message””: “”Oops, something unexpected went wrong.””}”;

#endif

Context.Result = new ErrorContentResult (content, “application/json”, context.Request);

**Controllers**

***Parameter binding attributes:***

*[FromBody] specifies that the variable marked should come from the body.*

*[FromUri] that the variable marked should come from the URI.*

*Both can be used before controller parameters to override the default api way to get the input.*

***HTTP Verb Attributes:***

*Used to define what http verb each controller marked uses. There can be multiple controllers with the same name, but with different HttpVerb.*

*[HttpGet], [HttpPost],[HttpPut],[HttpPatch],[HttpDelete],*

*[HttpOptions],[HttpHead],[HttpVerb]*

***Returns/Responses:***

* 1. *Void (Don’t use, no feedback etc)*
  2. *HttpResponseMessage: Straightforward but the message must be customized*
  3. *IHttpActionResult: Most common. Allows data validation and responses*

*for errors. Not easy to see what caused the error, though.*

*e.g.* var data = GetData();

if(data==null)

return InternalServerError(); //500

if(!ModelState.IsValid)

return BadRequest(ModelState); //400

return Ok(data); //200

* 1. *List<RequestedObject>: Also common. Strongly typed but doesn’t easily allow different returns for errors etc. A workaround can be exception handling.*

*e.g.* if (!validation)

throw new HttpResponseException(new HttpResponseMessage

{

StatusCode = HttpStatusCode.BadRequest,

Content = new StringContent(“MinPrice must be less than MaxPrice”)

}

***Common ActionResult Helper methods:*** *Most can return data in the parameters. For custom responses or less used codes, use h and i.*

* + - * 1. Ok()
        2. BadRequest()
        3. NotFound()
        4. InternalServerError()
        5. *Unauthorized()*
        6. *Created()*
        7. *Conflict()*
        8. *Content()*
        9. *StatusCode()*

***DTO (Data Transfer Object)****: custom entity that is represented by a simple class, which has as properties only what we want to send and not the whole DB entity, for speed and security reasons.*

*Create the DTO class corresponding to the DB entity*

*Copy the needed properties etc*

*Controller: save the data from the query in a select statement and a new DTO object.*

*Return the DTO to the View*

**Validations:** same as WebApp, with DataAnnotations and the use of ModelState in the Controller. Return ModelState in the helper method to get all the errors or set a custom message etc.

1. **GET**:
   1. Use either **IHttpActionResult** or **List<**T>
   2. Parameters can be filters, but many is not a good practice. If more are needed, we should use a POST method.
   3. We need access to the DB
   4. Store the data
   5. Return exceptions or action results depending on return type.
   6. The software converts them to JSON automatically
2. **POST**:
   1. Return type as GET
   2. Parameters can be a DTO or anything else we want to use as filter.
   3. Access to the DB
   4. When using a DTO, the query can contain filters from the DTO.
   5. Return as GET
3. **PUT:**
   1. Parameters should be Id and the object we want return

**Authorization**

* + - 1. [Authorize] before each controller needed or before the Class name for all
      2. [AllowAnonymous] to allow access without login
      3. JSON web tokens: SEE MORE!

**ROUTING**

**Regular**

We can add more routes to the RouteConfig or add constraints to the existing ones. This helps when we have several controllers that take different parameters and there is a conflict when they are used.

e.g. config.Routes.MapHttpRoute(

name: “DefaultApi”,

routeTemplate: “api/{controller}/{id}”,

defaults: new {id = RouteParameter.Optional},

constraints: new { id = @”\d+”}

*This will route numerical ids only. If needed, we create another rule which accepts {name} or else after the controller.*

**Attribute routing**

* + - 1. Configure the api to accept **Attribute Routing:**

Startup > ConfigureWebApi > add:

config.MapHttpAttributeRoutes();

**Documentation**

**1.Built in**

1. Include XML Comments
2. Check *XML Documentation File* in the Build tab in Settings
3. Download NuGet *Microsoft.AspNet.WebApi.HelpPage.* The project now includes MVCs to display the documentation
4. Replace *GlobalConfiguration.Configuration* with *Startup.HttpConfiguration,* if using OWIN, so that everything uses the same configuration.
5. App\_Start > GlobalAsax.cs > add *AreaRegistration.RegisterAllAreas();*
6. HelpPageConfig > Register > set the file path to the correct address
7. DomainName/Help to see the documentation
8. **Swashbuckle and Swagger**