[REST 2](#_Toc6809)

[Prism 5](#_Toc25132)

[Unity 6](#_Toc21588)

[SQL 7](#_Toc1701)

[Patterns 13](#_Toc27398)

[Testing 14](#_Toc19160)

[Architectures 15](#_Toc18350)

[ASP.NET 17](#_Toc29371)

[WCF 19](#_Toc27032)

REST

Advantages of ASP.NET Web API REST

* HTTP Actions(Get, Post0 are mapped automatically to methods.
* Responses are converted to either Json/XML depending on whats asked for in request.
* Attribute routing/Route prefixing. Route, Prefix, Http attributes and custom attributes enable this.
* Route constraints. {id:min(10)}.
* Global error handling. All unhandled exceptions can be caught in the one location.

**PUT and POST** differences. **PUT** updates a resource with a known ID. **POST** generates a new resources ID.

With all 4 HTTP verbs you pass either a collection or single record.

**GET, PUT** and **DELETE** are idempotent. That is they generate the same result no matter how many times they are called. If already deleted the user should not be returned an error. **POST** should create a new instance every time its called.

Only use **HTTP Status Codes** when returning status to user

**RMM** (**Rest Maturity Model**)

**Level 0**: Akin to WCF calls . Single uri for each method

**Level 1**: Using POST For all calls

**Level 2**: Using the verbs properly

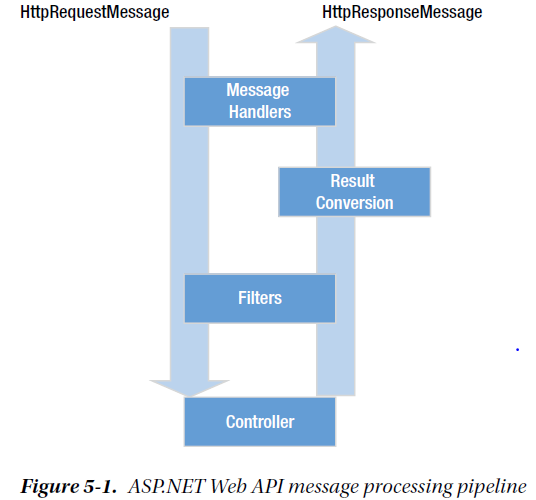
**Level 3**: Level 2 and returning a URI. Such as <link rel="self" href="/api/tasks/0987" method="GET" />

**HTTP Lifecycle in ASP.NET Web API**

1. When a **HttpRequestMessage** comes over the network and into IIS, IIS routes it to a worker process that’s hosting the ASP.NET runtime.
2. In this process, the ASP.NET Web API framework uses the routes configured in the WebApiConfig file to determine which controller should respond to the request.
3. When the correct controller is found the framework creates an instance of it and sends the **HttpRequestMessage** down the pipe to the controller.
4. The **HttpRequestMessage** goes through message handlers, filters and into the controller.
5. The controller then works on it to produce a **HttpResponseMessage**.
6. This **HttpResponseMessage** is then sent back up the pipe going through Filters, data conversion and message handlers.

IN our example we had Security check as a message handler

And a user audit logging for a filter



**Routing**

config.Routes.MapHttpRoute(...

routeTemplate: "api/{controller}/{id}",

...};

This means

1. The url must start with api/. (after http://IP:PORT/
2. After api/ must be a word that corresponds to a controller in the project
3. Then if there is a param after the controller this must correspond to a method param that is called id

**NOTE** If you remove the default route defined above, say to include a custom **IControllerSelect** class, ***you must then*** add the routes within the Controller class using the [**RouteAttribute**(“{id:int:max(10)}”, “GetTask”)] and also the [**RoutePrefixAttribute**(“api/”{Version}”/”{controller}”/tasks”)]

**Filters**

Filters execute pre and post behaviours around controller methods and inherit ActionFilterAttribute

**ActionFilterAttribute** has methods **OnActionExecuting**(pre) and **OnActionExecuted**(post) methods where you put your behaviour code.

HTTP objects used in ASP.NET REST.

**HttpRequestMessage:**

**HttpResponseMessage:**

**HttpContext: NHibernate** uses this to manage its **ISession** instances

REST Service interfaces

**IHttpControllerSelector** :Select a controller in code

**IHttpRouteConstraint** Checks if constraint is valid

**Using REST or SOAP**

**Use Rest when**  1. Limited bandwidth

1. Can deal with Stateless operations
2. If info can be cached

**Use Soap**  1. If you need reliability/security

1. Cant use HTTP

3 Stateful operations

**REST**: Using HTTP verbs to send data to and fro. Doesnt need to use HTTP...

Principles include **Uniform Interface** (you always access resources using URI’s), **Stateless** and **Cacheable**.

Within the rest call its best to have very little code. I like to have a data service instance passed in via construction parms, which is then called in the respective Controller method. This enables scalability..

You can addhandlers that will deal with each message coming up and the down the pipe such as for security and compression.

You then set up your routing

Controller

You have a controller that has methods. These methods will perform crud on resources. You access the controller by calling a url using http.

The controller is hosted as part of a service. The service needs be given a route on startup that will enable it to find the particular controller when its called. Ie

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

You will then call the ValuesController by <http://localhost:6666/api/Values/Method?id> = 123

In the controller I flag the method with its correct HTTP attribute.

You can also add **Route** attributes to the method that will override the default routes.

Ie

[**Route**(“AllBigValues”)]

[HttpGet]

Public IEnumerable<IValue> Biguns()

{...}

Would result in a call to <http://localhost:6666/AllBigValues/>

Adding a **RoutePrefix** attribute such as RoutePrefix[(“api/Prods”)] to the class definition would result in the following call

<http://localhost:6666/api/Prods/AllBigValues>

Passing in values for Post(Insert) , Put (update) and Partial you need to set the Http request header’s Content Type to something other than text. Ie ContentType: application/json; charset= utf8

Calling the Controller

The controller is called from the client side. You create an instance of **HttpClient**() and give it a base url and maybe a default header to accept json

Usual calls from client are **GetAsync, DeleteAsync, PutAsJsonASync, PostAsXMLAsync**

When you call you get a response which you then have to turn into the data type of your choice

Ie

public async Task<IEnumerable<Product>> GetAllProducts()

{

List<Product> products = new List<Product>();

var response = await this.GetServiceClient().GetAsync(

string.Format("{0}{1}", this.AbsoluteUrl, "Products/AllProducts"))

.ConfigureAwait(false);

response.EnsureSuccessStatusCode();

if (response.IsSuccessStatusCode)

{

var serviceResult = await response.Content.ReadAsAsync<IEnumerable<ProductDTO>>();

if (serviceResult != null)

{

products.AddRange( serviceResult.Select(

(dto) => ProductDataMapper.Instance.CreateModel(dto)));

}

}

return products;

.

.

.

Lifecycle of REST call

1. Call is send by client to service using ServiceClient
2. Service converts call into **HttpRequestMessage**
3. Message then gets sent down the Message handler pipeline which are classes that inherit **HttpMessageHandler**. They all have a **SendAsync** method which accepts a **HttpRequestMessage** and returns a **HttpResponseMessage.** They get added to the **GlobalConfiguration.Configuration.MessageHandlers** in **Global.asax**. The message leave the SendAsync method before it finishes and finishes SendAsync on the return.
   1. Handlers could be for security, compression
4. After handlers message goes thru any **ActionFilters** which have an **ActionExecuting** (and **ActionExecuted** for the way back). The filters get defined on the Controller class as attributes.
   1. Filters could be for transactions (begin, end)
5. The final handler is the **HttpControllerDispatcher** which calls Execute on the instance of the **APIController**. The result is converted into an **HttpResponseMessage** and goes back up the pipeline.

# 

# Prism

Partition your app into separate **Modules** ie Interface, Common, Model, Viewmodel View...

This is helpful if you have a team of devs, all can work on diff parts

Container can be Unity or MEF. A Container is software that you register your classes with and use to inject your classes into consumer classes.

Shell app is not a module but its the startup WPF project that wires up the application within its **Bootstrapper** class

You override a containers Bootstrapper class and from your app.xaml.cs you call **Bootstrapper.Run()**.

This sets up and configures the following:

**Logger, ModuleCatalog, Container, ServiceLocator, Region Mappings and Behaviours, FrameworkExceptions**

It also creates and initializes the shell.

You need to override **CreateShell** and **InitializeShell** with a main (parent) WPF window.

**Logger.** Unity provides 1 but you can create 1 by having a class that overrides **ILoggerFacade**

**ModuleCatalog** This is a list of modules used in the app. It can be defined in config, xaml, code or directory. With config you don’t need a reference in the project. (I’ve used the **IModuleCatalog** to see if a module has been started).

This **IModuleCatalog** is then used when configuring the modules in Bootstrapper (**ConfigureModuleCatalog()**). Nothing needs to be done if its a **ConfigurationModuleCatalog**, however a **DirectoryModuleCatalog** will need to be given the proper path.

Modules has a class that implements **IModule** (**initialise**()). The class is then defined in the module catalog

Within Module.Initialize you usually

* Register regions (defined in shell.xaml) with views
* Register types that will be injected into classes.

**IRegionManager** gives you control over the regions that you set up in the MainWindow.xaml.

It allows you to register/remove views to each region. And jump to regions.

<tabcontrol **prism:RegionManager.RegionName** = “MainRegion”/>

regionManager.RequestNavigate(“regionName”, uri )

regionManager.**RegisterViewWithRegion**("MainRegion", typeof(MainView));

And it also allows you to replace/remove views and activate views

regionManager.**Regions[**"MainRegion"**]**.Activate(null);

Communication

1. **EventAggregation** allows communication between modules. You subclass **CompositePresentationEvent**<T>.

Then in the 2 classes you publish and subscribe

**IEventAggregator** eventAggreg.Get<MyEvent>.**Publish**(new T);

**IEventAggregator** eventAgg.Get<MyEvent>().**Subscribe**(Action A, [What thread you want the event to come in on], [A predicate to filter the event);

1. **DelegateCommand** Prism implementation of RelayCommand. You have DelegateCommand which doesn’t accept a parm and DelegateCommand<T> which does.
2. **CompositeCommands** (multiple child commands which all get fired when main command fires)
3. **RegionContext:** Shares data between region host and views inside it.

Navigation

1. StateBasedNavigation. Views are updated by changing state in viewmodel
2. ViewBasedNavigation : View in a region is replaced by another view.

Container registers types as below so that it knows them when you are injecting them into class constructors

IRegionManager regionManager = this.Container.Resolve<IRegionManager>();

Container.**RegisterInstance**<ILoggerFacade>(Logger); Registers an existing instance. Good for **Singletons**

Container.**RegisterType**<IMenuBuilderService, XmlMenuBuilderService>(new ContainerControlledLifetimeManager()); Registers type which container will create when needed and will stay in scope for lifetime of container

Container.Resolve

RegisterTypeIfMissing(typeof(IModuleManager), typeof(ModuleManager), true);

IRegionManager regionManager = Container.Resolve<IRegionManager>();

Container.RegisterTypeForNavigation<AttendanceManagerMainView>();

ServiceLocater..CreateInstance. Service locator works like container. Used mainly in bootstrapper before container is created.

# SQL

Database normalization is the process of restructuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity

**SQL Data Types**

Exact numeric **(INT [4 bytes], NUMERIC, TINYINT [1 byte])**, character strings (**CHAR, VARCHAR)**,

Unicode character strings (**NCHAR, NVARCHAR**), approximate numeric (**FLOAT[8 bytes], REAL[4 bytes]**), binary strings (**BINARY, VARBINARY)**,

date and time (**DATE [3 bytes], TIME, DATETIME2[6 bytes], SMALLDATETIME[4 bytes], DATETIME[8 bytes],DATETIMEOFFSET)** etc.

Char(30) uses storage for 30 chars whether you use them or not.

Varchar(30) use storage for what you enter. Good for widely varying string sizes, and Read performance (as **less storage = faster reads)**

Char VS Nchar: C**har( and varchar) use 1 byte per char** and support english and 1 another language. **NChar(and NVChar) use 2 bytes per char** but deals with multi language. **Use N’abc’ to specify Nchars…**

**Choosing PKs**

**Identity**: Sequential : Less page fragmentation on single machines but bottlenecks on multi box systems

Issues with Identity:

Is not sequential if an insert fails

Created when record is created not before.

As you can enter identities manually doesn’t guarantee uniqueness

Doesn’t cycle if you reach limit of datatype

Use **@@Identity** or **IDENT\_CURRENT(<table>)** to get at identity just created

**Sequence**: Non sequential, More page fragmentation but better on multi box. **Note** fragmentation can be fixed by i**ndex rebuilds.**

Its treated as an individual object so it can be dropped.

*CREATE SEQUENCE MyTable.myID as int*

*MINVALUE 1*

*CYCLE*

*INCREMENT BY 15*

You then use it like *SELECT NEXT VALUE FOR MyTable.myID;*

Also *ALTER SEQUENCE MyTable.myID RESTART WITH 1; (*This command could come after a TRUNCATE command)

**GUID** : Always unique but 16 bytes. Any FK will have guids as well so more used space.

Where clause. **WHERE** number **NOT BETWEEN** 1 and 3..

Wildcards **%** - any number of chars; **\_** - 1 char; **[]** – chars within the brackets; **[^r, d]** – not r or d;

NULL – **WHERE** value **IS NULL–** returns all the unknown results. **WHERE** value **IS NOT NULL**

Use **CONTAINS** for a full text search – **WHERE CONTAINS(**DocumentSummary**,**'bicycle **AND NOT** reflectors'**);**

Table contains allocation units which contain 64kb Extents. Each extent contains 8 (8kb) pages

**Indexes –** each 1 requires an additional write when writing to the table

**Clustered Index** – 1 per table. Its sorted on

1st entry on each table page will go into a new page at index 1. This will then repeat(1st entry on each index 1 page will go into a new page at index 2). This repeats creating new levels until we are left with 1 page, the Root page. The root page is the starting point on a search

**NonClustered Index** > 0 per table

NCI create a sorted list of keys(based on the index) that point to the page

You need to create a list sorted on the column you want to create the NCI on. This is called the **leaf index.**  The rest is the same as CI

**Clustered index seek** : scans based on clustered index

**Clustered index scan:** must scan every row in the table

**Index seek** : scans based on (nonclustered) index

**Index scan**: must scan every row in the index

Explicitly converting data types

**CAST(value as datatype)**  ie select firstname + ‘ ID: ‘ + cast(id, nvarchar). Cast is ANSI compliant. It fails if Cast not successful. **Try\_Cast** returns a null.

**Convert**(datatype, value), allow you an optional style.

**PARSE**('1/2/2012' AS DATE USING 'en-US'). Allows an optional culture.

**Functions** – functions in where clause degrades performance

**ISNULL(value, replacementvalue)** i.e. select isnull(firstname, ‘Miko’) from names.

**String funcs**

**CONCATENATION -** SELECT empid, country **+** N',' **+** region. Any value thats null will result in complete string **NULL**. **Notice** the N’,’ means will change based on what char set is being used

**CONCAT**(country **+** N',' + region) substitutes nulls with empty strings.

**LEN and DATALENGTH** returns length of value and datalength ie LEN(‘hello’) = 5 .**DATALENGTH(‘HELLO’)** =10 as each char is 2 bytes.

**CHARINDEX(value, targetstring, [start posn])**  like C# IndexOf eg SELECT CHARINDEX(‘g’, ‘abcdefg’) = 6

**SUBSTRING(targetstring, start, length)** returns a portion of a string ie SELECT SUBSTRING(‘hello’ 1, 4) = hell

**UPPER, LOWER, LTRIM, RTRIM, FORMAT**

**Format:** You pass in a format string and optional culture.

**REVERSE**

**REPLACE(COLUMN, STRING TO REPLACE, NEW VALUE)**  ie replace (‘marcus’, ‘cus’, ‘k’) replaces marcus with mark

**REPLICATE**(‘Z’, 5) = ‘ZZZZZ’

**Date funcs**

**GETDATE() –** gets current date

**CURRENT\_TIMESTAMP** same as GetDate. Recommended.

To get current date or time use **CAST(SYSDATETIME()** **AS DATE**) or AS TIME.

**DATEADD(Datepart, value, date)** adds datepart to date **ie DATEADD(yy, 1, YEAR(GETDATE())**  adds 1 year to 2013

**DATEDIFF(Datepart, 1stDate, 2ndDate)** gets datepart difference in 2 dates **ie DATEDIFF(dd, getdate(), dateadd(mm, 1, getdate()) =** 31

**DATEPART(datepart, date)** returns a number ie **DATEPART**(mm, GetDate()) = 10 //October

**DATENAME(datepart, date)** returns a name ie **DATENAME**(mm, GetDate()) = October

**DAY(date), MONTH(date), YEAR(date)…**

**EOMONTH(**date**)** Last day of month

**DATEFROMPARTS**(YEAR(SYSDATETIME()), 12, 31) returns a date..

**MATH funcs**

**SQUARE(value), SQROOT…**

**ROUND(Number, length) -**  rounds out a number. I.e. **ROUND**(1234.6589**,-2**) = 1200.0000 ; **ROUND**(1234.6543,**2**) = 1234.6600

**RAND()** returns a float value between 0 and 1 ie SELECT CAST(RAND() \* 100 as INT) as ‘random #’

**(+-\*/%)** ie select (maxqty \* minqty) -10 as ‘wow’

**SYSTEM funcs**

**CASE :** similar to c# switch

SELECT Title,

CASE WHEN Title IN ('Ms.','Mrs.','Miss') THEN 'Female'

WHEN Title = 'Mr.' THEN 'Male'

ELSE 'Unknown' END AS Gender

**ROW\_NUMBER()** returns a row number of the results. **Row\_number() over (order by col1)**

**RANK()** similar to ROWNUMBER() except it gives same result to equal rows(based off ordering) **Rank() over (order by col1)**

**COALESCE** expression accepts list of expressions and returns first non null value or Null if all null.

COALESCE(Size, Color,'No color or size') AS 'Description'

**IF..THEN, WHILE..LOOP, FOR…LOOP**

**Joins**

**CROSS JOIN -**  every row from table 1 \* every row from table 2 (a Cartesian product). if 5 rows on each then 25 rows is result. Can be used in SP when you are access a lookup table.

**INNER JOIN –** returns only results that match the join

select c.CustomerID, c.StoreID, c.TerritoryID , p.FirstName, SOH.SalesOrderID

*from Sales.Customer as C*

*inner join Person.Person as P*

*on c.PersonID = p.BusinessEntityID*

*inner join Sales.SalesOrderHeader as SOH*

*on c.CustomerID = SOH.CustomerID;*

**OUTER JOIN-** retrieves all rows from 1 table along with the matching rows from other table. There’s no real reason to use RIGHT OUTER JOINS. The table which shows all results is to the left of the join statement. Table to the right shows only rows that match

If unsure of what table to have on left then choose table that has PK and FK

*SELECT c.CustomerID, s.SalesOrderID, s.OrderDate*

*FROM Sales.Customer AS c*

*LEFT OUTER ON Sales.SalesOrderHeader AS s ON c.CustomerID = s.CustomerID*

*WHERE c.CustomerID IN (11028,11029,1,2,3,4) ;* **HERE 1234 have no sales details but they will be included in results**

**Outer joins are good for finding rows with no match.** To return all customers with no sales

*select c.CustomerID, SOH.SalesOrderID, SOH.OrderDate*

*from Sales.Customer as C*

*LEFT OUTER JOIN Sales.SalesOrderHeader as SOH*

*on C.CustomerID = SOH.CustomerID*

***WHERE SOH.SalesOrderID is null;***

**FULL OUTER JOIN** - all the rows from each side of the join are returned

**SELF JOIN** Joins to itself eg *s***elect a.name, b.lastname from employee as a left join employee as b on a.empid = b.empid**

**SUBQUERY** can be used instead of a join

**SELECT c.name from customer c where c.customerid in (select p.customerid from product where name = ‘red product’)**

You can also use **not in**.

**Always** use **IS NOT NULL** in subquery otherwise no results will return

Need to use subquery when you want to find items that are not in other table

**UNION QUERY**  where you merge 2 result sets. Each individual query must contain the same number of columns and be of compatible types. Doesn't return duplicates

**UNION ALL**: Does return duplicates

*SELECT BusinessEntityID AS ID*

*FROM HumanResources.Employee*

*UNION*

*SELECT BusinessEntityID*

*FROM Person.Person*

**Derived Tables**: a query that appears in an outer queries FROM clause SELECT P.name der.city from Person P inner join (select id,city from Address) as der on P.id.3 = der.id

Basically it separates logic into another query

**GROUPING AND AGGREGATES**

**COUNT, MIN, MAX, SUM, AVG –** can’t be used in where clause need to use GROUP BY to use in select clause

**GROUP BY -**used when you have aggs and regular columns in the select. The results need to be grouped by the regular column

*SELECT* ***TerritoryID****, AVG(TotalDue) AS AveragePerTerritory*

*FROM Sales.SalesOrderHeader*

***GROUP BY TerritoryID****;*

You can also group by **expression (Not aggregates!!!)**

SELECT COUNT(\*) AS CountOfOrders, **YEAR(soh.OrderDate)** AS OrderYear

FROM Sales.SalesOrderHeader soh

GROUP BY **YEAR(soh.OrderDate);**

**HAVING** Can be an aggregate which isn’t in the SELECT clause. If not an agg then needs to be from GROUP BY clause but it should actually be in the WHERE clause.

SQL order of evaluation FROM->WHERE->GROUP BY->HAVING->SELECT->ORDERBY->LIMIT

**Correlated Subquery :** isolates an aggregate query ie have the agg query as a subquery

*SELECT CustomerID, SalesOrderID, TotalDue*

*FROM Sales.SalesOrderHeader AS soh*

*WHERE 10 =(SELECT COUNT(\*)*

*FROM Sales.SalesOrderDetail*

*WHERE SalesOrderID = soh.SalesOrderID);*

**CUBE and ROLLUP** give totals for the columns in the **GroupBy** . However Rollup wont total the last column in the groupby

Select t.type, avg(t.price) from titles t group by t.type with **cube**.

**GROUPING SETS** ???

**MANIPULATING DATA**

**INSERT 1 ROW:** *INSERT INTO table (col1, col2) values(1,2) or INSERT INTO table (col1, col2) select 1,2*

**INSERT >1 ROW : INSERT SELECT** *INSERT INTO table(col1, col2) select col1, col2 from table2*  **You could also use a union query**

**INSERT EXEC :** Inserts from a SP

**SELECT INTO**: Inserts rows into a new table defined by query rows

**INSERT MISSING ROW.** You need to link the table to insert into with another table that’s related and have a where clause of *where col is null*

**DELETE 1 row:** *DELETE table1 where col1 = ‘hello’;*

**DELETE table** *DELETE Table1*

**TRUNCATE table :** *TRUNCATE TABLE Table1*  this is faster as it drops and then recreates the table

**UPDATE TABLE** *UPDATE Table set col1 = ‘TEST’ where col2 = 666*

**MERGE TABLE :** You merge from a source to a target table.

You need to specify a predicate to match the source and target. If there’s a match you can only update/delete. If no match you can only insert.

**TRANSACTIONS**

**BEGIN TRAN**

**INSERT INTO…**

**COMMIT TRAN (or ROLLBACK TRAN)**

**@@TRANCOUNT.** Returns num of active trans. If 0 then none if >1 then nested trans are active.

SQL Server maintains transactional durability by using the database transaction log

**Locking**

**Shared locks** Used for sessions that read data—that is, for readers

**Exclusive locks** Used for changes to data—that is, writers

**User Defined Functions**

Can return either 1 (scalar) value or a table of values

1. (Scalar)

*CREATE FUNCTION dbo.udf\_Product(@num1 INT, @num2 INT) RETURNS INT AS*

*BEGIN*

*DECLARE @Product INT;*

*SET @Product = ISNULL(@num1,0) \* ISNULL(@num2,0);*

*RETURN @Product;*

*END;*

1. Table

***Stored Procedures***

Stored in cache.

*Some commands*

**DECLARE @VAR** VARCHAR(40) = ‘HELLO’

**SET** @VAR = ‘GOODBYE’

**PRINT LOWER(**@var) + ‘and so long’

**IF** @Count > 500 AND 1==1 **BEGIN**

**PRINT** 'The customer count is over 500.';

**ELSE BEGIN**

**PRINT ‘**More messages’

**END**;

**IF EXIST** (SELECT \* FROM Person where id=1) BEGIN

PRINT ‘ there is a record’

END;

Eg Create proc CRAPPROC (@id int **= 666**, **@result** char(15) **output**)

AS

**Return 0** <- this is a **Procedure return status** which is a usual way of returning status of the operation

***How to debug them…***

Add a parm at the of parm list @debug=0

Then in code if (@debug=1) print

Or you could print everything to a table

declare @log as table ( msg varchar(MAX) );

insert into @log values ('…

If in production you can use SQL Server Profiler which I know nothing about…

**New Functions learnt for HSBC**

ISNULL(p.color,'no color') replace null with..

**Concat**(p.col1, p.col2) still writes out answer if 1 is null.

**COALESCE** expression accepts list of expressions and returns first non null

CAST(soh.OrderDate AS Date) convert DT into date

DATEFROMPARTS(YEAR(SYSDATETIME()), 12, 31) returns a date.

select

**case when** (e.BusinessEntityID % 2 =0) then 'EVEN'

else 'ODD' end

from HumanResources.Employee e

select **top(3) [percent]** sod.OrderDate from… Add and order by for best results

OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY; skips first 50 and gets next 25 row.

Add Offset after the Order by which it needs.

Fetch needs an Offset.

IF you want to offset with out ordering using ORDER BY(SELECT NULL)

**How to optimize a slow running query in SQL Server**

**Analyze the query execution plan**  
First step in tuning the performance of a badly performing query is to take a look at the query execution plan. If an index is used to retrieve the result set, you will see **Index Seek**; if an index is not used, you will see **Table Scan** for a heap or a **Clustered Index Scan** in the case of an index-based table. Try adding appropriate indexes to the table being scanned, if you are not much familiar with the data model and the business needs, leverage Database Tuning adviser for appropriate index recommendations. However, if indexes already exist on the tables, try to find out why the indexes are not being used. If you see Clustered Index Scan, try to replace it with Index Seek by creating index on the most restrictive condition used in the query. Also, make sure that the indexes are not fragmented and the statistics on the tables are up to date. If the indexes are heavily fragmented or if statistics on the tables are not up to date, the optimizer will ignore the existing indexes and will likely generate a less efficient execution plan. You can find out index fragmentation using the sys.dm\_db\_index\_physical\_stats Dynamic Management Function(DMF) and to analyze statistics on indexes, you can use DBCC SHOW\_STATISTICS. If there are key lookup(also known as bookmark lookup) operations being performed, see if you can use a covering index to avoid key lookups. Key lookup requires access to both index pages and data pages and is a very expensive operation which might cause performance degradation, especially in the case of a large table.

**Analyze and tune the query**  
Once you are done analyzing the query execution plan, the next step step will be to optimize the query itself. Try to rewrite the query in another way if you think it will help improve the performance. Here are some of the general guidelines that I follow:

* 1. Operate on small result sets – Don’t use “SELECT \*”, instead limit the number of columns by including only those columns that are required. Also, try to use a highly restrictive condition in the WHERE clause to only include the required data. In short, retrieve only the rows and columns that are needed.
* 2. Avoid cursors – Avoid using cursors if you can, and try to use a **Temp table with identity column** to implement looping mechanism. I always create a temp table with an identity column and use a while loop to iterate over data sets.
* 2. Avoid using arithmetic operators or functions on WHERE clause column – Using an arithmetic operator or functions on a WHERE clause column prevents the optimizer from choosing the index on the column. Also, try to avoid using exclusion conditions(example !=, NOT EXISTS, NOT IN, OR) in the WHERE clause. Exclusion conditions can force the optimizer to  
  ignore indexes on the columns in the WHERE clause.
* 4. Fully qualify Database objects – Always fully qualify database objects with the owner. This will reduce the overhead of name resolution and might also avoid execution plan recompiles.
* 5. Avoid implicit datatype conversions in the query – Implicit conversions can prevent the indexes from being used by the optimizer and will also add overhead by costing extra CPU cycles for datatype conversions.
* 6. Don’t prefix stored procedure name with sp – Many Developers are used to prefixing stored procedure names with sp\_. If a stored procedure having an **sp\_** prefix is executed, SQL Server always looks in the master database first to find the stored procedure. Also, let’s say you have a stored procedure named sp\_Test, if Microsoft decides to use this name, all the references to this stored procedure will break, so never begin the name of a SP with sp\_.
* 7. Use SET NOCOUNT – SQL Server sends messages(count of the number of rows affected) to the client after each T-SQL statement is executed. If you are using stored procedure, there is no need pass this information and using this option will turn off the messages that are sent back to the client. Though this is not a huge thing, it is definitely something to consider.

Some Server functions

Sp\_help: gets details about an object

**Git Basic creation**

Create Repo up in Github. Take note of repo location for https i.e. <https://github.com/MarcusQuigley/MVVM_Basics.git>

Open up Git bash and go to path of repo parent i.e. C:\Programming\GitWorkspace

Type git clone <https://github.com/MarcusQuigley/MVVM_Basics.git> . This will create repo on local machine in ..\GitWorkspace\MVVM\_Basic folder

Create project within above folder

Commit initial project making sure to ignore .suo and everything within \bin and \obj folders.

Project is now commit to local repo

We now need to push it to remote repo (in <https://github.com/MarcusQuigley/MVVM_Basics.git>)

Type in git remote set-url origin <https://github.com/MarcusQuigley/MVVM_Basics.git>. This sets the remote repo for the current dir your in (in ..\GitWorkspace\MVVM\_Basic)

Type in git push origin master to push changes to remote repo

MVVM Northwind Project

Architecture

1 We linked Entity Framework (NW.Data) to UIDataProvider (NW.App) to NW.ViewModel. We used the EF data model as our data classes. EF contains its own change notification. This is not very scalable as NW.App is on the client

2 Then we added a Service layer between NW.Data and NW.App. We created data classes on the service and used these in the app instead of the EF data classes. This increases scalablity

3 We then added a model class to use as data classes replacing the dependency on the EF classes

**https://github.com/MarcusQuigley/Prism.Calculator.git**

# EntityFramework Core

SETUP CLASSES

add to data app

install-package microsoft.entityframeworkcore.<DATABASEPROVIDER> // such as sqlserver or sqllite or inmemory(for testing)

install-package microsoft.entityframeworkcore.tools

add to UI app

first Set as startup

run install-package microsoft.entityframeworkcore.design

CREATE DATABASE

Then run 'Add-Migration <MIGRATION-NAME>' in data project (In project with DbContext) -> this creates a schema about the database such as commands to create and delete the dtatabase. the create describes the tables, columns, constraints, indexes. Note the DB is not created yet

after this run 'update-database' making sure you're happy with the connection string as this is used to create the db. -> this is where the database is created off the migration information

Anytime you change the classes you need to rerun Add-Migration followed by update-database...'

add logging: in proj with DBContext run install-package microsoft.extensions.logging.console

CRUD on database

var samurai = new Samurai { Name = "Julie" };

var samuraiOther = new Samurai { Name = "Marcus" };

var battle = new Battle{name= "some battle"};

using(var addContext = new Context())

{

addContext.Add(samurai); //add one record

addContext.AddRange(samuraiOther, battle) //add differnet objects in one batch

addContext.SaveChanges()

}

using(var getContext = new Context())

{

var samurais = getContext.Samurais.ToList() // getting all records from Samurais table!

var lastSamuri = getContext.Samurai.OrderBy(n=>Name).LastOrDefault()

}

using(var updateContext = new Context())

{

var firstSamurai = updateContext.Samurais.FirstOrDefault();

firstSamurai.Name+='San';

updateContext.Samurais.Update(firstSamurai);

updateContext.SaveChanges();

}

//deleting

static void DeleteSamurais()

{

int samuraiId = 1;

using (var deleteContext = new SamuraiContext())

{

var samuraiToDelete = deleteContext.Samurais.Find(samuraiId);

if (samuraiToDelete != null)

{

deleteContext.Remove(samuraiToDelete);

deleteContext.SaveChanges();

}

}

}

# Patterns

**Service Locator**

A class that abstracts away logic for creating a class. **ViewModelLocator** is an example

**State machine** is any device that stores the status of something and can operate on input to change the status and/or cause an action or output to take place for any given change.

**Finite State machine**: Has a limited amount of inputs

**IOC** As opposed to you (your code) making the call you hand responsibility over to a framework which calls.. Its the Hollywood principle

Its main advantage is *To decouple the execution of a task from implementation*

Your constructor parameters are given to you by the IOC. Ie the framework decides what to inject into your class

In .Net it can be achieved by subscribing to events or the template method pattern.

**Dependency injection** means giving an object its instance variables (from outside).Through **Constructor**, **Setter** or **Interface Injection.**

DI is a pattern that utilizes IOC (think Template method).

Ioc deals with construction

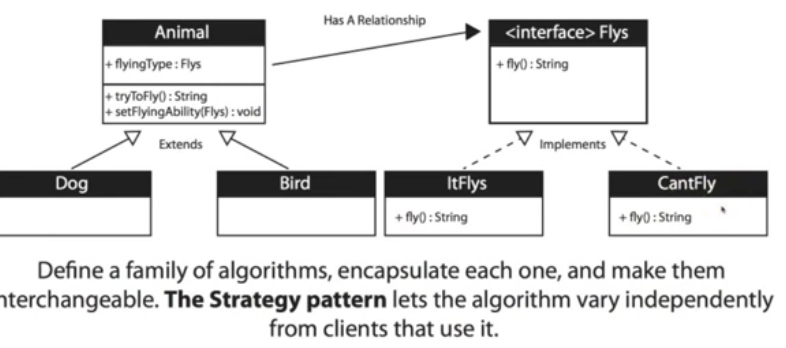
**Command:** allows you to decouple the requester of an action from the object that performs the action.

So decouples the wpf button (remote control) from the ICommand in the viewmodel(vendor class)

Decouples the waitress[takes order] from the cook[prepares meal]

**Strategy : Behavioral pattern**

Choose different implementation of an interface at runtime based on certain criteria



**Composite**: Group of items to be treated same way as an individual item.

**Façade**

Create a simplified interface that performs many other functions in the background

Action = Can I withdraw $ from bank

Subactions

Check if acct is valid

Check if funds are available

Withdraw and make changes

Example is the service layer(ie the controllers in a Rest app)

**Decorator**

Allows you to modify an object dynamically

Open for extension closed for modification.

More flexible than inheritance As its composition over inheritance

It does this by creating a new class which contains an instance of the existing class. It can then create new method implementations of the existing classes methods. Within these new methods it will define its own functionality along with calling the existing classes methods, thus decorating the existing class.

**Adapter**

Allows 2 incompatible interfaces to work together

You have a 3 pronged plug but only a two pronged socket. You use an adapter.

The client expects interface A. You have a class X. You create a class Y which implements A and contains an instance of X

**Observer**

1 to many relationship whereby the observer changes state all its dependents are notified.

Observer adds and removes dependents. Dependent is just an interface with an update method

**Template**

Subclass implementing abstract methods of base class.

**Factory**

Abstracting object creation to a different class by passing in a string

**Repository**

It queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source. A repository separates the business logic from the interactions with the underlying data source or Web service.

**Unit of Work pattern:** Groups one or more operations into a single unit of work. They either all pass or fail as one.

Data access Patterns

[T**ransaction Script**](https://app.pluralsight.com/player?course=architecting-applications-dotnet&author=cory-house&name=architecting-applications-dotnet-m3-bll&clip=3&mode=live) : procedural code that has class with both business logic and data access

**Table Module** : Each class models a table. Used by data sets and data tables

**Active Record:** Each class models a data row. You have a class for each table.

**Domain Model** : Classes represent the business object

**Singleton as Anti Pattern:** introduces global state. Hard to test. Overused as its an easy pattern to know.

# Testing

You should not test whole app. Just the Public interface. This presumes your using Single responsibility principle

**Stub**: A stub replaces an object so another object can be tested. The stub will not be tested. Dependency injection is used to add stubs (interface!!) **Uses state based testing** (Used for returning result??? Use Assert at the end??)



**Mock** : a fake object that decides whether test passed or failed. Usually only 1 per test. **Uses Interaction testing.**

The basic difference is that stubs can’t fail tests, and mocks can. Use Verify (for the most part) to see if mock passes



**State Based Testing:** Determines whether the tested method works correctly by examining state of the system and its methods after they’ve been used. Can be known of as result-driven testing

**Interaction testing**: How an object sends/receives input to other object. How the object interacts with other objects. Action driven testing

**Unit Testing** : Individual pieces of software

**Integration testing**: Tests that end to end works. No Mocks/stubs

**Subcutaneous Testing**: Test under the UI.

**UI Testing**: cover greater breadth of system

**Regression** : Tests updates to already tested code doesn’t introduce new bugs.

**Acceptence**: StakeHolder/End user testing and signoff.

Test Brittleness: How easy the test breaks when system changes. UI have higher brittleness, Unit tests less brittle.

**State verification** is a style of unit testing where you perform some action on the component under test, and then verify that the state of the application is as expected. In our first attempt we were verifying the state of our MainWindowViewModel.Customers property.

**Behavior verification** is a style of unit testing where you perform some action on the component under test, and use a mock object to verify that the component under test behaved as expected.

**XUnit**

Attributes around test methods:

1. **[Fact]** Tests a parameter-less method

2. [**Theory]**

**[InlineData(1,2)** Tests a method with 2 parameters

3. [**Theory]**

**[ClassData(typeof(someclass)** Tests a method with a data class that implements Ienumerable<Object[]>

4.[**Theory]**

**[MemberData** Tests a method with public static property/method that returns Ienumerable<Object[]>

Assert properties

# Architectures

**SOA**

Stateless,

Usually simple operations that are abstracted from user

Needs to be described and discoverable

# Architectural Patterns

**Client Server** : A server and multiple clients with nothing in between.

**Multi Tier Pattern** : The usual

**CQRS**: Command query pattern. Write commands separate from Query commands. Can be separate dbs as well. Then they need to be synced

**Event Source**: You don’t store state of object but the events that happened to it. Like Acounts ledger. Good for auditing

**Event Bus**: You have channels on a bus which sources send messages to. Subscribers receive these messages.

**Microservices:** Splitting services up into indivual pieces that get deployed separately. Great for scaling. Difficult to communicate betw services.

**Domain-Driven Design**

The modeling of the business domain as opposed to a **Table Module** pattern which is modeled on the databases tables.

Its better for more complex projects.. ORMS work better with Table Module as the orm classes are modeled on the tables..

**POCO** . Plain CLR object. Different from a DTO object in that it can have behaviour. The plain comes form the fact it doesnt have a dependency on the data layer.

**ESB** : An Architectural model that allows different systems to talk to each other while it takes care of the data translation (if any) and correct routing

**Networking**

. TCP is a core operates on the Transport level HTTP is an Application Layer protocol that is a request/response standard found in client server computing.

TCP provides communication services at an intermediate level between an application program and the IP; in HTTP there are a series of sessions in which the client sends a request and the server sends a reply message back to the client including the request, an error message, or another piece of information.

# ASP.NET

**ASP.NET -** aspnet\_wp.exe runs the asp process. 1 per webserver. Multiple per web farm

Controls have ‘runat=server’. Auto generates HTML on client side to display control and also has events that interact with server. And is OO

**Get** = sends data in url. **POST** sends data hidden

**AutoPostBack** if true then when control used will call back to server

Control inherits from **WebControl**

Each asp page inherits from **Page** object which inherits from control.

Properties of Page class

**Cache** : websites cache

**IsPostBack** : true if first time on page, false otherwise as it’ll be served

**Application** ; interact with data across entire site

**Server**: interact with server

**Session** : unique session for each user. Has a timeout property

**Request**  interact with http request

Members : Cookies : browsers cookies

Browser : info about the browser

Form : http form vars

Headers: http headers

HttpMethod : get or post

QueryString

RawURL

**Response** interact with HTTP response

Members: Output

Write() write to http stream

Redirect() redirect to new url

**Certain Page Lifecycle**

**Init :** set controls back to pre value based on state data

**Load :** Controls are ready and its now safe to interact with them

**Event that triggered postback**

**Prerender** Controls ready to put data into Response

**Unload –** You can cleanup here

**Error** occurs if an error is thrown

**Global.asax**. Allows you to handle events

**Application start/end**

**Application error** – this is the last place to catch an error

**Session start/end**

**ASP.NET State Management**

**Use ASP.NET view state.** Saved on page. Sent back and forth betw page and server. Stored in hidden field

**Use ASP.NET control state.**

**Define application-level data.**

**Use the cache object.** Is application wide. Can be inserted for a set time

**Define session-level data.** Saved on Server. User gets a new session ID when they hit a site

**Define cookie data.**  Stored locally. Used for user prefs. Can be perm or temp (will die when page is killed

ASP.Net Lifecycle

When a request comes in, IIS checks for the script map and routes the request to the aspnet\_isapi.dll.

From here the requests are routed into ASP.NETs processing pipeline. The ISAPI extensions and .NET also run inside the pool and communication in-process, thus highly efficient.

When a request hits, it is routed to the ISAPIRuntime.ProcessRequest() method. This method in turn calls HttpRuntime.ProcessRequest that does several important things (look at System.Web.HttpRuntime.ProcessRequestInternal with Reflector):

When we get to the ISAPIRuntime then we Create a new HttpContext instance for the request

Retrieves an HttpApplication Instance

Calls HttpApplication.Init() to set up Pipeline Events

Init() fires HttpApplication.ResumeProcessing() which starts the ASP.NET pipeline processing

HttpApplication is the outer container for your specific Web application and it maps to the class that is defined in Global.asax.

This all takes place in an Application pool for each virtual directory (client). When a new client sends its first request a new pool (which is a separate exe) is created.

This is very configurable and isolated, thus secure.

**HTTPModules :**Are event handlers that hook specific HttpApplication events

**HttpHandlers**: end point that gets called to handle ‘application level request processing’.

Both Modules and Handlers are loaded and attached to the call chain as part of the HttpApplication.Init() method call

# 

# ASP.NET CORE

**Directives**

**@page :** States that this page is a Razor page. Can be proceeded by variable that will be passed to the page (if its not optional “{id?:int}”

**@model**: Defines the model class for this page

**@{}:** Just a C# code block

**Tag Helpers**

**asp-for**: Extracts the name of the specified model property into the rendered HTML . <label **asp-for**="Restaurant.Cuisine"/> Works with <label> and <input> tags

**asp-items=”MODEL.items”**:Used within Select label to display collection

**asp-validation-for=”MODEL.Variable”**:looks in ModelState for any validation errors for MODEL.Variable

**asp-page:** used within <a> tag to set href to a page

**asp-route-XXX:** Used also within an <a> tagsets the XXX parameter to be passed along with the page

**ViewData**: Dynamic dictionary that ASP.net provides . Every page shares this object

**TempData** : Like ViewData except only holds data between 2 successive requests, after which their destroyed.

**ModelState**: used to record when binding occurs

Model Classes:

**OnGet and OnPost** methods: get fired when page is accessed by Get or post. Use IActionResult to return from these methods

[**BindingProperty**] attribute. Used for a property to signify that its to be bound to the page.

**[TempData]** attribute: Property is bound to value in **TempData** dictionary with same name as property.

WCF

Overview

DCOM and Remoting use a direct reference for local and a proxy for remote object. This caused issues such as reliability and security.

Address contains location and transport protocol.

**Contracts**: Service, Data, Fault and Message contracts. Avoid non default constructors as WCF won’t use them

**Hosting**: IIS5/6 (only http) Selfhost, WAS(offers app pooling, recycling, id mgmt.,isolation. Can also host websites), AppFabric (Specific for w services)

Every host can have multiple services and every service can be on multiple hosts.

Opening host launches WCF runtime and worker threads from the I/O completion tool.

Host closes gracefully. Usually waits 10 secs after Close() is called. This can be changed in config under Service>Host>timeout.

Host can be opened/closed Asynchronously if it’s a lengthy op.

Use Host Factory to interact with host in IIS and WAS

**Bindings**

**BasicHttp:** connectionless, no state. Receives request and sends response. Used for legacy non WCF systems

**TCP:**  Fast, reliable. Used when both ends use WCF

**NetPipes:** Used when both ends are WCF and are on same machine

**WS binding:** Over HTTP but uses WS\* standards(reliable, security, transaction) Used when other end supports WS\*. Can send an ID in message to identifying the client.

**MSMQ:** Support for disconnected queued calls

**WSDualBinding:** Same as WS but also supports duplex calls. Depreciated due to AppFabric

**Encoding:**  What you convert the message to before sending

**Text:** Supports both text and xml. Used by Basic and WS http. Can be Base64d. Slowest

**Binary:** Fastest. Only used by WCF to WCF so not interoperable.

**MTOM:** Used to transfer large binary files as is. Also can transfer textual form.

**Endpoints:** ABC. Every Service has >0 endpoint. Every endpoint has 1 ABC. Each address must be unique..

Base address must adhere to the binding. Eps can be created programmatically by Servicehost.AddServiceendpoint(C, B, A)

Conceptually C# has endpoints. A = memory address, B is the CLR. C is the interface

**Metadata**: This is the wsdl that contains the details of the service. Two ways to get it

1. Over http. Need to have a http address and set ‘serviceMetadata http(s)GetEnabled="true"’. This can only be accessed by WCF platforms. Ie SVCUtil
2. Using a MEX endpoint. Industry standard. Note need to set ‘serviceMetadata’ only in Behavior and can’t access mex address in browser

Proxy is generated using VS service references or SVCUtil

**Architecture**

**Call Process**

Client proxy serializes the call stack frame (i.e. the method) to a message and sends message down a series of channels.

Each channel can intercept and work on the message. Channels depend on the bindings (e.g. message can be encoded, manage reliable session, encrypt message, propagate a transaction).

Last client channel is the transport channel which sends the message to the host.

Host sends message back up through channels (decode message, unencrypt, propagate transaction, set security principal, activate service host)

Last host channel passes message to dispatcher which converts message to a stack frame (a method) and calls the service host instance.

The host the fires the method and returns control to dispatcher which then converts the values(if any) into a message and sends it back down the wire with the process reversed (ie the service channels encodes the message and the client channels decodes..etc.

**Host Architecture**

The host(iis, was, self) can contain many AppDomains. Each Appdomain has >-1 service instances. Each Service instance has >-1 **contexts.** The context is where the call gets processed

**Transport Session** Is a link between the client and service. Offered by TCP and pipes as all comms happens on the same link. HTTP doesn’t offer it as its stateless. WSHTTP can emulate it by passing a unique ID in the message. The session times out after 10 mins of inactivity.

**Reliability** – Not supported by Basichttp or msmq. Pipes binding is deemed reliable as there is never an intermediate hop.

Set within binding config **<reliableSession enabled = "true"/>**

**Transport Reliability** guarantees in order delivery of packets. TCP only

**Message Reliability**: like Transport except it doesn’t guarantee delivery of message. If not delivered it guarantees that the sender will know if a failure occurs . WSHttp only

**Service Contracts** [**ServiceContract**] for service [**OperationContract**] for method

Should not have too few or too many operations for each service. 6-10 is ideal. Anything over 20 is no good.

Operations can be overloaded but their Name properties must be different.

Services can inherit from each other but [ServiceContract] must be added to child interfaces. The client proxy will have no inheritance and will be flat but each Operations Action attribute will contain the ‘inheritance path’

**Data Contracts**

WCF can send primitive types across. Custom objects need to be flagged with [**DataContract**]. Their members(fields, properties) need to be flagged with [**DataMember**] This attribute is the serialize flag. The members need not be public. If property it needs to have getter and setters.

Data Types can be inferred if they are public. Not good practice

Marshall by Value is how you send data across the wire. Done by Serialization.

**Serialization** Done using reflection

Object gets Serialized and deserialized. This is the real performance bottleneck of WCF.

**DataContractSerializer** is what WCF uses to serialize the object. Any field of data

Data contract can override serialize events [**Onserializing**, **OnSerialized**, **OnDeserializing**, **OnDeserialized**]

No constructor is called during deserializing so use **OnDeserialized**. However don’t set values to Datamember fields as these will be overwritten!

[**KnownTypes**] is an attribute to flag subclass for the datacontract. [**ServiceKnownTypes**] can be applied on a method to allow the known type for that method only. It can also be applied at the **ServiceContract** level

If you define the DataContract as an interface then the ServiceKnownType must be used on the ServiceContract. You can’t use KnownType on a class as the interface will not be sent across

Data Contracts are equivalent if they have the same schema ie same names

**Ordering** – Good for keeping DateContract equivalency

Base members are ordered first

Then members that don’t have OrderAttribute set

Then members that do have OrderAttribute set. If 2 values have same OrderAttribute then their sorted alphabetically

**Versioning –** When client and server have different versions of the contract

3 types : New members: these will be ignored

Missing members : these will be given default (0 or null) values

Round tripping (When a new data contract is passed to and from a client to a server with an older version.

Data Member has an **IsRequired** attribute that will cause one side to throw an error.

Data contract class can implement **IExtensibleDataObject** which stores extra data in a linked list

Enums will be serialized and don’t need DataContract attribute

Collections

Generics are not to be used in WCF as their .NET specific

Collections are passed around as arrays. Ie IEnumerable<Contact> becomes Contact[].

Concrete collections are also passed as arrays only if they implement an Add(object obj) method

**CollectionDataContract** Attribute is best option. Use instead of DataContract. implement an Add(object obj) method. The result is a a List of objects.

Note IF you use **CollectionDataContract** with a Dictionary you will get a Dictionary on the client

**Behaviours**

They are local attributes(i.e. clients don’t know of service behaviours) that don’t affect communication.

2 types **ServiceBehaviour** which affects all services and operations and **OperationBehaviour** which only affect that operation

**Instance Management**

**3 types**  (they are Service behaviors and are called **InstanceContextMode**)

**PerCall** Every client request gets a new service instance. This is very scalable as there’s no continuous connection. No need to enforce consistency with the instance state. You will need to pass id to retrieve state. This affects performance. State should be global to be able to use load balancers

**PerSession :** Default mode.Service instance is maintained until client kills it. Suffers from scalability. Need to have a **reliable transport** session to work (i.e. wont work with HTTP and only works with WSHttp if reliable messageing or Message security is set). Client is informed by a ServiceContract attribute **SessionMode**

**Allowed** : default, session will default to percall if no transport session set

**Required** Error thrown if no transport session . This is the best choice

**NotAllowed** Does not use transport session. Cant work with TCP or pipes. Error will be thrown. No reason to use this. Set PerCall instead.

Session ID is used in the PerCall (can be used in other instance types but will be useless). Its got from the **OperationContext.Current.** Is accessed from client at **proxy.InnerChannel.SessionId**. Need to open proxy before getting ID

**Singleton** Dies only when the host dies. Needs to be thread synchronized. Only 1 client can access it at a time. Not scalable. Be wary of using it. Share state instead.

**Demarcating Operations :** Operation attributes **IsInitiating, IsTerminating** defaults to true and false

**Release Instance:**  Instances can be killed after operations using **ReleaseInstanceMode** attribute. Can also happen at runtime using **OperationContext.Current.ReleaseServiceInstance**

3 options **BeforeCall**

**AfterCall**

**BeforeAndAfterCall**

Should be avoided in general as it adds complexity

**Durable Services** Way to do this is to store data and therefore avoid keeping service state in memory. A serializabe and equitable(int, string, guid) id would be needed to access the data.

Couple of ways to do this:

1. Pass ID as parm in every method
2. Pass ID in message header
3. Use DurableService attribute. Set on ServiceContract which must also be serializable. All properties will then be saved

**Throttling** Limits the # of active connections. Accessed from config or through ServiceHost.

3 Types: **MaxConcurrentCalls**

**MaxConcurrentSessions**

**MaxConcurrentInstances**

**Operations –**

**Request-Reply** The usual way.

**One-Way** Attribute(**IsOneWay**) on OperationContract. Can still receive exceptions

**CallBack.** Create an interface with an OperationContract method

Then create a ServiceContract with its CallBackContract set to the interface

Need to implement the interface on the client

Call the callback from the service using the **OperationContext**.Current

Callbacks only work on TCP and pipes bindings

**Faults**

If error occurs on service its caught and sent across to client so as not to bring down service. Really a client shouldn’t worry about the error.

All errors that the client gets are **FaultExceptions** which are non WCF specific Soap exceptions.

Create a fault as a datacontract. In Operation add as such [ OperationContract]

[FaultContract(typeof(MyFault))]

double Add(double number1,double number2);

**Concurrency Management** – clients accessing services concurrently. Ignored for PerCall

**ConcurrencyMode** is a service behavior

3 modes

**Single** Only 1 client at a time can enter service. All others queue

**Multiple**  Any number of clients can enter(up to throttling limit). No Q’ing. Needs to be synchronized with lock/monitor

**Re-entry** Like singleton but a service can reenter a service its already on (ie A calls B which calls C which calls A – otherwise deadlock)

You shouldn’t share resources between services

**Synchronization Context** Call executes on the correct thread

Async calls (from client) need to be defined when generating proxy.

**Transactions**

**A** Atomic – least divisable

**C** Consistent – same result all the time

**I** Isolation - Nothing else can see transactions intermediate state

**D** Durability won’t unchanged

**Architecture notes**

**Overposting Attack** : Sending unneeded info down wire that can be manipulated .ie sending complete Person object which includes Salary prop. This can be changed even though Salary is not needed for call. Only send exactly what needs to be changed.