# 1. Key Components of Distributed Applications

- consume data from distributed data sources

Designed for:

* Scalability
* Low latency
* Availability
* Reliability
* Security and privacy

## Logical Layers of Distributed Applications

* Data Layer
* Business Layer
* User Interface Layer
* Service Layer

### Data can be represented in different models:

* Relational (databases, tables and columns)
* Hierarchical (XML, JSON)
* Object oriented (entities in code)

### There are many types of data store:

* Relational database
* File-systems and distributed file-systems
* Distributed caches
* No SQL databases
* Cloud storage
* In-memory stores

### .Net Data Technologies

* System.IO
* ADO.NET
* Entity Framework
* In-Memory cache (system.web)
* Windows AppFabric cache, Windows Azure caching

Http can also be used for accessing data

* OData services with WCF Data Services or ASP.NET WebAPI
* Windows Azure Storage

Declarative data queries in C# using LINQ

### SOAP and Http-based services

SOAP based

* Based on SOAP, an XML based format
* Use a Remote Procedure Call (RPC) approach
* Interoperable over HTTP, UDP, SMTP and AMQP

Http-based

* Support multiple content types (XML, Text, Images)]
* Uses a resource based approach
* Http is underlying protocol of the world wide web

### Cloud computing

* Infrastructure as a Service (IaaS) – virtual machines
* Platform as a Service (PaaS) ready to use platform that provide application hosting that can be cloned and scaled automatically
* Software as a Service (SaaS) – ready to use on demand software

### Windows Azure Cloud Services

* PaaS sollutions for your application:
  + Stateless
  + Provision ready
  + Scalable
* Cloud services host applicationson role:
  + Web role – IIS based applications
  + Worker role – host for any type of process, services, background processing etc

### Windows Azure Application Components

* Storage Services
* Storage Bus
* Access Control Service
* Distributed Cache
* Content Delivery Network
* SQL Databases as a Service

### Windows Azure Storage

* **Blob** **storage** – non-structured collection of objects accessed by a resource identifier
* **Table storage** – semi-structured collection of objects that can have fields but not relations between objects
* **Queue Storage** – provides a persistent messaging queue

# 2. Querying and manipulating data using Entity Framework

## ADO.Net Basic Objects

Ado.Net is the basic data access API in the .NET Framework. Contains data providers:

* System.Data.SqlClient
* System.Data.OleDb
* System.Data.Odbc
* System.Data.OracleClient

Each provider has an API that implement a set of common interfaces:

* Connection – IDbConnection
* Command – IdbCommand
* DataReader – IdataReader
* DataAdapter – IdataAdapter
* DataSet

Can access async via ExecuteXXAsync Methods and DbConnection.OpenAsync.

## Creating an Entity Data Model

Entity Framework is an ORM that maps application objects to database records. Development Approaches:

* Model-first and database-First
* Code-First

### Creating a DB Context.

DbContext is a wrapper around ObjectContext. Create a class that devives from DbContext, property of type DbSet<T> for each entity type that is mapped to your database schema.

### Creating the database if it does not exist.

When the DbContext is initialised, it detects if the target database already exists. If not you can create it using the CreateDatabaseIfNotExists<T> generic class.

### Updating the database with code first migrations

If the database was created by DbContext and you change something in domain model classes, EF will not update the database automatically.

### Data Annotations

Map an object in code to a table and columns in the database.

### Mapping Type Inheritance to Tables

* TPT – Table per Type
* TPF – Table per hierarchy
* TPC – Table per concrete type

### Table per Type

E.g.

Person

Teacher Student

### Table per Hierarchy

Person

(has a discriminator field)

### Table per Concrete Type

Teachers Students

### Mapping Classes to Tables using the Fluent API

Can override the OnModelCreating method or use a class derived from EntityTypeConfiguration<T>

## Querying Data

### LINQ to Entities

E.g.

var studentsQuery = from s in context.Students

where s.Name.ToLower().Contains(“a”)

select s;

As with LINQ to Objects, queries are not executed until they are enumerated.

### Entity SQL

var eSql = “SELECT VALUE prod FROM STORECONTEXT.Product AS prod ORDER BY prod.productName”;

var query = objectContext.CreateQuery<Product>(eSql);

List<Product> products = query.ToList();

### Retrieving Objects by Using Direct SQL

string sql = “select \* from Products where Price > 5000”;

var products = context.ExecuteStoreQuery<Product?(sql);

Executing SQL that returns a Scalar Value

Context.Database.ExecuteSqlComment(“update …etc”);

## Load Entities by Using Lazy and Eager Loading

Lazy loading and eager loading refer to the number of round trips EF makes to load data from the database.

When using lazy loading on the top level of the data is returned and nested levels are retrieved on demanded, e.g. Students.Courses – the courses will not be fetched.

When using egar leading EF returns the entire dataset.

When issuing a query , call the **Include** method to specify which entities should be eagerly loaded.

## Manipulating Data

Change tracking with EF

EF can track domain objects that you retrieve from the database, and we you call SaveChanges on the DbContext object it can update the database with changes. The state of the entity is recorded:

* Added
* Modified
* Unchanged
* Detached
* Deleted

# 3. Creating and consuming ASP.NET Web API Services

## Introduction to Http

Http is a fist class application protocol that was built to power the World Wide Web, taking into consideration caching and stateless architecture.

## Http Messages

Http is a simple request-response protocol, all Http Messages contain:

* Start-line
* Headers
* An empty line
* Body (optional)

### Request messages

Sent by the client to the server.

Request-line contains:

* HttpMethod (GET, POST etc)
* Request URI – URI to which the message is being sent
* Http version

#### Headers

Headers exist in both request and response messages, some readers are ised exclusively by one of them. E.g. Accept header used state the type of response the client would prefer to receive (content negotiation).

#### Body

Request message has no body

### Response messages

#### Status-Line

* Http version
* Status-code
* Reason-phrase – describes the status code

#### Headers

Similar to request, response also has headers. Some are server specific e.g. Cache-Control and Pragma.

#### Body

Returns a representation of a resource e.g. JSON

## Http Verbs

* GET
* HEAD – same result as GET but without returning a message body (Check validity, retrieve headers)
* OPTIONS – returns information about communication options of server
* POST – send an entity to a server
* PUT – store/update an entity
* DELETE
* TRACE – indicate to clients what is received at the server end
* COMMENT – used to start SSL tunnelling

## Introduction to REST

Representational State Transfer.

The Richardson Maturity Model:

Level 0 – Use http as a transport protocol by ignoring the capabilities of HTTP as an application layer protocol. Single address used as an endpoint e.g. SOAP and RPC based services

Level 1 – Identify resources by using URIs

Level 2 – Uses the different http verbs to allow the user to manipulate resources

Level 3 – Introduces Hypermedia for resources to describe their own state in additional to relation to other resources.

## Introduction to ASP.NET Web API

When WCF came in .Net 3.0 it was SOAP only. As Http services increased, .Net 3.5 added support for Http using the **WebHttpBinding**.

In 2009 MS released the WCF REST starter kit. This added **WebServiceHost** for hosting HTTP-based service.

Web API was then developed to provide a comprehensive solution:

* Better support for content negotiation and media types
* APIs to control every aspect of the HTTP messages
* Testability
* Integration with other frameworks

### Routing

ASP.Net WebAPI routes are defined using the MapHttpRoute extension method e.g.

MapHttpRoute(

name: “DefaultApi”,

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

defaults: new { ud = RouteParameter.Optional }

);

Unlike MVC there is no mapping for actions – this is because methods are mapped based on their prefix to Http Verbs.

### Responsibilities for ApiController

* **Action Selection** – the ApiController class is responsible for calling the Action Selector that executes the action method.
* **Applying filters**

### The HttpRequestMessageClass

ASP.Net Web API uses the HttpRequestMessage class to represent incoming HTTP message requests

### The HttpResponseMessgeClass

In order to control the HTTP response, you must create an action with HttpResponseMessage

# 4. Module 4 – Extending and Securing ASP.Net Web API Services

## Architecture Overview

The ASP.Net Web API processing architecture is made of three layers:

* Hosting
* Message handlers
* Controllers

### Hosting

Hosting layer communicates with infrastructure and creates the **HttpRequestMessage** and covering the **HttpResponseMessage**.

* Web-hosting in IIS
* Self-hosting using the WCF channel stack

### Message Handlers

Are chained to each other to form a pipeline. Each receives a HttpRequestMesssage object and performs some processing before passing to next handler in the pipeline.

After the hosting later creates the **HttpRequestMessage** it creates a new instance of **HttpServer –** a message handler. When this is intalized it creates a handle of message handlers, in order:

* **Custom Message Handlers**
* **HttpRoutingDispatcher** – finds the route that matches the **HttpRequestMessage**
* **HttpControllerDispatcher** – selects and creates the controller, then calls **ExecuteAsync** on the controller

### Controllers

When **ExecuteAsync** method is called it should result in processing of a request and returning a response.

* **Action Selection**
* Creating the filters pipeline
  + Authorization filters
  + Action filters
  + Exception filters

The filters pipeline contains two other components:

* **HttpActionBinding** – performs parameter binding and is executed after the authorization filters
* **ApiControllerActionInvoker**

### The DelegatingHandler Class

The main method for message handlers is the **SendAsync** method, which receives a **HttoRequestMessage**  and returns a **Task<HttpResponseMessage**.

Web API also providers the **DelegatingHanlder** class a base class for message handlers that include a property called **InnerHandler** and an implementation of the **SendAsync** that invokes the inner handler to simplify creating message handlers for a pipeline.

## Async Actions

E.g.

P

public async Task<string> Get()

{

var client = new HttpClient();

var response = await client.GetAync(“<http://someurl/>”);

return await response.Content.ReadAsStringAsync();

}

## Media Type Formatters

Web API supports content negotiating using media type formatters, derived from **MediaTypeFormatter** base class. Each has a property called **SupportedMediaTypes**.

public class CsvFormatter : MediaTypeFormatter

{

public CsvFormatter()

{

this.SupportedMediaTypes.Add(new MEdiaTypeHeaderValue(“text/csv”));

}

}

You can implement the process of reading or writing the data using **ReadFromSTreamAsync**  and **WriteToStreamAsync.**

## Creating OData Services

The OData Protocol is an HTTP-based data access protocol created by Microsoft. OData is designed for querying and updating data by using web technology such as HTTP and AtomPub. It is a RESTful implementation based on feeds.

### OData Query String Options

OData uses the querying string to perform query operations, e.g.

* **$oderby**
* **$top**
* **$skip**
* **$filter**

### Defining OData Actions

Web API supports OData Query String options using queryable actions that return IQueryable<T> and have the [Queryable] attribute.

### OData Controllers

To deal with OData formatting Web API introduces the **ODataController** base class.

Can also derive from **EntitySetController<TEntity, TKey**> which providers virtual methods you can override.

### Entity Data Model (EDM)

OData exposes the structure of its data model using a service metadata document, which is XML based. Web API needs an instance of an entity data model class implementing the **IEdmModel** interface. You can use the **ODataConventionModelBuilderclass**:

ODataConventionModelBuilder modelBuilder = new ODataConventionModelBuilder();

modelBuilder.EntitySet<Flights>(“Flights”);

IEdmModel = modelBuilder.GetEdmModel();

### OData Routes

After you have OData controllers and a EDM you need to provide a route using the MapODataRoute extension method:

Config.Routes.MapODataRoute(routeName: “OData”, routePrefix: “api/odata”, model: model);

### Consuming OData Services

Can add a reference and use local classes and linq for querying. You can use the Container class to consume the service and it exposes properties representing the different feeds:

var container = new OData.Container(new Uri(“<http://server/odata/>”));

var course = (from c in container.Courses

where c.Name == “WCF”

select c).FirstOrDefault();

# 5. Creating WCF Services

## Advantages of Creating Services with WCF

SOAP is a protocol spec for exchange of information between peers in a decentralized distributed environment. SOAP uses XML for its message formatting and usually HTTP for transmission.

### Benefits of SOAP based services

SOAP, a standard Remote Procedure Call (RPC) is maintained by the W3C. Soap is:

* Lightweight protocol (REST lighterweight)
* Communication between apps
* Designed for HTTP
* Any programming language
* XML based
* Simple and extensible
* **Versatility**
* **Security –** firewall/proxy friendly

### WCF Features that are not supported by ASP.Net Web API

* One way messaging
* Muticast messaging with UDP
* In-order delivery of messages
* Duplex services (pub-sub)
* Using message queues e.g. MSMW
* Runtime discovery of services
* Content-based message routing
* Distributed transactions support

## Creating and Implementing a Contract

The **service contract** is one of the fundamentals of WCF, defining operations supported by the service and other aspects such as error handling.

### Creating Service and Data Contracts

A WCF service contract is a standard interface but a **[ServiceContract]** attribute is added .

The interface methods that are exposed as service operations will each include on **[OperationContract]** attribute indicating that metheod is part of a service contract. In System.ServiceModel assembly.

Better to apply **[ServiceContract]** to an interface rather than a class as better SoC.

The **[DataContract]** attribute is applied to the class. The **[DataMember]** attribute is applied to class properties that will be included in the data contract. (In System.Runtime.Seralization assembly)

DataContract and DataMember attributes are optional, WCF will automatically try to serialize every public property and field it encounters in your class. Can take an included of exclusive approach – **[IgnoreDataMember]**

### Implementing a Service Contact

Concrete class implementing the ServiceContract, can control:

* **Instantiation** – when you send a request to the service the request is executed in an instance of the service class. The service instantiation controls when new instances of your service class are created
* **Concurrency** – Each request in WCF runs in its own tread. But when several requests running in different threads execute in parallel they might attempt to use the same service. The concurrency setting controls how many requests can use the same service instance concurrently.

These are controlled by the **[ServiceBehavior]** attribute, using the **InstanceContextMode** and **ConcurrencyMode** paramemeters.

**Instancing options:**

* **Per Call** – created and destroyed for each request
* **Single** – one instance for all requests, destroyed when service closes
* **Per Session** – new instance per client connection (session) destroyed on disconnect or idle

When Per Session or Single instancing modes, can use a **concurrency** mode:

* **Single** – (default) only single request can execute at a time
* **Multiple**
* **Reentrant** – single request a t a time but if method calls another service the instance is released rather than blocked

### Handling exceptions

WCF service cannot throw exceptions back to client:

* Client may not understand the exception
* .Net framework exception exposes sensitive details

SOAP fault messages are used instead. Can include .Net exception details in the fault with **IncludeExceptionDetailsInFaults** parameter of **[ServiceBehavior].** You can also throw a **FaultException**, if a string is not enough can use the **FaultException<T>** class where T specifies the data contract that holds extra information about the error. Need to use **[FaultContract]** attribute in this instance.

## Configuring and Hosting WCF Services

### Hosting WCF Services

The service host is responsible for opening ports and listening to requests, managing incoming requests, allocating resources, creating the service instance contact and passing this through the WCF runtime.

* **Self-hosting**
* **Web-hosting** – within IIS

Base class that manages WCF hosts in **ServiceHostBase**.

### Service endpoints

The endpoint is the entry point to the service, it receives messages from a communications cannel and transfers the message to the service. A service can have numerous endpoints, each listening to different types of communication. Each endpoint has a different address.

A service endpoints answers:

* **A**ddress - where the service resides, a URL
* **B**inding – how clients should communicate with the service: encoding, transport type, security, session support etc
* **C**ontract – operations supported by the endpoint, needs to match a contract interface

### Defining a service endpoint address

Endpoint address has:

* Scheme – HTTP, TCP
* Machine name or IP address
* Port (optional)
* Path (e.g./bookings)

You can only have one base address per URI scheme. Base address cannot be shared across bindings.

### Defining service endpoint bindings

Binding encapsulates all the technology decisions required to pass a message from point A to point B:

* Transport
* Encoding
* Protocols e.g. security, sessions

**Predefined bindings:**

* BasicHttpBiding
* WSHttpBinding
* NetTcpBinding
* NetNamedPipeBinding
* UdpBinding
* NetHttpBinding

### Defining Service Endpoint Contracts

If your service class implements more than one contact, you need to create multiple endpoints to expose the contracts to clients. They may use the same binding and binding configuration but have different addresses.

### Service Metadata

Can expose metadata by HTTP GET or Metadata Exchange (MEX) endpoints, can secure the metadata with the latter.

## Consuming WCF Services

To create proxy classes can add a service reference in visual studio via the WCF service WSDL.

### Creating a Service Proxy with ChannelFactory<T>

Another way to create a service proxy is using **System.ServiceModel.ChannelFactory<T>** generic class. Creates a proxy class at runtime, developer needs to provide service contract interface and the data contract classes.

To use ChannelFactory<T> you need access to the service contract interface and data contracts, through a shared library. Can use class in two ways, static CreateChannel or create an instance of ChannelFactory<T> then use the CreateChannel method of the instance.

# 6. Hosting Services

## Hosting On-Premises

When you want to host a web service on premises you can host it using a Windows Service or IIS.

### Self-Hosting WCF Services in Windows Services

* Create a windows service project and add the WCF hosting code to it
* Create an installer class for the windows service
* Install the windows service (**InstallUtil.exe**)

### Hosting WCF Services in IIS

Can benefit from health monitoring, app pools etc in IIS.

Do not need to use a base address – IIS always uses the URL of the .svc file as the base address for the service.

You can use the web.config to configure a service without using SVC files via the serviceActivations node.

### Self-Hosting ASP.NET Web API Services

You can use **HttpSelfHostServer** to host an WEB.NET Web API Service, for example inside a console app.

## Hosting Services in Windows Azure

* **Worker role** – hosts background processes for long tasks, similar to Windows service
* **Web role** – host services used as a front-end to the Internet
* **Web site** – similar to web role with fast deployment but fewer features
* Virtual machines

### Windows Azure Cloud Services

* Easy to create and deploy (PaaS)
* Automatic system updates
* Hardware monitoring

### Windows Azure Web Role

* IIS-based hosting environment – ASP.NET, WCF, Node.JS
* Can mix different web applications in a single deployment
* Exposed via Azure Load Balancer and Firewall

### Windows Azure Worker Role

* Do not have IIS enabled by default
* Hosting public services – hosts a WCF or Web API service and open the worker role for incoming communication
* Back end processing
* Windows Azure Queue storage
* Windows Azure service bus

### Windows Azure Web Sites

* Simple and fast deployment
* Support for source code repository deployment
* Share content and configuration across multiple instances

Limited features

* Not for multi-tired applications with background processing
* No staging/production environment

# 7. Windows Azure Service Bus

## Windows Azure Service Bus Relays

The Service Bus Relay is a middle-tier for connecting remote clients and cloud-based applications with on premises services.

### What is Windows Azure Service Bus Relay?

* Expose services without opening corporate firewall
* Connectivity to on prem from
  + Cloud based applications
  + Other corporate networks
  + Mobile devices
* Relay supports multiple message patterns
  + Unicast/muticast one way message forwarding
  + Request – response socket forwarding
  + Update to point-to-point socket
* Leverage WCF with new bindings

### One way messaging

Client sends message to service without expecting response. Because the client cannot communicate directly with the service, Service Bus is the mediator.

2 forms of one way:

* **NetOnewayRelayBinding** – only has one receiver
* **NetEventRelayBinding** – multiple receivers

### Request-Response Relays

Made possible with Service Bus by using socket-to-socket forwarding and rendezvous connections.

Socket forwarding looks like a normal socket but is slow. To instruct Service Bus relay to try establishing a direct connection between sender and receive you need to change the connection mode of the binding from a relayed connection to a hybrid connection.

### Creating Windows Azure Service Bus Namespaces

Service Bus namespace is a unique name used for addressing and resource configuration. URIS use the form sb://namespace.servicebus.windows.net

### Using WCF Relay Bindings

Service Bus Relay uses WCF as its underlying programming model. You build receives by implementing a simple WCF service and exposing endpoint that use relay bindings. You also need to authenticate against the relay by attaching a **TransportClientEndpointBehaviour**.

## Windows Azure Service Bus Queues

### What is brokered messaging?

* Producers and consumers to not have to be online at the same time
* Messages persisted until consumer ready to receive and process them
* Messages sent one-way
* Queues service as basic persistence store offer FIFO message delivery
* Topics offer muticast messaging depending on subscriptions and filters
* Topics used to implement the pub/sub pattern

### Windows Azure Service Bus brokered messaging

Core components of the Azure Brokered messaging infrastructure are:

* **Service Bus Queues** – basic persistent communication channels
* **Service Bus Topics** – special kind of queue where multiple consumers can receive a single message, can use to implement a publish/subscribe pattern

Creating a new queue in C# - **NamespaceManager.CreateFromConnectionString**

### Duplicate Detection

You can use duplicate detection to ensure that messages will only be sent once. **Request Duplication Detection** option when creating Queue or if in code, create a **QueueDescription** and set **RequiresDuplicateDetection** property.

### The BrokeredMessage Class

Represents a message transferred through queues and topics:

* GetBody
* Complete
* Abandon
* RenewLock

**Body** – Payment data, transparent to infrastructure

**Properties** – User defined key-value collection with data that is visible to infrastructure

**Metadata** – size, id, content-type etc

### Sending and Receiving Messages

After the queue is ready you can send and receive messages from the queue by using the **QueueClient** class:

var client = QueueClient.CreateStringConnectionString(conStr, queueName);

or

var factory = MessageFactory.Create(serviceUri, credentials);

var client = factory.CreateQueueClient(queueName);

var additionalClient = new QueueClient(queueName);

### Sending messages

Create an instance of **BrokeredMessage** and send it using the queue client.

### Sessions

You can use sessions to group message that belong to a certain logical group and receive them all on a dedicated receiver, senders can attach a **SessionId** to outgoing messages. Can use this to get around the 265kb message size limit.

### Receiving Messages

Call the Receive method of **QueueClass**, this will internally create an instance of a **MessageReceiever**. Call **Complete()** on the message to delete it from queue (peek-lock).

## Windows Azure Service Bus Topics

### Creating Service Bus Topics

Use the **NamespaceManager** class, from the namespaceMananger call **CreateTopic**(topicName)

### Creating Topic Subscriptions

var namespaceMananger = NamespaceMananger.CreatefromConnectionString(connStr);

namespaceManager.createTopic(“ProductUpdates”);

namespaceMananger.CreateSubscription(“ProductUpdates, “Inventory”);

### Creating filters

* **SqlFilter** – forwards messages based on sql like expression
* **CorrelationFilter** – based on CorrelationId property of the brokered message
* **TrueFilter**– messages always forwarded
* **FalseFilter** – messages never forwarded

namespaceMananger.CreateSubscription”ProductUpdates”, “Inventory”, new TrueFilter());

# 8. Deploying Services

### Web Deploy

- Web Deploy

- IIS Web Deployment packages

- MSDeploy – command line tool

### Deploying Services to Windows Azure Cloud Services

VS20120 supports deploying directly to windows Azure and creating Azure packages for Web/Worker roles

* Windows Azure packages can be manually deployed through Windows Azure Management Portal
* Deployment can be automated with PowerShell cmdlets

### Deploying to Windows Azure Web Sites

* Windows Azure Web Sites have publish profiles for quick publishing of web applications
* Profile contains
  + Destination server
  + Login credentials
  + Database information
* Use publish profiles with VS2012
  + Start publishing a web application
  + Import downloaded publish profile
  + Verify connection to destination server
  + Configure settings and publish

### Continuous Delivery with TFS and Git

* Decrease time to market
* Increase development teams confidence
* Reduce risk

## Best practises for deployment

### Web.config transforms

### Sharing Assemblies in Shared Hosting Scenarios

**aspnet\_intern** tool

### Windows Azure Upgrade Domains

When you deploy an app to an onprem scale out deployment, it is typical to take one server offline at a time, update then bring back online.

Similar scenarios is used in Azure when you deploy a new version of a role in additional to an existing deployment, an in-place update.

### Deploying to Staging and Production Environments

* Windows Azure services has two environments, production and staging
* Staging environments offer same hardware but have different VIP and service URL
* Upgrade production environments by swapping the VIP and service URL of the staging and production environments.