# 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();