# Contents

Overview ................................................................................................................................. 4
Objective ................................................................................................................................. 4
Architecture ............................................................................................................................ 4

## Design Principles ............................................................................................................ 9
- S.O.L.I.D .............................................................................................................................. 9
- Single Responsibility Principle ......................................................................................... 9
- Open-Closed Principle ..................................................................................................... 9
- Liskov Substitution Principle ......................................................................................... 9
- Interface Segregation Principle ...................................................................................... 10
- Dependency Inversion Principle (DIP) .......................................................................... 10

## Architecture Patterns .................................................................................................... 10
- Microservices .................................................................................................................... 10
- Back End for Front End .................................................................................................... 10
- Gateway Aggregation pattern ....................................................................................... 10
- CQRS ................................................................................................................................. 11
- Repository ......................................................................................................................... 11
- Dependency Injection ...................................................................................................... 11
- Repository Pattern .......................................................................................................... 11
- CQRS ................................................................................................................................. 11
- DDD ................................................................................................................................. 11
- REST ................................................................................................................................. 12
- Dispose Pattern ............................................................................................................... 13
- Command Pattern .......................................................................................................... 13
- Strategy Pattern ............................................................................................................... 13

## Best Practices .................................................................................................................. 13

- Coding Conventions ....................................................................................................... 13
  - Always Use Properties instead of Public Variables .................................................... 14
  - Use Nullable Data Types Whenever Required .......................................................... 14
  - Prefer Runtime Constants over Compile time Constants ............................................ 14
  - Prefer “is” and “as” Operators While Casting ............................................................ 14
  - Prefer `string.Format()` or `StringBuilder` for String Concatenation .................. 14
  - Properly Utilize `try/catch/finally` Blocks .............................................................. 14
- Catch Only that Exception that You Can Handle
- Use IDisposable Interface
- Split your Logic in Several Small and Simple Methods

API Design
Caching
Autoscaling
Background Jobs
Monitoring and diagnostics
Retry guidance for specific services
Transient fault Handling
Technology Stack
Solution Structure
Appendix I
Pre-Requisites
Overview
ODE has made a strategic decision to migrate their EAS applications to Microsoft Azure. Current EAS application is a monolithic multi-tier architecture. The current application components are tightly coupled and difficult to reuse and scale the application.

Objective
The primary objective if this project is to rearchitect the current EAS application to use the modern microservice architecture and cloud first approach. The goal is to break the monolithic application to the re-usable microservices so that other departments can re-use them if needed. Also they can scale the microservices as needed.

Architecture
The following diagram shows the high level architecture of this application. The application business logs as well as the UI components are broken down into several microservices and they are deployed to Azure Service Fabric.
Azure Service Fabric

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers. Service Fabric also addresses the significant challenges in developing and managing cloud native applications.

Developers and administrators can avoid complex infrastructure problems and focus on implementing mission-critical, demanding workloads that are scalable, reliable, and manageable.

Service Fabric enables you to build and manage scalable and reliable applications composed of microservices that run at high density on a shared pool of machines, which is referred to as a cluster. It provides a sophisticated, lightweight runtime to build distributed, scalable, stateless, and stateful microservices running in containers. It also provides comprehensive application management capabilities to provision, deploy, monitor, upgrade/patch, and delete deployed applications including containerized services.

Azure Application Gateway

Azure Application Gateway gives you application-level routing and load balancing services that let you build a scalable and highly-available web front end in Azure. You control the size of the gateway and scale your deployment based on your needs. Application Gateway also protect your application from common web vulnerabilities and exploits like SQL Injection or Cross site scripting. Customize rules to reduce false positives.

The EAS infrastructure and application component are isolated inside an Azure Virtual Network. Azure Virtual Network gives you an isolated and highly-secure environment to run your virtual machines and applications. Use your private IP addresses and define subnets, access control policies, and more. Use Virtual Network to treat Azure the same as you would your own datacenter.
Traffic between Azure resources in a single region, or in multiple regions, stays in the Azure network—intra-Azure traffic doesn’t flow over the Internet. In Azure, traffic for virtual machine-to-virtual machine, storage, and SQL communication only traverses the Azure network, regardless of the source and destination Azure region. Inter-region virtual network-to-virtual network traffic also flows entirely across the Azure network. Use Virtual Network to extend your on-premises IT environment into the cloud, like you set up and connect to a remote branch office. You have options to securely connect to a virtual network—choose an IPsec VPN or a private connection by using Azure ExpressRoute.

Application Insights

Application Insights is an extensible Application Performance Management (APM) service for web developers on multiple platforms. Use it to monitor your live web application. It will automatically detect performance anomalies. It includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app. The impact on your app’s performance is very small. Tracking calls are non-blocking, and are batched and sent in a separate thread.
Microsoft Azure Service Bus

Microsoft Azure Service Bus is a reliable information delivery service. The purpose of this service is to make communication easier. When two or more parties want to exchange information, they need a communication facilitator. Service Bus is a brokered, or third-party communication mechanism. This is similar to a postal service in the physical world. Postal services make it very easy to send different kinds of letters and packages with a variety of delivery guarantees, anywhere in the world.

Service Bus supports two distinct messaging patterns: Azure Relay and Service Bus Messaging.

**Azure Relay:** The relay service supports traditional one-way messaging, request/response messaging, and peer-to-peer messaging. It also supports event distribution at Internet-scope to enable publish-subscribe scenarios and bi-directional socket communication for increased point-to-point efficiency. In the relayed messaging pattern, an on-premises service connects to the relay service through an outbound port and creates a bi-directional socket for communication tied to a particular rendezvous address.

**Brokered messaging:** Service Bus messaging with queues, topics, and subscriptions can be thought of as asynchronous, or "temporally decoupled." Producers (senders) and consumers (receivers) do not have to be online at the same time. The messaging infrastructure reliably stores messages in a "broker" (for example, a queue) until the consuming party is ready to receive them. This enables the components of the distributed application to be disconnected, either voluntarily; for example, for maintenance, or due to a component crash, without affecting the entire system. Furthermore, the receiving application may only have to come online during certain times of the day, such as an inventory management system that only is required to run at the end of the business day.
**Azure SQL Server**

Azure SQL Database is a relational database-as-a-service using the Microsoft SQL Server Engine. SQL Database is a high-performance, reliable, and secure database you can use to build data-driven applications and websites in the programming language of your choice, without needing to manage infrastructure.

**Azure Cosmos DB**

Cosmos DB is a database for extremely low latency and massively scalable applications anywhere in the world, with native support for NoSQL. Azure Cosmos DB allows you to use key-value, graph, column-family, and document data in one service. Azure Cosmos DB automatically indexes all data, and allows you to use your favorite API including SQL, JavaScript, Gremlin, MongoDB, and Azure Table Storage to access your data.

**Azure Blob Storage**

Microsoft Azure Storage is a Microsoft-managed cloud service that provides storage that is highly available, secure, durable, scalable, and redundant. Azure Storage consists of Blob storage, File Storage, and Queue storage.

Common uses of Blob storage include:

- Serving images or documents directly to a browser
- Storing files for distributed access
- Streaming video and audio
- Storing data for backup and restore, disaster recovery, and archiving
- Storing data for analysis by an on-premises or Azure-hosted service

**Redis Cache**

Azure Redis Cache is based on the popular open source Redis cache. It gives you access to a secure, dedicated Redis cache, managed by Microsoft and accessible from any application within Azure. Azure Redis Cache helps your application become more responsive even as customer load increases. It takes advantage of the low-latency, high-throughput capabilities of the Redis engine. This separate, distributed cache layer allows your data tier to scale independently for more efficient use of compute resources in your application layer.
Service Architecture

Design Principles

S.O.L.I.D

Single Responsibility Principle
A class should have one and only one reason to change, meaning that a class should have only one job.

Open-Closed Principle
Objects or entities should be open for extension, but closed for modification.

Liskov Substitution Principle
Derived classes must be substitutable for their base classes

Let $q(x)$ be a property provable about objects of $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$ where $S$ is a subtype of $T$. 
Interface Segregation Principle
A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

Dependency Inversion Principle (DIP)
Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.


Architecture Patterns
Microservices
Microservices is an approach to application development in which a large application is built as a suite of modular services. Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other sets of services.

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.

[http://microservices.io/patterns/microservices.html](http://microservices.io/patterns/microservices.html)

Back End for Front End
Create separate backend services to be consumed by specific frontend applications or interfaces. This pattern is useful when you want to avoid customizing a single backend for multiple interfaces.

Gateway Aggregation pattern
Use a gateway to aggregate multiple individual requests into a single request. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation. Use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.
CQRS
Repository
Dependency Injection

https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation

Repository Pattern
Use a repository to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer. For example, the data source layer can be a database, a SharePoint list, or a Web service.


CQRS
Segregate operations that read data from operations that update data by using separate interfaces. This can maximize performance, scalability, and security. Supports the evolution of
the system over time through higher flexibility, and prevent update commands from causing merge conflicts at the domain level.

https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs

DDD

**Domain-driven design (DDD)** is an approach to software development for complex needs by connecting the implementation to an evolving model. The premise of domain-driven design is the following:

- placing the project's primary focus on the core domain and domain logic;
- basing complex designs on a model of the domain;
- initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.


http://dddsample.sourceforge.net/patterns-reference.html

REST

**RESTful** Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

The table below describes the most commonly used HTTP methods, their semantics, and whether they're defined to be safe and idempotent.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Safe</th>
<th>Idempotent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Requests a specific representation of a resource</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PUT</td>
<td>Create or update a resource with the supplied representation</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes the specified resource</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>Submits data to be processed by the identified resource</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HEAD</td>
<td>Similar to GET but only retrieves headers and not the body</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Returns the methods supported by the identified resource</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Dispose Pattern

All programs acquire one or more system resources, such as memory, system handles, or database connections, during the course of their execution. Developers have to be careful when using such system resources, because they must be released after they have been acquired and used.

The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the `IDisposable` interface.

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern

Command Pattern

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. This pattern is commonly used in CQRS and DDD

http://www.dofactory.com/net/command-design-pattern

Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

http://www.dofactory.com/net/strategy-design-pattern

Best Practices

Coding Conventions

You should prefer proper naming conventions for consistency of your code. It is very easy to maintain the code if consistent naming is used all over the solution. Here are some naming conventions which are generally followed by .NET developers:

- Always use Camel case (A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized) while declaring variables.
- Use Pascal (A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized) naming format while declaring Properties.
- Avoid all uppercase or lowercase names for properties, variables or method names. Use all uppercase when declaring const variables.
- Never use a name that begins with a numeric character.
Always prefer meaningful names for your class, property, method, etc. This will be very useful for you to maintain the code in future. For example, “P” will not give proper meaning for a class. You will find it difficult to know about the class. But if you use “Person”, you will easily understand by it.

Never build a different name varied by capitalization. It is a very bad practice. It will not be useful while developing code, as you will not know what is “person” class and what is “Person” class!!! But from the above scenario, it can be very easily understandable that “person” is an instance variable of “Person” class.

Don’t use the same name used in .NET Framework. People who are new to your code have great difficulty to understand it easily.

Avoid adding prefixes or suffixes for your identifiers. Though in some guidelines, they use “m_” and in some other they use “_” as the prefix of variable declaration. I think it is not that much useful. But, it depends on your organizational coding practices. This point is contradictory based on various organizations and there is no strict guidance on it.

Always use “I” as prefix for Interfaces. This is a common practice for declaring interfaces.

Always add “Exception” as suffix for your custom exception class. It will give better visibility to your exception class.

Never prefix or suffix the class name to its property names. It will unnecessarily increase the property name. If “Firstname” is a property of “Person” class, you can easily identify it from that class directly. No need to write “PersonFirstname” or “FirstnameOfPerson”.

Prefix “Is”, “Has” or “Can” for boolean properties like “IsVisible”, “HasChildren”, “CanExecute”. These give proper meaning to the properties.

Don’t add prefix for your controls, instead write proper name to identify the control.

Always Use Properties instead of Public Variables

Use Nullable Data Types Whenever Required

Prefer Runtime Constants over Compile time Constants

Prefer “is” and “as” Operators While Casting

Prefer string.Format() or StringBuilder for String Concatenation

Properly Utilize try/catch/finally Blocks

Catch Only that Exception that You Can Handle

Use IDisposable Interface

Split your Logic in Several Small and Simple Methods
**API Design**

**Caching**

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. If this fast data storage is located closer to the application than the original source, then caching can significantly improve response times for client applications by serving data more quickly.

Caching is most effective when a client instance repeatedly reads the same data, especially if all the following conditions apply to the original data store:

- It remains relatively static.
- It's slow compared to the speed of the cache.
- It's subject to a high level of contention.
- It's far away when network latency can cause access to be slow.


**Autoscaling**

Autoscaling is the process of dynamically allocating resources to match performance requirements. As the volume of work grows, an application may need additional resources to maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.


**Background Jobs**

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction—the application can start the job and then continue to process interactive requests from users. This can help to minimize the load on the application UI, which can improve availability and reduce interactive response times.


**Monitoring and diagnostics**

Distributed applications and services running in the cloud are, by their nature, complex pieces of software that comprise many moving parts. In a production environment, it's important to be able to track the way in which users utilize your system, trace resource utilization, and generally monitor the health and performance of your system. You can use this information as a diagnostic aid to detect and correct issues, and also to help spot potential problems and prevent them from occurring.

Retry guidance for specific services

Most Azure services and client SDKs include a retry mechanism. However, these differ because each service has different characteristics and requirements, and so each retry mechanism is tuned to a specific service. This guide summarizes the retry mechanism features for the majority of Azure services, and includes information to help you use, adapt, or extend the retry mechanism for that service.

Transient fault Handling

All applications that communicate with remote services and resources must be sensitive to transient faults. This is especially the case for applications that run in the cloud, where the nature of the environment and connectivity over the Internet means these types of faults are likely to be encountered more often. Transient faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay it is likely succeed.

Technology Stack

<table>
<thead>
<tr>
<th>Technology</th>
<th>Help link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure</td>
<td>(<a href="https://portal.azure.com">https://portal.azure.com</a>)</td>
</tr>
<tr>
<td>Angular</td>
<td><a href="https://angular.io/">https://angular.io/</a></td>
</tr>
<tr>
<td>Bootstrap</td>
<td><a href="https://getbootstrap.com/">https://getbootstrap.com/</a></td>
</tr>
<tr>
<td>.NET CORE</td>
<td><a href="https://github.com/dotnet/core">https://github.com/dotnet/core</a></td>
</tr>
<tr>
<td>Entity Framework Core</td>
<td><a href="https://github.com/aspnet/EntityFrameworkCore">https://github.com/aspnet/EntityFrameworkCore</a></td>
</tr>
<tr>
<td>Dapper</td>
<td><a href="https://github.com/StackExchange/Dapper">https://github.com/StackExchange/Dapper</a></td>
</tr>
<tr>
<td>Azure Service Bus</td>
<td><a href="https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions">https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions</a></td>
</tr>
<tr>
<td>Cosmos DB</td>
<td><a href="https://docs.microsoft.com/en-us/azure/cosmos-db/introduction">https://docs.microsoft.com/en-us/azure/cosmos-db/introduction</a></td>
</tr>
</tbody>
</table>
Solution Structure

The solution is currently hosted in the VSTS. [https://eascloud.visualstudio.com/EASCloud](https://eascloud.visualstudio.com/EASCloud)

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>.git</td>
<td>3/2/2018 3:00 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>.vs</td>
<td>3/2/2018 12:06 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>cli</td>
<td>2/28/2018 11:11 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>2/26/2018 9:07 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>packages</td>
<td>2/23/2018 12:40 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>ServiceFabric</td>
<td>3/2/2018 2:10 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>src</td>
<td>2/23/2018 12:41 PM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>Tests</td>
<td>2/26/2018 11:50 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>.gitignore</td>
<td>3/2/2018 2:50 PM</td>
<td>Text Document</td>
<td>2 KB</td>
</tr>
<tr>
<td>EAS.sln</td>
<td>3/2/2018 12:03 PM</td>
<td>Visual Studio Solution</td>
<td>19 KB</td>
</tr>
<tr>
<td>EAS_All.sln</td>
<td>3/2/2018 12:49 PM</td>
<td>Visual Studio Solution</td>
<td>24 KB</td>
</tr>
<tr>
<td>EAS_CaseManagement.sln</td>
<td>3/2/2018 2:10 PM</td>
<td>Visual Studio Solution</td>
<td>16 KB</td>
</tr>
<tr>
<td>README.md</td>
<td>2/23/2018 12:39 PM</td>
<td>MD File</td>
<td>2 KB</td>
</tr>
</tbody>
</table>

**Cl**i folder contains command line scripts, for example the script for building the angular project

**ServiceFabric** Folder contains the Deployment definitions for the Service Fabric Project. For each of the Microservice you develop, create a Service Fabric Application Project here. Update the ApplicationManifest.xml, Cloud.xml and Local.1Node.xml
Src folder contains the MicrosService and Web components.

Any new Microservices created should be inside the src\Services folder.

Create a separate folder for the Microservices, for example CaseManagement for CaseManagementDetail.Api Service

MicrosService Project folder Name should end with .Api

Micoservice Assembly Name should end with Service as shown below
Make sure to update the Default Namespace to Ode.EAS.Services.[ServiceName]

Any new Web application created should be inside the src\Web folder.

Create a separate folder for the Microservices, for example CaseManagement for CaseManagementWeb Service

Create a Build script for the angular project, the sample script in cli folder.

**Microservice Project Structure**

All your controller code goes to the **Controller** Folder

Add the Produces, Route and Authorize attribute to your controller class
Add the HTTP Action attribute and Response Type attribute to your Action Method

```csharp
[HttpPost("AuthorizedAssets")]
[ProducesResponseType(typeof(AssetViewModel), (int)HttpStatusCode.OK)]
public IActionResult GetAuthorizedAssets(AssetRequestViewModel authorizeVm)
{
    try
    {
        var assets = _assetRepository.GetAuthorizedAssets(authorizeVm.OrgKey,
                                                        authorizeVm.ApplicationKey,
                                                        authorizeVm.PersonKey,
                                                        authorizeVm.AudienceTypeKey);
        if (assets == null)
        {
            return BadRequest("Unable to get assets for the application : {authorizeVm.ApplicationKey}"");
        }

        var assetVm = Mapper.Map<List<Asset>, AssetViewModel>(assets);
        return Ok(assetVm.ToAssetViewModel());
    }
    catch
    {
        return BadRequest("Unable to get assets for the application : {authorizeVm.ApplicationKey}"");
    }
}
```

Return `IActionResult` or `Task<IActionResult>` from the Controller methods.

For example:

Add your extension methods to the `Extensions` folder

Filters go to the `Filters` folder.

`AuthorizationHeaderParameterOperationFilter` is a required for the authorization.
**Infrastructure** folder contains the database related code. The Entity Framework Models and Repositories go into that folder. Always create interfaces for your repositories and inject into the code as needed.

**Mappings** folder contains your AutoMapper profiles.

**Models** folder contains any non-database models required for the applications.

**Service** folder contains your application specific internal services code.

**ViewModels** are your request/response model for the REST API.

**PackageRoot** contain your Service Fabric Manifest. This is an autogenerated file, but you can update it as needed.

The Code Package Program name must be the same as your assembly Name. The Service Type name and EndPoint name must match your end point name in the program.cs file.

```xml
<CodePackage Name="Code" Version="1.0.0"/>
</ServiceManifest>
``` 

Open the StaticEventSource.cs and update the Event Source Name.
Open the Startup.cs file and add necessary configurations, check out and exerting project for details.

On ConfigureServices methods
Add

```csharp
services.AddCors
services.AddDbContext
services.AddAutoMapper
services.AddDistributedRedisCache
services.AddAuthentication
services.AddSwaggerGen
services.AddLogging
services.AddMvc
```

On Configure method
Add

```csharp
app.UseAuthentication
app.UseStaticFiles
app.UseSwagger
app.UseSwaggerUI
app.UseMiddleware
app.UseCors
app.UseMvcWithDefaultRoute
```

Open the Service Definition class, update theWebHostBuilder initialization code
Add
Web Project Structure

Create folder called app

Copy your angular code into that folder

Create a Build script for the angular project, the sample script in cli folder.

```bash
@rem
@echo on

cd %1app

call npm install

call npm run build

if not exist %1wwwroot mkdir %1wwwroot

xcopy /S /Q /Y %1app\dist\* %1wwwroot\*
```
Add a pre build event to run the build script

Database
## Appendix I
### Pre-Requisites

<table>
<thead>
<tr>
<th>Developer Tool</th>
<th>Installer Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2017</td>
<td>(Available in developer machine, no install necessary)</td>
</tr>
<tr>
<td>.NET Core SDK 2.xx</td>
<td><a href="https://www.microsoft.com/net/download/windows">https://www.microsoft.com/net/download/windows</a></td>
</tr>
<tr>
<td>VS Code</td>
<td><a href="https://code.visualstudio.com/">https://code.visualstudio.com/</a></td>
</tr>
<tr>
<td>Microsoft Service Fabric SDK (for VS 2017)</td>
<td>Install from Web Platform Installer</td>
</tr>
<tr>
<td>Linq Pad</td>
<td><a href="http://www.linqpad.net/">http://www.linqpad.net/</a></td>
</tr>
<tr>
<td>Docker Container</td>
<td><a href="https://store.docker.com/editions/community/docker-ce-desktop-windows">https://store.docker.com/editions/community/docker-ce-desktop-windows</a></td>
</tr>
<tr>
<td>Redis Cache</td>
<td>From Command Line run the following command: docker run –name eas-cloud –d redis</td>
</tr>
<tr>
<td></td>
<td>Run the following command to confirm: docker ps</td>
</tr>
<tr>
<td></td>
<td>Redis cache will now accept connection on container port 6379</td>
</tr>
</tbody>
</table>