JENSEN yrkeshögskola
TRÄNING FÖR VERKLIGHETEN

OverwatchNext

Node.js och Microsoft

FRK22G

VT-2024

Carl Bergman

Handledare: Klas Hermodsson

Sammanfattning

I detta arbete tar jag upp en del av de tekniker jag använt för att bygga om frontend koden för en webbapp vid namn Overwatch.
Overwatch är en applikation som används ute på fabrikerna för att få mail eller sms när ett värde når en bestämd setpoint.

Overwatch frontend är asp.net och Visual Basic i bakgrunden vilket är det jag har bytt ut till Next.JS, stora delar av logiken till Overwatch ligger dock inte i Overwatch frontend utan i HistoricalData vilket är ett klassbibliotek som är byggt i VB.net v4.0, detta bibliotek har jag till hög grad återanvänt men även detta har krävts sina beskärda uppdateringar.

I detta projektet så implementerade jag node.js i ett intranät som nästan uteslutet lever i Microsofts ekosystem, koden är skriven i .net, hemsidorna hostas på IIS, AD används för auktorisering och access, mm...
Därav blev jag tvungen att implementera ett antal nischade tekniker inom Node.js och Microsoft.
De två stora teknikerna jag går igenom i detta arbete är IISNode och Edge.js.
IISNode är en modul som fungerar som en brygga mellan Node.js och IIS.
IIS tar emot HTTP anropen, IISNode läser av dem och skickar informationen till Node.js som sedan kan ta hand om logiken för sidan.
Edge.js är ett bibliotek skapat av samma utvecklare som skapade IISNode (Tomas Tjanczuk). Biblioteket ger dig möjligheten att köra .net funktioner i din Node.js process (Funkar även med en rad vanilla språk så som C#, Pyhon och F#).

Jag tar även upp hur datan är sparad och vad som krävs för att kunna uppdatera, eller lägga till ny data till databasen.

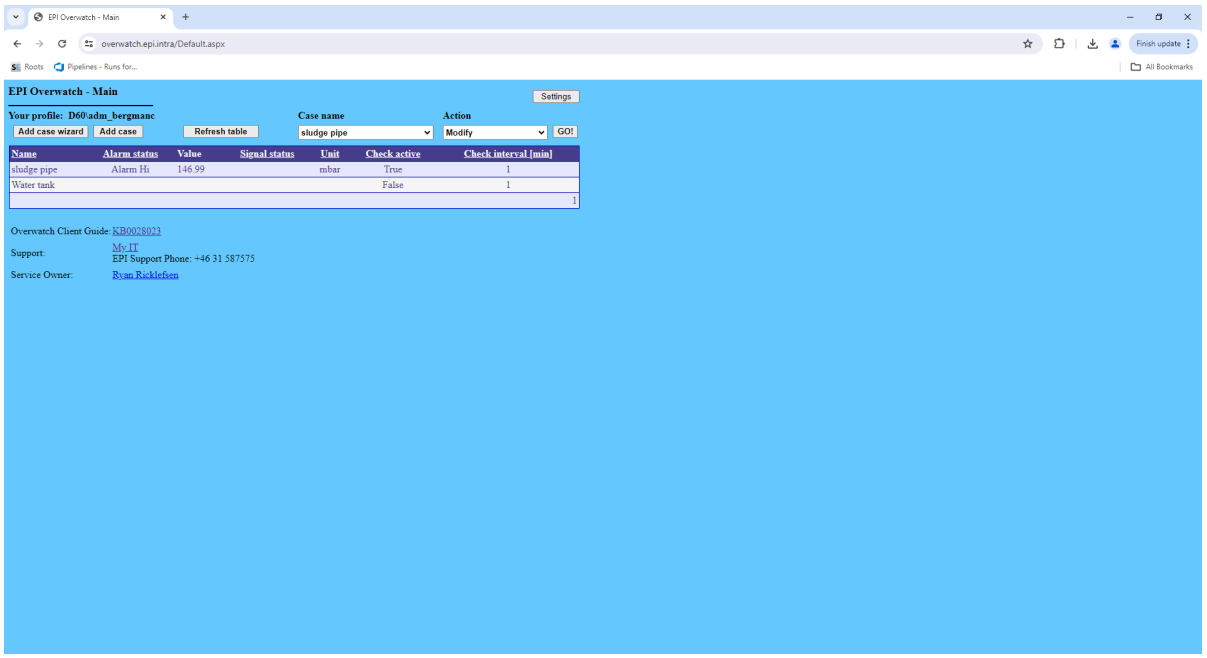
Innehåll

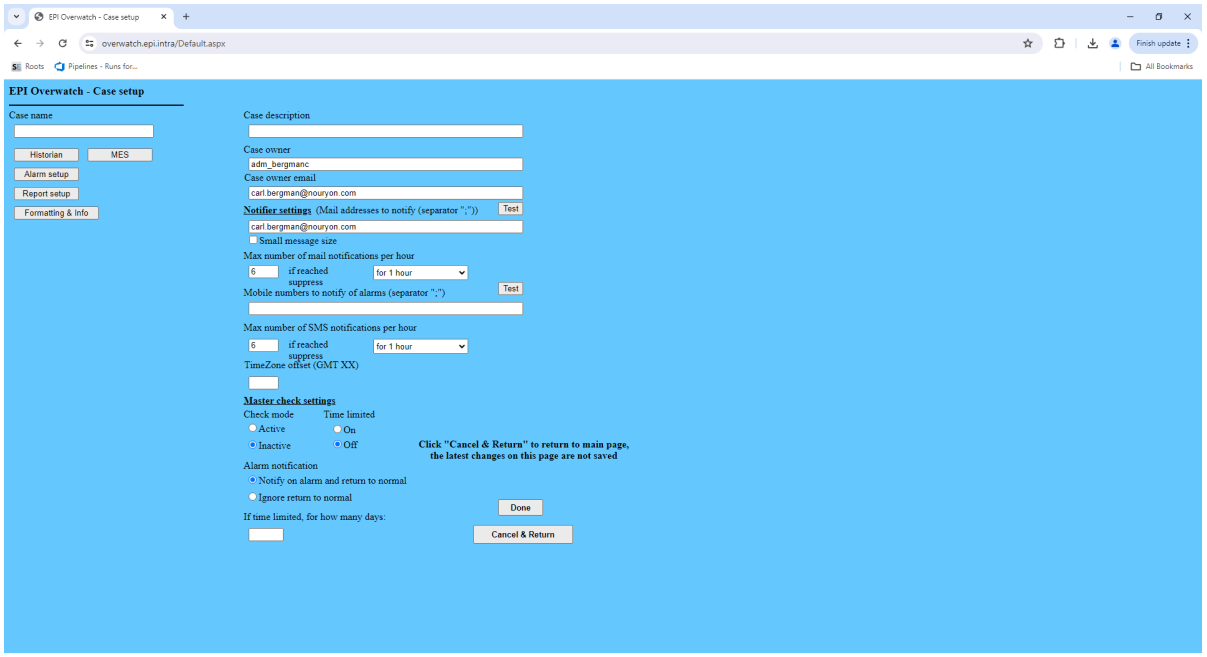
1. Inledning

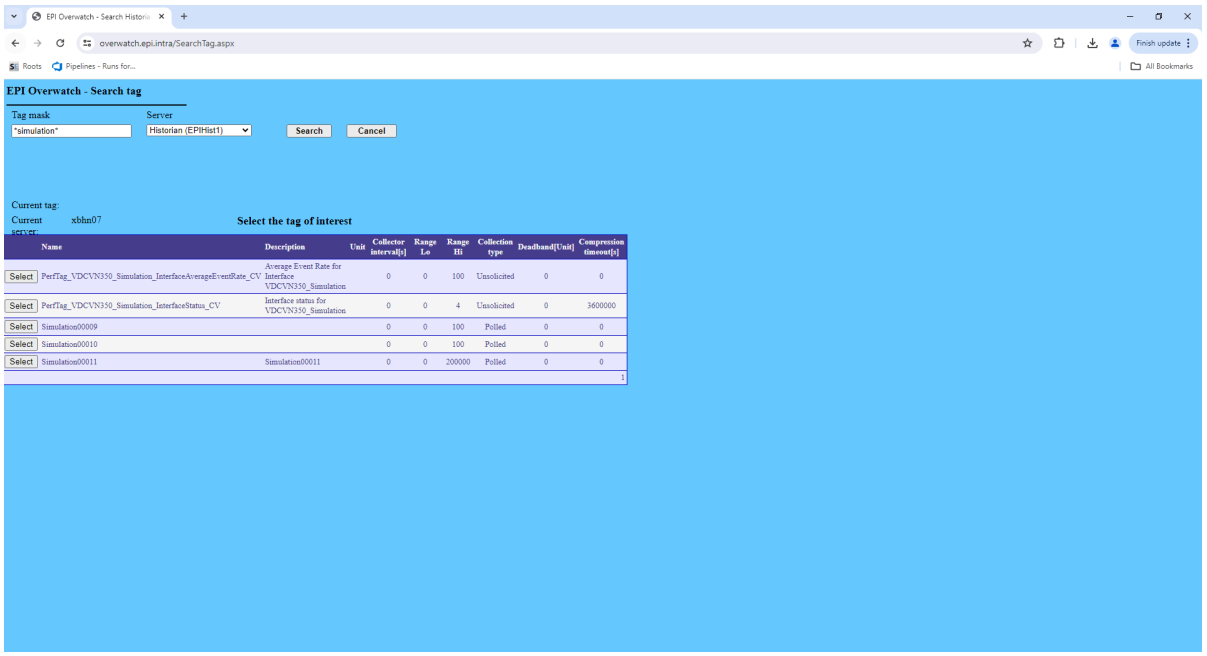
1.1 Overwatch

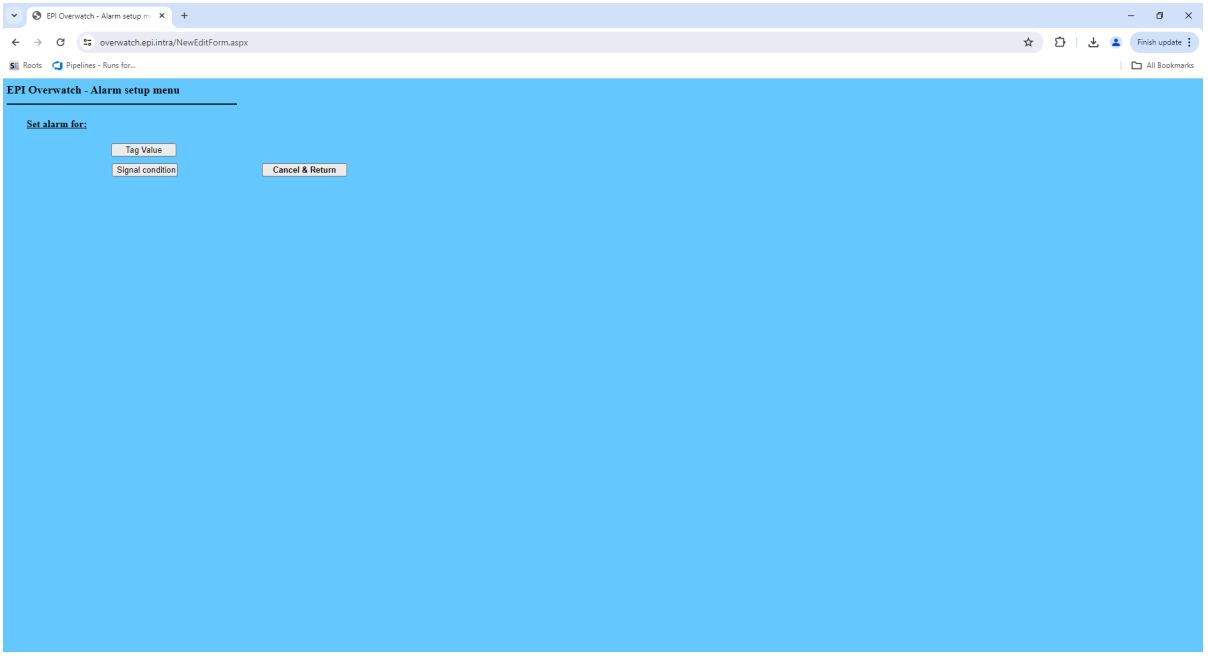
Under min LIA så fick jag och en kursare i uppdrag att bygga om frontend delen av en webb app vid namn Overwatch.
Overwatch är ett SaaS larmsystem för Nouryons kemikalie fabriker som gör det möjligt för fabriksarbetare att sätta upp så kallade “Cases” för alla värden som finns runt om i fabriken. När användaren sätter upp ett Case så får användaren flera olika alternativ för att konfigurera ett Case men för enkelhetens skull så väljer vi att använda “current value”.
I den här konfigurationen så ska användaren välja vilken tag (se förklaring 1.3 terminologi) han vill arbeta med. Efter detta så får användaren en möjlighet att sätta hög och låg setpoint.
När taggens värde når hög/låg setpoint så kommer användaren få ett sms i mobilen alternativt ett e-mail med vilket larm som gick av och vad taggens värde är på.
Se bilderna nedanför för att få en bättre förståelse av Overwatch och hur du sätter upp ett case på både nya och gamla Overwatch.

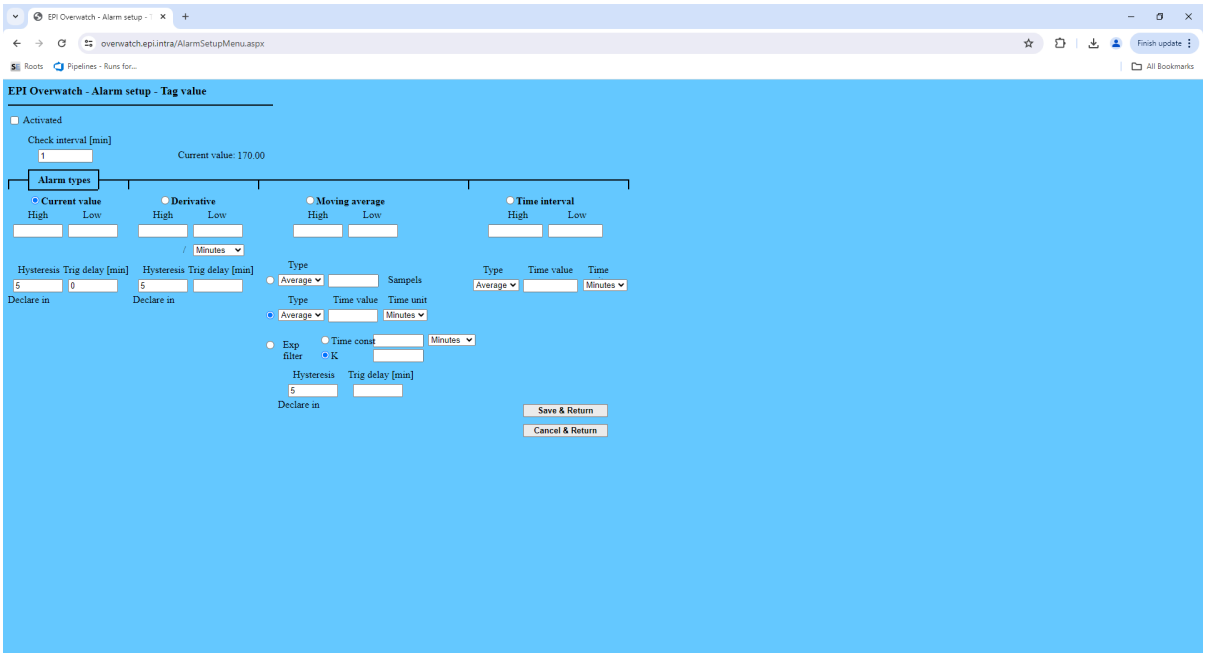
1.1.1 Original Overwatch



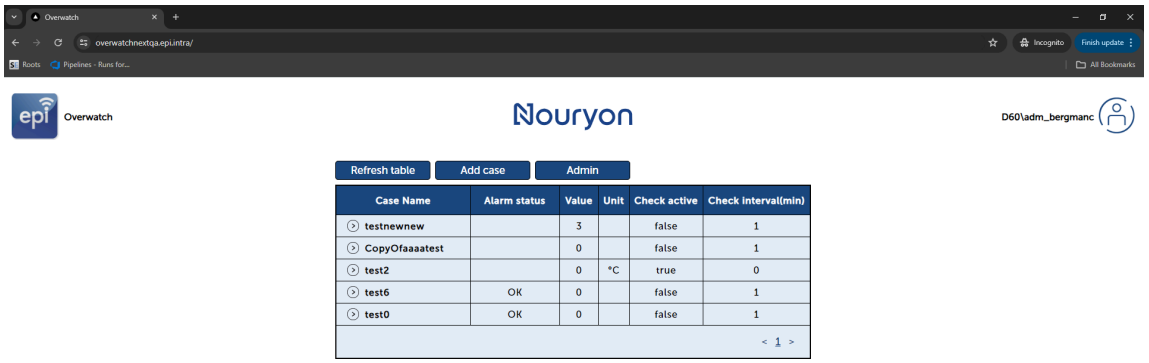


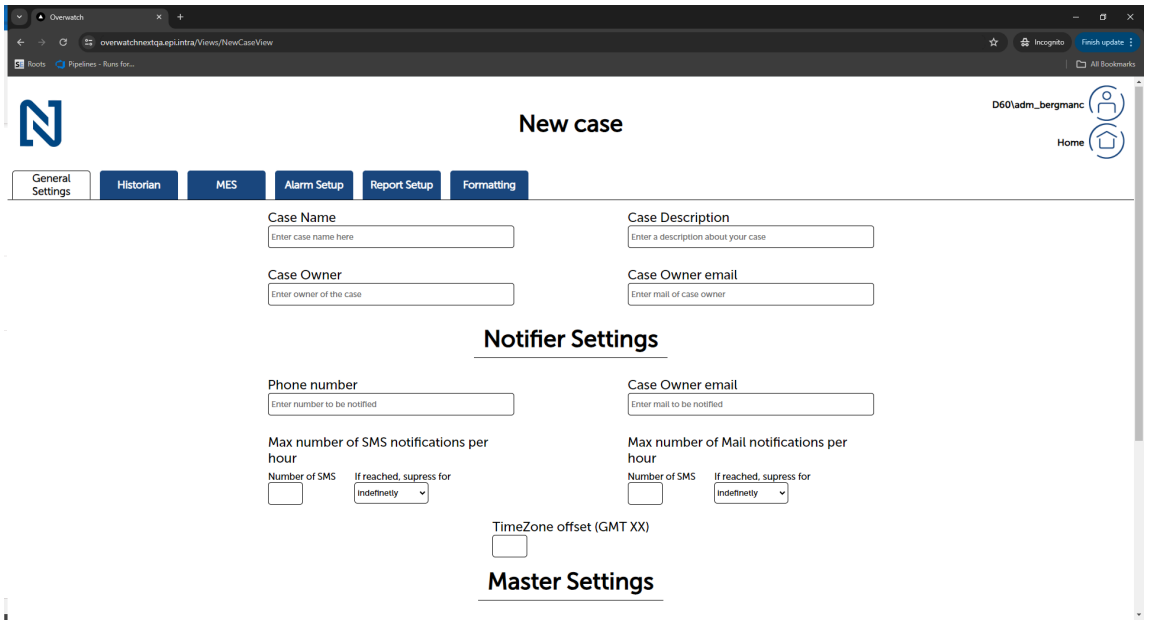


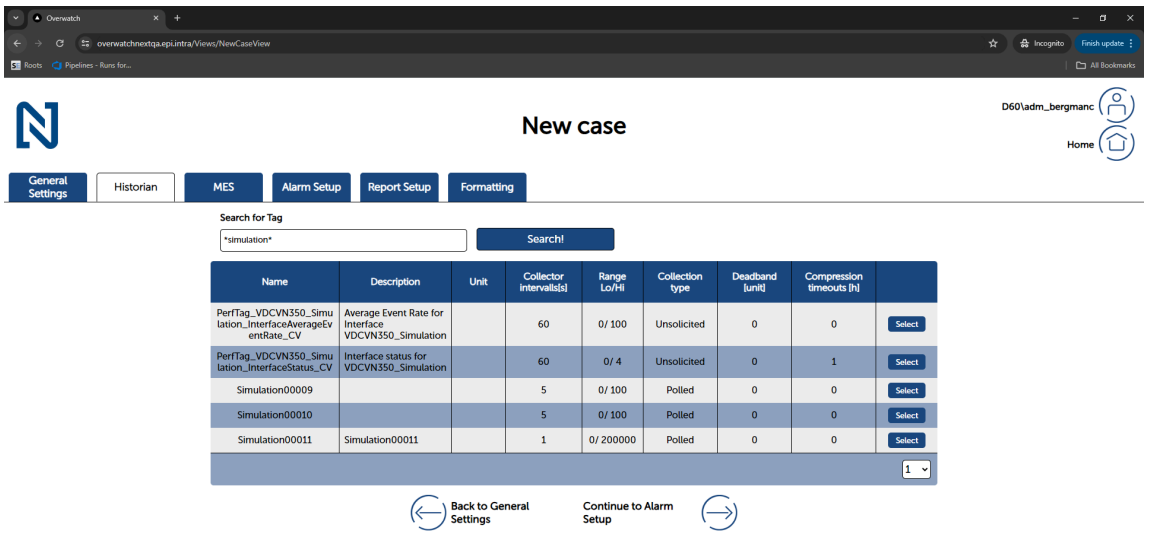

EPI Overwatch - Alarm setup menu
Set alarm for:
Tag Value
Signal condition
Cancel & Return

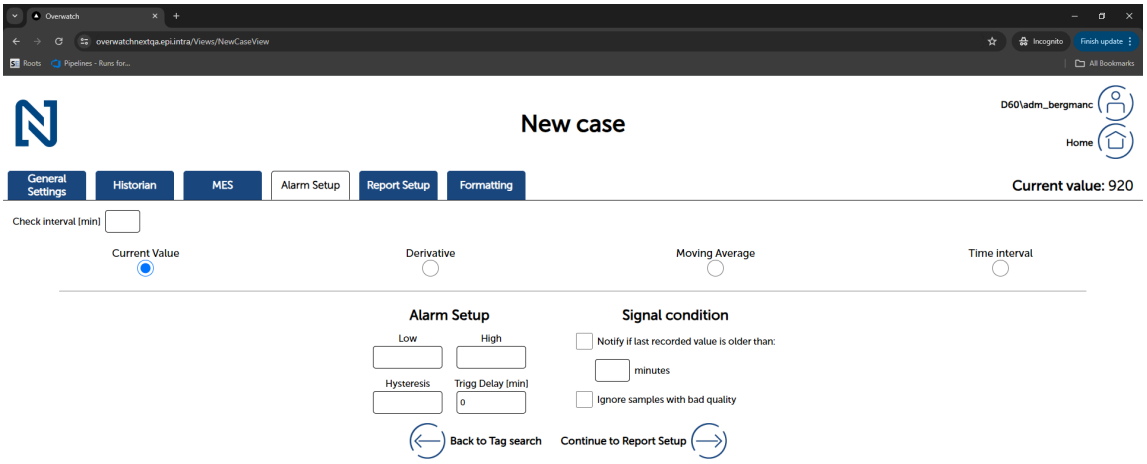

EPI Overwatch - Alarm setup - Tag value
Activated
Check interval [min]
Current value: 170.00
Alarm types
Current value
Derivative
Moving average
Time interval
High
Low
Minutes
Hysteresis
Trig delay [min]
Declare in
Type
Samples
Time value
Time unit
Time
Average
Exp filter
Time const
K
Save & Return
Cancel & Return

1.1.2 Nya Overwatch









1.2 Teknisk information

Overwatch första version släpptes 2007 och har efter detta blivit sparsamt uppdaterad, därav kan vi förvänta oss en del förlegade tekniker.
Overwatch byggdes först i VB.net 2.0 men fick en uppgradering till VB.net 4.0 2012 och efter det blivit sparsamt uppdaterat.
Applikationen är hostad på en av företagets Windows- servrar med hjälp av IIS.
Användarens data sparas i en filbaserad databas där datan representeras med hjälp av xml.

OverwatchNext byggdes i Nextjs 14.1 och kommer därav köras av nodejs. Backend Kod är orörd förutom några mindre tillägg i koden och lite konfigurations modifikationer.

1.3 Terminologi

I denna uppsats kommer jag använda x antal termer som är specifika till Overwatch och Nouryons system, men också en del andra tekniker som inte är “common knowledge".

Namn	Förklaring
Historian	Ett system som hämtar information från sensorer, kontroller och andra verktyg som finns ute i fabrikerna, lagrar informationen i en databas.
Tag	Varje sensor/kontroll/mätare har en tag kopplad till sig. Denna tag innehåller namn, värde, om sensorn är aktiv och om det är bra kvalitet på värdet.
Case	Case är vad användaren skapar när han sätter upp ett larm.
Settings	Information som användaren kan skriva in för att värden som tex email och username ska automatiskt skrivas in när användaren skapar ett nytt Case.

OW	Original applikationen Overwatch
OWN	Nya applikationen OverwatchNext
AD	Active directory, ett system utvecklat av microsoft för att ge personer och grupper tillgång till mappar/servrar/nätverk.
IIS	Internet information services, en Microsoft applikation som hjälper dig att hosta och konfigurera hemsidor
HistoricalData	En .net lösning med VB.net klassbibliotek för att hämta taggar, värden av taggar, kalkylationer av värden, hämta labb data, mm…
OverwatchWinService	En windows service som övervakar Cases, skickar ut mail/SMS när en setpoint är nådd och hanterar uppdateringar av användarens data.

2. Syfte och frågeställningar

- I denna rapport har jag som syfte att:
    - Visa upp problem som kan uppstå när du arbetar med legacy kod i ett slutet microsoft ekosystem
    - Visa upp de mindre vanliga tekniker som jag behövt implementera men även de som redan fanns i systemet.
- Utifrån detta syfte vill jag ha svar på följande tre frågor:
    - Vilka problem kan man stöta på när man arbetar med legacy kod
    - Vilka problem kan man stöta på när man implementerar node.js in i ett microsoft ekosystem
    - Var Next.JS en bra väg att gå för detta projekt

3. Metod

För att få en bättre förståelse av OW och teknikerna som används i applikationen så har jag tagit hjälp av kollegor som varit med och skapat/underhållit applikationen, läst i källkoden, dokumentation och microsofts officiella dokumentation har varit till stor hjälp eftersom OW är byggd för och med Microsofts toolkit.

Informationen om hur diverse tekniker/ramverk/språk/paket fungerar kommer ifrån respektive dokumentation, när det kommer till implementationen av paketen så kommer jag vid vissa tillfällen ta mig friheten att dra egna slutsatser av hur man använder paketen då de inte alltid fungerar som de ska med den tech stack och konfiguration jag under detta projekt använder.

Stack overflow och microsoft community har också varit en stor hjälp i att förstå mig på problemen som uppstått. Vill tillägga att Microsoft community är ett community där Microsoft anställda ger svaren och kan därav anses vara en pålitlig källa.
När det kommer till Stack Overflow så kommer jag inte ta in lösningar/svar som fakta om svaret inte är skrivet av en utvecklare som arbetar på tekniken som diskuteras.

4. Resultat

4.1 Starten

När vi började detta projekt visste vi inte vad vi skulle kunna förvänta oss att stöta på eftersom det var otroligt många tekniker som var nya för oss. Men vi insåg tidigt att vi kommer behöva skriva nya API:er för att ha möjligheten att arbeta med historian och spara cases. Därav valde vi NextJS som med sin nya app router har ett intuitivt routing system och ett smidigt sätt att skriva API:er utan att skapa ännu ett projekt.

Efter att ha startat ett NextJS projekt så var det dags att sätta upp detta på Development servern och se till att “bare minimum” fungerar.
Tidigare version av OW är hostad på IIS så vi väljer att hålla oss till företagets principer och hostar även vår applikation på IIS.

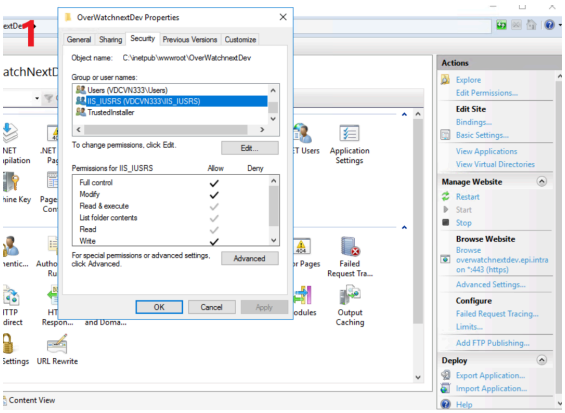
4.2 IIS

Internet Information Services är en web server för Windows-system som hanterar och visar begärda HTML sidor.
Eftersom IIS är skapat av Microsoft så fungerar asp.net nativt på IIS och ger möjlighet till att använda funktioner såsom autentisering och session-hantering. IIS passar särskilt bra att hosta en webbapp som är del av ett intranät på grund av dess möjligheter att hantera Active Directory. [5]

4.2.1 Konfigurering

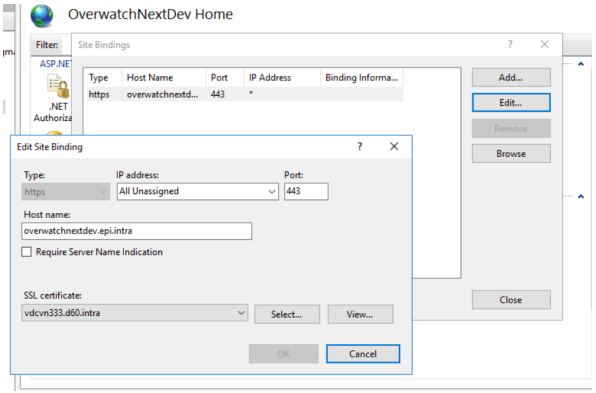
Jag börjar att sätta upp en IIS sida, konfigurera permissions(1), bindings(2), appPool(3) och Settings(4).

1.



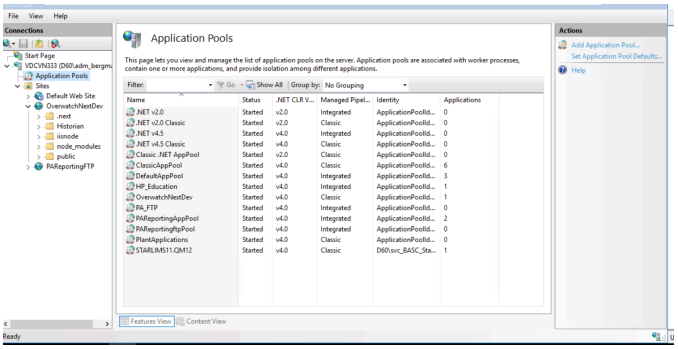
Att sätta rätt access till rätt grupper är viktigt för säkerheten då informationen som finns i detta system kan vara känslig.
I detta fall räcker det att ge projektet IIS_IUSRS gruppen som är en grupp skapad av IIS vilket ger projektet tillgång till all relevant information på servern.

2. Bindings



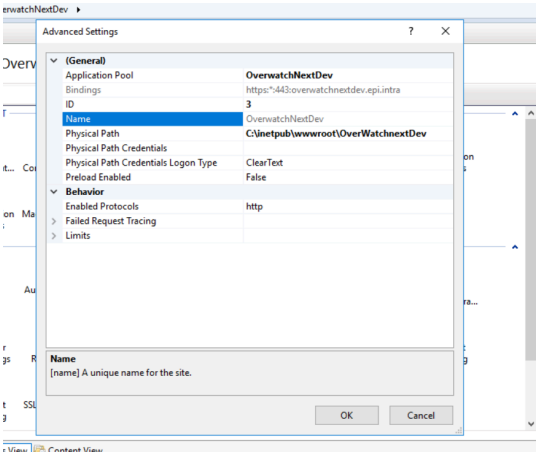
I bindings sätter vi host namn, protokoll, portnummer och SSL certifikat.
SSL certifikat används för att kryptera all data som överförs och skyddar användaren mot “man-in-the-middle” attacker. Eftersom OWN hittills endast hostas på en development server så använder jag ett generellt certifikat för den specifika servern.
Vi sätter portnummer till 443 vilket är standardporten för HTTPS.
Hostnamnet sätter vi för att IIS ska veta vilken URL den ska lyssna efter anrop mot.

3. Application Pool



Application Pool används för att isolera webbapplikationen och optimera användandet av serverns resurser (CPU/minne).
Vi sätter managed pipeline till classic eftersom integrated som är det senaste och att föredra inte fungerar med impersonate som vi kommer behöva senare för autentisering.
Identity behåller vi som ApplicationPoolIdentity eftersom denna inte kommer ha någon inverkan på vår webbapp när vi har konfigurerat Impersonate rätt.

4. Settings



Här deklarerar vi vilken Application Pool vi vill att OWN ska köra på och vilken map processen ska köra.

4.3 IISNode

Här stöter jag på det första problemet. Jag har satt upp ett NextJS projekt och laddat upp det till IIS men det funkar inte.
Tidigare har vi konstaterat att IIS är en Microsoft produkt och fungerar nativt .net kod. Detta betyder även att Node.js processer inte fungerar nativt.
Det betyder att vi behöver IISNode vilket är en IIS modul som fungerar som en brygga mellan våran node.js process och IIS. [3][4]

4.3.1 Konfigurering

Varje webbapp som hostas på IIS kräver en web.config fil där utvecklaren kan

deklarera bindings, endpoints, nycklar, begränsningar, autentisering, mm…
OW web.config sträckte sig upp till 260 rader men vi valde av flera anledningar att hålla config filen för OWN så grundlig som var möjligt då nodejs/IISNode hade stora problem att läsa specifika settings (mer om detta i edge-js).

4.3.1.1 server.js

```
const { createServer } = require("http");
const { parse } = require("url");
const next = require("next");

const dev = process.env.NODE_ENV !== "production";
const port = process.env.PORT || 3000;
const app = next({ dev });
const handle = app.getRequestHandler();

app.prepare().then(() => {
  createServer((req, res) => {
    var username = req.headers['x-iisnode-auth_user'];
    var authenticationType = req.headers['x-iisnode-auth_type'];
    // Be sure to pass `true` as the second argument to `url.parse`.
    // This tells it to parse the query portion of the URL.
    const parsedUrl = parse(req.url, true);
    const { pathname, query } = parsedUrl;

    if (pathname === "/a") {
      app.render(req, res, "/a", query);
    } else if (pathname === "/b") {
      app.render(req, res, "/b", query);
    } else {
      handle(req, res, parsedUrl);
    }
  }).listen(port, (err) => {
    if (err) throw err;
    console.log(`> Ready on http://localhost:${port}`);
  });
});
```

Eftersom node.js inte fungerar direkt med IIS så omdirigeras alla anrop mot OWN till server.js så att en node.js process kan starta. I denna filen deklareras även eventuella headers vi vill göra tillgängliga för applikationen. I detta fallet behöver OWN veta vem som loggar in på sidan och vilken typ av autentisering användaren använder.

4.3.1.2 web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <rule name="myapp">
          <match url="/*" />
          <action type="Rewrite" url="server.js" />
        </rule>
      </rules>
    </rewrite>
    <iisnode node_env="production" nodeProcessCommandLine="C:/inetpub/wwwroot/OverWatchnextDev/node.exe;"
    interceptor="&quot;%programfiles%\iisnode\interceptor.js&quot;" promoteServerVars="AUTH_USER,AUTH_TYPE"/>
  </system.webServer>

  <location path="" overrideMode="Deny">
    <system.webServer>
      <handlers>
        <add name="iisnode" path="server.js" verb="*" modules="iisnode" />
      </handlers>
    </system.webServer>
  </location>

  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
      <identity impersonate="true" />
  </system.web>
</configuration>
```

1. URLRewrite är en IIS modul som gör det möjligt att skriva om, förenkla och omdirigera URL:er. I detta fall omdirigerar vi alla anrop mot OWN till server.js så att IISNode kan hantera anropet. [6]
2. I iisnode elementet deklareras vilken node.exe som ska starta våran applikation, vilka header variabler som ska skapas (dessa översätter iisnode

till x-iisnode-auth_user/x-iisnode-auth_type som kan ses i server.js).

3. Eftersom alla http requests är omdirigerade till server.js är path satt till server.js för att lyssna efter anrop emot just server.js. När ett kommer ett anrop plockar denna "handler" upp det, skickar det till IISNode för hantering och sedan skickar det till node processen som skapar en respons som skickas tillbaka till IISNode som tillslut skickar responsen till webbservern (IIS) som sedan kan visa upp resultatet.
4. Konfigurationen för autentisering är relativt simpel. I detta fall tillåts alla med ett företagskonto att logga in på OWN. Impersonate är vad som gör att Application Pool identiteten ändras vilket ger användaren tillgång till rätt resurser med hjälp av företagets AD.

4.4 HistoricalData

Nu har vi en fungerande webbapp som kör på en node.js runtime och hostas på IIS. Nästa problem är att det inte finns några API:er för att hämta taggar eller värdet av taggarna.
I OW har detta lösts med hjälp av HistoricalData (se terminologi).
I OWN funkar inte detta då node.js inte kan hantera VB.net kod.

4.4.1 Edge

Edge är ett bibliotek skapat av Tomasz Tjanczuk (vidareutvecklas sedan 2017 av Agracio) som gör det möjligt att kompilera och köra C#/F#/Python direkt från din server komponent. Edge gör det även möjligt att köra kompilerad .net kod direkt i komponenten.
Det är just detta vi vill ha eftersom vi redan har färdiga bibliotek som vi ska använda. [2]

4.4.2 Hur fungerar Edge?

I teorin är Edge ett mycket simpelt bibliotek att använda där du helt enkelt pekar mot din komprimerade .net kod, specificerar den specifika funktionen du vill ha och kan sedan använda den i din kod.

```
var edge = require("edge-js");
import { headers } from "next/headers";
let functionFound = true;
let errorToDisplay;
try {
  // imports class that gets information about a selected tag
  var dotNetFunction = edge.func({
    assemblyFile: "./Historian/HistoricalDataClient.dll",
    typeName: "HistoricalDataClient.Proxies.HistoricalDataClientProxy",
    methodName: "GetCurrentValue",
  });
} catch (error) {
  console.log(error);
  errorToDisplay = error;
  functionFound = false;
}
```

I ovan exempel hämtar vi funktionen GetCurrentValue som ligger i projektet HistoricalDataClient, namespacet proxies och klassen HistoricalDataClientProxy som vi sedan ska kunna använda för att hämta värdet på önskad tagg.
Tyvärr var det inte riktigt så enkelt som vi önskade. För att API:et ska fungera så kräver det att funktionen är Async. .net4.0 som HistoricalData är skriven i stödjer inte asynkrona funktioner.
Att höja hela HistoricalData till en senare version skulle troligtvis vara ett för stort projekt så jag att istället starta ett nytt projekt med en senare version av .net (v4.7.2) i HistoricalData lösningen som wrappar funktionerna i en async funktion vilket gör det möjligt att köra funktionen async.

```
4 references
Public Function GetCurrentValue(ServerName As String, TagName As String) As HistoricalDataEntities.DataPoints.ExtendedDataPoint Implements IHistoricalData.GetCurrentValue
    Dim pServerName = ServerName
    Dim pTagName = TagName

    Return CallOperation(Function() InnerChannel.GetCurrentValue(pServerName, pTagName))
End Function
```

GetCurrentValue i VB.net4.0

```
Public Async Function GetCurrentValueAsync(payload As Object) As Task(Of Object)
    Dim currentValue = Await Task.Run(Function() _client.GetCurrentValue(payload.ServerName, payload.TagMask))

    Return currentValue
End Function
```

GetCurrentValue importerad och wrappad i en async funktion.

```
1  var edge = require("edge-js");
2  import { headers } from "next/headers";
3  let functionFound = true;
4  let errorToDisplay;
5  try {
6    // imports class that gets information about a selected tag
7    var dotNetFunction = edge.func({
8      assemblyFile: "./Historian/HistoricalAsync.dll",
9      typeName: "HistoricalAsync.ProxiesAsync.HistoricalDataClientProxyAsync",
10     methodName: "GetCurrentValueAsync",
11   });
12 } catch (error) {
13   console.log(error);
14   errorToDisplay = error;
15   functionFound = false;
16 }
```

Nu kan vi importera funktionen och köra den async.

```
47      // result1 has information about desired tag. See mock example
48      const result1 = await new Promise((resolve, reject) => {
49        // run imported dll, pass payload in first argument and funct
50        dotNetFunction(payload, function (error, result) {
51          if (error) {
52            reject(error);
53          } else {
54            console.log(result);
55            resolve(result);
56          }
57        });
58      });
59      return new Response(JSON.stringify(result1));
```

I första argumentet skickar vi med payload som i detta fallet innehåller Tag namnet och vilken server vi vill hämta datan ifrån, det andra argumentet är där vi kör funktionen och får tillbaka ett resultat som vi sedan kan skicka till anropet.

4.4.3 Konfiguration

HistoricalData hämtar data via SecureHistoricalDataService som samlar all tag-data och distribuerar det via en tcp connection.

```
<system.serviceModel>
  <bindings>
    <netTcpBinding>
      <binding name="NetTcpSecureBinding_IHistoricalData" closeTimeout="00:01:00" openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:02:00" transactionFlow="false" transferMode="Buffered"
      transactionProtocol="OleTransactions" hostNameComparisonMode="StrongWildcard" listenBacklog="10" maxBufferPoolSize="2147483647" maxBufferSize="2147483647" maxConnections="10" maxReceivedMessageSize=
      "2147483647">
        <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384" maxBytesPerRead="4096" maxNameTableCharCount="16384" />
        <reliableSession ordered="true" inactivityTimeout="00:10:00" enabled="true" />
        <security mode="Message">
          <transport clientCredentialType="Windows" protectionLevel="EncryptAndSign" />
          <message clientCredentialType="Windows" />
          <!--<message clientCredentialType="Windows"
              negotiateServiceCredential="true"
              algorithmSuite="Default"
              establishSecurityContext="true" /> -->
        </security>
      </binding>
    </netTcpBinding>
  </bindings>
  <client>
    <endpoint address="net.tcp://vdcvn338:8003/HistoricalDataProvider/Services/SecureHistoricalDataService" binding="netTcpBinding" bindingConfiguration="NetTcpSecureBinding_IHistoricalData" contract=
    "HistoricalDataServiceContract.IHistoricalData" behaviorConfiguration="standardBehaviour" name="NetTcpBinding_IHistoricalData">
   <identity>
      <servicePrincipalName value="HDP/vdcvn338" />
    </identity>
    </endpoint>
  </client>
  <behaviors>
    <endpointBehaviors>
      <behavior name="standardBehaviour">
        <dataContractSerializer maxItemsInObjectGraph="6553600" />
        <clientCredentials>
          <windows allowedImpersonationLevel="Delegation" />
          <!--<windows allowedImpersonationLevel="Impersonation" />-->
        </clientCredentials>
      </behavior>
      <behavior name="basicBehaviour">
        <dataContractSerializer maxItemsInObjectGraph="6553600" />
      </behavior>
    </endpointBehaviors>
  </behaviors>
</system.serviceModel>
```

Ovan har vi en del av OW:s web.config som deklarerar endpoint och tcp binding. TCP (Transmission Control Protocol) är ett protokoll som används för att ordna dataöverföring och anslutning mellan två enheter på ett nätverk. I vårt fall mellan OWN/HistoricalData och HistoricalDataProvider. [7]

Efter flera misslyckade försök att implementera konfigurationen i OWN:s web.config konstaterar jag att detta inte kommer fungera och är tvungen att ta en annan approach. Om det har och göra med att HistoricalData funktionerna körs på en node server och därav inte hittar config filen eller om det är något annat.

För att försäkra mig om att HistoricalData ska kunna hämta den data som krävs valde jag att skriva konfigurationen programmatiskt i HistoricalAsync klassen jag tidigare skapat. Nedanför kan ni se konfigurationen omformulerad till VB.net kod.

```
1 reference
Function CreateNetTcpBinding() As NetTcpBinding
    Dim binding As New NetTcpBinding With {
    .Name = "NetTcpSecureBinding_IHistoricalData",
    .CloseTimeout = TimeSpan.FromMinutes(1),
    .OpenTimeout = TimeSpan.FromMinutes(1),
    .SendTimeout = TimeSpan.FromMinutes(2),
    .TransactionFlow = False,
    .TransferMode = TransferMode.Buffered,
    .TransactionProtocol = TransactionProtocol.OleTransactions,
    .HostNameComparisonMode = HostNameComparisonMode.StrongWildcard,
    .ListenBacklog = 10,
    .MaxBufferPoolSize = 2147483647,
    .MaxBufferSize = 2147483647,
    .MaxConnections = 10,
    .MaxReceivedMessageSize = 2147483647
}
    binding.ReaderQuotas.MaxDepth = 32
    binding.ReaderQuotas.MaxStringContentLength = 8192
    binding.ReaderQuotas.MaxArrayLength = 16384
    binding.ReaderQuotas.MaxBytesPerRead = 4096
    binding.ReaderQuotas.MaxNameTableCharCount = 16384
    'binding.Security.Mode = MessageCredentialType.Windows
    binding.Security.Mode = SecurityMode.Message
    binding.Security.Transport.ProtectionLevel = ProtectionLevel.EncryptAndSign
    binding.Security.Transport.ClientCredentialType = TcpClientCredentialType.Windows
    binding.Security.Message.ClientCredentialType = TcpClientCredentialType.Windows
    binding.ReliableSession.Ordered = True
    binding.ReliableSession.Enabled = True
    binding.ReliableSession.InactivityTimeout = TimeSpan.FromHours(10)
    Return binding
End Function
1 reference
Function CreateEndpointAddress() As EndpointAddress
    Dim uri As New Uri("net.tcp://vdcvn338.D60.intra:8003/HistoricalDataProvider/Services/SecureHistoricalDataService")
    Dim identity As New SpnEndpointIdentity("HDP/vdcvn338")
    Dim epa As New EndpointAddress(uri, identity)
    Return epa
End Function

ReadOnly contractDesc As ContractDescription = ContractDescription.GetContract(GetType(IHistoricalData))
ReadOnly binding As NetTcpBinding = CreateNetTcpBinding()
ReadOnly endpointAddress As EndpointAddress = CreateEndpointAddress()
Dim _client As New HistoricalDataClientProxy(binding, endpointAddress)
```

När jag importerar HistoricalDataClientProxy klassen som har önskade funktioner så kör jag instansen av klassen med en ny konstruktor som sätter konfigurationen för HistoricalDataClient att kommunicera med SecureHistoricalDataService.

```
Public Class HistoricalDataClientProxy
    Inherits ProxyWrapper(Of HistoricalDataServiceContract.IHistoricalData)
    Implements HistoricalDataServiceContract.IHistoricalData

    2 references
    Public Sub New(binding As NetTcpBinding, endpointAdress As EndpointAddress)
        MyBase.New(binding, endpointAdress)
    End Sub
```

4.5 Cases

4.5.1 xml2js

Cases och Settings sparas som tidigare konstaterats i xml-filer. För att vi smidigt ska kunna arbeta smidigt med användarens data vill vi konvertera xml filen till ett js objekt.
xml2js är ett npm-paket som gör just detta.

4.5.2 Hämta xml

```
import { parseString as parseStringCallback } from "xml2js";
import { promisify } from "util";
import { readFileSync } from "fs";

const parseString = promisify(parseStringCallback);

export async function GET() {
  filePath =
    "C:/Users/bergmanc/OneDrive - Nouryon/Desktop/nextjs code/EXAMPLE SETTINGS CASES.xml";

  const fileContents = readFileSync(filePath);

  const jsonData = await parseString(fileContents, {
    explicitArray: false,
  });

  return Response.json(jsonData);
}
```

I bilden ovan är en nedskalad version av API:et som hanterar att hämta Cases/settings och konvertera xml-filen till ett js-objekt. Funkar i princip som tänkt "out of the box" förutom att vi vill lägga till "explicitArray: false" när vi konverterar datan, detta görs för att Cases ska hamna i en array.
Här fick vi ett väldigt otydligt error meddelande när vi hämtar vissa av användarnas filer som säger att det är något fel i filen.

```
Error: Error: Non-whitespace before first tag.
Line: 0
Column: 1
Char: ■
    at error (webpack-internal:///(rsc)/./node_modules/sax/lib/sax.js:620:14)
    at strictFail (webpack-internal:///(rsc)/./node_modules/sax/lib/sax.js:642:13)
    at beginWhiteSpace (webpack-internal:///(rsc)/./node_modules/sax/lib/sax.js:895:13)
    at SAXParser.write (webpack-internal:///(rsc)/./node_modules/sax/lib/sax.js:944:21)
    at Parser.parseString (webpack-internal:///(rsc)/./node_modules/xml2js/lib/parser.js:338:39)
    at Parser.eval [as parseString] (webpack-internal:///(rsc)/./node_modules/xml2js/lib/parser.js:6:23)
    at exports.parseString (webpack-internal:///(rsc)/./node_modules/xml2js/lib/parser.js:380:23)
    at node:internal/util:430:21
    at new Promise (<anonymous>)
    at node:internal/util:416:12
    at GET (webpack-internal:///(rsc)/./src/app/api/getUserXml/route.js:56:40)
    at async C:\Users\bergmanc\OneDrive - Nouryon\Desktop\OWEM\OWEM\overwatchnext\node_modules\next\dist\compiled\next-server\app-route.runtime.dev.js:6:63809
    at async eU.execute (C:\Users\bergmanc\OneDrive - Nouryon\Desktop\OWEM\OWEM\overwatchnext\node_modules\next\dist\compiled\next-server\app-route.runtime.dev.js:6:53964)
    at async eU.handle (C:\Users\bergmanc\OneDrive - Nouryon\Desktop\OWEM\OWEM\overwatchnext\node_modules\next\dist\compiled\next-server\app-route.runtime.dev.js:6:65062)
    at async doRender (C:\Users\bergmanc\OneDrive - Nouryon\Desktop\OWEM\OWEM\overwatchnext\node_modules\next\dist\server\base-server.js:1333:42)
    at async cacheEntry.responseCache.get.routeKind (C:\Users\bergmanc\OneDrive - Nouryon\Desktop\OWEM\OWEM\overwatchnext\node_modules\next\dist\server\base-server.js:1555:28)
```

När vi sparar användarnas xml fil från OWN så sparas de med UTF-16 encoding medan OW sparar datan som UTF-8. Detta ställer till problem för konverteraren då den har UTF-8 som default.

```
try {
  const fileContents = readFileSync(filePath);

  const jsonData = await parseString(fileContents, {
    explicitArray: false,
  });

  return Response.json(jsonData);
} catch (err) {
  // if xml file is not UTF-8 we try for UTF-16
  try {
    const fileContents = readFileSync(filePath, "utf16le");
    // converts xml to json
    const jsonData = await parseString(fileContents, {
      explicitArray: false,
    });
    return Response.json(jsonData);
  } catch (error) {
    console.error("Error:", error);
    return Response.json(error);
  }
}
```

För att testa efter UTF-16 så deklarerar vi att filen kommer att vara UTF-16 när vi hämtar den. [1]

4.5.3 Uppdatera xml

I OW så hanterades uppdateringen av xml-filen med hjälp av ett xml diffgram som skickas till en mapp som övervakas av OverwatchWinService. När diffgrammet hamnar i den övervakade mappen plockar WinServicen upp filen, jämför den med existerande xml-fil och lägger till den nya informationen.
Ett Diffgram är en xml-fil som beskriver skillnaden mellan två xml-filer. I vårt fall skapar vi Diffgrammet genom att jämföra hur användarens data såg ut när vi började att redigera och hur den ser ut när vi är klara. Båda versionerna skickas till en .net funktion som gör om xml:en till två separata dataset som jämförs och skapar ett diffgram som sedan konverteras till ett xml-Diffgraml som OverwatchWinService kan läsa och implementera i databasen.

```
public class createDiffGram
{
    0 references
    public async Task<object> Invoke(dynamic input)
    {
        var orgXml = (string)input.orgXml;
        var newXml = (string)input.newXml;
        var diffGramPath = (string)input.outputPath;
        try
        {
            var orgDataSet = new DataSet();
            orgDataSet.ReadXmlSchema("./Historian/DataSetSchema.xsm");

            using (var reader = new StringReader(orgXml))
            {
                orgDataSet.ReadXml(reader);
                orgDataSet.AcceptChanges();
            }

            var newDataSet = new DataSet();
            newDataSet.ReadXmlSchema("./Historian/DataSetSchema.xsm");
            using (var reader = new StringReader(newXml))
            {
                newDataSet.ReadXml(reader);
                newDataSet.AcceptChanges();
            }
            newDataSet.Merge(orgDataSet, true);
            newDataSet.WriteXml(diffGramPath, XmlWriteMode.DiffGram);

            return diffGramPath;
        }
        catch (Exception ex)
        {
            return new
            {
                error = true,
                message = ex.Message,
                stackTrace = ex.StackTrace
            };
        }
    }
}
```

Innan vi skapar Diffgrammet så jämför vi även våra dataset med ett schema som deklarerar vilka värden som måste vara med och vilken datatyp värden ska ha.

5. Slutsatser och diskussion

5.1 Varför Edge?

I avsnitt 4.4.1 Edge förklarade jag att jag använder Edge för att köra .net kod i node. Men var detta verkligen bästa alternativet?
I vårt fall fanns det två alternativ, att sätta upp ett nytt projekt i HistoricalData lösningen där vi skriver våra API:er i .net för att hämta vår data eller alternativet jag valde att just använda Edge för att köra vår .net kod.

Hade jag valt att skriva API:er i .net hade jag inte behövt wrappa HistoricalData funktionerna i async (som jag förklara i avsnitt 4.4.2)funktioner och jag hade kunnat skriva konfigurationen med xml vilket följer företagets och projektets linje bättre.
Fördelarna med att jag valde att gå vägen med node.js API:er/Edge och integrera det i OWN är att jag nu har en mindre applikation som ska underhållas och köras.
Så var node/Edge rätt väg att gå? Med kunskaperna jag har nu efter att ha implementerat det skulle jag kunna argumentera både för och emot men det som tippar över det till ett "ja" är att HistoricalData är också på gång att få en uppdatering så därav var det inte lönt att rota för mycket i den lösningen.

5.2 Node.exe

I avsnitt 4.3 IISNode kan man se att node.exe ligger i samma mapp som OWN-projektet. Varför inte i "programs" där den brukar vara?
Nouryon har en till applikation som kör på node (v16.1) men som använder en äldre version. Att uppdatera från v16.1 till v20.13.1 skulle högst troligt ställa till problem för den redan existerande produkten.
Första tanken var att använda nvm (node version manager) som gör det möjligt att köra två olika versioner av Node samtidigt på en och samma server.
Efter flera dagars testande konstaterade jag att detta kommer fungera perfekt. Båda applikationer kan stänga av/på, uppdateras, raderas och installeras utan att ställa till problem för varandra.
Men så kom konfigurationen för HistoricalData (som förklaras i avsnitt 4.4.3). Enligt Tomasz Janczuks svar i stack overflow tråden [8] så ska det vara fullt möjligt för Edge att dra nytta av en config fil. Filen ska heta node.exe.config och ligga bredvid node.exe. Men jag vill inte ha en config fil liggandes bredvid en node.exe som vilken applikation som helst på servern kan och kommer läsa av när node.exe kör.
Alternativet som återstår är att ta en kopia av node.exe och köra den direkt från projekt mappen.
Detta löste inte problemet med konfigurationen men processen blir isolerad och gör det möjligt för servern att köra två olika node versioner samtidigt utan nvm.

5.3 Filbaserad databas

I avsnitt 4.5 Cases så förklarar jag för processen hur OWN:s data konverteras från/till dataset och XML och sedan sparas i databasen. Denna processen kan ses som krånglig, hade det inte varit bättre att sätta upp en SQL databas istället?
Ja, det hade det varit. Detta var en diskussion vi (jag och stakeholders) hade innan projektet startades och där vi konstaterade att det skulle ta för lång tid att bygga om databasen och backenden som hanterar datan.
Finns det några fördelar med en filbaserad databas?

Den är enkel att förstå och navigera runt inom och simpel för en otränad att sätta upp.

5.4 Var Next.JS en bra väg att gå?

Så infrastrukturen på företaget är uppbyggd just nu med IIS som hostar applikationer, filbaserad databas och en backend som är byggd för en .net frontend så skulle jag säga nej. I detta fallet skulle det vara simplare och mer naturligt att lyfta .net versionen på OW och bygga om gränssnittet för att vara mer användarvänligt och intuitivt.
Men just nu jobbar företaget för en övergång till Microsoft Azure och dess molnlösningar. Jag anser att node.js är vägen att gå framåt när det kommer till webbappar och med Azure App Services kommer många av problemen vi stötte på med att hosta OWN på IIS att försvinna.
I början av en så pass stor process det är att konvertera ett stort företag till molnet så måste man börja någonstans och implementera tekniker som kommer stå sig bra i många år med de nya tekniker som kommer.
Så var Next.JS en bra väg att gå?
Troligtvis inte. .net är Microsofts egenproducerade ramverk och kommer troligtvis alltid ha en fördel inom Microsofts ekosystem. Men det får tiden utvisa.

6.Källhänvisning

1. Leonidas-from-XIV (2023) Dokumentation av npm paketet xml2js
https://www.npmjs.com/package/xml2js

2. Agracio (2024) Dokumentation av biblioteket Edge
https://github.com/agracio/edge-js

3. Tomasz Tjanczuk (2022) Dokumentation av IISNode
https://github.com/Azure/iisnode

4. Tomasz Tjanczuk (2011) Hosting node.js applications in IIS on Windows
https://tomasz.janczuk.org/2011/08/hosting-nodejs-applications-in-iis-on.html

5. https://www.techtarget.com/searchwindowsserver/definition/IIS

6. URLRewrite
https://www.iis.net/downloads/microsoft/url-rewrite

7.
https://internetstiftelsen.se/guide/introduktion-till-ip-internet-protocol/tcp-och-udp-nivan/

8. Node.exe.config
https://stackoverflow.com/questions/34955798/in-iisnode-make-edge-js-pull-connection-string-from-web-config