Informatik

# BIF-SWE 1
# Embedded Sensor Cloud Projekt

## Kompensationsbericht

**Autor**

Johannes Bauer (if18b152)

# Plugin Manager

Der Plugin Manager erstellt die Plugins mithilfe von .dll Dateien, die aus einem statischen Verzeichnis in der Projektstruktur geladen werden.

Zuerst werden alle .dll Files im Pfad gesucht die, ein bestimmtes Plugin implementieren. Die Plugins werden dann als IEnumerable gespeichert und mithilfe eines Assemblys erstellt.

```
public void LoadPluginsFromPath()
{
    try
    {
        if (Directory.Exists(PluginPath))
        {
            string[] files = Directory.GetFiles(PluginPath, "*.dll");
            // selectMany() -> create a single sewuence from a sequence in which all of the elements are seperate
            IEnumerable<IPlugin> allPlugins = files.SelectMany(singlePath =>
            {
                Assembly assemblyPlugins = LoadPlugin(singlePath);
                return CreateAllPlugins(assemblyPlugins);
            }).ToList();

            Plugins = allPlugins;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
}
```

Mit GetPluginFromPath können die Plugins geladen werden.

```
public IPlugin GetPluginFromPath(string pluginName)
{
    // create plugin path string
    string pluginPath = PluginPath + "/" + pluginName + ".dll";
    // create assembly for plugin
    Assembly pluginAssembly = LoadPlugin(pluginPath);

    if(pluginAssembly != null)
    {
        List<IPlugin> assemblyPlugins = CreateAllPlugins(pluginAssembly).ToList();
        return assemblyPlugins.First(); // OrDefault(); -> exceptions: do not return null!
    }
    else
    {
        return null;
    }
}
```

# StaticFilePlugin

Mithilfe des StaticFilePlugins können Dateien, von einem Ordner aus, vom Webserver eingelesen und dem Benutzer im Browser angezeigt werden. Unterstützte Dateiformate sind zum Beispiel html, css, js, txt und json. Das Plugin umfasst die Methoden CanHandle, Handle sowie einen String, der den Namen des Plugins beinhaltet.

Die wichtigste Methode des Plugins ist Handle. Sie ist folgendermaßen aufgebaut:

```
/// <summary>
/// Handles the request, returns the static file
/// </summary>
/// <param name="req"></param>
/// <returns>Valid response with response code...</returns>
34 Verweise | 16/25 bestanden
public IResponse Handle(IRequest request)
{
    string filePath = request.Url.Path;
    Response response = new Response();

    if(request.Url.RawUrl == "/")
    {
        filePath = "index.html";
    }
    if (File.Exists("./static-files/" + filePath))
    {
        Console.WriteLine("./static-files/" + filePath);
        response.StatusCode = 200;
        response.SetContent(File.OpenRead("./static-files/" + filePath));
        string fileExtension = Path.GetExtension("./static-files/" + filePath).Trim('.');
        Console.WriteLine("File Ext: " + fileExtension);
        Console.WriteLine("File Path: " + "./static-files/" + filePath);
        response.ContentType = response.KnownFileExtensions[fileExtension] ?? "text/plain";
    }
    else
    {
        response.StatusCode = 404;
        response.SetContent(response.Status);
    }
    return response;
}
```

Das Plugin bekommt eine Request als Parameter mitgeliefert. Aus dieser wird die Url extrahiert, um den Dateipfad des gewünschten Dokuments herauszulesen. Danach wird im static-files Ordner nach der Datei gesucht und die File Extension extrahiert. Die Response gleich noch ab ob der Dateityp unterstützt wird (die Dateitypen sind in einem Dictionary in der Response Klasse gespeichert). Danach wird die Datei mithilfe der Response Klasse und dem Statuscode 200 zurückgeliefert.

Existiert die Datei nicht wird der Statuscode 404 zurückgegeben.

# ToLowerPlugin

Mithilfe des ToLowerPlugin können Strings verarbeitet werden. Das Ziel ist es den übergebenen String in Kleinbuchstaben zurückzugeben. Um das ToLowerPlugin verwenden zu können muss auch das StaticFilePlugin verwendet werden, da die Weboberfläche nur so geladen werden kann.

Die Handle Methode des Plugins:

```
/// <summary>
/// Handles the request, returns lowerCase string
/// </summary>
/// <param name="req"></param>
/// <returns>Valid response with response code...</returns>
34 Verweise | 16/25 bestanden
public IResponse Handle(IRequest req)
{
    if (req == null)
    {
        return new Response { StatusCode = 404 };
    }

    if(req.IsValid && req.ContentString == "text=")
    {
        Response response = new Response { StatusCode = 200 };
        response.SetContent("Bitte geben Sie einen Text ein");
        return response;
    }

    // returns lower case request content string
    else if (req.IsValid && req.Url.Segments.Length == 1 && req.Url.Segments[0].ToLower() == "tolower" && req.Method == "POST")
    {
        Response response = new Response { StatusCode = 200 };
        response.SetContent(req.ContentString.ToLower());
        return response;
    }
    else
    {
        return new Response { StatusCode = 404 };
    }
}
```

Zuerst wird überprüft ob ein String mit der Request mitgegeben wurde. Ist dies nicht der Fall, wird eine Response mit dem Code 404 zurückgegeben. Ist die Request valide, dann wird der Content String mit der Methode ToLower() einfach in einen String in Kleinbuchstaben verwandelt. Der String wird danach mithilfe der Response Klasse an den Benutzer zurückgegeben und auf einer HTML-Seite ausgegeben.

# NaviPlugin

Mithilfe des Navi Plugins kann ein Benutzer nach einer Straße suchen und alle Orte zurückzubekommen, in denen es die gesuchte Straße gibt.
Die Handle Funktion kann zwei Arten von Requests entgegennehmen:

1. **Navi**: Der Request body ist mit einem Straßennamen gefüllt. Die Handle Funktion ruft GetNavi auf, um eine Stadt zurückzuliefern.
2. **Reload**: Der Request stellt eine Anfrage an den Server, die den Reload der Map Datei triggern soll. Ist der Server gerade dabei die Map Datei einzuladen, antwortet der Server mit dem Status Code 503. Dies verhindert, dass der Server nicht mit unzähligen Requests überlastet werden kann. Ebenfalls wird zu Beginn des Ladevorgangs ein neuer Thread erstellt.

Kann eine Request verarbeitet werden, antwortet der Server mit dem Status Code 200. Wird eine ungültige Anfrage gestellt hat die Response den Status Code 404.

Das Laden der Map Datei erfolgt in drei verschiedenen Funktionen. Als erstes wird LoadMapData aufgerufen. Diese Funktion lädt das gewünschte xml File aus dem Dateisystem und liest es aus.

```
while (xml.Read())
{
    // check for a xml node named osm
    if(xml.NodeType == XmlNodeType.Element && xml.Name == "osm")
    {
        ReadOsmTree(xml);
    }
}
```

ReadOsmTree liest danach einen osm Subtree aus dem xml File. Werden beim Lesevorgang die gewünschten Tags gefunden, wird ReadAnyOsmElement getriggert. Diese Methode liest die Adress-Parameter im osm Tree aus und speichert dies in einer Instanz der Address Klasse, welche nur zum Speichern der Informationen dient.

```
public void ReadOsmTree(XmlReader xml)
{
    var osm = xml.ReadSubtree();
    while (osm.Read())
    {
        if (osm.NodeType == XmlNodeType.Element && (osm.Name == "node" || osm.Name == "way"))
        {
            ReadAnyOsmElement(osm);
        }
    }
}
```

Im letzten Schritt werden die gefundenen Adressen in einem Dictionary gespeichert. Hier werden Straßennamen gespeichert und mit den dazugehörigen Städten in Verbindung gesetzt.

```
public void ReadAnyOsmElement(XmlReader osm)
{
    //ReadAnyOsmElement
    Address address = new Address();
    var element = osm.ReadSubtree();

    while (element.Read())
    {
        if (element.NodeType == XmlNodeType.Element && element.Name == "tag")
        {
            //Read the tags
            string tagType = element.GetAttribute("k");
            string value = element.GetAttribute("v");

            switch (tagType)
            {
                case "addr:city":
                    address.City = value;
                    break;
                case "addr:postcode":
                    address.Zip = value;
                    break;
                case "addr:street":
                    address.Street = value;
                    break;
            }
        }

    }
```

# Server Klasse

Die Server Klasse öffnet Verbindungen über Sockets und verwendet Multithreading, um die verschiedenen Requests abzuarbeiten. Dies wird von der Listen Methode aus gestartet:

```
public void Listen()
{
    Console.WriteLine("Starting Server...");
    TcpListener tcpListener = new TcpListener(IPAddress.Any, Port);
    Console.WriteLine("Listening on Port " + Port);
    tcpListener.Start();

    while(true)
    {
        Socket s = tcpListener.AcceptSocket();
        Thread thread = new Thread(()=> ProcessRequest(s));
        thread.Start();
    }
}
```

In der ProcessRequest Methode werden die Anfragen an den Server dann verarbeitet. Hier wird zuerst ermittelt welches Plugin verwendet werden soll.

```
private void ProcessRequest(Socket s)
{
    Stream stream = new NetworkStream(s);
    Request request = new Request(stream);
    Response response = null;
    IPlugin selectedPlugin = null;
    float maxScore = 0.0f;
    string message = "";

    foreach (var plugin in PluginManager.Plugins)
    {
        var score = plugin.CanHandle(request);
        if (score > maxScore)
        {
            maxScore = score;
            selectedPlugin = plugin;
        }
    }
```

# Lessons Learned/Probleme

- Unit Tests
  - Die Unit Tests geben zwar eine gute Richtung vor, jedoch schränken sie auch ein wenig in der Implementierung des Webservers ein
  - Eigene Unit Tests zu finden war nicht einfach, da schon viel durch die vorgegebenen Unit Tests getestet wurde
  - Anfangs wurden nur die Testfälle abgearbeitet, dadurch sieht man relativ wenig Fortschritt
- Umfang und Schwierigkeitsgrad
  - Bei einem Umfangreicheren Projekt zahlt es sich aus rechtzeitig zu beginnen
  - Da bei dem Projekt für mich unbekannte Aufgaben verlangt waren habe ich mir nicht so leicht getan die Implementierung des Servers vorzunehmen
  - Viele kleinere Fehler haben mich teilweise frustriert, hier zahlt es sich aus einen Kollegen über den Code schauen zu lassen → vier Augen sehen mehr als zwei
  - Ein größeres Problem war, dass der Networkstream sich nach jeder Anfrage geschlossen hat, dadurch konnte immer nur ein File vom Server mit dem StaticFilePlugin geladen werden. Somit konnte ich zuerst kein JavaScript und keine Stylesheets verwenden.
- Browserkommunikation
  - Debuggen schwer
  - Wie müssen die Dateien übergeben werden?
- .osm Files
  - Die großen .osm Files brauchen sehr lange zum Einlesen, dadurch braucht man zum Testen länger, da man in jedem Testdurchlauf warten muss, bis die Datei eingelesen ist
  - Ein Fehler im Code beim Einlesen des Files hat viel Zeit gekostet → debuggen schwer
- Positives
  - Ich habe im Verlauf des Projekts viel dazugelernt, vergleichbare Projekte habe ich bisher noch nie gemacht.
  - Ich habe neue Tools, wie zum Beispiel Doxygen kennengelernt.
  - Das Programmieren an sich hat Spaß gemacht und ich bin mit dem Endergebnis zufrieden.