



netidee

PROJEKTE

SIKOSI

Entwicklerdokumentation

Dokumentation | Call 14 | Projekt ID 4533

Lizenz: CC-BY-SA

Inhalt

1	Einleitung	3
1.1	Allgemeines/Struktur und Aufbau	3
1.2	Source-Code	3
2	SDKs	4
2.1	SIKOSI Auth – Authentication and Authorization	4
2.2	SIKOSI SRP.....	4
2.3	SIKOSI Secure Services.....	5
2.4	SIKOSI CRYPTO	8
2.5	SIKOSI Datenbank Services.....	11
3	Beispielprojekte.....	12
3.1	Sample 01 – Auth.....	12
3.2	Sample 02 – Secure Remote Password.....	13
3.3	Sample 03 – Crypto Server	14
3.4	Sample 04 – SignalR	14
3.5	Sample 05 – Multi Factor Authentication.....	15
3.6	Sample 06 – Beispiel für fehlerhafte Implementierung.....	16
3.7	Sample 07 – Verschlüsselter Chat und Speicherung verschlüsselter Dateien	20
3.8	Sample 08 – IOT.....	21
4	Demo-Applikationen	21
	Sample 01 – WebApplikation	21
	Sample 02 – Smartphone Applikation für Android und iOS	22
5	Sichere Datenhaltung.....	24
5.1	Konfiguration von Always Encrypted ohne Secure Enclaves	24
5.2	Verschlüsselungstypen	24
6	HOW-TO	26
6.1	SRP SDK EINBINDEN.....	26
6.2	Externe Logins.....	34

1 Einleitung

Diese Entwicklerdokumentation gibt einen Überblick über die im Rahmen des Netidee-Projektes „SIKOSI“ entwickelten Software-Bibliotheken sowie Demoprojekte, die sich mit Entwicklung sicherer Softwaresysteme auseinandersetzt.

Dies SDKs und Demoprojekte sollen Software-Entwickler dabei helfen sichere Systeme zu schaffen.

1.1 Allgemeines/Struktur und Aufbau

Es wurden eine Vielzahl an Software-Projekten entwickelt (siehe Abbildung 1) die in den folgenden Kapiteln näher beschrieben werden und zukünftige Entwicklungen als Ausgangsbasis und Nachschlagewerk dienen.

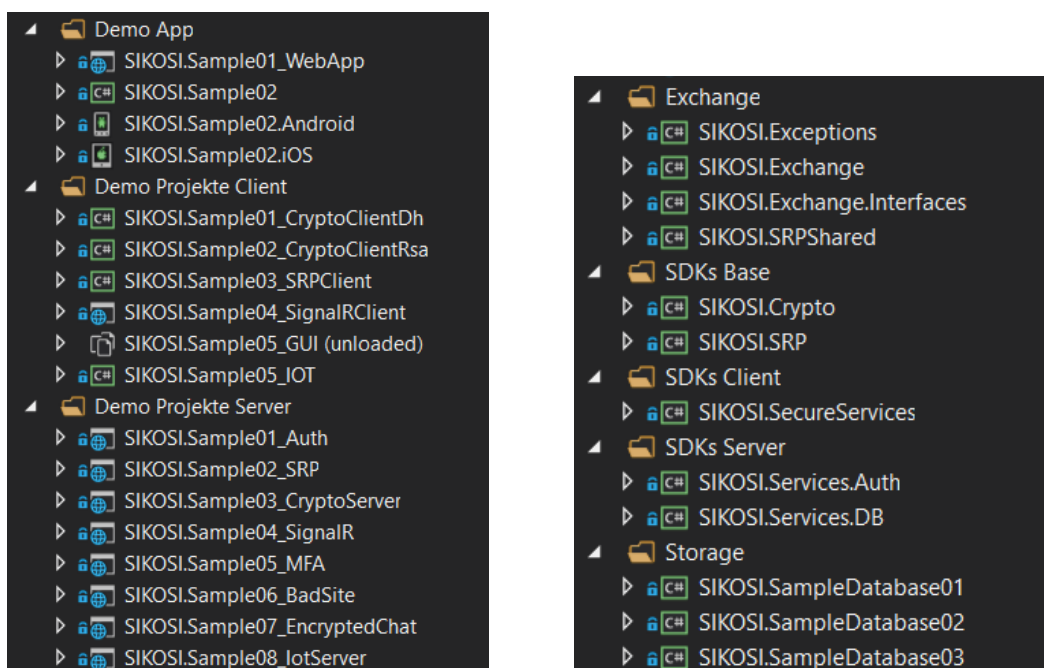


Abbildung 1: Projekte

1.2 Source-Code

Der Source-Code befindet sich unter GitHub

<https://github.com/FotecGmbH/SIKOSI>

Der Source-Code ist dokumentiert, detaillierte Informationen über einzelne Funktionen kann somit direkt im Source-Code nachgelesen werden.

2 SDKs

Folgend die entwickelten SDKs für eine sichere Kommunikation.

2.1 SIKOSI Auth – Authentication and Authorization

Authentifizierung bedeutet, der gegenüberliegenden Stelle (Server) mitzuteilen wer der Benutzer der Software ist und dies auch zu beweisen. Dies kann im einfachsten Fall die Übermittlung eines Benutzernamens und eines Passwortes sein, dass im Vorfeld bereits durch eine Registrierung bekannt gegeben wurde. So weiß der Server Bescheid, welcher Benutzer auf das System über die Schnittstellen zugreifen möchte. Damit der Server bei jedem Aufruf einer Schnittstellen Bescheid weiß, welcher Benutzer diesen Aufruf durchführt, stellt dieser bei der Authentifizierung einen Token zu Verfügung über diesen der Client eindeutig identifizierbar ist. Dieser Token hat eine begrenzte Lebensdauer und muss vom Client sicher gespeichert werden. Bei jedem Schnittstellen-Aufruf die der Client durchführt, muss dieser Token mit angegeben werden.

Autorisierung bedeutet, dass Schnittstellen am Server geschützt sind und nur von berechtigten Benutzern aufgerufen werden können. Durch die Angabe des Tokens kann der Server bestimmen welcher User den Aufruf getätigt hat, ob dieser berechtigt ist und somit entscheiden welche Daten für den Client bestimmt sind und welche nicht.

SDK SIKOSI.Services.Auth

Diese Bibliothek dient dazu Benutzer

- **Sicher Anzulegen** (mit Passwort-Hash und Salt!)
- Zu **Authentifizieren**
- Zu löschen
- Zu Aktualisieren

Die Bibliothek wird in folgenden Beispielprojekten verwendet

- **SIKOSI.Sample01_Auth**

Technologie: .NET CORE

2.2 SIKOSI SRP

Dieses SDK implementiert das als Secure Remote Password (SRP) bekannte Protokoll, eine Art PAKE-Protokoll.

Die PAKE-Protokollfamilie ist eine Familie von kryptografischen Protokollen, mit denen ein kryptografischer Schlüssel auf diese Weise generiert werden kann.

Ein passiver Angreifer, d.h. ein Angreifer der nur zuhört ohne selbst Daten zu senden, ist nicht in der Lage genügend Informationen zu extrahieren um den resultierenden Schlüssel zu berechnen oder effektiv zu erraten.

SDK SIKOSI.SRP

Die SDK besteht aus den folgenden vier Klassen

- SrpClient.cs
- SrpServer.cs
- SRPGroup.cs
- ExtensionsAndHelpers.cs

Der Client und der Server repräsentieren, wie die Namen andeuten, die Client- und Serverteile der Protokollimplementierung.

Um sowohl den Client als auch den Server zu instanzieren, muss eine SRP-Gruppe angegeben werden.

Die Extensions-Klasse bietet Erweiterungsmethoden für die System.Numerics.BigInteger-Klasse, da dieses SDK die BigInteger-Klasse verwendet. Des Weiteren bietet es auch eine Berechnungslogik, die sowohl vom Server als auch vom Client benötigt wird, um verschiedene Schlüssel und Werte zu generieren.

Die SRP-Gruppe ist Teil des SRP-Protokolls und enthält einen Generator und einen Wert N. Es wird sowohl vom Client als auch vom Server für verschiedene Berechnungen verwendet.

Die Bibliothek wird in folgenden Beispielprojekten verwendet

- **SIKOSI.Sample02_SRP**

Technologie: .NET CORE

2.3 SIKOSI Secure Services

Diese Bibliothek bietet Client-Services für eine vereinfachte verschlüsselte Kommunikation zwischen Client und Server. Speziell gibt es auch einen Login-Service sowie einen Registrierungs-Service, der auf dieser verschlüsselten Kommunikation aufbaut.

EncryptedCommunicationService

Diese Klasse ist die Basis für die verschlüsselte Kommunikation zwischen Client und Server. Vom Client aus können hier verschiedene verschlüsselte Requests an den Server versandt werden und die jeweilige Antwort des Servers verarbeitet werden.

Vor Verwendung der Methoden dieser Klasse müssen die beiden Properties „ReceiverPublicKey“ und „Encryption“ initialisiert werden. „ReceiverPublicKey“ stellt den öffentlichen Schlüssel des Empfängers (Servers) dar und muss bekannt sein. „Encryption“ stellt die asymmetrische Verschlüsselung (siehe Kapitel SIKOSI Crypto) dar, die verwendet werden soll. Sie beinhaltet unter anderem den eigenen öffentlichen Schlüssel und die Funktionen zum Ver- und Entschlüsseln.

Derzeit stehen 3 unterschiedliche Methoden zur Verfügung.

„TryEncryptedJsonPost“ wird verwendet, wenn ein Model verschlüsselt an den Server gesendet werden soll und man ein Antwort-Model erwartet. Als klassisches Beispiel dient der Login-Vorgang, wo die Daten zur Authentifizierung (üblicherweise Benutzername und Passwort) an den Server versandt werden sollen und als Antwort erwartet man seine benötigten Benutzerdaten (wie z.B. eine ID, einen Token oder ähnliches).

„TryEncryptedJsonPostWithoutResultModel“ wird verwendet, wenn ein Model verschlüsselt an den Server gesendet werden soll und man KEIN Antwort-Model erwartet. Als Beispiel hierfür dient die Registrierung auf einer Webseite. Der Benutzer sendet alle notwendigen Daten an den Server und bekommt als Antwort lediglich, ob der Vorgang erfolgreich war oder nicht. Danach kann man sich üblicherweise mit den entsprechenden Daten einloggen.

„TryEncryptedJsonGet“ sendet keine zusätzlichen Daten an den Server (GET-Methode), aber man erhält durch diesen Get-Request Daten in Form eines Models. Als Beispiel wäre hier die Abfrage des öffentlichen Schlüssels des Servers erwähnt.

All diese Methoden senden die entsprechenden Requests verschlüsselt mit dem zur Verfügung gestellten „HttpClient“ an den Server und erwarten auch verschlüsselte Nachrichten als Antwort. Das bedeutet die Antworten des Servers müssen ebenfalls entsprechend verschlüsselt werden. Am einfachsten unter Verwendung der Klasse „ModelToEncryptedBytesConverter“ (siehe Kapitel SIKOSI Crypto).

EncryptedLoginService

Diese Klasse stellt eine Methode „TryEncryptedLogin“ zur Verfügung, die versucht den Benutzer einzuloggen.

Vor Verwendung dieser Methode müssen die beiden Properties

„GetReceiverPublicKeyFunc“ und „Encryption“ initialisiert werden.

„GetReceiverPublicKeyFunc“ stellt eine Funktion dar, die den öffentlichen Schlüssel des Empfängers (Servers) ermittelt und muss bekannt sein. Diese Funktion wird dann zum gegebenen Zeitpunkt abgerufen. „Encryption“ stellt die asymmetrische Verschlüsselung (siehe Kapitel SIKOSI Crypto) dar, die verwendet werden soll. Sie beinhaltet unter anderem den eigenen öffentlichen Schlüssel und die Funktionen zum Ver- und Entschlüsseln.

Neben den Login-Daten (Login-Model) werden der „HttpClient“ und die Route des Controllers (wenn nicht schon im HttpClient enthalten) benötigt. Wenn die Antwort des Servers eine Zeichenkette (String) „Token“ enthält („IToken“-Interface) wird dieser Token auch gleich in der übergebenen „HttpClient“-Instanz im Authorization-Header gesetzt, um die zukünftige Kommunikation dieser „HttpClient“-Instanz mit demselben Server zu unterstützen.

EncryptedRegistrationService

Diese Klasse stellt zwei Methoden für die verschlüsselte Registrierung eines Benutzers bei einem Server dar.

Vor Verwendung dieser Methode müssen die beiden Properties

„GetReceiverPublicKeyFunc“ und „Encryption“ initialisiert werden.

„GetReceiverPublicKeyFunc“ stellt eine Funktion dar, die den öffentlichen Schlüssel des Empfängers (Servers) ermittelt und muss bekannt sein. Diese Funktion wird dann zum gegebenen Zeitpunkt abgerufen. „Encryption“ stellt die asymmetrische Verschlüsselung (siehe Kapitel SIKOSI Crypto) dar, die verwendet werden soll. Sie beinhaltet unter anderem den eigenen öffentlichen Schlüssel und die Funktionen zum Ver- und Entschlüsseln.

Die beiden Methoden „TryEncryptedRegistration“ und „TryEncryptedRegistrationWithoutResultModel“ funktionieren im Prinzip ähnlich wie die Login-Methode (siehe voriges Kapitel). Allerdings wird kein Token in den „HttpClient“ gesetzt, da es sich nur um eine Registrierung handelt und um keinen Login-Vorgang. Den Unterschied zwischen den beiden Methoden der Registrierung sieht man schon im Namen. Einmal wird nur ein „OK“ oder „Nicht-Ok“ als Antwort erwartet und einmal werden zusätzliche Daten in der Antwort des Servers erwartet.

SecureServiceResults

Alle in diesem Kapitel beschriebenen Methoden retournieren ein Objekt der Klasse „SecureServiceResult“ oder „SecureServiceResultNoContent“. Anhand des Namens der beiden Klassen kann man erkennen, dass „SecureServiceResultNoContent“ verwendet wird, wenn keine zusätzlichen Daten des Servers erwartet werden. In ihr kann man nur erkennen, ob die Antwort des Servers OK war oder nicht. In „SecureServiceResult“ können auch zusätzliche Antwort-Daten des Servers beinhaltet sein.

Die Bibliothek wird in folgenden Beispielprojekten verwendet

- SIKOSI.Sample01_WebApp
- SIKOSI.Sample02 (Mobile Applikation)

2.4 SIKOSI CRYPTO

Diese Bibliothek bildet die Basis für die Verschlüsselung von Daten.

Prinzipiell werden immer Byte-Arrays verschlüsselt und entschlüsselt. Bei der Entschlüsselung (Decrypt) der Daten wurde darauf Wert gelegt, dass, wo möglich, nur der verschlüsselte Output der Verschlüsselungsmethode (Encrypt) benötigt wird. Dazu werden bei der Verschlüsselung alle benötigten Daten in ein Byte-Array zusammengefasst, welches dann der Entschlüsselungsmethode übergeben wird. Bei der Verschlüsselung mittels Passwortes (oft für Verschlüsselung von Dateien) muss klarerweise zusätzlich das Passwort bei der Entschlüsselung angegeben werden, welches auch bei der Verschlüsselung benutzt wurde.

Asymmetrische Verschlüsselung

Für die asymmetrische Verschlüsselung stehen insgesamt 4 verschiedene Klassen zur Verfügung, die allesamt das Interface „ISecureEncryption“ implementieren. Dadurch können auch eigene Klassen entwickelt werden, die dieselben Funktionalitäten bereitstellen.

Zwei dieser Klassen verwenden das Diffie-Hellman-Schlüsselaustauschverfahren („SecureEncryptionDh“ und „SecureEncryptionDhCrossPlatform“) und die anderen zwei das RSA-Verfahren („SecureEncryptionRsa“ und „SecureEncryptionRsaCrossPlatform“). Wie die Namen schon verraten ist jeweils eine der beiden Klassen auf unterschiedlichen Plattformen lauffähig, die andere hauptsächlich unter Windows und zum Teil auf Linux-Systemen.

Für die Verschlüsselung der Daten ist hier der öffentliche Schlüssel (Property „PublicKey“) des Empfängers notwendig. Dadurch kann nur dieser Empfänger mittels seines geheimen privaten Schlüssels diese Daten entschlüsseln.

Symmetrische Verschlüsselung

Für die symmetrische Verschlüsselung stehen zwei Klassen zur Verfügung („SecureEncryptionAesGcm“ und „SecureEncryptionAesCrossPlatform“). Auch hier wieder der Unterschied der unterschiedlichen Kompatibilität auf verschiedenen Systemen.

Diese Klassen bieten die Möglichkeit Daten mittels eines geheimen Schlüssels (ByteArray) oder eines Passwortes (String) zu verschlüsseln. Sie implementieren das Interface „ISecureSymmetricEncryption“, wodurch wieder eigene Implementationen möglich sind. Die Klasse „SecureEncryptionAesGcm“ verschlüsselt die Daten mittels AES im GCM-Modus. Dieser Modus ermöglicht gleichzeitig die Authentifizierung der Daten und verwendet intern die Klasse „Aes-Gcm“ der .NET-Bibliothek „System.Security.Cryptography“ – deshalb auch die eingeschränkte Verwendung.

Die Klasse „SecureEncryptionAesCrossPlatform“ verschlüsselt die Daten mittels AES im CBC-Modus mit „PKCS7-Padding“. Die Authentifizierung erfolgt hier extra über den Algorithmus „HMACSHA256“ der gleichnamigen .NET-Klasse, da sie im CBC-Modus nicht enthalten ist.

Für die Verschlüsselung der Daten ist hier ein (geheimer) Schlüssel (Byte-Array) notwendig (Methode „EncryptData“). Alternativ kann auch ein Passwort (String) verwendet werden,

aus dem dann ein geheimer Schlüssel berechnet wird (Methode „EncryptDataWithPassword“). Hier kann dann auch ein „Salt“ mitangegeben werden oder es wird ein eigenes erstellt.

Für die jeweilige Entschlüsselung ist dann entweder der geheime Schlüssel wieder notwendig oder eben das benutzte Passwort zur Wiederherstellung des geheimen Schlüssels.

CryptoResult

Bei allen Ver- und Entschlüsselungsmethoden wird als Ergebnis ein Objekt der Klasse „CryptoResult“ retourniert. Dieses Objekt gibt Aufschluss darüber, ob die jeweilige Methode zum Ver- oder Entschlüsseln erfolgreich verlaufen ist (Bool'scher Wert „Success“). Wenn nicht erfolgreich, kann der Grund im Property „CausingException“ ausgelesen werden. Wenn erfolgreich, stehen die ver- oder entschlüsselten Daten im Property „ResultBytes“. Beim Verschlüsseln ist anzumerken, dass die „ResultBytes“ nicht allein aus den verschlüsselten Daten bestehen, sondern auch aus den mitgelieferten Daten, die zur Entschlüsselung der Daten notwendig (aber nicht geheim - gemäß Kerckhoff's Prinzip ist die Sicherheit der Daten allein von der Geheimhaltung des geheimen bzw. privaten Schlüssels abhängig) sind. Daher folgt, dass das Byte-Array nach der Verschlüsselung etwas größer ist als die ursprünglichen Daten.

CryptoBytes

In der Klasse „CryptoBytes“ werden die verschlüsselten Daten mit den benötigten zusätzlichen Daten in ein Byte-Array zusammengefasst („GetConcatenatedBytes“) bzw. wieder aus einem Byte-Array herausgelesen („SplitBytes“).

IDiffieHellmanKeyExchange

Die Klasse „SecureEncryptionDh“ und „SecureEncryptionDhCrossPlatform“ stellen jeweils neben dem parameterlosen Konstruktor auch einen weiteren Konstruktor zur Verfügung. Dieser nimmt ein Objekt mit dem Interface „IDiffieHellmanKeyExchange“ entgegen. Auf diese Weise kann ein „eigenes“ Diffie-Hellman-Schlüsselaustausch-Verfahren implementiert werden. Es gibt auch einige Bibliotheken, die diese Funktionalitäten zur Verfügung stellen. In SIKOSI wurde dieses Verfahren mittels vier verschiedener externer

Bibliotheken umgesetzt und daher gibt es vier verschiedene Implementierungen des Interface „IDiffieHellmanKeyExchange“, nämlich „EcdhBouncyCastle“, „EcdhInferno“, „EcdhNaCl“ und „EcdhNsec“. Der Namensteil nach „Ecdh“ gibt Aufschluss darüber welche externe Bibliothek jeweils verwendet wurde, um den Diffie-Hellman-Schlüsselaustausch zu implementieren. Es steht also Entwicklern auch frei hier ihre eigene Bibliothek/Implementierung zu verwenden.

ModelToEncryptedBytesConverter

Diese Hilfsklasse im Namespace „SIKOSI.Crypto.Helper“ kann verwendet werden um ein beliebiges Objekt (Model) zu ver- und entschlüsseln. Das Ergebnis beim Verschlüsseln („ConvertFromModelToEncryptedByteArray“) ist ein Byte-Array welches den öffentlichen Schlüssel des Senders sowie das Model verschlüsselt enthält. Beim Entschlüsseln („ConvertFromEncryptedByteArrayToModel“) wird dieses Byte-Array wieder in ein Objekt zurück konvertiert und entschlüsselt, das die Daten und öffentlichen Schlüssel des Senders enthält.

Die Bibliothek wird in folgenden Beispielprojekten verwendet

- **SIKOSI.Sample01_WebApp**
- **SIKOSI.Sample02 (Mobile Applikation)**
- **SIKOSI.Sample07_EncryptedChat**

2.5 SIKOSI Datenbank Services

Hier befinden sich die Datenbankzugriffe.

3 Beispielprojekte

Folgend eine Auflistung der erstellten Beispielprojekte, die erstellte Bibliotheken verwendet.

3.1 Sample 01 – Auth

Projektname: SIKOSI.Sample01_Auth

Technologie: ASP.NET CORE

Art: SERVER API

In diesem Beispielprojekt wird die SDK SIKOSI.Services.Auth verwendet.

Implementiere Routen

- **Authenticate/Login**
Autorisierung eines Users
- **Manage**
Benutzerdaten abfragen
- **Register**
Registrierung eines Benutzers

Geschützte Routen, bei denen ein Benutzer authentifiziert sein muss, müssen mit dem Schlüsselwort `[Authorize]` gekennzeichnet werden. Nicht geschützte Routen wie das Login und die Registrierung müssen mit dem Schlüsselwort `[AllowAnonymous]` gekennzeichnet werden. Bei sämtlichen geschützten Routen muss clientseitig der Autorisierungs-Token (JWT-Token) mitgeschickt werden, der beim Login generiert wird.

Sollte ein Benutzer nicht ... sein ...

Account		▼
POST	/api/account/login	
GET	/api/account/logout	
GET	/api/account/manage	
POST	/api/account/register	

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- SIKOSI.Services.Auth

3.2 Sample 02 – Secure Remote Password

Projektname: SIKOSI.Sample02_SRP

Technologie: ASP.NET CORE

Art: SERVER API

Beispielhafte Implementierung der ...

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- **SIKOSI.SRP**

The SRP Protokoll kann auf folgende drei Schritte aufgeteilt werden

1. Registrierung
2. Authentifizierung
3. Verifizierung

Registrierung

Während der Registrierung wählt ein Benutzer seinen Benutzernamen und sein Kennwort und fordert basierend auf dem Benutzernamen dieses Benutzers ein generiertes Salt an. Anschließend wird vom Benutzer eine Verifizierung generiert, der dann an den Server gesendet wird, der diese Verifizierung zusammen mit dem generierten Salt und dem Benutzernamen beibehält, um den Registrierungsprozess abzuschließen.

Diese Verifizierung kann neben Benutzername und Benutzersalt in der Serverdatenbank gespeichert werden.

Authentifizierung

Während der Authentifizierung gibt der Benutzer seine Anmeldeinformationen auf der Clientseite ein.

Die Anwendung sendet dann eine Anforderung an den Server und fordert das mit diesem Benutzernamen verknüpfte Salt an.

Sobald der Benutzer sein Salt kennt, berechnet er zwei kurzlebige Werte, A und a.

Der Wert A, im SDK als clientPublicValue bekannt, wird an den Server gesendet, während der Wert a, im SDK als clientPrivateValue bekannt, im Client-Speicher gespeichert wird.

Der Server berechnet dann selbst 2 kurzlebige Werte, die als B und b bekannt sind.

Wie beim Client wird der Wert B, auch als serverPublicValue bezeichnet, an den Client gesendet, während der Wert b, der als serverPrivateValue bezeichnet wird, im Serverspeicher gespeichert wird.

Nach diesen Schritten berechnen sowohl der Client als auch der Server einen Sitzungsschlüssel und beenden den Authentifizierungsschritt.

Verifizierung

Während der Überprüfung beweisen sich sowohl der Server als auch der Client gegenseitig, dass sie denselben Sitzungsschlüssel berechnet haben.

Zunächst berechnet der Client seinen Beweis anhand seines eigenen öffentlichen Werts, des öffentlichen Werts des Servers sowie des Sitzungsschlüssels.

Anschließend sendet der Client den berechneten Proof zur Überprüfung an den Server. Um den Client-Proof zu überprüfen, berechnet der Server denselben exakten Proof unter Verwendung der Clients sowie seiner eigenen öffentlichen Werte und des Sitzungsschlüssels. Wenn und nur wenn sowohl der Client als auch der Server übereinstimmende Sitzungsschlüssel berechnet haben, stimmt der Client-Proof mit dem überein, was der Server als erwarteter Client-Proof berechnet hat.

Somit hat der Server überprüft, dass der Benutzer den richtigen Sitzungsschlüssel berechnet hat und daher im Besitz des Kennworts sein muss, ohne das Kennwort jemals direkt über das Netzwerk senden zu müssen.

Nach Überprüfung der Legitimität des Benutzers berechnet der Server seinen eigenen Beweis anhand des öffentlichen Werts des Clients, des vom Server generierten Sitzungsschlüssels sowie des vom Client generierten und gesendeten Beweises.

Nachdem der Client seinen eigenen Proof an den Client gesendet hat, berechnet er, wie der Server zuvor mit dem Client-Proof, seinen erwarteten Proofwert anhand derselben exakten Formel, die der Server verwendet hat.

Nur wenn sowohl Client als auch Server über identische Sitzungsschlüssel verfügen, stimmen der erwartete Beweiswert und der tatsächliche Beweiswert überein.

Zu diesem Zeitpunkt ist der Überprüfungsschritt abgeschlossen, da beide Parteien sicher wissen, dass die Berechnungen der anderen Parteien zu einem identischen Sitzungsschlüssel geführt haben.

Somit kann dieser Sitzungsschlüssel von nun an zum Ver- und Entschlüsseln von Nachrichten verwendet werden.

3.3 Sample 03 – Crypto Server

Projektname: SIKOSI.Sample03_CryptoServer

Technologie: ASP.NET CORE

Art: SERVER API

Ein Mini-Projekt, das beispielhaft einen Controller eines Servers zeigt, von dem der öffentliche Schlüssel (entweder RSA oder Diffie-Hellman-Verfahren) abgefragt werden kann und an den dann eine verschlüsselte Nachricht versendet werden kann.

„SIKOSI.Sample01_CryptoClientDh“ und „SIKOSI.Sample02_CryptoClientRsa“ können hier als Client dienen.

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- SIKOSI.Crypto

3.4 Sample 04 – SignalR

Projektname: SIKOSI.Sample04_SignalR

Technologie: ASP.NET CORE

Art: SERVER API

SignalR bietet eine Möglichkeit in Echtzeit Daten auszutauschen. In diesem Projekt wird gezeigt welche Maßnahmen bezüglich SignalR ergriffen werden muss, damit der Datentransport auch autorisiert stattfinden kann.

3.5 Sample 05 – Multi Factor Authentication

Projektname: SIKOSI.Sample05_MFA

Technologie: ASP.NET CORE

Art: SERVER API

Die Nuget Packages **Microsoft.AspNetCore.Identity.EntityFrameworkCore** und **Microsoft.AspNetCore.Identity.UI** bieten eine einfache Möglichkeit, Authentifizierung und Autorisierung in eine Web-Applikation einzubauen. Zusätzlich bietet das Framework die Möglichkeit, Two Factor Authentication in eine Applikation einzubinden.

Aktivieren der Möglichkeit, 2FA in der App zu nutzen

In [diesem](#) Dokumentationstext von Microsoft wird gut beschrieben, wie die Möglichkeit der 2FA in die eigene App integriert werden kann. Dieser Abschnitt fasst die Schritte zusammen. Um den QR-Code zu generieren, ist eine Javascript Library notwendig. Microsoft verweist hierbei auf [diese Javascript-Bibliothek](#). Nach dem herunterladen muss diese entpackt, und der Ordner in das Verzeichnis `wwwroot/lib` des Projektes gezogen werden.

Einrichten von 2FA

Um die Two Factor Authentication einzurichten, sobald die im Artikel beschriebenen Schritte durchgeführt wurden, muss zuerst ein Benutzeraccount erstellt werden. Die dafür notwendige Eingabemaske wird von Identity generiert. Sobald ein Account besteht und sich in diesen eingeloggt wurde, ist es möglich per Klick auf den eigenen Benutzernamen in ein Menü zu gelangen. In diesem Menü sind diverse Optionen aufgelistet, darunter Profil, Externe

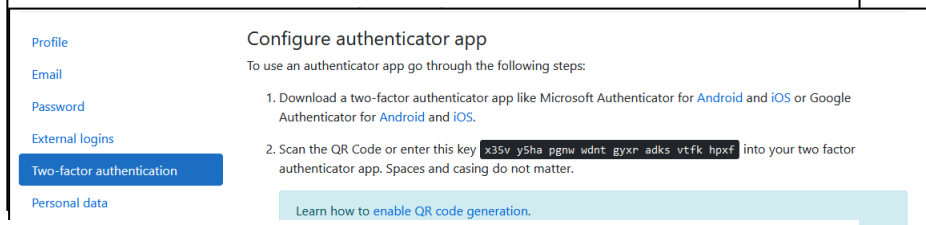
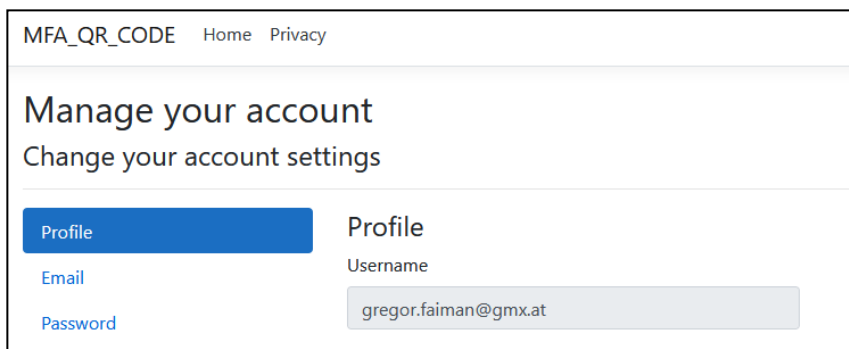


Abbildung 3: Benutzermenü

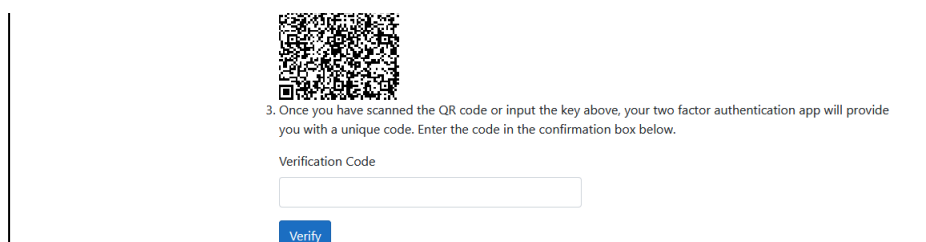


Abbildung 2: 2FA aktivieren per QR Code Scan

Logins, sowie auch Two-Factor Authentication. Per Klick auf die Option Two-Factor Authentication ist es möglich, unter der Option **Add Authenticator** eine neue Authenticator App zu registrieren. Um diesen Schritt durchzuführen wird eine Authenticator App wie Google Authenticator, oder Microsoft Authenticator für das Smartphone benötigt. Bei auswählen der Option wird ein QR Code im Browser generiert. Dieser QR Code ist mit der gewählten Authenticator App zu scannen. Damit ist für den aktuellen Benutzeraccount die Two-Factor Authentication aktiviert, wodurch bei jedem Login ein sechsstelliger Code einzugeben ist, der in der Authenticator App generiert und jeweils alle 30 Sekunden erneuert wird.

Zugriff auf eine Ressource auf jene Accounts beschränken, die 2FA aktiviert haben

Um den Zugriff auf eine Ressource nur jenen Benutzern zu erlauben, die ihrem Account 2FA hinzugefügt haben, kann bei der **OnGet** Methode der code Behind Datei einer Razor Page in den User Claims abgefragt werden, ob für den jeweiligen Benutzer 2FA aktiviert ist. Wenn 2FA

```
public IActionResult OnGet()
{
    var isToFactorEnabled = this.User.Claims.FirstOrDefault(t => t.Type == "amr");

    if (isToFactorEnabled != null && "mfa".Equals(isToFactorEnabled.Value))
    {
        return Page();
    }
    else
    {
        return Redirect("/Identity/Account/AccessDenied");
    }
}
```

aktiviert ist, wird es dem Benutzer gestattet auf die Seite zuzugreifen (oder es geschieht ein Redirect, um eine andere Aktion auszuführen, potentiell die Aufforderung den Code erneut einzugeben). Andernfalls wird der Benutzer umgeleitet, weil ihm aufgrund von nicht eingerichteter 2FA die Berechtigungen fehlen, um auf eine gewisse Ressource zuzugreifen.

3.6 Sample 06 – Beispiel für fehlerhafte Implementierung

Projektname: SIKOSI.Sample06_BadSite

Technologie: ASP.NET CORE

Art: SERVER API

Beispielhafte Fehl-Implementierung einer Seite, die nicht über XSRF geschützt ist.

XSRF – Cross Site Request Forgery

Cross Site Request Forgery gehört neben SQL Injection und Cross Site Scripting zu den drei großen Angriffsvektoren auf Web Applikationen. Bei einer Cross Site Request Forgery Attacke, versucht sich der Angreifer ein Authentication Cookie zunutze zu machen. Das klassische Beispiel um eine XSRF Attacke zu verbildlichen ist jene einer Banküberweisung.

Man nehme an, ein Benutzer führt eine Banküberweisung auf der Seite X aus. Nach dem Einloggen in die Applikation bekommt der Benutzer einen Token, welcher im Browser als ein Cookie gespeichert ist, und mit dessen Hilfe der Benutzer gegenüber der Bankapplikation beweisen kann, dass er über die notwendigen Berechtigungen verfügt. Nach dem Durchführen der Überweisung loggt sich der Benutzer nicht aus, wodurch das Authentication Cookie nicht seine Gültigkeit verliert. Ein Problem wird dies dann, wenn der Benutzer auf eine bössartige Seite stößt, welche entweder auf Knopfdruck, Stichwort PopUps mit dem Text „Sie

haben gewonnen“, oder vollautomatisch durch einfaches Anklicken und Laden der Seite ein unsichtbares Form generiert, und dieses Form an den Webserver der Bank Applikation schickt. Weil sich der Benutzer nicht ausgeloggt hat, ist in diesem Fall das gültige Authentication Cookie noch im Browser des Benutzers gespeichert. Aus Sicht der Bankapplikation wäre bei diesem Request, sofern kein Anti-Forgery Schutz implementiert ist, also alles in Ordnung, weil es scheinbar vom Benutzer initiiert wurde.

Die nachfolgenden Abbildungen stellen eine XSRF Attacke dar. Zunächst loggt sich der Benutzer in die Applikation, und führt die Überweisung durch.

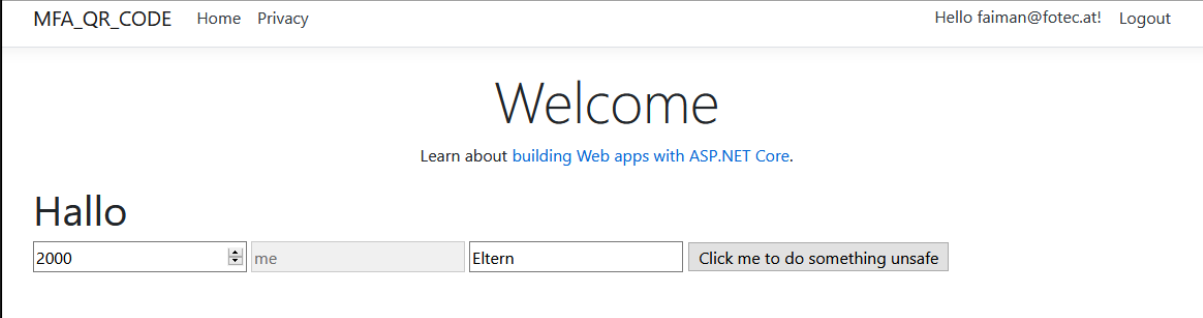


Figure 1: Eingabemaske der legitimen Bankapplikation.

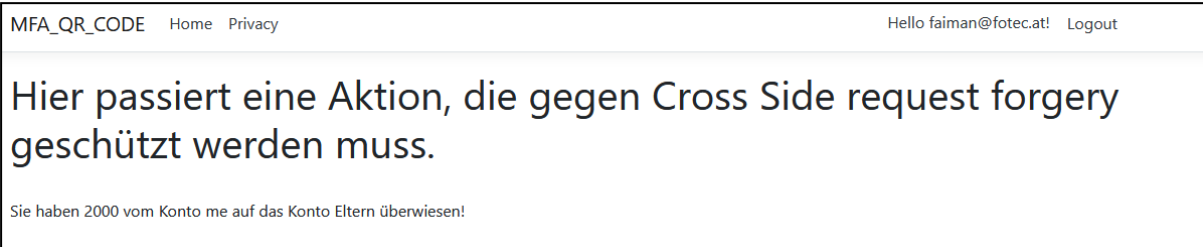


Abbildung 2: Benutzer hat die legitime Überweisung durchgeführt.

Danach vergisst er sich auszuloggen, und surft weiter im Web, wobei er unglücklicherweise auf eine bössartige Seite stößt. Auf dieser bössartigen Seite sticht ihm sofort

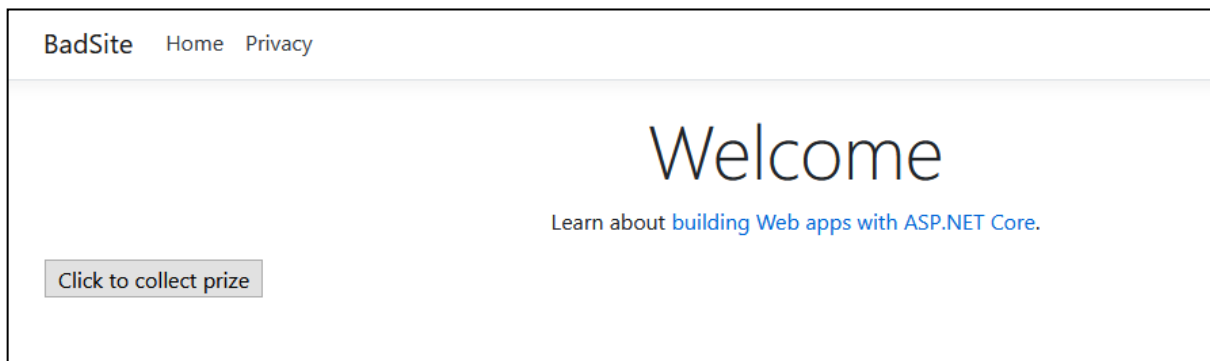


Abbildung 3: Bössartige Seite mit PopUp

ein Pop-Up in die Augen, auf dem Text in ähnlichem Format wie „Sie haben gewonnen, klicken Sie HIER um Ihren Preis zu erhalten, Sie haben noch 60 Sekunden!!!!“ aufscheint. Der Benutzer, nicht ahnend, dass das

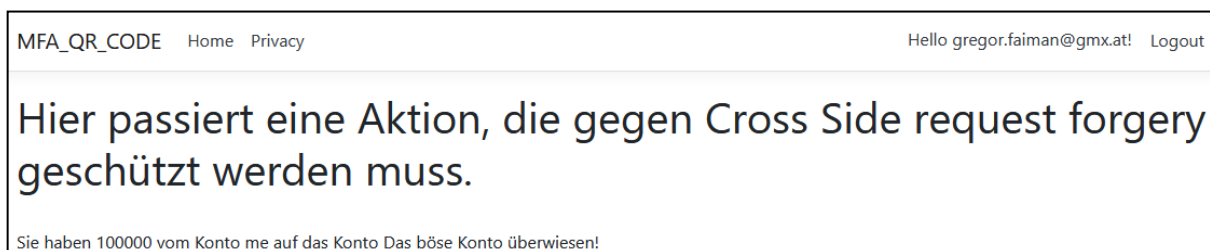


Figure 2: Erfolgreiche XSRF Attacke

Internet ein gefährlicher Ort sein kann, klickt auf den Knopf, wodurch im Hintergrund ein POST Request generiert wird, welches an die Bankapplikation geschickt wird. Weil wie bereits angesprochen, das Session Cookie des Benutzers noch im Browser gespeichert und von der Bankapplikation noch nicht annulliert wurde, ist das Request aus Sicht der Applikation ein autorisiertes Request, wodurch der Angreifer einen beliebigen Betrag auf ein beliebiges Konto überweisen kann.

Gegen XSRF Attacke schützen (in ASP.NET Core)

Die leichteste Variante um sich gegen eine XSRF Attacke zu schützen wäre, dass sich jeder Benutzer verlässlich aus Applikationen ausloggt. Weil darauf nicht immer Verlass ist, benötigt es jedoch andere Sicherheitsstrategien. Glücklicherweise liefert Microsoft eine fertig implementierte Sicherheitsstrategie in Form eines Anti Forgery Tokens mit, welcher bereit ist

```
[ValidateAntiForgeryToken]
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [HttpGet]
    public IActionResult Index()
    {
        return View(new TransactionAmountModel());
    }

    [Route("Home/Privacy")]
    [Route("Privacy")]
    [Authorize]
    [HttpGet]
    public IActionResult Privacy()
    {
        return View();
    }

    [Route("Home/DoSomethingUnsafe")]
    [Authorize]
    [HttpPost]
    public IActionResult DoSomethingUnsafe(TransactionAmountModel model)
    {
        return View(model);
    }

    [Route("Home/DoSomethingSafe")]
    [Authorize]
    [HttpPost]
    [IgnoreAntiforgeryToken]
    public IActionResult DoSomethingSafe()
    {
        return View();
    }

    [Route("Home/Error")]
    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    [HttpGet]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

Abbildung 4: Controller mit Anti forgery Token Attribut.

verwendet zu werden.
 Um diesen Anti Forgery Token zu verwenden muss in der `Startup.ConfigureServices` Methode die Anti Forgery Middleware hinzugefügt werden:

- AddMvc
- MapRazorPages
- MapControllerRoute

- MapBlazorHub

Um den Token zu verwenden, ist bei einer Controller oder Methodensignatur das Attribute `ValidateAntiForgeryToken` zu setzen.

Die von Microsoft empfohlene Best-Practice ist hier jedoch, das Attribut `AutoValidateAntiForgeryToken` zu verwenden, und dieses auf Controllerebene zu setzen. Das Attribut `AutoValidateAntiForgeryToken` sorgt dafür, dass nur Methoden die als sicher gelten, weil sie keine Daten modifizieren (`GET`, `HEAD`, `OPTIONS`, `TRACE`) ohne Anti Forgery Token ausgeführt werden können. Requests die sämtliche andere Methoden ausführen, brauchen einen validen Anti Forgery Token.¹

Anti Forgery Token mitschicken

Um einen Anti Forgery Token in ASP.Net Core mitszuschicken, gibt es zwei Möglichkeiten. Wenn ein Form mittels HTML Helper generiert wird, erzeugt ASP.Net Core automatisch einen Anti Forgery Token, wenn die Methode des Forms eine andere ist, als HTTP GET. Wird das Form Element von HTML verwendet, kann ein Anti Forgery Token mittels Aufruf von `HTML.AntiForgeryToken()` hinzugefügt werden. Sollte das `action` Attribut des Forms leer und die Methode POST sein, wird der Anti Forgery Token dem Form automatisch hinzugefügt. Wenn die Anti-Forgery Validierung aktiviert ist, generiert der Webserver für den Client auf Anfrage des Forms zwei Validierungstoken, die an den Client mitgeschickt werden. Diese Token haben die Eigenschaft, dass sie kryptografisch generiert sind, sodass es fast unmöglich ist, sie zu erraten, und beide vom Client beim schicken des ausgefüllten Forms inkludiert werden müssen, damit dem Form Gültigkeit geschenkt wird. Dadurch wäre es der böartigen Seite zwar möglich, einen `AntiForgeryToken` im Form zu inkludieren, jener wäre jedoch mit äußerst sicherer Wahrscheinlichkeit falsch, wodurch das Request nicht autorisiert würde.

```
<form action="https://localhost:44373/Home/DoSomethingUnsafe" method="post">
  @Html.AntiForgeryToken()
  <input type="number" hidden name="Amount" value="100000" />
  <input type="text" hidden name="DestinationBankAccount" value="Das böse Konto" />
  <input type="submit" value="Click to collect prize" />
</form>
```

Figure 3: Selbst mit eigenem generierten `AntiForgeryToken`, hat die böse Seite keine Chance.

3.7 Sample 07 – Verschlüsselter Chat und Speicherung verschlüsselter Dateien

Projektname: SIKOSI.Sample07_EncryptedChat

Technologie: ASP.NET CORE

Art: SERVER API

Beispielhafte Implementierung der asymmetrischen Verschlüsselung unter Verwendung der entsprechenden SIKOSI-Bibliotheken.

Dieses Beispielprojekt verwendet folgende Bibliotheken:

¹ Offizielle Microsoft Dokumentation: <https://docs.microsoft.com/de-de/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.1>

- SIKOSI.Crypto

Dieser Server kann genutzt werden um über SignalR verschlüsselte Chat-Nachrichten an alle verbundenen Benutzer weiterzuleiten. Das soll demonstrieren, dass die Nachrichten sicher verschlüsselt sind und nur vom „anvisierten“ Benutzer entschlüsselt werden können. Bei allen anderen schlägt die Entschlüsselung der Nachricht fehl und kann nicht gelesen werden.

Außerdem stellt dieser Server Controller zur Verfügung die einen verschlüsselten Login- und Registrierungsvorgang zeigen.

Zusätzlich können eingeloggte Benutzer ihre Daten ändern und verschlüsselte Files speichern und wieder abrufen. Der Server selbst hat hier zu keinem Zeitpunkt die Möglichkeit an die unverschlüsselten Dateien und Chatnachrichten zu kommen. Er dient hier also nur dem Zweck der Speicherung und/oder Weiterleitung von verschlüsselten Daten. Die Kommunikation mit dem Server selbst findet ebenfalls verschlüsselt statt.

3.8 Sample 08 – IOT

Projektname: SIKOSI.Sample08_IOTServer

Projektname: SIKOSI.Sample05_IOT

Technologie: ASP.NET CORE

Art: SERVER API

Beispielhafte Implementierung der ...

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- SIKOSI.Secure Services

Dies ist eine Demoapplikation, um zu zeigen, dass die Bibliotheken auch auf IOT Geräten zur Anwendung kommen kann.

Die Applikation nimmt Daten, die von einem LoraWan-Gerät gesammelt werden, entgegen und überträgt diese Daten sicher an einen Server.

Eine Installationsanleitung findet sich im Projekt in der Datei README.md

4 Demo-Applikationen

Sample 01 – WebApplikation

Projektname: SIKOSI.Sample01_WebApp

Technologie: ASP.NET CORE

Art: Webapplikation

Beispielhafte Implementierung der asymmetrischen und symmetrischen Verschlüsselung unter Verwendung der entsprechenden SIKOSI-Bibliotheken.

Starten Sie gleichzeitig das Projekt SIKOSI.Sample07_EncryptedChat und achten Sie auf eine übereinstimmende Adresse und Port in der Datei „launchSettings.json“ des Servers und in der Datei „appSettings.json“ der WebApp unter „ServerBaseAddress“.

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- **SIKOSI.Crypto**
- **SIKOSI.SecureServices**

Diese Webapplikation (Blazor) zeigt einen verschlüsselten Login- und Registrierungsvorgang unter Verwendung der Bibliothek „SIKOSI.SecureServices“. Anschließend können in einem Chat verschlüsselte Nachrichten (inklusive Bilder und Videos) an die verschiedenen registrierten Benutzer (oder alternativ an die ganze Gruppe - Gruppenchat) versandt werden. Die Nachrichten werden dabei nur über SignalR weitergeleitet und nicht am Server zwischengespeichert. Das heißt, nicht eingeloggte Benutzer erhalten die Nachrichten nicht.

Außerdem werden die verschlüsselten Nachrichten immer an alle anderen eingeloggten Benutzer weitergeleitet. Allerdings können nur die entsprechenden Empfänger (mit dem richtigen Schlüsselpaar) die Nachrichten entschlüsseln (persönliche Nachricht) bzw. bei Gruppennachrichten können alle mit dem zuvor verteilten Gruppenschlüssel die Nachricht entschlüsseln. Der Server (hier SIKOSI.Sample07_EncryptedChat) hat zu keiner Zeit die Gelegenheit die Nachrichten zu entschlüsseln, da er nie den entsprechenden Schlüssel besitzt.

Weiters können Dateien „passwort-verschlüsselt“ und an den Server zur Speicherung gesendet werden. Ebenso können diese Datei wieder heruntergeladen und mit der Eingabe des richtigen Passworts entschlüsselt werden. Der Server hat auch hier zu keiner Zeit Kenntnis über den Inhalt der Datei, da sie immer verschlüsselt ist.

Ebenso können die Benutzerdaten beispielhaft geändert und verschlüsselt übermittelt werden.

[Sample 02 – Smartphone Applikation für Android und iOS](#)

Projektname: SIKOSI.Sample02

Technologie: .NET Standard und Xamarin

Art: Mobile Applikation

Beispielhafte Implementierung der asymmetrischen und symmetrischen Verschlüsselung unter Verwendung der entsprechenden SIKOSI-Bibliotheken.

Starten Sie gleichzeitig das Projekt SIKOSI.Sample07_EncryptedChat und achten Sie auf einen übereinstimmenden Port in der Datei „launchSettings.json“ des Servers und im definierten „HttpClient“ in der App.xaml.cs Datei der mobilen Applikation.

Dieses Beispielprojekt verwendet folgende Bibliotheken:

- **SIKOSI.Crypto**
- **SIKOSI.SecureServices**

Diese mobile Applikation zeigt einen verschlüsselten Login- und Registrierungsvorgang unter Verwendung der Bibliothek „SIKOSI.SecureServices“. Anschließend können in einem Chat verschlüsselte Nachrichten (inklusive Bilder) an die verschiedenen registrierten Benutzer (oder alternativ an die ganze Gruppe - Gruppenchat) versandt werden. Die Nachrichten werden dabei nur über SignalR weitergeleitet und nicht am Server zwischengespeichert. Das heißt, nicht eingeloggte Benutzer erhalten die Nachrichten nicht.

Außerdem werden die verschlüsselten Nachrichten immer an alle anderen eingeloggten Benutzer weitergeleitet. Allerdings können nur die entsprechenden Empfänger (mit dem richtigen Schlüsselpaar) die Nachrichten entschlüsseln (persönliche Nachricht) bzw. bei Gruppennachrichten können alle mit dem zuvor verteilten Gruppenschlüssel die Nachricht entschlüsseln. Der Server (hier SIKOSI.Sample07_EncryptedChat) hat zu keiner Zeit die Gelegenheit die Nachrichten zu entschlüsseln, da er nie den entsprechenden Schlüssel besitzt.

Weiters können Dateien „passwort-verschlüsselt“ und an den Server zur Speicherung gesendet werden. Der Server hat auch hier zu keiner Zeit Kenntnis über den Inhalt der Datei, da sie immer verschlüsselt ist.

5 Sichere Datenhaltung

Always Encrypted ist eine Art der Datenverwaltung, bei der bestimmte besonders sensible, in der Datenbank persistierte Daten, verschlüsselt sind, wodurch eine Trennung zwischen dem Besitzer der Daten, und dem Verwalter der Daten geschaffen wird. Die Daten die verschlüsselt in der Datenbank liegen, bleiben dort verschlüsselt. Eine benötigte Entschlüsselung passiert nur am Client, nie auf dem Datenbankserver selbst. Dies führt dazu, dass der Server niemals die Möglichkeit bekommt, die Daten zu lesen. Das bedeutet jedoch auch, dass die Daten immer erst vom Server geladen werden müssen, bevor diese entschlüsselt und somit gelesen werden können. Es gibt eine Version von Always Encrypted, welche dieses Problem löst, und sogenannte Secure Enclaves verwendet. Eine Secure Enclave bezeichnet einen geschützten Bereich im Speicher des Servers, welcher von außen wie eine Black Box wirkt. Diese Black Box kann verwendet werden, um Daten darin zu entschlüsseln, lesen und zu modifizieren, ohne dass der Server die Möglichkeit hat, in diesen geschützten Bereich Einsicht zu erlangen.

5.1 Konfiguration von Always Encrypted ohne Secure Enclaves

Um Always Encrypted zu konfigurieren wird das Microsoft SQL Server Management Studio (MSSMS) benötigt. Dieses Tool steht online zum Download zur Verfügung.

Die Schritte wie Always Encrypted grundsätzlich funktioniert belaufen sich auf:

1. MSSMS installieren
2. Verbindung zum Datenbankserver herstellen (Kann auch local zum Testen sein)
3. Rechtsklick auf Datenbank → Tasks → Encrypt Columns
4. Auswählen der Spalten, welche verschlüsselt werden sollen, sowie den [Typ](#) der Verschlüsselung.
5. Auswählen des Schlüssels (entweder bereits vorhandenen Schlüssel verwenden, oder neuen generieren lassen)
6. Wahl wie der Schlüssel zu speichern ist, entweder in der Azure Cloud, oder Lokal am Rechner.
7. Verschlüsselung der Spalten durch das Tool, mit anschließender Validierung.

Nachdem Daten verschlüsselt wurden, ist der Connection String der Applikation um folgendes zu ergänzen: **Integrated Security=true;Column Encryption Setting=enabled**

Nun kann die Datenbank in gewohnter Weise verwendet werden, jedoch ist es von außen nicht möglich, die Spalten die verschlüsselt wurden zu lesen. Damit wird effektiv unterschieden, zwischen dem Besitzer der Daten, welcher gleichzeitig die Rechte haben soll, diese zu lesen, und dem Datenbank Administrator und anderen Befugten, welche die Datenbank zwar verwalten, aber keinen lesenden Zugriff auf die Daten selbst haben sollen.

Anmerkung: Es ist sowohl möglich, Microsoft Certificate Store, als auch Azure Key Vault für die Speicherung des Masterschlüssels zu verwenden. Die genaue Beschreibung, wie Always Encrypted zu konfigurieren ist, kann [hier](#) (Microsoft Certificate Store) sowie [hier](#) (Microsoft Azure Key Vault) eingesehen werden.

5.2 Verschlüsselungstypen

Deterministisch

Deterministische Verschlüsselung generiert immer den selben Schlüsseltext für einen bestimmten Input, wie auch Hash Funktionen das tun. Vorteile davon sind, dass mehr

Datenbankoperationen auf den verschlüsselten Daten ausgeführt werden können. Nachteile davon sind, dass unbefugte Nutzer durch analysieren von Mustern eventuell Informationen über verschlüsselte Spalten bekommen könnten.

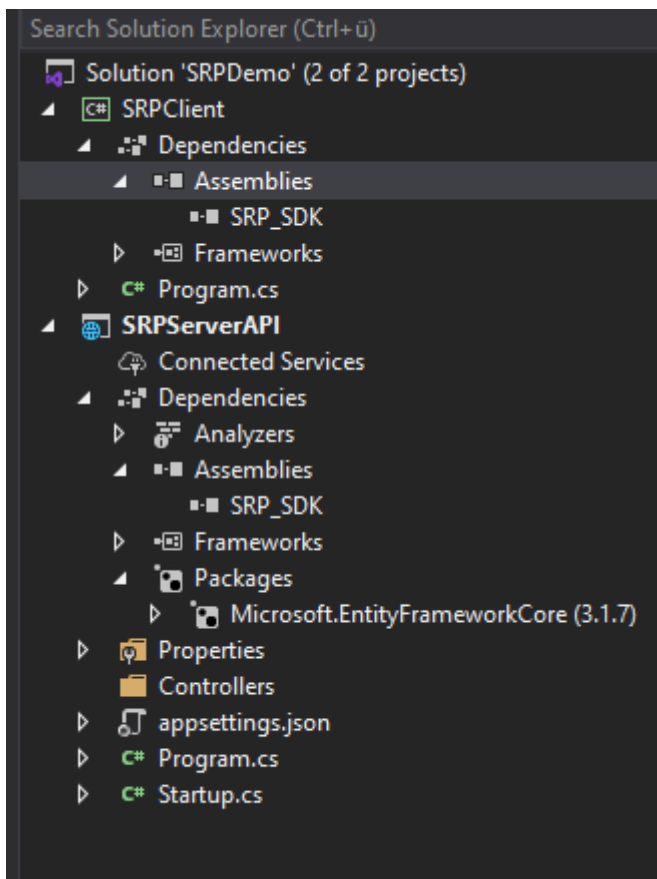
Randomisiert

Randomisierte Verschlüsselung verschlüsselt Daten auf eine unvorhersehbarere Art und Weise. Dadurch wird die Sicherheit erhöht, da keine Muster bei den verschlüsselten Daten erkennbar sind. Dafür sind einige Datenbankoperationen, wie Joins, Indexing, Pattern Matching oder Searching auf Spalten die randomisiert verschlüsselt sind, unmöglich.

6 HOW-TO

6.1 SRP SDK EINBINDEN

Der erste Schritt um das SDK einzubinden ist es, zwei neue Projekte zu erstellen, um die Funktionalität für Server und Client einzubauen. Die Demoprojekte aus denen in diesem Artikel immer wieder Screenshots auftauchen werden, sind als Konsolenapplikation für den Client, und als Web API für den Server aufgesetzt, jedoch können hier beliebige Projekttypen gewählt werden. Nach dem Erstellen der Projekte ist der nächste Schritt, das SDK in beide Projekte einzubinden. Dies ist per Rechtsklick auf das jeweilige Projekt, unter dem Punkt



Manage Nuget Packages möglich. Zusätzlich zum SRP SDK, ist in das Server Projekt noch ein Packet einzubinden, das eine Nutzerverwaltung ermöglicht. Das Demo Projekt verwendet hierzu **Entity Framework Core und Microsoft SQL Server**. Die Wahl der Datenbank ist jedoch jedem Entwickler selber überlassen. Empfehlenswert. Sind sämtlich notwendige Referenzen in beide Projekte eingebunden, kann das eigentliche Integrieren des SRP SDKs nun beginnen.

Implementieren des Servers

Der erste Schritt am Server ist es, per Rechtsklick auf das Projekt – Hinzufügen – Controller einen neuen Controller hinzuzufügen. Dieser Controller wird sämtliche Anfragen zu SRP entgegennehmen, und sich um SRP spezifische Logik kümmern. Diesem Controller sind nun einige Methoden hinzuzufügen.

Registration

Der erste Schritt im Protokoll ist es, dass ein Benutzer, egal ob er sich Registrieren oder Authentifizieren möchte, ein Salt anfragt, das der Server basierend auf seinem Benutzernamen generiert. Hierfür benötigt es eine **http GET Methode**, die besagtes Salt zurückliefert. Die Generierung des Salts übernimmt die Implementierung des SRP Servers. Die

hierfür benötigte Methode ist `byte[] SrpServer.GenerateSalt(string username)`. Falls ein Benutzername dem Server bereits bekannt ist, weil sich ein Benutzer mit diesem Namen bereits registriert hat, kann der Server den bereits bekannten Salt-Wert zurückliefern, und muss diesen nicht neu generieren. Daher ergibt sich nachfolgender Code.

```
/// <summary>
/// Action that gets invoked when the user requests the salt for a given user name.
/// </summary>
/// <param name="userName">The specified user name.</param>
/// <returns>The salt for this particular user if successful. Otherwise a bad request result.</returns>
[HttpGet]
[Route("/api/user/getsalt/{userName}")]
public async Task<IActionResult> GetSaltAsync(string userName)
{
    byte[] salt;
    ApplicationUser user;

    if (userName == null)
    {
        return BadRequest("Username was null");
    }

    user = await this.context.Users.Where(u => u.Username == userName).FirstOrDefaultAsync();

    if (user != null)
        salt = user.Salt;
    else
        salt = srpServer.GenerateSalt(userName);

    return Content(Convert.ToBase64String(salt));
}
```

Um die Registration abzuschließen benötigt der Controller eine http POST Methode, die es einem Client erlaubt, Daten die für die Registration benötigt werden, an den Server zu übermitteln. Hierfür werden zwei Dinge benötigt. Einerseits die Methode, die das Request am Controller annimmt, andererseits eine Model Klasse, welche die zur Registration benötigten Daten kapselt. Diese benötigten Daten sind:

1. Das Salt des Benutzers, gespeichert als byte array.
2. Den Benutzernamen, gespeichert als string
3. Den Verifier, der vom Client generiert wurde, gespeichert als byte array.

Die Controller Methode hat die Aufgabe, zu prüfen ob das Model gültig ist, und falls dies gegeben ist und der Benutzer noch nicht existiert, einen neuen Benutzer mit den Daten aus dem Model anzulegen.

Authentifizierung

Ist ein Benutzer bereits registriert und möchte sich authentifizieren, generiert dieser zunächst einen Wert, den er an den Server schickt. Hierfür wird eine weitere http POST Methode am Server benötigt. An diesem Punkt ist es Zeit eine weitere Model Klasse zu erstellen, um Login spezifische Daten zu kapseln. Die beim Login benötigten Daten belaufen sich auf den Benutzernamen, und den auf Client Seite generierten einmaligen Wert.

Nachdem der Client seinen berechneten Wert an den Server übermittelt hat, muss der Server seinerseits einen einmaligen Wert berechnen, und diesen an den Client übermitteln. Die Berechnungslogik ist im SDK inbegriffen, und befindet sich in der `BigInteger GenerateBValues(BigInteger verifier, out BigInteger serverPrivateValue)` Methode. Was im SDK nicht inbegriffen ist, ist die Persistierung von bereits vom Server berechneten Werten. Das Demoprojekt beinhaltet eine `DataCache` Klasse, welche benutzt wird, um eben jene Werte

im Arbeitsspeicher des Servers zu behalten, damit sie später wieder abgerufen werden können. Die Überlegung wie dieses Problem angegangen wird, fällt aber im Endeffekt auf den jeweiligen Entwickler zurück.

Implementierung des DataCache

Diese Sektion ist nur relevant, wenn der wertere Leser die Idee des **DataCaches** aufgreifen möchte. Andernfalls kann mit der nächsten Sektion fortgeführt werden.

Um die Idee des **DataCaches** aufzugreifen, braucht es eine Klasse, welche pro Client gewisse Werte speichern kann. In meiner Implementierung verwende ich hierfür ein **Dictionary<Tkey, TValue>**, wobei als Schlüssel der Client Verifier (**BigInteger**) dient, und als Wert eine Instanz einer Klasse, die sämtliche relevanten Werte für einen Client kapselt, nachfolgend **ClientSessionValues** genannt. Relevante Daten die pro **ClientSessionValues** Instanz gespeichert werden müssen, sind:

- die beiden generierten Server Werte
- der generierten Client Wert
- der Client Benutzername
- der generierte Session Key

Sobald eine Klasse erstellt wurde, die alle fünf dieser Werte kapselt, kann der DataCache Klasse ein privates Feld vom Typ **Dictionary<BigInteger, ClientSessionValues>** hinzugefügt werden.

Da in meinem Fall das Dictionary lediglich in einem privaten Feld gespeichert ist, und nicht öffentlich nach außen sichtbar gemacht wird, braucht es diverse Methoden, um auf Daten in diesem Dictionary zu greifen. Unterstützte Funktionalitäten müssen sein:

- Das Hinzufügen sowie Löschen von einem neuen User Record
- Die beiden generierten Server Werte, public und private, abzufragen
- Die beiden generierten Server Werte, public und private, zu speichern
- Den Session key zu persistieren und wieder abzufragen

Meine eigene Implementierung kann dem Demoprojekt entnommen werden.

Fortsetzung Authentifizierung

Nachdem sichergestellt ist, dass bereits berechnete Werte persistiert und wieder abgefragt werden können, ist von dieser Funktionalität gleich gebrauch zu machen. Wie bereits angesprochen, schickt der Client im Rahmen der Authentifizierung einen einmalig berechneten Wert an den Server. Nach Empfangen dieses Wertes, muss der Server einen Record für diesen Client anlegen, und den eben empfangenen Wert speichern. Anschließend sind per Aufruf der Methode **BigInteger GenerateBValues(BigInteger verifier, out BigInteger serverPrivateValue)** die beiden Serverwerte zu generieren und persistieren. Der öffentliche Wert ist hierbei auch gleichzeitig an den Client zur weiteren Verarbeitung zu retournieren.

Server Proof Generieren

Die letzte benötigte Methode des Controllers ist wiederum eine http POST Methode, welche die Funktionalität bieten soll, dass der Client seine Version des Proofs dem Server zeigt, jener diesen Proof zu validieren versucht, und bei erfolgreicher Validierung, den eigenen Proof zurücksendet. Für diesen Schritt, wie auch für die beiden Schritte davor, wird ein eigenes Model benötigt. In diesem Model müssen der generierte Client Proof, sowie der Benutzername gespeichert sein.

Damit der Server eine eigene Version des Client Proofs berechnen kann, um zu überprüfen ob die eigene Berechnung sich mit der vom Client übermittelten Berechnung deckt, muss zunächst der Session Key berechnet werden. Hierfür ist die Methode `SrpServer.ComputeSessionKey(BigInteger verifier, BigInteger publicClientValue, BigInteger privateServerValue, BigInteger publicServerValue)` zu verwenden. Der verifier sollte im User Objekt in der Datenbank selbst gespeichert sein, die anderen drei Werte im Arbeitsspeicher des Servers (siehe Sektion DataCache). Sobald der Session Key des Servers generiert ist, kann mittels eines Methodenaufrufs von `BigInteger CalculateExpectedClientProof(byte[] sessionKey, byte[] clientPublicValue, byte[] serverPublicValue)` der erwartete Client Proof generiert werden. Der hieraus resultierende Wert ist jetzt mit dem vom Client übermittelten Wert abzugleichen. Decken sich die beiden Werte, hat sich der Client erfolgreich Authentifiziert, und hat mit Sicherheit den selben Session Key generiert, wie der Server. Für den Fall, dass sich die beiden Werte nicht abdecken, muss die Validierung sofort Serverseitig terminiert werden.

Wenn der Client Proof erfolgreich validiert werden konnte, muss der Server nun seine eigene Version des Proofs generieren, und an den Client übermitteln. Die hierfür zu verwendende Methode ist `byte[] CalculateServerProof(byte[] sessionKey, byte[] clientProof, byte[] clientPublicValue)`. Nach Generierung ist dieser Wert an den Server zu schicken, damit der Client den Server verifizieren kann.

Implementierung des Clients

Zuallererst ist das Client Projekt zu erstellen. Für mein Demoprojekt verwende ich eine Konsolenapplikation, grundsätzlich kann hier jedoch nach belieben gewählt werden. Sämtliche benötigte Clientfunktionalität befindet sich in der Klasse `SrpClient` des SDKs. Um diese korrekt nutzen zu können, benötigt es eine Wrapper-Klasse, welche Informationen wie Benutzername und Passwort, sowie die Logik des Netzwerkprotokolls kapselt. Im Demoprojekt ist diese Klasse als `SrpApplicationClient` betitelt. Dieser `SrpApplicationClient` braucht zusätzlich zu diversen Properties für Name, Passwort, sowie eine Instanz des `SrpClients`, Methoden für Login, Registration, sowie Generieren des Proofs. Zusätzlich bietet es sich an, die Logik für das Anfordern des Salts in eine eigene Methode zu verpacken.

```

/// <summary>
/// Requests a salt associated with the specified user name from the server via a HTTP request.
/// </summary>
/// <param name="saltRequestUri">The resource location on the API.</param>
/// <returns>The salt associated with the user.</returns>
public async Task<string> RequestSaltAsync(string saltRequestUri)
{
    if (saltRequestUri == null)
        throw new ArgumentNullException(nameof(saltRequestUri), "Salt request uri must not be null");

    HttpResponseMessage salt;

    using (var client = new HttpClient())
    {
        client.BaseAddress = this.clientBaseAddress;
        salt = await client.GetAsync(saltRequestUri);
    }

    return await salt.Content.ReadAsStringAsync();
}

```

Abbildung 5: Logik zum Anfordern des Salts.

Registration

Je nach gewähltem Netzwerkprotokoll unterscheidet sich die Logik dieser Methoden. Im Fall von http clients empfiehlt es sich, bereits im Konstruktor die Top-Level-Domain des Web APIs mit welchem der Datenaustausch stattfinden soll in einem privaten Feld zu speichern. Dies erleichtert insgesamt die nachfolgenden http Aufrufe. Die Registrationslogik selbst beläuft sich darauf, zunächst das Salt vom Server anzufragen, und anschließend den Verifier zu generieren. Der Verifier kann mithilfe der Methode `BigInteger SrpClient.GenerateVerifier(string username, string password, byte[] salt)` berechnet werden. Ist der Verifier generiert, muss dieser per http Request an den Server übermittelt werden.

```
/// <summary>
/// Registers the client with the server.
/// </summary>
/// <param name="userName">The chosen user name.</param>
/// <param name="password">The chosen password.</param>
public async Task RegisterAsync(string saltRequestUri, string registrationUri)
{
    if (password == null)
        throw new ArgumentNullException(nameof(password), "Password must not be null");

    if (registrationUri == null)
        throw new ArgumentNullException(nameof(registrationUri), "Registration uri must not be null.");

    if (saltRequestUri == null)
        throw new ArgumentNullException(nameof(saltRequestUri), "Salt request uri must not be null.");

    BigInteger verifier;
    byte[] salt;

    using (var client = new HttpClient())
    {
        client.BaseAddress = this.clientBaseAddress;

        salt = Convert.FromBase64String(await this.RequestSaltAsync(saltRequestUri));
        verifier = this.SrpClient.GenerateVerifier(this.Username, this.Password, salt);

        var model = new SrpRegistrationModel(this.Username, salt, verifier.ToByteArray());
        var result = await client.PutAsJsonAsync<SrpRegistrationModel>(registrationUri, model);

        if (!result.IsSuccessStatusCode)
        {
            throw new HttpRequestException("Registration could not be completed.");
        }
    }
}
```

Abbildung 6: Registrations Logik

Authentifizierung

Um den Vorgang der Authentifizierung abzuschließen muss der Client zunächst sein Salt vom Server abfragen, und während der Abfrage seine beiden einmaligen Werte generieren, und den öffentlichen Wert an den Server übermitteln. Die Methode zum generieren der

```
using (var client = new HttpClient())
{
    SrpAuthenticationModel model;

    client.BaseAddress = this.clientBaseAddress;

    saltRequestTask = this.RequestSaltAsync(saltRequestUri);

    valueA = this.SrpClient.GenerateClientValues();
    model = new SrpAuthenticationModel(valueA.ToByteArray(), this.Username);

    authenticationResponse = await client.PostAsJsonAsync<SrpAuthenticationModel>(serverValueExchangeUri, model);
}
```

Abbildung 7: Paralleles abfragen des Salts und generieren der Client Werte

Clientwerte heißt
[BigInteger](#) `SrpClient.GenerateClientValues()`. Nachdem der eigene Wert an den Server übermittelt ist, muss auf den vom Server generierten öffentlichen Wert sowie das Salt gewartet werden, um anschließend den Session Key aus Benutzernamen, Passwort, Salt, sowie dem vom Server an den Client geschickten Wert generiert werden. Der Session Key kann mittels einem Aufruf von `byte[] SrpClient.ComputeSessionKey(string userName, string password, byte[] salt, BigInteger serverPublicValue)` generiert werden.

```
if (!authenticationResponse.IsSuccessStatusCode)
{
    throw new HttpRequestException("Client could not post generated value A to the server.");
}

valueBStringRepresentation = await authenticationResponse.Content.ReadAsStringAsync();
valueB = Convert.FromBase64String(valueBStringRepresentation).ToBigInteger();

saltBase64StringRepresentation = await saltRequestTask;
salt = Convert.FromBase64String(saltBase64StringRepresentation);

// It is important to set the ServerPublicValue Property of the SRP client before attempting to generate
// the client proof.
this.SrpClient.ServerPublicValue = valueB;

// At this point the client has its key, however it does not know as of yet if the server has calculated the same key.
sessionKey = this.SrpClient.ComputeSessionKey(this.Username, password, salt, this.SrpClient.ServerPublicValue).ToByteArray();

return sessionKey;
```

Abbildung 8: Erhalten des Serverwerts sowie des Salts und generieren des Keys.

Verifizierung

Bei der Verifizierung berechnet zuerst der Client seine Version des Proofs, und übermittelt diese an den Server, zur dortigen Verifizierung. Falls die Verifizierung am Server erfolgreich war, übermittelt jener seine eigene Version des Proofs an den Client, wo wiederum eine Validierung stattfinden muss, um zu gewährleisten, dass beide Parteien untereinander verifiziert sind.

Der Client Proof wird mithilfe der Methode `byte[] SrpClient.ComputeProof(byte[] sessionKey)` generiert. Der erwartete Server Proof kann mithilfe der Methode `byte[] SrpClient.GenerateExpectedServerProof(byte[] clientProof, byte[] sessionKey, int padLength)` berechnet werden. Um den Wert für den Parameter *padLength* zu erhalten, ist der

Wert N der SRP-Gruppe (zu finden im Property `SrpClient.SrpGroup.N`) in ein byte array umzuwandeln, und die Länge dieses byte arrays anzugeben.

```
byte[] clientProof;
byte[] serverProof;
byte[] expectedServerProof;
string serverProofStringRepresentation;

Task<HttpResponseMessage> postClientProofTask;
HttpResponseMessage clientProofTaskResult;
var padLength = this.SrpClient.SrpGroup.N.ToByteArray().Length;
SrpProofModel model;

clientProof = this.SrpClient.ComputeProof(sessionKey);
```

Abbildung 9: Variablendeklarationen und Berechnung des Proofs.

```
using (var client = new HttpClient())
{
    client.BaseAddress = this.clientBaseAddress;

    model = new SrpProofModel(clientProof, username);
    postClientProofTask = client.PostAsJsonAsync<SrpProofModel>(validateProofUri, model);
    expectedServerProof = this.SrpClient.GenerateExpectedServerProof(clientProof, sessionKey, padLength);
    clientProofTaskResult = await postClientProofTask;
```

Abbildung 10: Senden des Client Proofs und Berechnung des erwarteten Server Proofs.

```
if (!clientProofTaskResult.IsSuccessStatusCode)
    throw new CryptographicException("Client proof could not be validated on the server.");

serverProofStringRepresentation = await clientProofTaskResult.Content.ReadAsStringAsync();
serverProof = Convert.FromBase64String(serverProofStringRepresentation);

if (serverProof.ToBigInteger() == expectedServerProof.ToBigInteger())
    return true;
else
    return false;
```

Abbildung 11: Validieren des Server Proofs.

Testen der Implementierung

Da die Implementierung nun fertig ist, ist es an der Zeit diese zu testen. Dafür ist in die Eigenschaften der Solution zu navigieren, und sowohl das Serverprojekt, als auch das Clientprojekt als Startprojekte auszuwählen. **Vor dem Testen ist sicherzustellen, dass die Referenzen auf die Bibliotheken BouncyCastle.NetCore 1.8.6, Inferno 1.6.2 sowie Microsoft.AspNetCore.WebApi.Client 5.2.7 hergestellt sind, sowie eine Datenbank existiert in die erstellte Benutzerobjekte persistiert werden können.** Um die Implementierung zu testen, verwende ich folgenden Code.


```
public static async Task Main(string[] args)
{
    var client = new SrpApplicationClient("Hugo", "Coolespassword", new SRP_SDK.SrpClient(new SRP_SDK.SRPGroup()), new Uri("https://localhost:44373/"));
    string saltRequestUri = $"api/user/getsalt/{client.Username}";
    byte[] sessionKey;
    bool isKeyValid;

    await client.RegisterAsync(saltRequestUri, "api/user/registration");
    Console.WriteLine("Client registriert. Drücke Taste um mit Login fortzufahren.");
    Console.ReadKey(true);

    sessionKey = await client.LoginAsync(saltRequestUri, "api/user/login/postvalue");
    Console.WriteLine("Session Key generiert, aber noch nicht verifiziert. Drücke Taste um fortzufahren.");
    Console.ReadKey(true);

    isKeyValid = await client.ComputeProof(sessionKey, "api/user/proof/postproof");

    Console.WriteLine($"Is Session key valid: {isKeyValid}.");
    Console.ReadKey(true);
}
```

Abbildung 12: Code zum testen der Implementierung.

Beim starten der Applikation sehe ich folgenden Output.

```
Client registriert. Drücke Taste um mit Login fortzufahren.
Session Key generiert, aber noch nicht verifiziert. Drücke Taste um fortzufahren.
Is Session key valid: True.
```

Abbildung 13: Output des Tests.

Nach erfolgreicher Registrierung ist auch in der Datenbank ein registrierter Benutzer zu sehen.

ID	Verifier	Username	Salt
4	0x8B7C5F9F7A7...	Hugo	2BF293A15B7319A

Abbildung 14: Der erstellte Benutzer in der Datenbank.

Von nun an kann sich dieser Benutzer mit seinen jeweiligen Daten einloggen, ohne dass das Passwort an den Server übermittelt werden muss. **Achtung, bei erneutem Registrationsversuch mit einem bestehenden Benutzer meldet sich eine Exception, da ein Benutzername jeweils nur einmal vergeben werden kann.**

Senden von Nachrichten

Sämtlicher geschriebene Code befindet sich im Projekt Sikosi. Der Client Code ist zu finden im File **Sample01_SRPCClient**, der Server Code ist zu finden im File **SRPServerAPI**. In der Program

```
Enter message: SendBack:Hello me
Plaintext to server sent: SendBack:Hello me
Encrypted client message: ????>??niK???Zp
?v*??)
??q/?X?@1ZA?^"A?e??B F?????ov@c^@A?@
Encrypted server message: %?@?6@?@?@kH?@?h?@?sawGiB?@?!E?=R?? g???>??>r
?F@Xud#?>>
Plaintext server received: Hello me
Enter message: SendBack:Das ist ein geiler unverschlüsselter Text XYZ!
Plaintext to server sent: SendBack:Das ist ein geiler unverschlüsselter Text XYZ!
Encrypted client message: ?g@+$m? ???f?#? G??zj?@q?t=?S?????_+e0?@iD|.????&??#@@?S????D??????0?P? ???f?@?Zd#??@Y??gr?@P ?'>?
Encrypted server message: ???+??@17?@???_Fd?kaB?N?m?N%?8$?I?@?~@"?N'a@h??~@r?k&Y?q`?r????g@#V?D`???[6?@>?/???@?z????
Plaintext server received: Das ist ein geiler unverschlüsselter Text XYZ!
```

Klasse des Client Projekts, ist nach dem Verifizieren des Session Keys eine Schleife zu finden, die es ermöglicht, testhalber verschlüsselte Nachrichten zwischen Server und Client zu schicken. Das Format einer Nachricht muss hierbei stets sein: ***Nachricht*:*Nachricht***. Der Server schickt sämtlichen Nachrichtentext zurück, der nach dem Doppelpunkt angeführt wird. Der Output besteht pro Nachricht stets aus vier Dingen:

- Dem unverschlüsselten Nachrichtentext des Clients
- Dem verschlüsselten Nachrichtentext des Clients
- Dem verschlüsselten Nachrichtentext des Servers
- Dem unverschlüsselten Nachrichtentext des Servers

6.2 Externe Logins

Dieses Dokument bietet einen Überblick darüber, wie externe Login Anbieter in ein ASP.NET Core Web Projekt eingebunden werden können. Die Anbieter, die besprochen werden sind Google, Facebook, Microsoft und Apple.

Vorweg lässt sich sagen, dass die Anbieter Google, Facebook und Microsoft jeweils sehr ähnlich einzubinden sind, da alle das Protokoll AuthO unterstützen. Lediglich Apple erfordert mehr Schritte.

Alle der besprochenen Anbieter sind in demselben Projekt implementiert, die Basis des Projekts bietet Microsoft.AspNetCore.Identity.²

Die offiziellen Microsoft Docs zu diesem Thema sind [hier](#) zu finden.

Login mit Facebook

Folgende Dinge werden für den Login mit Facebook benötigt:

- Microsoft.AspNetCore.Authentication.Facebook Nuget Package³
- Facebook Developer Account⁴
- Facebook Account um Login zu testen.

Nach dem Einloggen in den Facebook Developer Account, ist auf dem Homescreen auf die Option Neue App Hinzufügen zu klicken, wodurch ein Fenster erscheint in dem verschiedene

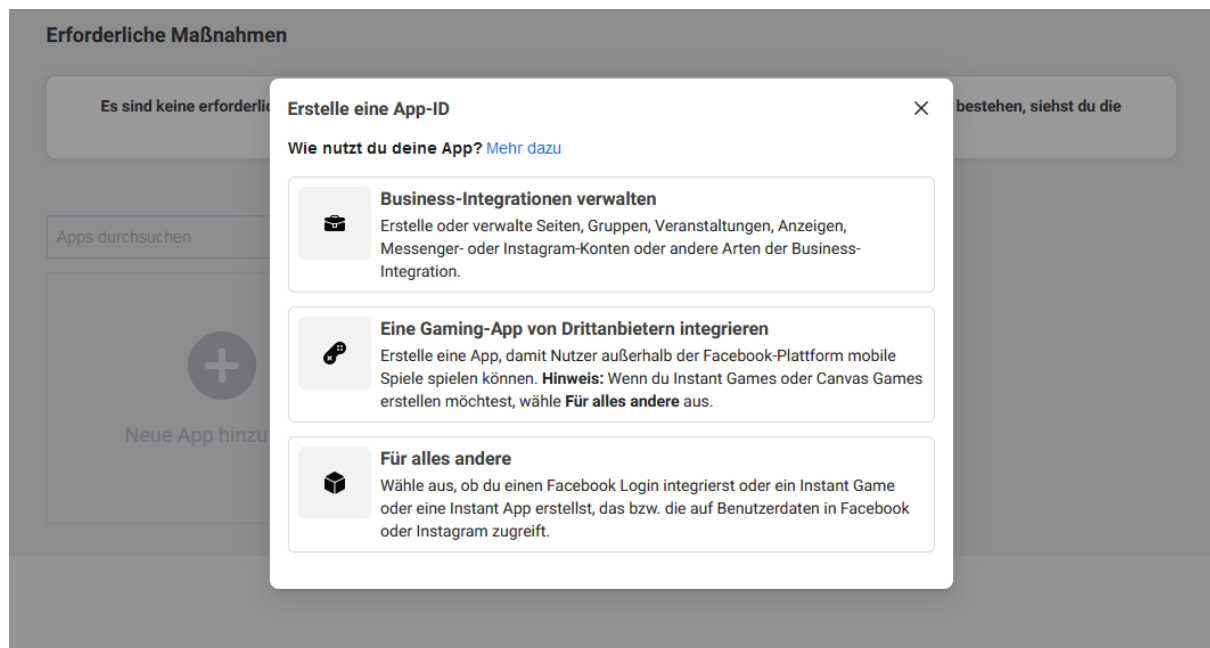


Abbildung 15: Neue App-ID erstellen

Optionen

Applikation mit Facebook Developer registrieren.

auswählbar sind. Um den Facebook Login zu implementieren wird die Option **Für alles andere** benötigt. In dem nun erscheinenden Fenster sind Name der Applikation sowie Kontaktadresse anzugeben. Sind die Applikationsdaten zufriedenstellend eingegeben worden, kann durch

² <https://www.nuget.org/packages/Microsoft.Extensions.Identity.Core/>

³ <https://www.nuget.org/packages/Microsoft.AspNetCore.Authentication.Facebook>

⁴ https://developers.facebook.com/?locale=de_DE

drücken auf den Knopf **App-ID** erstellen und nachfolgendem Sicherheitscheck eine neue Applikation erstellt werden.

Login mit Facebook der Applikation hinzufügen

Damit gelangt man auf eine Seite, auf der verschiedenste Produkte angeboten werden. Navigiert man hier zum Reiter **Facebook Login**, und drückt auf **Einrichten**, öffnet sich ein Dialog, in dem verschiedene Optionen, darunter iOS, Android und Web, angeboten werden. Diese sind jedoch gänzlich zu ignorieren, stattdessen ist links unter dem Reiter **Facebook Login** auf **Einstellungen** zu klicken. Hier öffnen sich die Konfigurationseinstellungen, in denen

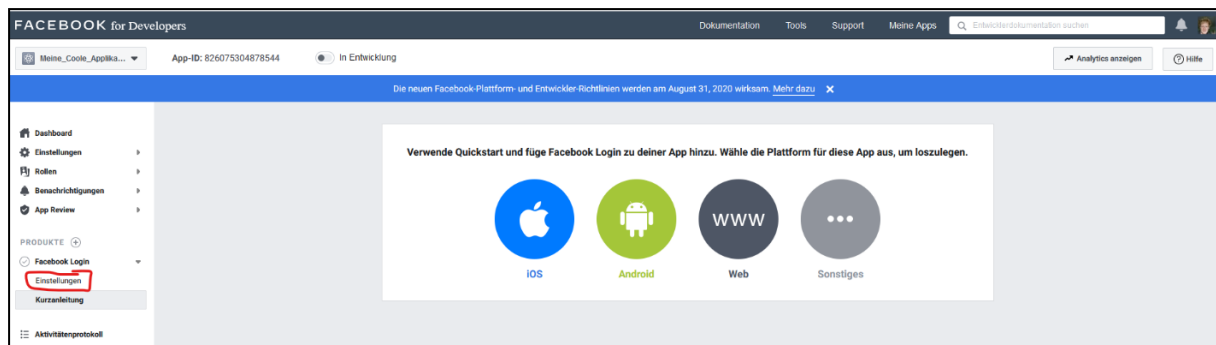


Abbildung 16: Facebook Login hinzufügen

Redirect URIs spezifiziert werden können. Die einzige Option die hier vorerst benötigt wird, ist die Option **Gültige OAuth Redirect URIs**. Hier ist es möglich, für das Testen lokale URLs anzugeben. Dabei ist darauf zu achten, dass an die die spezifizierte URI ein **/signin-facebook** anzuhängen ist. In meinem Testfall ist die Redirect URI <https://localhost:80/signin-facebook>.

App Secret und App ID generieren

Um den Login verwenden zu können, werden ein App Secret, sowie eine App ID benötigt, die im Code der Applikation zu hinterlegen sind. Unter Einstellungen – Allgemeines auf der Homepage der Applikation sind sowohl die App-ID als auch der App-Geheimcode einsehbar.

Login mit Facebook im Code einbinden (Visual Studio)

Um im Code der Web Applikation den Facebook Login zu implementieren sind einige Konfigurationen im **Startup.cs** notwendig.

In der Methode **ConfigureServices**, ist ein Aufruf des APIs **AddAuthentication** notwendig, um ein **AuthenticationBuilder** Objekt zu generieren. Auf diesem **AuthenticationBuilder** muss nun die **AddFacebook** Methode aufgerufen werden. Dieser Methode ist eine anonyme Funktion zu übergeben, welche einen Parameter vom Typ **FacebookOptions** übernimmt. Diese können verwendet werden um eine Vielzahl von Dingen zu konfigurieren. Jedenfalls zu spezifizieren sind jedoch App-ID und App-Secret.

```
services.AddAuthentication()
    .AddFacebook(options => { options.AppId = "Your APP ID"; options.AppSecret = "Your App Secret"; });
```

Facebook Login testen

Startet man nun die Applikation und klickt auf Login, ist neben dem lokalen Login eine Liste mit externen Login Providern zu sehen. In dieser Liste ist Facebook auszuwählen.

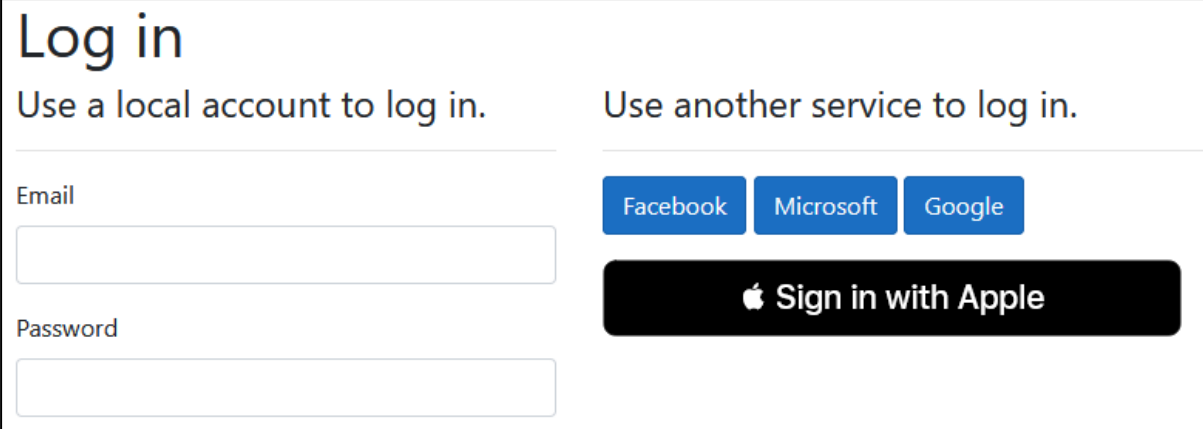


Abbildung 17: Login Fenster mit lokalen und externen Login Möglichkeiten.

Anschließend wird der Benutzer auf die Facebook Login Seite umgeleitet und muss dort den Login Vorgang vollziehen. Nach erfolgreichem Login und akzeptieren der benötigten Rechte der App, ist der Benutzer Authentifiziert.

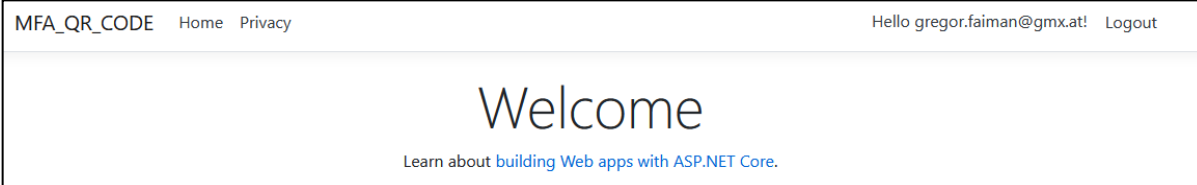


Abbildung 18: Benutzer mit Facebook authentifiziert.

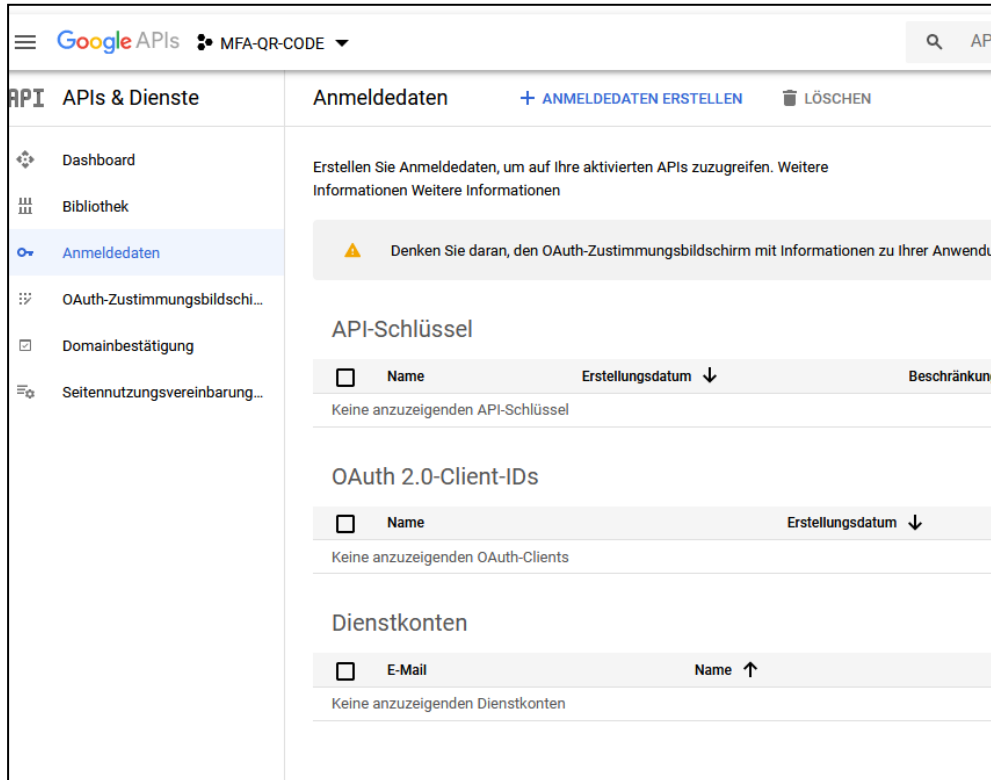
Login mit Google

Folgende Dinge werden für den Login mit Google benötigt:

- Microsoft.AspNetCore.Authentication.Google Nuget Package⁵
- Google Account

⁵ <https://www.nuget.org/packages/Microsoft.AspNetCore.Authentication.Google>

Den ersten Schritt um den Google Login zu implementieren, bildet das Einloggen in den Google Developer Account.⁶ Anschließend ist per drücken auf den Knopf **PROJEKT ERSTELLEN** unter Angabe eines Projektnamens ein neues Projekt anzulegen. Unter dem Menüpunkt



ANMELDEDATEN ERSTELLEN --> **OAuth-Client-ID** ist es möglich, eine APP-ID, sowie ein Client Secret zu generieren, welche für den Login Vorgang von Nöten sind. Bevor Google dies jedoch

Abbildung 19:Anmeldedaten hinzufügen.

zulässt, braucht es einige Konfigurationseinstellungen.

⁶ <https://console.developers.google.com/apis/credentials>

OAuth-Zustimmungsbildschirm

Wählen Sie aus, wie Sie Ihre Anwendung konfigurieren und registrieren möchten, einschließlich Ihrer Anwendergruppe. Sie können nur eine Anwendung mit Ihrem Projekt verknüpfen.

User Type

Intern [?]

Nur für Nutzer aus Ihrer Organisation verfügbar. Sie müssen Ihre Anwendung nicht zur Bestätigung einreichen.

Extern [?]

Für jeden Nutzer mit einem Google-Konto verfügbar.

ERSTELLEN

[Let us know what you think](#) about our OAuth experience

Abbildung 20: Zustimmungsbildschirm konfigurieren.

Autorisierte Domains [?]

Google lässt nur Anwendungen zu, die sich per OAuth für die Nutzung von autorisierten Domains authentifizieren. Die Links Ihrer Anwendungen müssen in autorisierten Domains gehostet werden. [Weitere Informationen](#)

Geben Sie die Domain ein und drücken Sie die Eingabetaste, um sie hinzuzufügen

Domain ungültig: das Protokoll (http:// oder https://) darf nicht angegeben werden.

Link zur Startseite der Anwendung

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Ungültige Domain: URL muss in einer Domain gehostet werden, die im Abschnitt "Autorisierte Domains" aufgelistet ist.

Link zur Datenschutzerklärung der Anwendung

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Ungültige Domain: URL muss in einer Domain gehostet werden, die im Abschnitt "Autorisierte Domains" aufgelistet ist.

Link zu den Nutzungsbedingungen der Anwendung (Optional)

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Abbildung 21: Localhost ist keine gültige URL

Zunächst gilt es einen User Typ auszuwählen, die Möglichkeiten hierfür belaufen sich auf Intern und Extern. Nach auswählen des Punktes Extern, müssen einige URLs spezifiziert werden. Benötigt sind ein Link zur Startseite der Applikation, ein Link zur

Autorisierte Domains [?]

Google lässt nur Anwendungen zu, die sich per OAuth für die Nutzung von autorisierten Domains authentifizieren. Die Links Ihrer Anwendungen müssen in autorisierten Domains gehostet werden. [Weitere Informationen](#)

Geben Sie die Domain ein und drücken Sie die Eingabetaste, um sie hinzuzufügen

Link zur Startseite der Anwendung

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Link zur Datenschutzerklärung der Anwendung

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Link zu den Nutzungsbedingungen der Anwendung (Optional)

Wird auf dem Zustimmungsbildschirm angezeigt. Muss in einer autorisierten Domain gehostet werden.

Speichern

Abbildung 22: Mit ngrok kann ein Tunnel ins WWW erzeugt werden.

Datenschutzerklärung der Applikation, sowie eine autorisierte Top Level Domain, die dem Entwickler gehört und auf der die Applikation läuft.

Da sich mein Testprojekt auf dem lokalen Rechner befindet, und localhost Adressen nicht gültig sind, verwende ich das Tool ngrok⁷, um einen Tunnel von meinem lokalen Rechner ins WWW zu hosten, damit ich die von ngrok generierte Adresse, als Domain verwenden kann.

Nach Speichern der Einstellungen ist es nun möglich, beim Menüpunkt Anmeldedaten neue Anmeldedaten zu erstellen. Im anschließend aufscheinenden Fenster sind der Typ der Anwendung, der Name des Clients, sowie eine Weiterleitungs-URI anzugeben. Bei der Weiterleitungs-URI ist es möglich, eine localhost Adresse anzugeben, jedenfalls muss an die Adresse ein /signin-google angehängt werden.

Ab sofort können sowohl die Client ID, als auch das Client Secret abgerufen werden.

Implementieren des Google Sign In im Code

Im letzten Schritt ist im `Startup.cs` im Code bei der Methode `ConfigureServices`, nach einem Aufruf von `services.AddAuthentication()`, wie auch beim Facebook Login, das zuständige API, hier `AddGoogle(Action<Microsoft.AspNetCore.Authentication.GoogleOptions> configureOptions)` aufzurufen. In den Optionen sind jedenfalls die Properties Client ID und Client Secret zu setzen.

Login mit Microsoft

Folgende Dinge werden für den Login mit Microsoft benötigt:

- Microsoft.AspNetCore.Authentication.Microsoft Nuget Package⁸
- Microsoft Account

Da der Prozess des Implementierens eines Microsoft Logins ähnlich jener Prozesse von Google und Facebook ist, werden die Schritte nicht mehr im Detail beschrieben. Um den Login einzubinden müssen die Schritte App registrieren, App Geheimnis und ID generieren, und App ID sowie Geheimnis im Code einfügen, durchgeführt werden.

App registrieren

- Navigation zur App Registrations Seite des Azure Web Portals⁹
- Erstellen einer neuen App mittels klicken auf Neue Registrierung
- Namen und gewünschten Kontotyp auswählen, in meinem Fall Option 2
- Umleitungs URL spezifizieren und /signin-microsoft anhängen
- Registrieren drücken

App Geheimnis und ID generieren

- Auf der Homepage des Projekts im Menü links Zertifikate & Geheimnisse auswählen
- Mit dem Knopf Neuer geheimer Clientschlüssel ein neues Secret anlegen.
- Nach dem erstellen des Secrets unbedingt dieses persistieren, da wenn die Seite neu geladen wird, das Geheimnis nicht mehr ersichtlich ist.
- Die Client ID ist unter dem Menüpunkt Übersicht einsehbar, und ist als Anwendungs-ID betitelt.

Den Code konfigurieren

⁷ <https://ngrok.com/download>

⁸ <https://www.nuget.org/packages/Microsoft.AspNetCore.Authentication.MicrosoftAccount/>

⁹ <https://go.microsoft.com/fwlink/?linkid=2083908>

- Wie auch bei Google und Microsoft, nach dem Aufruf von **AddAuthentication**, die Methode **AddMicrosoft** rufen, und ihr in den Optionen die client ID, und das Client Secret übergeben.

Login mit Apple

Apple Login noch nicht vollständig dokumentiert. Als Hilfe zu Implementierung siehe folgende Links:

<https://sarunw.com/posts/sign-in-with-apple-4/>

<https://sarunw.com/posts/sign-in-with-apple-3/#how-to-verify-the-token>

https://developer.apple.com/documentation/sign_in_with_apple/sign_in_with_apple_js/configuring_your_webpage_for_sign_in_with_apple

Den Apple Login zu implementieren ist leider vergleichsweise aufwändig und mit einigen zusätzlichen Schritten verbunden. Um den Login zu implementieren sind folgende Dinge nötig:

- Apple ID zum Testen vom Login
- Apple Developer Account
- Einen Webserver auf dem die Applikation gehostet wird, oder ein Tool, das die lokale Website nach außen tunnelt

Um einen Login mit Apple ID in die eigene Applikation einzubauen, sind jene Schritte notwendig:

- Erstellen eines Apple Developer Accounts, wenn noch nicht vorhanden
- Erstellen einer App-ID mit zugehöriger Service-ID
- Hinzufügen eines Apple Logins zur erstellten Service-ID

In den nachfolgenden Sektionen wird auf sämtliche benötigte Schritte genauer eingegangen.

Erstellen der Service-ID

Ähnlich wie bei anderen besprochenen Login-Anbietern ist es auch bei Apple notwendig, die eigene Applikation mit Apple zu registrieren, um einen Login in dieser Applikation nutzen zu können. Dies ist nach Einloggen in den Apple Developer Account unter der Option **Certificates, Identifiers & Profiles** möglich. Hier sind eine Beschreibung sowie ein Identifier anzugeben. Dieser Identifier hat die gleiche Funktion, wie die Client-ID bei den anderen Anbietern.

Apple Login konfigurieren

Im Menü in dem die Service-ID zu konfigurieren ist, existiert ein Eintrag mit Namen **Sign In with Apple**, mit einem **Configure** Knopf daneben. Durch drücken auf diesen Knopf gelangt