

State of the Art Webapps mit Blazor

Gamification Content selber Hosten

Bachelorarbeit

Zur Erlangung des
Bachelor of Education (BEEd)

an der School of Education der
Paris-Lodron-Universität Salzburg



Eingereicht von
JOHANNES BAISCHER
12127340

Betreuer: Dr. Rudolf Schraml
Fachbereich: Computer Science

Salzburg, September 2025

Inhaltsverzeichnis

1	Vorwort	III
2	Abstract	IV
3	Das Blazor Framework	1
3.1	Getting Started	1
4	Server Side	5
4.1	Startup-Logik	5
4.2	SignalR	8
4.3	Authentifizierung und Autorisierung	13
4.4	Email Verifizierung und Verschlüsselung	21
5	Hosting	24
5.1	Basic Linux-ASP.NET Hosting	24
5.2	Reverse Proxy mit Apache2	26
5.3	Domain und Zertifikate	28
5.4	PostgreSQL Datenbank	29
6	Client Side	36
6.1	WebAssembly	36
6.2	MVC und MVVM Patterns	38
6.3	Blazor Syntax	40
6.4	Dependency Injection	43
6.5	JS Interop	46
6.6	File Management	48
6.7	Localization	53
6.8	Benutzeroberfläche	55
7	Cross Platform Development	64
7.1	Razor Class Library	64
7.2	Blazor MAUI	67
7.3	Plattform abhängiger Code	68
8	Gamification mit H5P-Content	75
8.1	Gamification Konzept	75
8.2	H5P-Content	76
8.3	Lumi H5P Editor	79
8.4	H5P-Standalone	83

9	Fazit	86
	Literatur	vi

1 Vorwort

Die Informatik ist eine der am schnellsten wachsenden Disziplinen in der heutigen Welt. Dieses, im Kontext der Weltgeschichte, extrem junge Fachgebiet, entwickelt sich in allen Bereichen rasant weiter. Seit der Einführung der ersten „Webtechnologien“ vor etwas mehr als 50 Jahren, hat sich die Art und Weise, wie wir Informationen austauschen, kommunizieren und interagieren, grundlegend verändert. Die Anforderungen wurden komplexer, dynamischer, schneller und benutzerorientierter. Hier am Zahn der Zeit zu bleiben und seine begrenzte Ressourcen sinnvoll einzusetzen, ist die wichtigste Tugend der heutigen Webentwickler:innen. In dieser Arbeit und dem damit verbundenen Projekt habe ich versucht, diesen Anforderungen gerecht zu werden und mit einem in den letzten Jahren aufstrebendes Framework – dem Blazor Framework – eine moderne, leistungsstarke Webanwendung zu entwickeln. Als besonderer Anwendungsfall dient die Bereitstellung von interaktiven Lerninhalten für Schüler:innen.

Besonderer Dank gilt meinem Betreuer, Dr. Rudolf Schraml, für die Möglichkeit, dieses Thema und Projekt zu bearbeiten und für die reibungslose Betreuung dieser Arbeit und seine wertvolle Unterstützung und Rückmeldungen.

Johannes Baischer
Salzburg, 4. September 2025

(Blazor, ASP.NET, C#, H5P, Gamification)

2 Abstract

Thema dieser Arbeit ist die Entwicklung einer modernen Anwendung für das Bereitstellen von interaktiven Lerninhalten mittels Microsoft Blazor WebAssembly. Konkret werden interessante Lösungen für die Implementierung verschiedenster Komponenten vorgestellt, die eine rollenbasierte Authentifizierung mit ASP.NET Identity und eine sichere Verwaltung von Nutzerdaten sowie Speicherung dieser in einer PostgreSQL-Datenbank ermöglichen. Weiters wird die Bereitstellung der Applikation durch einen selbst konfigurierten Linux-Server mithilfe von Apache2 als Reverse Proxy und DuckDNS zusammen mit Let's Encrypt für eine verschlüsselte HTTPS-Verbindung erleutert. Ziel ist es, interaktive H5P-Inhalte zu integrieren und dadurch gamifizierte Lernmodule zu erstellen, mit etwa Multiple-Choice-Quizzes oder Drag-and-Drop-Aufgaben. Wir werden das verwendete Framework Blazor auch auf seine Wiederverwendbarkeit von Code untersuchen, um eine plattformübergreifende Nutzung zu ermöglichen, etwa als Webanwendung oder Windows-App mittels .NET MAUI. Besondere Herausforderungen, wie eine tokenbasierte Kommunikation zwischen Client und Server oder Responsive Design, werden durch die geschickte Verwendung aller durch die ASP.NET Core Plattform bereitgestellten Funktionen und günstige Strukturierung des Codes elegant gelöst.

Der gesamte Code ist in einem Git-Repository verfügbar und kann unter <https://github.com/Johannes-Baischer/State-of-the-Art-Webapps-mit-Blazor> eingesehen werden.

3 Das Blazor Framework

Durch die Tatsache, dass Blazor eine relativ neue Technologie ist und sich doch sehr grundlegend von anderen Webtechnologien unterscheidet, gibt es noch nicht viel umfassende Literatur zu diesem Thema. Daher wurde bei der Entwicklung und wird bei der Erstellung dieser Arbeit hauptsächlich auf die umfassende offizielle Dokumentation von Microsoft zurückgegriffen und zu gegebener Stelle auf passende Online-Ressourcen verwiesen [15].

Zu aller erst wollen wir uns mit der Frage beschäftigen, was Blazor eigentlich ist und welche Ziele es verfolgt. Kurz gesagt ist Blazor Microsofts Antwort auf die unzähligen, meist auf JavaScript basierenden Webframeworks, die in den letzten Jahren das Web dominiert haben. Es gab von verschiedenen Seiten immer wieder Versuche, alternative Ansätze zu entwickeln, die sich alle mehr oder weniger durchsetzen konnten. Bis heute bleibt JavaScript die dominierende Sprache für die Entwicklung von Frontend-Anwendungen, mit all seinen Vor- und Nachteilen. Es kam durch den Trend vom Wechseln von Frontend- und Backend-Entwickler:innen hin zu Fullstack-Entwickler:innen aber immer wieder der Wunsch auf, eine einheitliche Programmiersprache für die gesamte Webentwicklung zu haben. Mit den besonderen Anforderungen an Backend-Server, die mit Datenbanken, Schnittstellen und Dateimanagement in einem sicher und vor allem einheitlich umgehen müssen, fühlten sich viele Entwickler:innen mit JavaScript als nicht typstrenge Programmiersprache nicht ganz wohl. Steve Sanderson und sein Team bei Microsoft haben sich daraufhin die Aufgabe gestellt, ihre solide Backend-Technologie ASP.NET Core mit einer Frontend-Technologie zu vereinen, die es ermöglicht, Webanwendungen in der typenstrengen Sprache C# zu schreiben. Das Ergebnis ist Blazor, ein Framework, das es ermöglicht, Webanwendungen sowohl auf der Serverseite als auch auf der Clientseite in C# zu entwickeln. Seit der Veröffentlichung 2019 hat sich Blazor durch die reibungslose Integration der bestehenden ASP.NET Anwendungen und die Möglichkeit, bestehende C#-Bibliotheken zu nutzen, schnell etabliert und wird mittlerweile von vielen Unternehmen und Entwickler:innen weltweit eingesetzt [1]. Für die clientseitige App-Logik steht die entwickelte Razor-Syntax zur Verfügung, die es ermöglicht, HTML und C#-Code in einer Datei zu kombinieren. Dies erlaubt das nahtlose Einbinden von Ablauf- und Datenstrukturen innerhalb der HTML-Struktur und erleichtert so das Schreiben von dynamischen Webanwendungen.

3.1 Getting Started

Für die Entwicklung von Blazor-Anwendungen empfiehlt es sich, eine beliebige, nicht zu alte Version von Visual Studio zu verwenden. Es ist natürlich auch möglich, einen beliebigen Code-Editor zu verwenden, da aber viele Quality-of-Life-Features wie das dynamische Nachladen von Razor-Komponenten, Syntax-Highlighting und IntelliSense und besondere

Debugging-Features nur in Visual Studio verfügbar sind, wird die Verwendung von Visual Studio empfohlen. Auch das initiale Erstellen eines Blazor-Projekts ist in Visual Studio sehr einfach. Nach der erfolgreichen Installation von Visual Studio durch Auswahl der benötigten Komponenten (= ASP.NET, C# und Blazor) kann ein neues Projekt erstellt werden. Hier stehen für Blazor zwei verschiedene Projektvorlagen zur Verfügung: Blazor Server und Blazor WebAssembly. Auch wenn sich die beiden Vorlagen in der Entwicklung und letztendlich in der Anwendung sehr ähnlich sind, gibt es doch einige grundlegende Unterschiede, die bei der Auswahl der richtigen Vorlage berücksichtigt werden sollten. Blazor Server ist dabei die klassische ASP.NET Core Webanwendung [12], bei der die Logik auf dem Server ausgeführt wird und die Benutzeroberfläche über das sog. „SignalR“-Protokoll an den Client übertragen wird. Damit ist Blazor Server die beste Wahl für alle Entwickler, die bereits Erfahrung mit traditionellen Server-Client-Webanwendungen haben und diesem Modell treu bleiben wollen. Mit Blazor WebAssembly, der zweiten Projektvorlage, stößt Microsoft in bislang selten genutzte Features des modernen Webs vor. WebAssembly ist ein offener Standard, der es ermöglicht, Programmcode in vom World-Wide-Web-Konsortium (W3C) standardisierten Formaten zu kompilieren und im Browser auszuführen. Seit 2017 wird WebAssembly als Standard in Chrome, Edge, Firefox und Webkit Browsern unterstützt, und 2019 in der Version 1 offiziell festgelegt. Es entstehen durch die Verwendung von WebAssembly einige interessante Möglichkeiten, die vor allem für Apps mit sehr komplexer Anwendungslogik interessant sind. In der Praxis muss hier nicht bei jeder Änderung der dargestellten Inhalte auf der Website der Server kontaktiert werden, da die Ablauf- und Renderlogik im Bytecode (beim erstmaligen Laden der Seite) bereits an den Client übermittelt worden ist. Das hat für den Nutzer den Vorteil einer schnelleren Reaktionszeit und für den Anbieter den Vorteil, dass die Leistungsanforderungen an den Server mit steigender Nutzerzahl nicht stark steigt. Im Gegenzug muss der Client mit dem verwendeten Endgerät und Webbrowser die nötige Leistung aufbringen, um die WebAssembly-Anwendung auszuführen. Das ist in der Regel aber kein Problem, da die meisten modernen Endgeräte und Browser leistungsstark genug sind, um WebAssembly-Anwendungen problemlos auszuführen. Auch muss beim ersten Aufruf der Website die gesamte Anwendung an den Client übertragen werden, was im Vergleich zu Blazor Server zu einer längeren Ladezeit führen kann. Das ist aber in der Regel nur beim ersten Aufruf der Website der Fall, da die Anwendung danach im Browser-Cache gespeichert wird und bei späteren Aufrufen nicht erneut geladen werden muss, sollte sich die Anwendung nicht verändert haben. Mit diesem Wissen und einigen Details, die wir im Kapitel 6 *Client Side* noch genauer betrachten werden, lassen wir uns nun eine neue Blazor WebAssembly Anwendung erstellen.

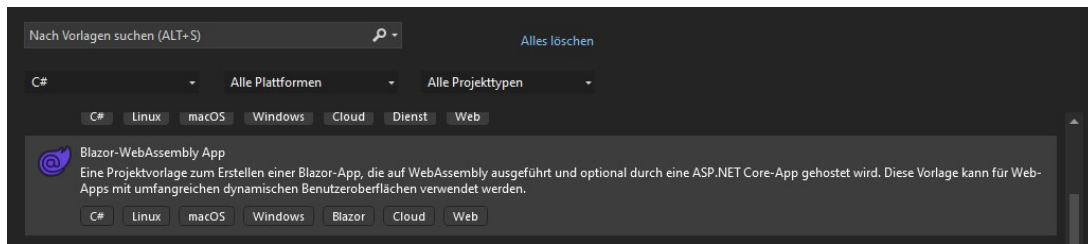


Abbildung 1: Erstellen der WebAssembly-App

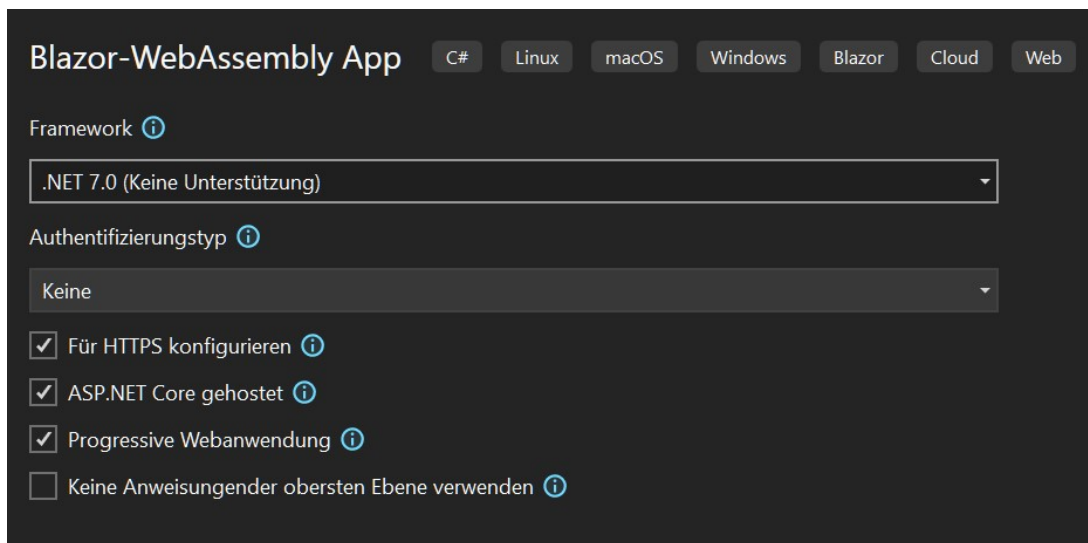


Abbildung 2: WebAssembly-App Konfiguration

In Abbildung 2 sehen wir einige Optionen, die bei der Erstellung der in Abbildung 1 ausgewählten WebAssembly-App konfiguriert werden können. Die wichtigsten Optionen sind hier die Auswahl des HTTPS-konformen Templates und der ASP.NET Core Hosted Option, welche zusammen mit der Client-Anwendung gleich eine ASP.NET Core Server-Anwendung erstellt, die die WebAssembly-App bereitstellt. Würde man diese Option abwählen, könnte man wie gewöhnlich jede andere Art von Hosting-Server verwenden, wie es auch bei traditionellen Webanwendungen der Fall ist. Da wir aber den besonderen Vorteil einer gemeinsamen Codebasis für die Client- und Server-Anwendung nutzen wollen, wählen wir diese Option aus. Die letzte Option, die „Progressive Web App“ Option, ist eine interessante Möglichkeit, die es ermöglicht, die WebAssembly-App als Progressive Web App (PWA) zu erstellen. Das bedeutet, dass die App im Browser die Möglichkeit anbietet, sich auf dem Endgerät zu installieren und wie eine native App funktioniert. Würde man besonderen Wert darauf legen, die Anwendung auch offline verfügbar zu

machen, sollte man diese Option unbedingt auswählen. In unserem Fall wollen wir für bestimmte Datenbankabfragen und Dateizugriffe ohnehin Kontakt zum Server halten, wodurch eine Offline Variante nicht funktionieren würde. Nichtsdestotrotz werden wir die PWA-Funktionalität aktivieren, um den Nutzer:innen die Möglichkeit zu geben, die Anwendung auf ihrem Endgerät zu installieren und wie eine native App zu verwenden (ähnlich Installation einer App aus dem App Store).

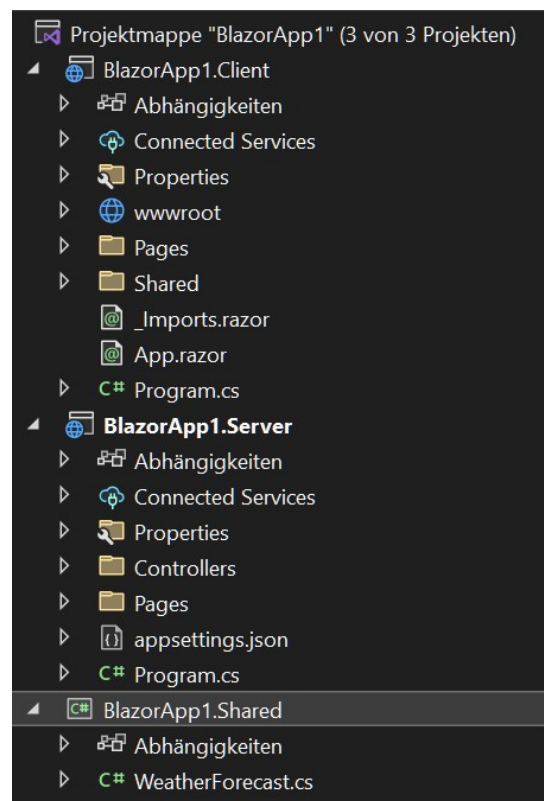


Abbildung 3: Blazor WebAssembly Projektvorlage

Damit erhalten wir wie in Abbildung 3 ersichtlich eine funktionstüchtige Blazor WebAssembly Anwendung, mit drei Bestandteilen: der Client-Anwendung, der Server-Anwendung und einer gemeinsamen Code-Bibliothek, die es ermöglicht, Code zwischen der Client- und Server-Anwendung zu teilen. In weiterer Folge werden wir diese Bibliothek zur Blazor spezifischen sog. „Razor Class Library“ umbauen, die es ermöglicht, auch ganze Razor-Komponenten zu teilen. Im nächsten Schritt werden wir den Sprung von dieser Einführung zum derzeitigen Stand der Anwendung machen und die wichtigsten Komponenten, Features und Einstellungen der Anwendung vorstellen.

4 Server Side

Um den logischen Ablauf der Anwendung am besten zu verstehen, betrachten wir zu aller erst die serverseitige Komponente. Durch die oben erwähnte Entscheidung, hin zur WebAssembly Version von Blazor, beschäftigt sich unser Server mit nur einigen wenigen Aufgaben. Diese werden wir im Detail in den folgenden Unterkapiteln betrachten.

4.1 Startup-Logik

Die Startup-Logik der Anwendung ist in der Datei `Program.cs` zu finden (siehe Listing 1). Hier wird die ASP.NET Core Webanwendung initialisiert und konfiguriert. Die wichtigsten Schritte sind die Konfiguration der verwendeten Dienste (=vorgefertigte Bibliotheken), die Konfiguration der Middleware und die Konfiguration der Routen und Ressourcen, auf die die Anwendung zugreifen können soll.

```
1 var builder = WebApplication.CreateBuilder(args);
2
3 // Add configuration files
4 builder.Configuration
5     .AddJsonFile("appsettings.json")
6     .AddJsonFile("secrets.json")
7     .AddUserSecrets(Assembly.GetExecutingAssembly(), true);
8
9 //Authorization and Authentication
10 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
11     .AddJwtBearer(options =>
12     {
13         //... Configure JWT Bearer authentication
14     });
15 builder.Services.AddAuthorization(options =>
16 {
17     options.AddPolicy("EmailIsVerified", policy =>
18     {
19         //... Define policy for email verification
20     });
21 });
22
23 // Add services to the container.
24 builder.Services.AddControllersWithViews();
25 builder.Services.AddRazorPages();
26 builder.Services.AddOptions();
27
28 //...
```

Listing 1: Startup-Logik (Server): program.cs 1/2

Der Server richtet im Listing 1 als Erstes alle benötigten Ressourcen ein, auf die er zugreifen soll, wie etwa Konfigurationsdateien oder Informationen über die verwendete Datenbank. Danach wird die Authentifizierung und Autorisierung der Anwendung konfiguriert und zuletzt werden vom Framework benötigte Default-Dienste hinzugefügt.

```
1  //...
2
3  var app = builder.Build();
4
5  // Configure the HTTP request pipeline.
6  if (app.Environment.IsDevelopment())
7  {
8      app.UseWebAssemblyDebugging();
9  }
10 else
11 {
12     app.UseExceptionHandler("/Error");
13     app.UseHsts();
14 }
15
16 app.UseHttpsRedirection();
17
18 app.UseBlazorFrameworkFiles();
19 app.UseStaticFiles(new StaticFileOptions
20 {
21     // Adds static file serving for all file types (e.g. .h5p)
22     ServeUnknownFileTypes = true
23 });
24 app.UseRequestLocalization(LocalizationInfo.Instance.GetRequestLocalizationOptions());
25
26 app.UseRouting();
27 app.UseAuthentication();
28 app.UseAuthorization();
29
30 app.MapRazorPages();
31 app.MapControllers();
32 app.MapFallbackToFile("index.html");
33
34 app.Run();
```

Listing 2: Startup-Logik (Server): program.cs 2/2

In weiterer Folge erstellt der Server die Server-App in Listing 2 mit der *builder.Build()*; Methode. Daraufhin kann er alle weiteren Konfigurationen zur Bereitstellung der Client-Application vornehmen. Darunter fällt etwa die Konfiguration des Dateimanagements, einbinden der Authentifizierung und Autorisierung, die Konfiguration der Routen und die Bereitstellung der statischen Dateien. Schlussendlich wird die Anwendung mit der *app.Run()*; Methode gestartet.

4.2 SignalR

Nachdem unser Server nun konfiguriert und bereit ist, die Blazor WebAssembly Anwendung auszuliefern, stellt sich noch die Frage, wie die Kommunikation zwischen dem Server und der Client-Anwendung funktioniert. Hier müssen wir abermals zwischen den zwei Blazor-Typen unterscheiden. Da bei der Blazor-Server Variante die gesamte Logik am Server ausgeführt wird, braucht der Client eine Möglichkeit, über vollendete Prozesse oder Änderungen an der Benutzeroberfläche Bescheid zu bekommen. Hier wird im Framework das von Microsoft entwickelte, freie und quelloffene System „SignalR“ verwendet. Damit hat der Server die Möglichkeit, asynchrone Benachrichtigungen an der Client zu senden, wenn beispielsweise die Oberfläche neu gerendert wurde und neu geladen werden muss. Auch sämtliche API-Abfragen laufen hier über dieses System. In Abbildung 4 ist die Architektur von SignalR in einer Blazor-Server Anwendung dargestellt.

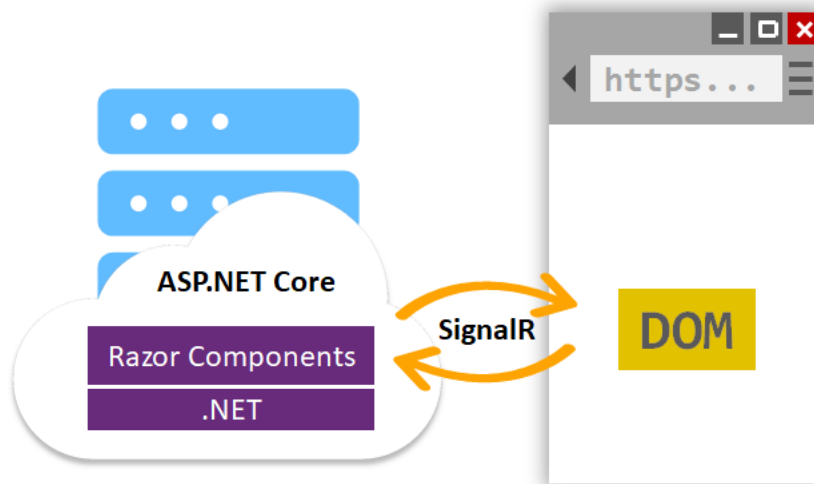


Abbildung 4: SignalR Architektur [11]

Da es sich bei Blazor-WebAssembly um die vom Server unabhängige Variante handelt, ist man hier nicht standardmäßig an das SignalR Protokoll gebunden. Wir erinnern uns, dass wir zum Erstellzeitpunkt des Projekts die Auswahlmöglichkeit hatten, ob wir überhaupt gemeinsam mit dem Client eine Serverapplikation erstellen wollen. Es würde sich auch hier sehr einfach gestalten, eine SignalR-Verbindung zwischen Server und Client aufzubauen. Für unsere Zwecke reicht aber eine herkömmliche API-Schnittstelle vollkommen aus und erleichtert uns zudem die Logik für die Übermittlung von beispielsweise Datenbankabfragen ungemein. [19]

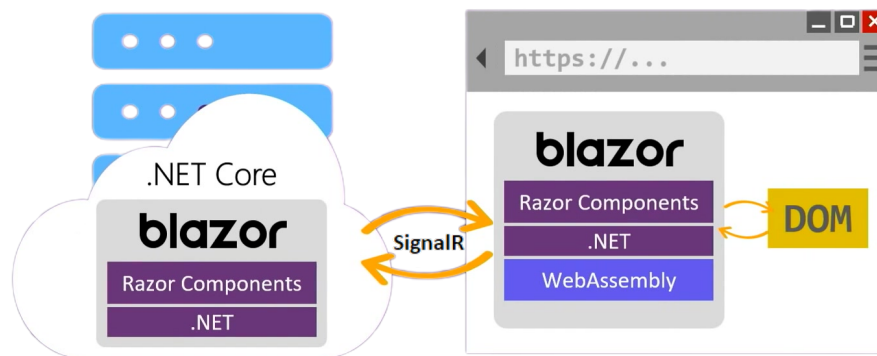


Abbildung 5: SignalR Architektur [22]

Wie in Abbildung 5 zu sehen ist, wird bei Blazor WebAssembly die SignalR-Verbindung beispielhaft verwendet, um mit der Client-Anwendung, welche direkt im Browser läuft und somit direkt Zugriff auf die Benutzeroberfläche hat, zu kommunizieren. Diese Kommunikation könnte im Diagramm durch eine beliebige API-Schnittstelle oder andere Art der Server-Client-Kommunikation ersetzt werden. In unserem Fall befindet sich hier am Server eine ASP.NET Core Controller-Schnittstelle, welche beispielsweise für Datenbankabfragen oder Dateizugriffe kontaktiert werden kann.

```

1 namespace BlazorWasmHost.Controller.PostgreSQL
2 {
3     /// <summary>
4     /// API Controller for BlazorWasm that calls Methods as Server
5     /// </summary>
6     [ApiController]
7     [Route("internal/[controller]/[Action]")]
8     public class UserCredentialsController : ControllerBase
9     {
10         [HttpPost]
11         [Authorize(Roles = "System, Student, Teacher")]
12         [ActionName("Insert")]
13         [ProducesResponseType(typeof(UserCredential), 201)]
14         public async Task<IActionResult?> Internal_UserCredentials_Insert_Post(
15             [FromBody] JsonElement body)
16         {
17             //...
18         }
19
20         [HttpPost]
21         [Authorize(Roles = "Student, Teacher")]
22         [ActionName("Upsert")]
23         [ProducesResponseType(typeof(UserCredential), 200)]
24         public async Task<IActionResult?> Internal_UserCredentials_Upsert_Post(
25             [FromBody] JsonElement body)
26         {
27             //...
28         }
29
30         [ActionName("Read")] //...
31         [ActionName("ReadSalt")] //...
32         [ActionName("CheckUserExists")] //...
33         [ActionName("CheckPasswordResetTokenMatch")] //...
34         [ActionName("ResetPassword")] //...
35     }
36 }

```

Listing 3: Skeleton des User Credential Controllers

Die Schnittstelle für Datenbankabfragen weist derzeit sieben Endpunkte auf, die erlauben, bestimmte Informationen über die Nutzer:innen zu erhalten oder auch zu verändern. Listing 3 zeigt hier den Aufbau des zuständigen Controllers. Hier bedienen wir uns der idealen Einbindung der über Jahre erprobten ASP.NET Technologien für API-Schnittstellen und Controller, welche durch den typensicheren Umgang mit den Daten und die Kombination mit Clientseitigem C# Code eine sehr einfache und übersichtliche Schnittstelle ermöglichen.

Sowohl in der Nutzung durch den Client, welcher die Endpunkte ohne unerwartete Artefakte bei der Typkonvertierung nutzen kann, als auch in der Entwicklung ohne die Notwendigkeit, Formatfehler oder fehlerhafte Konvertierungen debuggen zu müssen, wird einem so das Leben erleichtert. Verdeutlicht wird das in Listing 4, in welchem typenstrenge Dienste über alle Endpunkte hinweg zur Verfügung gestellt werden.

```
1 namespace BlazorWasmHost.Controller.PostgreSQL
2 {
3     /// <summary>
4     /// API Controller for BlazorWasm that calls Methods as Server
5     /// </summary>
6     [ApiController]
7     [Route("internal/{controller}/{Action}")]
8     public class UserCredentialsController : ControllerBase
9     {
10         private SharedComponents.PostgreSQL.UserCredentialsController
11             GetSharedController(JsonElement services)
12         {
13             IPlatformInfo platformInfo = services[0].Deserialize<WebPlatformInfo>();
14             HttpClient httpClient = services[1].Deserialize<HttpClient>();
15
16             string token = HttpContext.Request.Headers["Authorization"]
17                 .ToString().Replace("Bearer ", "");
18             httpClient.DefaultRequestHeaders.Authorization
19                 = new AuthenticationHeaderValue("Bearer", token);
20
21             return new SharedComponents.PostgreSQL.UserCredentialsController(
22                 platformInfo, httpClient, null);
23         }
24
25         ///... Read UserCredential method
26     }
27 }
```

Listing 4: User Credential Controller

```

1 namespace BlazorWasmHost.Controller.PostgreSQL
2 {
3     /// <summary>
4     /// API Controller for BlazorWasm that calls Methods as Server
5     /// </summary>
6     [ApiController]
7     [Route("internal/[controller]/[Action]")]
8     public class UserCredentialsController : ControllerBase
9     {
10         ///... GetSharedController method
11
12         [HttpPost]
13         [Authorize(Roles = "System, Student, Teacher")]
14         [ActionName("Read")]
15         [ProducesResponseType(typeof(UserCredential), 200)]
16         public async Task<IActionResult> Internal_UserCredentials_Read_Post(
17             [FromBody] JsonElement body)
18         {
19             var services = body[0];
20             var args = body[1];
21
22             //calls SharedComponents Controller
23             SharedComponents.PostgreSQL.UserCredentialsController ucc
24                 = GetSharedController(services);
25             //calls Methode as Server (callerIsServer = true)
26             UserCredential? uc = null;
27             HttpStatusCode? statusCode = HttpStatusCode.OK;
28
29             try
30             {
31                 uc = await ucc.Read(
32                     args[0].Deserialize<string>(),
33                     args[1].Deserialize<string>(),
34                     true);
35             }
36             catch (HttpRequestException ex)
37             {
38                 statusCode = ex.StatusCode;
39             }
40             return StatusCode((int)statusCode, uc);
41         }
42     }
43 }

```

Listing 5: Read Endpunkt des User Credential Controllers

Betrachten wir nun exemplarisch den in Listing 5 dargestellten Read-Endpunkt, der es

dem Client ermöglicht, die Informationen über den aktuell angemeldeten Nutzer abzurufen. Ohne zu weit in das Kapitel Cross Platform Development vorzudringen, sehen wir hier eine Besonderheit in der von mir gewählten Architektur. Der Server agiert hier tatsächlich nur als reine API-Schnittstelle, ohne selbst in Besitz von Ablauflogik oder Daten zu sein. Die gesamte Logik zur tatsächlichen Datenbankabfrage liegt im geteilten Code zwischen Client und Server. Die Serverschnittstelle entsteht hier rein aus der Notwendigkeit, eine separate Anwendung anstelle des Browsers mit der Verbindung zu „sensibler“ Infrastruktur zu beauftragen, da der Browser aus Sicherheitsgründen keine direkten Datenbankabfragen oder Dateizugriffe zulässt. Der Client „beauftragt“ den Server also damit, die Datenbankabfrage mit derselben Logik, auf die er selbst auch Zugriff hätte, durchzuführen und die Ergebnisse als HTTP-Response zurückzuliefern. Anders als der Browser haben native Anwendungen wie etwa Windows-Apps durchaus die Erlaubnis, selbstständig Datenbankabfragen durchzuführen, weshalb dieser Teil der Logik überhaupt erst im geteilten Code existiert. Eine Windows-Anwendung überspringt also die Abfrage per HTTP-Request und führt die Datenbankabfrage direkt aus. Genauer dazu im Kapitel 7.

Das Einrichten der Schnittstelle ist in ASP.NET Core sehr einfach und übersichtlich. Der Controller wird mit dem `[ApiController]` Attribut versehen, welches die Controller-Logik für API-Schnittstellen optimiert, alternativ kann auch noch eine bestimmte Route mit dem `[Route(" ")]` Attribut versehen werden, im Falle des `UserCredential-Controllers` hört dieser auf den Endpunkt `„internal/UserCredentials/*“` wobei der Stern für die jeweilige Aktion steht. Definiert wird die Route in der Regel dynamisch zu Beginn des Controllers: `[Route("internal/{controller}/{Action}")]`. Die Endpunkte selbst werden mit den HTTP-Methoden Attributen wie `[HttpGet]`, `[HttpPost]` usw. versehen, definieren ihren Namen mit dem `[ActionName(" ")]` Attribut und die Parameter der Endpunkte werden automatisch aus der Anfrage extrahiert. Es ist auch Best-Practice, die Rückgabewerte und den erwarteten Rückgabe-Code des Endpunktes mit dem `[ProducesResponseType]` Attribut zu definieren, um die API-Schnittstelle für andere Entwickler:innen verständlicher zu machen. In unserem Fall wird der Endpunkt `„Read“` mit dem `[ProducesResponseType(typeof(UserCredentials), 200)]` Attribut versehen, was bedeutet, dass der Endpunkt ein `UserCredentials`-Objekt mit Code 200 zurückliefert, wenn die Anfrage erfolgreich war. Zu guter Letzt kann man Endpunkte auch noch gegen unerwünschte Anfragen absichern, indem man das `[Authorize]` Attribut verwendet, um sicherzustellen, dass nur autorisierte Benutzer:innen Zugriff auf bestimmte Endpunkte haben. Diesen Punkt werden wir in weiterer Folge nicht nur für die API-Schnittstelle, sondern auch für die gesamte Anwendung betrachten.

4.3 Authentifizierung und Autorisierung

ASP.NET und damit auch Blazor bietet eine Vielzahl von Möglichkeiten, um die Authentifizierung und Autorisierung innerhalb einer Anwendung zu implementieren. Neben bekannten

Standards wie OAuth2 oder Token basierter Authentifizierung, wie etwa JWT (JSON Web Token), bietet ASP.NET auch eine eigene Implementierung der Authentifizierung und Autorisierung an, das sog. *Identity Framework*. Im Identity Framework sind nahezu alle gängigen Ablaufstrukturen und Features die man für verschiedene nutzerbasierte Anwendungen benötigt, bereits vollständig implementiert. [17]

Zudem würde Identity auch die Möglichkeit bieten, selbst die Datenbankstruktur zum Speichern von Nutzer:innendaten bereitzustellen, ohne selbst die nötige Logik dafür definieren zu müssen. Für manche Anwendungen kann es auch sinnvoll sein, den Nutzer:innen die Möglichkeit zu geben, sich mit bestehenden Accounts von Drittanbietern wie Google, Facebook oder Microsoft anzumelden. Auch das ist mit dem Identity Framework sehr einfach möglich. In unserem Fall übernehmen wir die Strukturierung und Aufbewahrung der Nutzer:innendaten selbst. Wir verwenden das Identity Framework nur für das Definieren bestimmter „Rollen“ und „Claims“ für die Nutzer:innen, auf die wir dann in der Anwendung zugreifen können, um den Zugriff auf bestimmte Ressourcen und Features zu verwalten.

Im Beispiel oben haben wir gesehen, wie wir das *[Authorize]* Attribut verwenden können, um den Zugriff auf bestimmte Endpunkte der API-Schnittstelle zu beschränken. In unserem Fall ist der Endpunkt „Read“ nur für angemeldete Nutzer:innen zugänglich, da er mit dem *[Authorize]* Attribut versehen ist. „Angemeldet“ bedeutet in diesem Fall erst einmal nur, dass der Client zu Beginn der Anwendung durch das Framework erfasst wurde, und hinterlegt wurde, gegensätzlich dazu würde ein anonymes Aufrufen des Endpunktes (=z.B. als URL im Browser eingeben) zu diesem Zeitpunkt abgefangen werden. Alternativ und viel interessanter für eine API-Schnittstelle ist die Möglichkeit, sich auch über einen Token zu authentifizieren. Tatsächlich wäre auch jede andere Art der Authentifizierung denkbar, da wir im nächsten Schritt die Implementierung eines eigenen sog. „AuthStateProvider“ betrachten werden, der die Logik zur Authentifizierung der Nutzer:innen bereitstellt.

```

1 namespace SharedComponents.Services
2 {
3     public class EdubaiAuthStateProvider : AuthenticationStateProvider
4     {
5         private IPlatformInfo PlatformInfo { get; set; } = default!;
6         private ILocalStorageService LocalStorage { get; set; } = default!;
7         private HttpClient Http { get; set; } = default!;
8
9         public EdubaiAuthStateProvider(ILocalStorageService localStorage,
10             HttpClient http, IPlatformInfo platformInfo)
11         {
12             this.LocalStorage = localStorage;
13             this.Http = http;
14             this.PlatformInfo = platformInfo;
15         }
16
17         public override async Task<AuthenticationState> GetAuthenticationStateAsync()
18         {
19             return await GetAuthenticationStateAsync(null, null);
20         }
21
22         public async Task<AuthenticationState> GetAuthenticationStateAsync(
23             IEnumerable<Claim>? claims, string passwordHash)
24         {
25             ...
26         }
27
28         /// <summary>
29         /// Sets the Jwt in local storage from the given claims
30         /// </summary>
31         /// <param name="claims">Enumerable Collection of Claims</param>
32         /// <returns>true, if the operation was successfull</returns>
33         public async Task<bool> SetJwtFromClaims(IEnumerable<Claim> claims,
34             string passwordHash)
35         {
36             ...
37         }
38
39         #region Static Helper Methods
40         //-----
41         //----- Helper Methods -----
42         //-----
43
44         private async Task<string> GetDefaultAuthenticationStateJwt(){...}
45         public static async Task<IEnumerable<Claim>> ParseClaimsFromJwt(string jwt){...}
46         private async Task<string> GenerateJwtFromClaims(IEnumerable<Claim> claims,
47             string passwordHash){...}
48         private async Task<string> GetValidOrDefault(string token){...}
49         private static byte[] ParseBase64WithoutPadding(string base64){...}
50
51         #endregion
52     }
53 }

```

Der in Listing 6 dargestellte `AuthstateProvider` implementiert den von ASP.NET bereitgestellten *AuthenticationStateProvider*, der mit dem Identity Framework zusammenarbeitet, um den aktuellen Authentifizierungsstatus der Nutzer:innen zu verwalten. Dafür muss zumindest die Methode *GetAuthenticationStateAsync()* implementiert/überschrieben werden, die den aktuellen Authentifizierungsstatus der Nutzer:innen zurückliefert. In unserem Fall ermitteln wir diesen Status, indem wir die Anfrage an den Server zusammen mit dem mitgesendeten Token auslesen. Der Token ist nach dem JWT-Standard aufgebaut und enthält die Informationen über den Nutzer, die Rolle und über sog. „Claims“, die der Nutzer zu seinen Berechtigungen macht. Der Token wird in der Regel bei der Anmeldung generiert und an den Client zurückgesendet, welcher ihn lokal (im Beispiel eines Browsers im *Local Storage* bzw. als *Cookie*) abspeichert und ihn dann bei jeder Anfrage an den Server mitsendet. Der `AuthstateProvider` überprüft den Token auf Echtheit bzw. Gültigkeit und liefert die Informationen als *AuthenticationState*-Objekt zurück, auf welches das Identity Framework dann reagieren kann.

```

1 public async Task<AuthenticationState> GetAuthenticationStateAsync(
2     IEnumerable<Claim>? claims, string passwordHash)
3 {
4     string token = null;
5
6     if(claims == null)
7     {
8         if (PlatformInfo.Platform == Platform.Maui)
9         {
10             token = EdubaiLocalStorage.Instance.Get("edubaiauthstatetoken",
11                 "", PlatformInfo);
12         }
13         else
14         {
15             token = await EdubaiLocalStorage.Instance.GetAsync("edubaiauthstatetoken",
16                 "", PlatformInfo, LocalStorage);
17         }
18
19         token = await GetValidOrDefault(token);
20
21         // Decrypt the JWT
22         EdubaiAESEncryptionController aes = new EdubaiAESEncryptionController(
23             PlatformInfo, Http, this);
24         token = await aes.Decrypt(token);
25
26         claims = await ParseClaimsFromJwt(token);
27     }
28
29     if (token == null)
30     {
31         token = await GenerateJwtFromClaims(claims, passwordHash);
32     }
33
34     ClaimsIdentity identity = new ClaimsIdentity(claims, "jwt");
35     Http.DefaultRequestHeaders.Authorization = null;
36     Http.DefaultRequestHeaders.Authorization =
37         new AuthenticationHeaderValue("Bearer", token!.Replace("\\"", ""));
38
39
40     var user = new ClaimsPrincipal(identity);
41     var state = new AuthenticationState(user);
42
43     NotifyAuthenticationStateChanged(Task.FromResult(state));
44
45     return state;
46 }

```

Listing 7: GetAuthenticationStateAsync

Die in Listing 7 implementierte Methode *GetAuthenticationStateAsync()* zeigt den schematischen Ablauf der Tokenvalidierung. Grundsätzlich gibt es zwei Fälle, in denen der Authentifizierungsstatus der Nutzer:innen ermittelt wird. Im ersten Fall handelt es sich um eine explizite Abfrage durch die eigens implementierte Logik während der Anmeldung. In diesem Fall werden die Anmeldedaten und Berechtigungen direkt an die Methode überliefert, woraus in weiterer Folge der Token generiert und an passender Stelle lokal gespeichert wird. Im zweiten Fall fordert das Identity Framework Informationen über den aktuellen Status der Nutzer:innen an. Hier wird der zuvor gespeicherte Token ausgelesen und die Berechtigungen ermittelt. In beiden Fällen liefert die Methode den *AuthenticationState* zurück, wodurch das weitere Verhalten der Anwendung bestimmt werden kann.

Schauen wir uns diesen Ablauf nun an einem konkreten Beispiel, wie es in der Anwendung implementiert ist, an. Grundsätzlich wollen wir bestimmte Ressourcen und Features der Anwendung nur für angemeldete Nutzer:innen zugänglich machen. Andere Teile der Anwendung sollen/müssen jedoch auch anonym zugänglich sein, wie etwa die Startseite oder die Login-Seite. Um das zu erreichen, und in Zukunft auch zu ermöglichen, verschiedenen Gruppen verschiedene Ressourcen zur Verfügung zu stellen, führen wir ein Rollen-System ein. Derzeit gibt es in der Anwendung drei verschiedene Rollen:

- **System:** Default Rolle für alle Anfragen, die nicht mit einem bestimmten Nutzer verknüpft sind
- **Student:** Nutzer:innen, die zum Zeitpunkt der Anmeldung die Rolle „Student“ ausgewählt haben
- **Teacher:** Nutzer:innen, die zum Zeitpunkt der Anmeldung die Rolle „Teacher“ ausgewählt haben

Grundsätzlich handelt es sich derzeit bei den Rollen *Student* und *Teacher* um Nutzerrollen, die auf alle Lernressourcen zugreifen können. In weiterer Folge lässt die Unterscheidung aber weitere Features, die nur für Lehrpersonen zugänglich sein sollen, besonders einfach zu. Die *System*-Rolle unterscheidet sich insofern von einem rein anonymen Zugriff dadurch, dass man sicherstellen kann, dass der Zugriff/die Anfrage von der Applikation selbst, oder zumindest von einer Person mit Zugriff auf die Applikation kommt. Das ungefragte, automatische Auslesen aller API-Endpunkte durch automatisierte Systeme wird so zumindest oberflächlich unterbunden.

```
1 @page "/learningapps"
2 [Authorize(Roles = "Student, Teacher")]
3 [Authorize(Policy = "EmailIsVerified")]
4
5 @layout MainLayout
6
7 @using SharedComponents.Pages.LearningApps.Model
8 @inherits LearningAppsBase
9
10 <PageTitle>@Localizer[Localization.learning_apps]</PageTitle>
11
12 <AuthorizeView>
13     <NotAuthorized>
14         //...
15     </NotAuthorized>
16
17     <Authorized>
18         //...
19     </Authorized>
20 </AuthorizeView>
```

Listing 8: LearningApps Component

Die Razor-Komponente *LearningApps* in Listing 8 ist eine Beispielkomponente, die nur für Nutzer:innen mit der Rolle *Teacher* oder *Student* zugänglich sein soll. Wir versehen daher die Komponente oben mit dem `@attribute [Authorize(Roles = "Student, Teacher")]` Attribut, wodurch das Identity Framework automatisch überprüft, ob der/die aktuell angemeldete Nutzer:in eine der erlaubten Rollen besitzt. Wir haben hier durch das folgende `<AuthorizeView>` Element die Möglichkeit, den Inhalt der Komponente basierend auf dem Authentifizierungsstatus der Nutzer:innen zu verändern. Im Falle, dass die Nutzer:innen die Rollenerfordernis nicht erfüllen, können wir in der `<NotAuthorized>` Sektion eine entsprechende Nachricht anzeigen, welche nur für diese Nutzer:innen sichtbar ist. Hat der/die Nutzer:in jedoch eine der erlaubten Rollen, wollen wir den eigentlichen Inhalt, definiert durch die `<Authorized>` Sektion, anzeigen. Dadurch können wir sehr effizient und übersichtlich den Zugriff auf bestimmte Ressourcen und Features der Anwendung steuern.

```

1 namespace SharedComponents.Policies
2 {
3     public class EmailVerifiedRequirement : ClaimsAuthorizationRequirement
4     {
5         private string ClaimType { get; set; } = default!;
6         private IEnumerable<string>? AllowedValues { get; set; } = default!;
7
8         public EmailVerifiedRequirement(string claimType, IEnumerable<string>? allowedValues)
9             : base(claimType, allowedValues)
10        {
11            this.ClaimType = claimType;
12            this.AllowedValues = allowedValues;
13        }
14
15        protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
16            ClaimsAuthorizationRequirement requirement)
17        {
18            if(context.User.HasClaim(c => c.Type.Equals("EmailIsVerified")))
19            {
20                Claim EmailIsVerified = context.User.Claims.Where(
21                    c => c.Type.Equals("EmailIsVerified")).FirstOrDefault();
22
23                if (EmailIsVerified != null && AllowedValues.Where(
24                    v => v.Equals(EmailIsVerified.Value)).Count() > 0)
25                {
26                    // EmailIsVerified claim exists and is true (the only allowed value)
27
28                    context.Succeed(requirement);
29                    return Task.CompletedTask;
30                }
31            }
32
33            context.Fail();
34            return Task.CompletedTask;
35        }
36    }
37 }

```

Listing 9: EmailVerifiedRequirement

Reichen nun alle vorgefertigten Systeme des Identity Frameworks, wie das oben vorgestellte Rollen-System, nicht aus, haben wir die Möglichkeit, eigene Authentifizierungs- und Autorisierungslogik zu implementieren. Dafür definieren wir sog. „Policies“, also bestimmte Richtlinien, die für den Nutzer beim Zugriff auf eine Ressource gelten müssen. In unserem Fall wollen wir sicherstellen, dass die Nutzer:innen ihre Email-Adresse verifiziert haben, bevor sie auf bestimmte Ressourcen zugreifen können. Dafür definieren wir eine eigene

EmailVerifiedRequirement Klasse (siehe Listing 9), die die *ClaimsAuthorizationRequirement* Klasse implementiert. Diese Klasse enthält eine Methode *HandleRequirementAsync()*, welche vom Identity Framework zu gegebener Zeit aufgerufen wird und entscheidet, ob der/die aktuelle Nutzer:in eine bestimmte Richtlinie erfüllt. In unserem Fall überprüfen wir, ob wir den *EmailVerified* Claim zuvor zu den Authorisierungsinformationen des Users hinzugefügt haben. Finden wir einen solchen Claim, bestätigen wir das Einhalten der Richtlinie mit dem Aufruf der *context.Succeed(requirement)* Methode, andernfalls wird die Richtlinie nicht erfüllt, was wir mit dem Aufruf von *context.Fail()* signalisieren. Wie in Listing 8 Zeile 3 zu sehen ist, wird diese Komponente zusätzlich zur Rollenüberprüfung nur angezeigt, wenn die Richtlinie *EmailIsVerified* erfüllt ist, signalisiert durch das *@attribute [Authorize(Policy = "EmailIsVerified")]* Attribut.

Mit diesem System haben wir die Möglichkeit den Zugriff auf Ressourcen sehr kompakt und übersichtlich zu steuern, ohne dass wir in der Komplexität der Autorisierungslogik eingeschränkt werden.

4.4 Email Verifizierung und Verschlüsselung

Um gleich beim Thema Email-Verifizierung zu bleiben, betrachten wir kurz einen weiteren Dienst, welcher nicht vom Blazor WebAssembly Client ausgeführt werden kann, sondern der Server übernehmen muss. Um zu verhindern, dass sich Nutzer:innen mit ungültigen Emailadressen anmelden, die ihnen nicht gehören, implementieren wir eine mittlerweile gängige Verifizierungsmethode. Bei der Registrierung wird dem/der Nutzer:in bevor er/sie Zugriff auf die geschützten Ressourcen bekommt eine Email an die angegebene Adresse geschickt, die einen Link enthält, der auf einen Endpunkt des Servers verweist. Vor dem Versenden erstellen wir einen sog. „Verification Token“ und hinterlegen diesen in der Datenbank, zusammen mit der Emailadresse des/der Nutzer:in. Auch die Email, bzw. der Link enthält den Token. Wenn also der/die Nutzer:in tatsächlich Zugriff auf diesen Link hat, können wir beim Aufruf des Endpunktes den Token auslesen und mit dem in der Datenbank gespeicherten Token vergleichen. Stimmt der Token überein, können wir die Email-Adresse des/der Nutzer:in als verifiziert markieren und ihm/ihr Zugriff auf die geschützten Ressourcen gewähren. Das Senden selbst geschieht wie in Listing ?? über den .NET *SmtplibClient*, welcher aber unter Blazor WebAssembly nicht unterstützt wird und daher in unserer Architektur eine Aufgabe des Servers ist.

```
1  SmtplibClient smtpClient = new SmtplibClient("smtp.gmail.com")
2  {
3      Port = 587,
4      EnableSsl = true,
5      Credentials = new NetworkCredential(EdubaiMail, GetEdubaiMailPassword())
6  };
7
8  var mailMessage = new MailMessage
9  {
10     From = new MailAddress(EdubaiMail),
11     Subject = subject,
12     Body = message,
13     IsBodyHtml = true,
14 };
15 mailMessage.To.Add(toEmail);
16
17 try
18 {
19     smtpClient.Send(mailMessage);
20 }
21 catch (Exception ex)
22 {
23     EduBaiMessaging.ConsoleLog($"(2) SMTP Error! {ex.Message}", "Email Controller");
24     return false;
25 }
26
27 return true;
```

Listing 10: Email Verification Service

Dasselbe Problem haben wir auch bei der Verschlüsselung von z.B. der Signatur unserer JWT-Tokens, die wir für die Authentifizierung der Nutzer:innen verwenden. Es wäre ein großes Sicherheitsrisiko, wäre diese Verschlüsselung nativ im Browser möglich, da in diesem Fall auch die privaten Schlüssel im Browser gespeichert werden müssten. Daher ist auch hier der Server für die Verschlüsselung und Signatur der Tokens zuständig. Wir verwenden dafür in Listing 11 die im System.Security.Cryptography Namespace enthaltenen Klassen, die es uns ermöglichen eine symmetrische Verschlüsselung der Tokens durchzuführen. Wir halten uns dabei an die empfohlenen Standards, wie man mit diesen Klassen eine ausreichend sichere Verschlüsselung durchführt.

```
1 // Encrypt the plain text
2 string key = GetValueFromConfig("EdubaiJwtEncryptionKey");
3
4 byte[] encrypted;
5 using (Aes aesAlg = Aes.Create())
6 {
7     aesAlg.Key = Encoding.UTF8.GetBytes(key);
8     aesAlg.IV = new byte[16];
9     aesAlg.Mode = CipherMode.CBC;
10    aesAlg.Padding = PaddingMode.PKCS7;
11    ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
12    using (MemoryStream msEncrypt = new MemoryStream())
13    {
14        using (CryptoStream csEncrypt = new CryptoStream(
15            msEncrypt, encryptor, CryptoStreamMode.Write))
16        {
17            using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
18            {
19                swEncrypt.Write(plainText);
20            }
21            encrypted = msEncrypt.ToArray();
22        }
23    }
24 }
25
26 return Convert.ToBase64String(encrypted);
```

Listing 11: Symmetrische Token Verschlüsselung

5 Hosting

Bevor wir uns dem Client-Teil der Anwendung zuwenden, wollen wir zusätzlich zur Logik des Servers auch noch alle dazugehörigen Infrastruktur- und Hostingelement betrachten. Wir werden im folgenden über die Bereitstellung der Anwendung sprechen, nötige Konfigurationen für einen sicheren und reibungslosen Zugriff betrachten und den Aufbau der zugrunde liegenden Datenbank und den Zugriff darauf erläutern.

5.1 Basic Linux-ASP.NET Hosting

Eine der einfachsten, kostengünstigsten und doch verhältnismäßig stabilen Möglichkeiten, eine Webanwendung zu hosten, ist die Verwendung eines eigens verwalteten Netzwerkservers. *Achtung: In diesem Kapitel ist ab sofort mit dem Begriff „Server“ der Netzwerkservers, also die Infrastruktur gemeint, nicht die Server-Anwendung, die wir in den vorherigen Kapiteln betrachtet haben.* Ich habe mir hier rein aus Bequemlichkeitsgründen für einen verfügbaren HP-Elitedesk Mini-PC entschieden, welcher mit einem Intel i5 Prozessor und 8GB RAM für unsere Zwecke mehr als ausreichend ist. Vor allem auch durch die Eingangs erwähnte Tatsache, dass wir die schwere Ablauflogik auf den Client auslagern. Der Server läuft mit *Xubuntu*, einer leichtgewichtigen Linux-Distribution, basierend auf Ubuntu 22.04., welches als besonders stabil und benutzerfreundlich gilt. Denkbar wäre hier aber auch jede andere gängige Linux-Distribution, im speziellen auch Server-Distributionen ohne grafische Oberfläche, da wir den Server großteils Remote über die Kommandozeile verwalten können. Für die erste Installation und Konfiguration hat mir die grafische Oberfläche aber sehr geholfen und einige Zeit an Einarbeitung erspart. Die Installation selbst soll hier aber nicht Gegenstand dieser Arbeit sein, dazu gibt es zahlreiche Ressourcen im Internet.



Abbildung 6: HP Elitedesk Mini-PC [8]

Das Hosten der Anwendung selbst, ist durch die jahrelange Weiterentwicklung der ASP.NET Core Plattform sehr einfach geworden. Verwendet man für die Entwicklung Visual Studio, kann man das gesamte Projekt mit einem Klick für die Bereitstellung auf einem Server vorbereiten. Für unsere Blazor-WebAssembly Anwendung, müssen wir im Speziellen die Host-Anwendung „veröffentlichen“. Hier haben wir dann die Option auszuwählen, in

welches Verzeichnis und für welches (Betriebs-)System wir die Applikation bereitstellen wollen. Standardmäßig veröffentliche ich den aktuellsten Stand wie in Abbildung 7 in ein temporäres Verzeichnis, mit der Option, dass die Anwendung auf einem Linux-Server mit der passenden Version von .NET Core laufen soll. Das Ergebnis ist ein Verzeichnis, welches alle benötigten Dateien enthält, um die Anwendung auf dem Server zu starten, welche man nun nurmehr auf den Server kopieren muss.

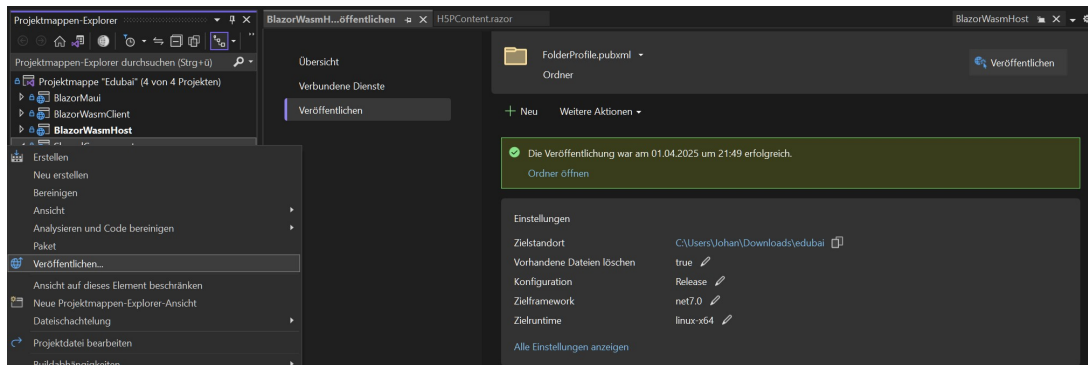


Abbildung 7: Projekt veröffentlichen

Sind alle Dateien auf dem Server verfügbar, müssen wir nur noch die Anwendung mittels ASP.NET Core starten. Dafür sollte man einmalig kontrollieren, ob eine passende Version des Frameworks auf dem Server installiert ist, oder es gegebenenfalls z.B. mit dem Befehl `apt-get install aspnetcore-runtime-7.0` installieren. Danach kann die Anwendung aus dem richtigen Verzeichnis heraus mit dem Befehl `dotnet run` gestartet werden. Das Ergebnis ist eine laufende ASP.NET Core Anwendung, die standardmäßig auf Port 5000 lauscht und über die IP-Adresse des Servers erreichbar ist. Damit ist die Anwendung nun grundsätzlich bereit, von außen aufgerufen zu werden. Um die Anwendung für den Endnutzer zugänglicher zu machen, werden wir uns in Kapitel 5.2 noch um die Konfiguration eines Webserver kümmern. Dieser macht die Anwendung über eine Domain erreichbar und sichert die Kommunikation über HTTPS ab. Zuvor wollen wir aber noch eine schnelle Maßnahme gegen Stromausfälle und andere unerwartete Ereignisse treffen, da es sich noch immer um einen einzelnen Server handelt. Um nicht manuell den Server neu starten zu müssen, wenn dieser einmal abstürzt oder neu gestartet wird, installieren wir einen *systemd* Service unter Ubuntu, der es ermöglicht, die Anwendung als Dienst zu starten und zu verwalten. Dafür erstellen wir im Verzeichnis `/etc/systemd/system/` eine neue Datei mit der Endung `.service`, in unserem Fall `edubaihost.service`.

```
1 [Unit]
2 Description=EduBai Webservice
3
4 [Service]
5 WorkingDirectory=/var/www/edubai
6 ExecStart=/var/www/edubai/BlazorWasmHost
7 Restart=always
8 # Restart service after 10 seconds if the dotnet service crashes:
9 RestartSec=10
10 KillSignal=SIGINT
11 SyslogIdentifier=dotnet-edubai
12 User=www-data
13 Environment=ASPNETCORE_ENVIRONMENT=Production
14
15 [Install]
16 WantedBy=multi-user.target
```

Listing 12: edubaihost.service

In Listing 12 definieren wir alle Ausgangsparameter, die der Dienst beim Starten des Service annehmen soll. Unter anderem definieren wir hier das Arbeitsverzeichnis, in welchen der den Dienst starten soll, welchen Befehl er ausführen soll, und als welcher Benutzer der Dienst laufen soll. Es reicht hier beim Befehl die *BlazorWasmHost.dll* Datei anzugeben, da er diese im angegebenen Arbeitsverzeichnis dann automatisch mit der *dotnet*-Laufzeit ausführen wird. Nachdem wir diesen Service fertig konfiguriert haben, können wir ihn mit dem Befehl *sudo systemctl enable edubaihost.service* aktivieren, wodurch er beim nächsten Start des Servers automatisch gestartet wird. Für uns bedeutet das, solange der Server läuft, läuft auch die Anwendung und stellt den Nutzer:innen die Blazor WebAssembly Website zur Verfügung. Der Name „edubaihost“ lässt hier übrigens schon auf den Namen der Website schließen. Diese soll in weiterer Folge unter der Domain „www.edubai.duckdns.com“ erreichbar sein, wobei *edubai* ein einfaches Kunstwort aus „Education“ und „Baischer“ ist.

5.2 Reverse Proxy mit Apache2

Apache2 ist ein weit verbreiteter, freier und quelloffener Webserver, der es selbstständig ermöglicht, Webanwendungen über das HTTP-Protokoll bereitzustellen. Grundsätzlich sind wir mit der ASP.NET Core Anwendung bereits in der Lage, die Blazor WebAssembly Anwendung über HTTP bereitzustellen, allerdings ist es für den Endnutzer nicht sehr komfortabel, die Anwendung über eine IP-Adresse aufzurufen. Wir können hier Apache2 nutzen, um letztendlich die Anwendung über eine Domain erreichbar zu machen. Dazu

installieren wir Apache2 auf dem Server mit dem Befehl `sudo apt-get install apache2`. Konfigurationsdateien für Apache2 befinden sich im Verzeichnis `/etc/apache2/sites-available/`. Hier können wir eine neue Konfigurationsdatei für unsere Anwendung erstellen. Wir erstellen hier die Datei `edubai.conf`, siehe Listing 13.

```

1 <VirtualHost *:80>
2     ServerAdmin johannes.baischer@gmail.com
3     ServerName www.edubai.duckdns.org
4     ServerAlias edubai.duckdns.org
5     DocumentRoot /var/www/edubai
6
7     ProxyRequests          On
8     ProxyPreserveHost      On
9     ProxyPassMatch         ^/_blazor/(.*) http://localhost:5000/_blazor/$1
10    ProxyPass               /_blazor ws://localhost:5000/_blazor
11    ProxyPass               / http://localhost:5000/
12    ProxyPassReverse        / http://localhost:5000/
13
14    ErrorLog ${APACHE_LOG_DIR}/error.log
15    CustomLog ${APACHE_LOG_DIR}/access.log combined
16 RewriteEngine on
17 RewriteCond %{SERVER_NAME} =edubai.duckdns.org [OR]
18 RewriteCond %{SERVER_NAME} =www.edubai.duckdns.org
19 RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [END,NE,R=permanent]
20 </VirtualHost>

```

Listing 13: edubai.conf

Wir nutzen Apache2 hier als Reverse-Proxy, welcher alle Anfragen an den Server an verschiedene Anwendungen weiterleiten kann. In unserem Fall läuft bis jetzt zwar nur die ASP.NET Core Anwendung, wir können aber durch dieses Setup auf bestimmte Informationen der Serveranfrage reagieren. Genauer gesagt, wollen wir auf den angefragten Servernamen reagieren. In der Konfigurationsdatei definieren wir, dass alle Anfragen an den Server mit dem Servernamen „www.edubai.duckdns.com“ (alternativ auch ohne www) and unsere Blazor-App die auf Port 5000 lauscht, weitergeleitet werden. Dadurch werden alle typischen Anfragen durch einen Browser mit dieser Domain an die ASP.NET Core Anwendung weitergeleitet. Hier sollten wir aber auch gleich noch eine wichtige Tatsache feststellen: *Würde der Nutzer zu diesem Zeitpunkt die Domain im Browser aufrufen, würde er noch immer nicht bei unserem Server landen!* Dieses Problem werden wir im nächsten Schritt beheben, indem wir die Domain auf unseren Server verweisen lassen.

5.3 Domain und Zertifikate

Die erwähnte Domain „www.edubai.duckdns.com“ lässt schon darauf schließen, dass wir hier einen Domain-Provider verwenden. Dieser bestimmte Anbieter, zu sehen in Abbildung 8, ist nämlich für bis zu fünf Domains kostenlos mit dem kleinen Beigeschmack, dass sein Name in der Domain enthalten ist. Es handelt sich hier um einen sog. Dynamic DNS Anbieter, das heißt er ermöglicht es, eine Domain auch für einen Server, welcher ab und an seine IP-Adresse ändert, zu verwenden. Das ist besonders praktisch für Home-Server wie unseren, da es ab und an sein kann, dass der Internetanbieter die IP-Adresse des Heims ändert. DuckDNS bietet hierfür eine Vielzahl an verschiedenen Betriebssystemen und Geräten eine einfache Schnittstelle an, an der sich der Server regelmäßig melden kann, um seine aktuelle IP-Adresse bekanntzugeben. Sollte sich die Adresse geändert haben, passt DuckDNS das Routing automatisch an, sodass die Nutzer:innen wieder richtig an den Server weitergeleitet werden.

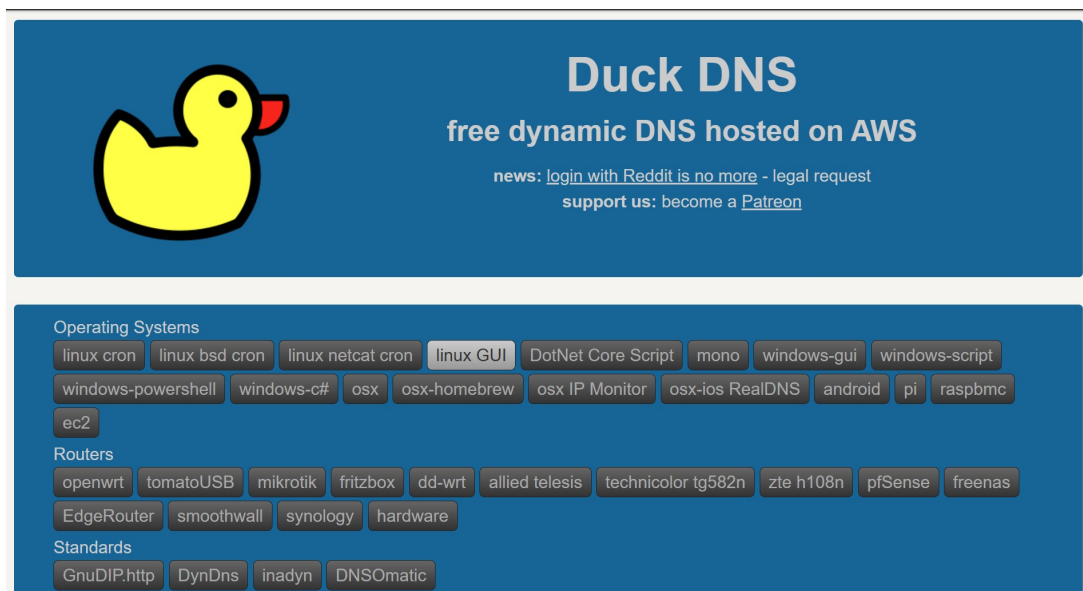


Abbildung 8: DuckDNS Website

Da die Installationsanleitungen [3] von DuckDNS sehr gut und ausführlich sind und stark von den verwendeten Betriebssystemen und Geräten abhängen, werden wir hier nicht näher darauf eingehen. Stattdessen wollen wir uns mit einem letzten sehr wichtigen Punkt beschäftigen, der für die Sicherheit der Anwendung und der Nutzer:innen von großer Bedeutung ist: der Verschlüsselung der Kommunikation zwischen dem Server und den Nutzer:innen. Dafür ist bekannterweise anstelle des HTTP-Protokolls das HTTPS-Protokoll notwendig, welches aber für die Verschlüsselung des Datenverkehrs ein gültiges

Zertifikat benötigt. Ein solches Zertifikat kann man für den tatsächlichen Einsatz nicht einfach selbst erstellen, sondern es muss von einer vertrauenswürdigen Zertifizierungsstelle ausgestellt werden. Es entsteht in diesem System eine Art „Kette des Vertrauens“, da jede Zertifizierungsstelle von einer höheren Stelle zertifiziert sein muss. Oft übernehmen Websiteanbieter die Aufgabe ein gültiges Zertifikat aufrecht zu halten. In der Regel kosten diese Zertifikate nämlich Geld, etwa 15€ - 100€ pro Jahr. Eine der bekanntesten und bei Webentwickler:innen sehr beliebten Zertifizierungsstellen ist „Let’s Encrypt“, welche es durch die Finanzierung von Spenden ermöglicht, kostenlose Zertifikate für Domains auszustellen. Diese Zertifikate sind in der Regel zwar nur für 90 Tage gültig und müssen danach erneuert werden, auch diesen Prozess kann man aber ganz leicht automatisieren. Hierfür gibt es den von der Electronic Frontier Foundation (EFF) entwickelten Client „Certbot“ [5], der es ermöglicht, die Zertifikate automatisch zu beantragen und zu erneuern. Die genaue Konfiguration hängt hier wieder stark vom verwendeten DDNS Betreiber ab und ist auf der offiziellen Website sehr gut dokumentiert.

5.4 PostgreSQL Datenbank

Sobald man von einer statischen Website zu einer Webanwendung wechselt, die auf verschiedene Benutzer dynamisch reagieren soll, bestimmte Daten oder Zustände speichern muss, oder auch nur Benutzerkonten für die Anmeldung benötigt, braucht man zwangsläufig eine Möglichkeit diese Daten zu speichern. Für eine überschaubare Menge an Daten kann eine individuelle Lösung in Form von geschicktem Dateimanagement oder auch nur das Speichern in einer JSON-Datei durchaus ausreichen. Sobald es aber um größere Datenmengen oder komplexere Datenstrukturen geht, ist es ratsam, auf eine klassische Datenbank zurückzugreifen. Da wir Benutzerkonten verwalten und deren Rechte und Rollen einschränken und bearbeiten können wollen, benötigen auch wir eine Datenbank. Für EduBai haben wir uns für die Open-Source Datenbank „PostgreSQL“ entschieden, welche Nutzer für ihre hohe Stabilität und Flexibilität schätzen. PostgreSQL ist eine objektrelationale Datenbank, das bedeutet, dass sie sowohl relationale Datenbankstrukturen (wie Tabellen) als auch objektorientierte Konzepte (wie Vererbung und Polymorphismus) unterstützt, was einen klaren Unterschied zum etwas bekannteren rein relationalen Datenbanksystem „MySQL“ darstellt. PostgreSQL ist bekannt für seine hohe Leistung, Skalierbarkeit und Unterstützung für komplexe Abfragen und Transaktionen. Auch wenn wir diese Funktionen nicht sehr weit ausreizen werden, stellt PostgreSQL eine sehr gute Basis für unsere Anwendung dar.

Nach einer herkömmlichen Installation der PostgreSQL Datenbank auf dem gewünschten System [25] haben wir zwei Möglichkeiten für die Konfiguration der Datenbank. Zum einen können wir die Datenbank über die Kommandozeile verwalten, was für einfache Abfragen und Konfigurationen durchaus ausreicht, vorausgesetzt man kennt sich mit den

SQL-Befehlen genügend aus. Zum anderen, und das ist die von mir bevorzugte Variante, können wir eine grafische Oberfläche verwenden, um die Datenbank zu verwalten. Hierfür gibt es eine Vielzahl an Tools, die sich für PostgreSQL eignen. Das wohl bekannteste für PostgreSQL ausgelegte Tool ist „pgAdmin“ (siehe Abbildung 9).



Abbildung 9: pgAdmin Logo

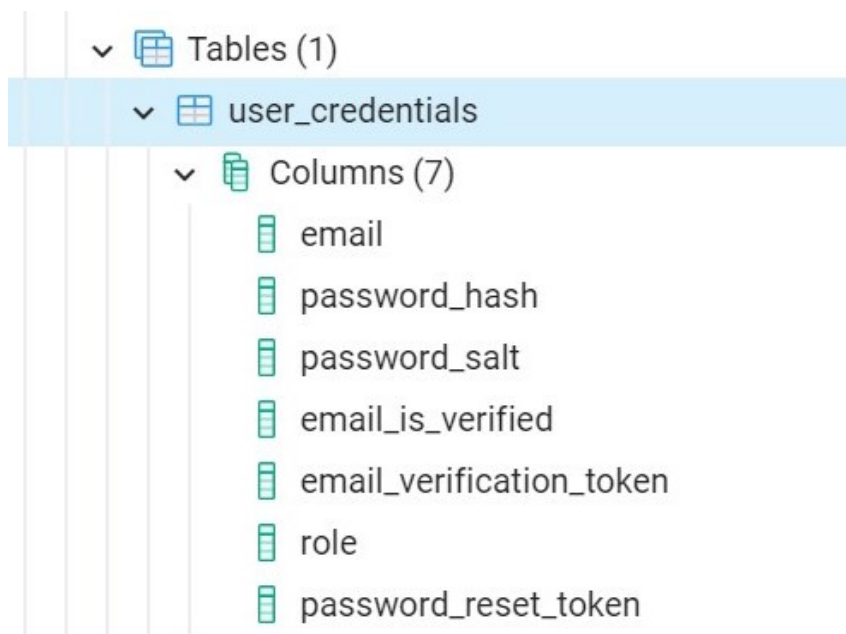


Abbildung 10: pgAdmin Menü

Das Erstellen bestimmter Datenbankstrukturen, wie etwa Tabellen, ist in pgAdmin sehr intuitiv über verschiedenen Menüs (siehe Abbildung 10) möglich. Optionen die in SQL-Befehlen oft ein Keyword an der richtigen Stelle benötigen, sind hier in Form von Checkboxen und Dropdown-Menüs verfügbar. Auch das Erstellen von Abfragen ist in pgAdmin sehr einfach. Das Anzeigen der gespeicherten Einträge einer Tabelle lässt sich beispielsweise über ein Rechtsklickmenü bewerkstelligen (siehe Abbildung 11).

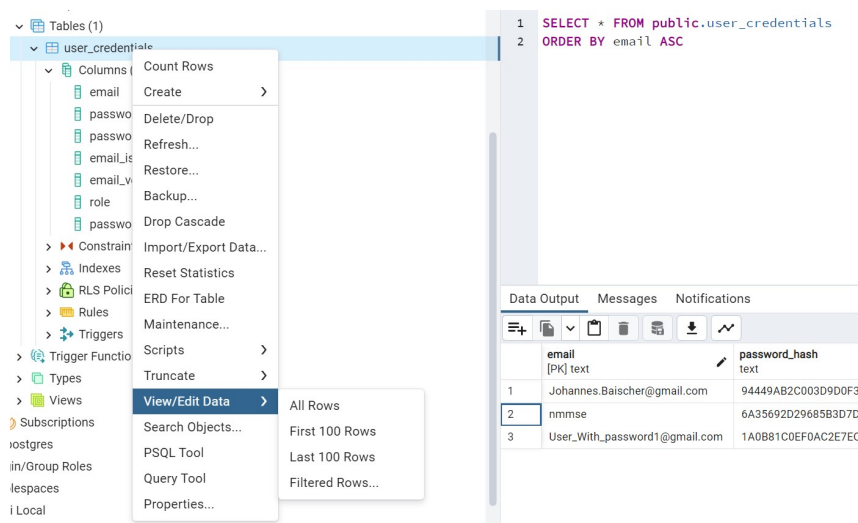


Abbildung 11: pgAdmin Abfrage

Zum jetzigen Zeitpunkt befindet sich in der Datenbank nur eine Tabelle, die die Nutzer:inneninformationen speichert. Diese Tabelle *user_credentials* enthält die Felder:

Feldname	Datentyp	Beschreibung
email (Primärschlüssel)	Text	Eindeutige Email-Adresse des Nutzers
password_hash	Text	Hash des Passworts des Nutzers
password_salt	Text	Salt (Zufallswert) für randomisierte Passwort-Hashes
email_is_verified	Boolean	Ob die Email-Adresse des Nutzers verifiziert ist
email_verification_token	Text	Token für die Email-Verifizierung
role	Text	Rolle des Nutzers (z.B. „Student“ oder „Teacher“)
password_reset_token	Text	Token für die Passwort-Zurücksetzung

Tabelle 1: Felder der Tabelle *user_credentials*

Ein besonderer Vorteil der Verwendung einer relationalen Datenbank, neben der einfachen Struktur und weit verbreiteten Nutzung, ist die Möglichkeit, die verwendeten Datenstrukturen in Programmcode abzubilden. Wir haben hier abermals ein von Microsoft entwickeltes, langjährig erprobtes System, das sog. „Entity Framework“, welches es ermöglicht, die Datenbankstruktur in C# Klassen abzubilden und so eine typensichere Abfrage der Datenbank zu ermöglichen.

Nach der erfolgreichen Installation des Entity Frameworks, beispielsweise als Commandline-Tool mit dem Befehl *dotnet tool install -global dotnet-ef*, können wir die in pgAdmin erstellte Datenbankstruktur in C# Klassen abbilden.

1. Mittels hinterlegtem Connection String:

```
dotnet ef dbcontext scaffold "Name=ConnectionStrings:EdubaiPostgreSQL" Npgsql.EntityFrameworkCore.PostgreSQL -o PostgreSQL -f
```

2. Mittels Connection String direkt in der Kommandozeile:

```
dotnet ef dbcontext scaffold "Host=homebai.duckdns.org:5432;Username=edubai; Password=<Password>;Database=edubai" Npgsql.EntityFrameworkCore.PostgreSQL -o PostgreSQL -f
```

Konnte das Entity Framework die Datenbankstruktur erfolgreich auslesen, wird es die entsprechenden C# Klassen im angegebenen Verzeichnis *PostgreSQL* erstellen. Diese Klassen spiegeln exakt die Struktur der Daten der Datenbank wider, wie sie auch nach einer SQL-Abfrage zurückgeliefert werden würden.

```

1 namespace SharedComponents.PostgreSQL;
2 public partial class EdubaiContext : DbContext
3 {
4     public EdubaiContext(){}
5     public EdubaiContext(DbContextOptions<EdubaiContext> options)
6         : base(options){}
7     public virtual DbSet<UserCredential> UserCredentials { get; set; }
8     protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
9         => optionsBuilder.UseNpgsql(ConnectionString.ConnectionStringValue);
10    protected override void OnModelCreating(ModelBuilder modelBuilder)
11    {
12        modelBuilder.Entity<UserCredential>(entity =>
13        {
14            entity.HasKey(e => e.Email).HasName("user_credentials_pkey");
15            entity.ToTable("user_credentials", tb
16                => tb.HasComment("Table containing user data"));
17            entity.Property(e => e.Email).HasColumnName("email");
18            entity.Property(e => e.EmailIsVerified)
19                .HasComment("true if email verification process was succesful")
20                .HasColumnName("email_is_verified");
21            entity.Property(e => e.EmailVerificationToken)
22                .HasComment("token to be sent via email for verification")
23                .HasColumnName("email_verification_token");
24            entity.Property(e => e.PasswordHash).HasColumnName("password_hash");
25            entity.Property(e => e.PasswordResetToken)
26                .HasComment("temporary token for password reset request")
27                .HasColumnName("password_reset_token");
28            entity.Property(e => e.PasswordSalt).HasColumnName("password_salt");
29            entity.Property(e => e.Role).HasColumnName("role");
30        });
31
32        OnModelCreatingPartial(modelBuilder);
33    }
34    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
35 }

```

Listing 14: DbContext

```
1 namespace SharedComponents.PostgreSQL;
2 /// <summary>
3 /// Table containing user data
4 /// </summary>
5 public partial class UserCredential
6 {
7     public string Email { get; set; } = null!;
8     public string PasswordHash { get; set; } = null!;
9     public string PasswordSalt { get; set; } = null!;
10    /// <summary>
11    /// true if email verification process was succesful
12    /// </summary>
13    public bool EmailIsVerified { get; set; }
14    /// <summary>
15    /// token to be sent via email in case of verification or password reset
16    /// </summary>
17    public string? EmailVerificationToken { get; set; }
18    public string Role { get; set; } = null!;
19    /// <summary>
20    /// temporary token for password reset request
21    /// </summary>
22    public string? PasswordResetToken { get; set; }
23 }
```

Listing 15: UserCredentials Entity Framework Klasse

In Listing 14 sehen wir die automatisch generierte *DbContext* Klasse, die das Entity Framework für die Datenbank erstellt hat. Diese Klasse ist das Herzstück des Entity Frameworks und ermöglicht es, auf die Datenbank zuzugreifen und Abfragen durchzuführen. In unserem Fall enthält der *DbContext* einen Builder für die *UserCredentials*-Tabelle welcher alle Eigenschaften beschreibt. Daraus entsteht dann die in Listing 15 dargestellte C#-Klasse. Diese Klasse können wir nun direkt in unserem Code verwenden, um auf die Datenbank zuzugreifen und Abfragen durchzuführen. Am Beispiel einer „Upsert“-Methode (siehe Listing 16) sehen wir, wie einfach und vor allem eindeutig es ist, bestimmte Eigenschaften der Nutzer:innen zu ändern oder zu erstellen.

```

1  var newUserCredential = new UserCredential
2  {
3      Email = Model.Email,
4      PasswordHash = passwordHash,
5      PasswordSalt = salt,
6      EmailIsVerified = false,
7      EmailVerificationToken = UserCredentialsController.GenerateSalt(),
8      Role = Model.Role.ToString(),
9  };
10
11  /// <summary>
12  /// Creates a new UserCredential in the Database
13  /// </summary>
14  /// <param name="newUserCredential">Usercredentials to be added</param>
15  /// <param name="callerIsServer">Flag for dedicated call from Server</param>
16  /// <returns></returns>
17  public async Task<UserCredential?> Upsert(UserCredential newUserCredential,
18      bool callerIsServer = false)
19  {
20      #region Redirect to Server
21      if (callerIsServer == false && PlatformInfo!.Platform == Platform.Web)
22      {
23          //...
24      }
25      #endregion
26
27      try
28      {
29          bool userExists = await CheckUserExists(newUserCredential.Email, callerIsServer);
30
31          //DB Call
32          using (var db = new EdubaiContext())
33          {
34              if (userExists)
35              {
36                  //Update
37                  db.UserCredentials.Update(newUserCredential);
38              }
39              else
40              {
41                  //Insert
42                  db.UserCredentials.Add(newUserCredential);
43              }
44              db.SaveChanges();
45          }
46
47          return newUserCredential;
48      }
49      catch (Exception ex)
50      {
51          EduBaiMessaging.ConsoleLog(ex.ToString(), "Exception/UserCredentials Controller");
52          EduBaiMessaging.Throw(ex, "UserCredentials Controller", callerIsServer);
53          return null;
54      }
55  }

```

6 Client Side

Kommen wir nun zum eigentlichen Herzstück der Anwendung und damit auch einem Großteil der Codebasis, der Blazor WebAssembly Client-Anwendung. Wir werden im Folgenden einige besondere Aspekte des Blazor Frameworks kennenlernen, die es von anderen bekannten Web-Technologien unterscheiden. Auch werden wir bekannte Themen wie etwa die Verwendung von CSS oder den Umgang mit Mehrsprachigkeit in Webanwendungen betrachten.

6.1 WebAssembly

Eingangs haben wir bereits die Vorteile von WebAssembly angesprochen. Es ist ein offener Standard, der es ermöglicht, Programmcode in einem standardisierten Format zu kompilieren und im Browser auszuführen. Der Standard wurde vom World Wide Web Consortium (W3C) entwickelt und wird seit der ersten Veröffentlichung 2017 von allen modernen Webbrowsern unterstützt. WebAssembly (kurz WASM) verfolgt eine ähnliche Philosophie wie es Java mit der Java Virtual Machine (JVM) getan hat, indem man einen plattformunabhängigen Zwischencode erstellt, welcher dann von den verschiedensten Plattformen unterstützt werden kann, ohne dass Entwickler selbst Code für jede Plattform schreiben müssen. Viele Programmiersprachen unterstützen mittlerweile die Kompilierung nach WebAssembly (siehe Abbildung 13). Dies ermöglicht es, Anwendungen die zuvor eine Desktopmaschine und Installation benötigt haben, jetzt direkt im Browser auszuführen. Das senkt die Hürde für die Nutzung vieler Tools drastisch. Aus Entwicklersicht muss man aber dennoch beachten, dass die endgültige Ausführung des Codes in der Regel ein Browser übernimmt, man also auch mit dessen Limitationen umgehen können muss, wie bereits im Kapitel Server Side mehrmals erwähnt. Die wohl größte und anschaulichste Einschränkung ist wohl die Tatsache, dass WebAssembly, obwohl es direkt im Browser läuft, nicht auf das Dokument-Objekt-Modell (DOM) zugreifen kann. Es gibt aber sehr viele auftretende Limitationen eine Lösung. Für das DOM-Manipulationsproblem besteht diese darin, auf JavaScript auszuweichen, die Schnittstelle zwischen WASM und JavaScript nennt sich „JS Interop“ und wird im Kapitel 6.5 genauer betrachtet. Um mit einem positiven Punkt zu schließen, kann man noch ein Feature erwähnen, dass durch die mittlerweile sehr fortgeschrittene Integration von WASM entsteht. Viele Browser erlauben es, geeignete WASM-Anwendungen als sog. *Progressive Web App (PWA)* auf dem Endgerät „zu installieren“. Dadurch wird der im Browser zugängliche Bytecode heruntergeladen und kann direkt vom Betriebssystem ausgeführt werden, vorausgesetzt die Anwendung und das Betriebssystem unterstützen das [7]. Viele bekannte Webanwendungen sind mittlerweile als PWA verfügbar (siehe Abbildung 13), was einem, wenn gewünscht, einige Klicks beim Aufruf der Anwendung einsparen kann.

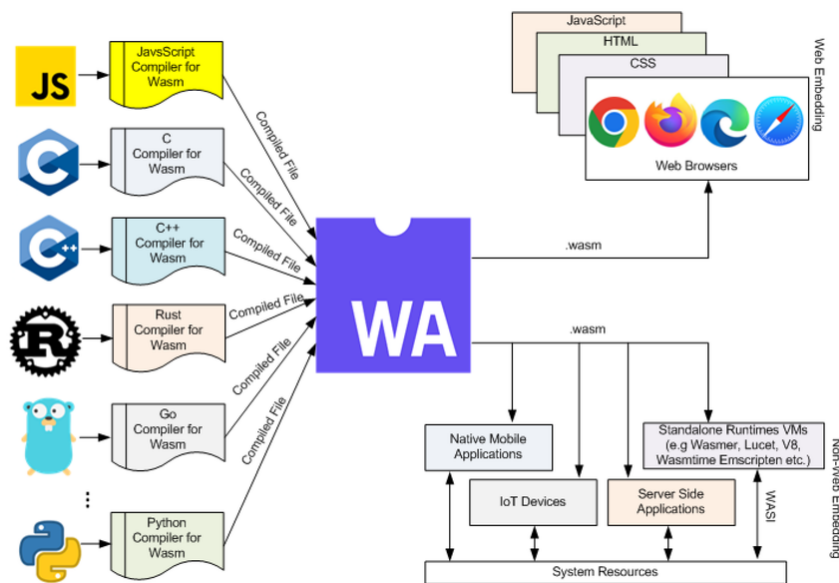


Abbildung 12: WebAssembly Flowchart [21]

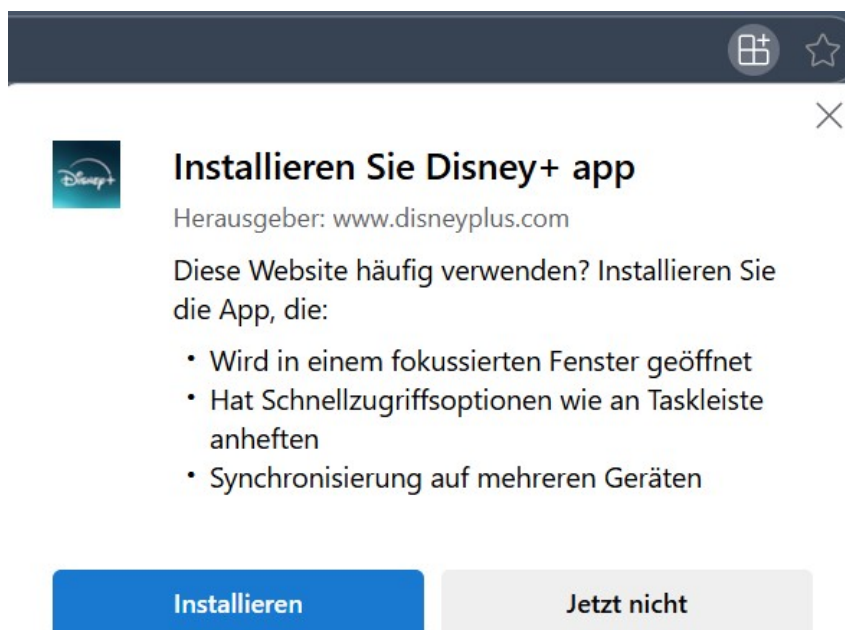


Abbildung 13: WebAssembly PWA Installation

6.2 MVC und MVVM Patterns

Bevor wir uns mit tatsächlichem Code beschäftigen, wollen wir uns ganz kurz die gewählte Strukturierung der Anwendungslogik ansehen. Die Teilung von Server- und Client-Code haben wir bereits betrachtet, die grundsätzliche Positionierung und den Grund dafür, werden wir im Kapitel Cross Platform Development noch genauer betrachten. Für uns interessant ist jetzt die Strukturierung aller Codeteile eines logisch zusammengehörenden Blocks. Wer sich schon einmal in der Webentwicklung in das Minenfeld der *Design Patterns* begeben hat, weiß, dass hier weniger Eindeutigkeit herrscht, als man es gern hätte. Eines der bekanntesten und am weitesten verbreiteten Muster ist das *Model-View-Controller* (MVC) Pattern (siehe Abbildung 14), welches die Anwendungslogik in drei Teile unterteilt: Model, View und Controller. Das Model repräsentiert dabei die Daten der Anwendung, die View ist für die Darstellung der Daten verantwortlich und der Controller vermittelt zwischen den beiden. Es gibt hier von Framework zu Framework aber bereits unterschiedliche Interpretationen, in welchen der Layer der Großteil der Logik untergebracht werden soll - Controller oder Model [10]. Noch dazu hat Microsoft und andere Frameworks in den letzten Jahren das *Model-View-ViewModel* (MVVM) Pattern populär gemacht, welches die Kommunikation der drei Teile leicht abändert und nahelegt, komplizierte Logik ganz auszulagern und über *Services* abzurufen. Wir werden uns für unser Projekt nicht zu sehr mit den Einzelheiten der Patterns beschäftigen. Wir werden stattdessen eine für uns günstige Aufteilung wählen und nur die wichtigste Grundidee streng verfolgen, nämlich die Entkoppelung der Benutzeroberfläche von der Anwendungslogik (siehe Abbildungen 15 & 16). Halten wir die Schnittstelle zwischen diesen zwei Teilen so schlank wie möglich, haben wir bei der fortlaufenden Entwicklung, welche vermutlich viele Oberflächenänderungen mit sich bringen wird, hoffentlich so wenig Integrationsprobleme wie möglich. Alle Komponenten, die nicht direkt mit einer Benutzeroberfläche zusammenhängen, werden wir passend gruppieren und strukturieren.

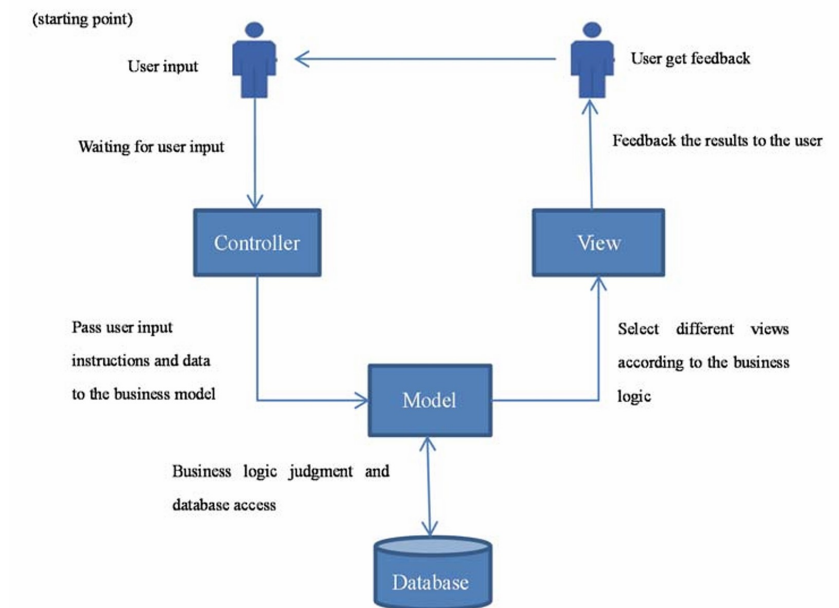


Abbildung 14: MVC Pattern [9]

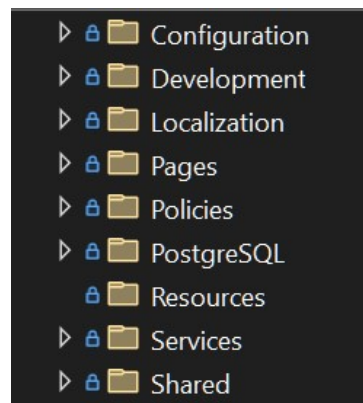


Abbildung 15: Dateistruktur

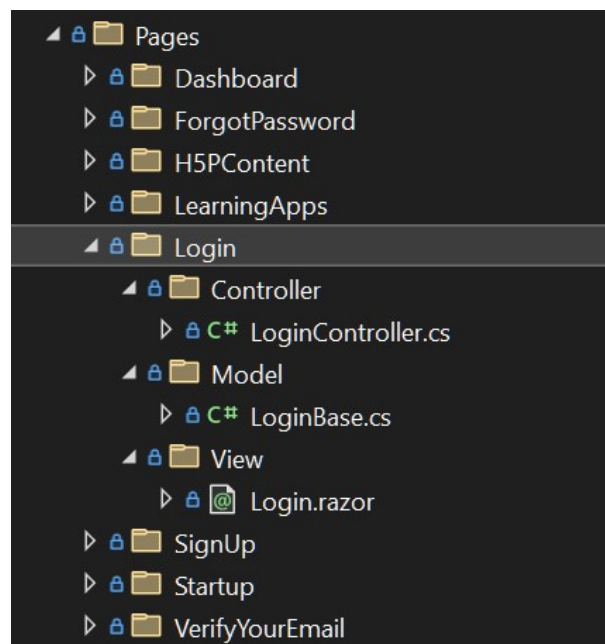


Abbildung 16: Pages MVC Teilung

6.3 Blazor Syntax

Blazor führt, neben seinen technischen Besonderheiten bei der Ausführung und Bereitstellung der Anwendung, auch eine eigene Syntax ein mit seinen sog. „Razor-Komponenten“ [18]. Eine Razor-Komponente [14] ist eine Datei mit der Endung *.razor*, die HTML zusammen mit speziellem C# Code unterstützt. Dadurch ist es einerseits möglich, auf ganz herkömmliche Weise HTML zu schreiben, für komplexe und dynamische Inhalte steht aber auch die mächtige C# Syntax zur Verfügung. In der einfachsten Variante würde das z.B. bedeuten, Methoden direkt in der Razor-Komponente zu definieren, und dann mit einem Button darauf zuzugreifen. Genau dieses Prinzip erhält man bei der Standardvorlage für das seitliche Navigationsmenü, das sich auf- und zuklappen lässt, wenn die Bildschirmgröße zu klein wird.

```

1  @attribute [AllowAnonymous]
2
3  <div class="top-row ps-3 navbar navbar-dark">
4      <div class="container-fluid">
5          <a class="navbar-brand" href="">EDUBAI</a>
6          <button title="Navigation menu" class="navbar-toggler" @onclick="ToggleNavMenu">
7              <span class="navbar-toggler-icon"></span>
8          </button>
9      </div>
10 </div>
11
12 <div class="@NavMenuCssClass nav-scrollable" @onclick="ToggleNavMenu">
13     <nav class="flex-column">
14         ...
15     </nav>
16 </div>
17
18 @code {
19     private bool collapseNavMenu = true;
20
21     private string? NavMenuCssClass => collapseNavMenu ? "collapse" : null;
22
23     private void ToggleNavMenu()
24     {
25         collapseNavMenu = !collapseNavMenu;
26     }
27 }

```

Listing 17: Sidebar Razor Komponente

In Listing 17 wird mit dem `@` Präfix der C# Code in die Razor-Komponente eingebettet. Ganz unten erstellen wir so einen Block `@code`, in dem wir die Methode „ToggleNavMenu()“ definieren, um sie oben auf Knopfdruck mit dem `@onclick` Attribut aufzurufen.

Besonders mächtig wird diese Syntax dann, wenn wir bekannte Ablaufstrukturen wie Schleifen oder Bedingungen für das Rendern der Oberfläche verwenden. Wir betrachten als Beispiel die Komponente „LearningApps.razor“, die eine zuvor unbekannte Anzahl an Lernapp-Kacheln darstellen soll. Wir können hier die Razor-Syntax nutzen, um eine Schleife zu definieren, die über eine Liste von Lernapps iteriert und für jede App ein Kachel-Element rendert.

```

1  ...
2  <AuthorizeView>
3      ...
4      <Authorized>
5          @if (LearningApps == null)
6          {
7              // TODO: Add a loading spinner
8              <p>@Localizer[Localization.loading]</p>
9          }
10         else
11         {
12             <div class="learningapps">
13                 <div class="learningapps__selection">
14                     ...
15                 </div>
16
17                 <div class="learningapps__all">
18                     <h1>@Localizer[Localization.learning_apps]</h1>
19                     <div class="learningapps__all__apps">
20                         @foreach (LearningAppData la in LearningApps!)
21                         {
22                             <LearningAppCard Data=la NavigationManager=NavigationManager />
23                         }
24                     </div>
25                     <div class="learningapps__all__overlay"></div>
26                 </div>
27             </div>
28         }
29     </Authorized>
30 </AuthorizeView>

```

Listing 18: LearningApps Razor Komponente

Wir können in Listing 18 Zeile 20 mit dem *@foreach*-Statement über die Liste der Lernapps iterieren und direkt im HTML-Block auf die Eigenschaften des jeweiligen Objektes zugreifen und diese passend anzeigen. In unserem Fall überliefern wir das gesamte Datenobjekt an eine weitere Razor-Komponente, die dann jeweils eine Instanz der Kachel-Komponente für jede App erstellt und rendert. Es gibt für verschiedenste weitere Anwendungsfälle eine passende Razor-Syntax (siehe Listing 19), um so ohne viel Aufwand, viele Dinge ohne den Einsatz von sonst notwendigem JavaScript-Code zu erledigen. Es gibt jedoch auch immer Sonderfälle, die noch nicht direkt in Razor abgebildet werden können. Bevor wir uns aber ansehen, wie wir für diese Fälle doch auf JavaScript zurückgreifen, wollen uns in Kapitel Dependency Injection ansehen, warum und wie das in Blazor überhaupt möglich ist.

1	<code>@page "/learningapps"</code>	<code>@*Markiert Pfad*@</code>
2	<code>@attribute [Authorize(Roles = "Student, Teacher")]</code>	<code>@*Autorisierung*@</code>
3	<code>@layout MainLayout</code>	<code>@*Layout Vorlage einbinden*@</code>
4	<code>@using SharedComponents.Pages.LearningApps.Model</code>	<code>@*Namespaces einbinden*@</code>
5	<code>@inherits LearningAppsBase</code>	<code>@*CodeBehind File verknüpfen/vererben*@</code>

Listing 19: Razor Syntax Beispiele

6.4 Dependency Injection

Viele Dienste, die für die Entwicklung von Webanwendungen nützlich sind oder sogar benötigt werden, werden nicht direkt mit dem Framework mitgeliefert. Die gängigste Möglichkeit, diese Dienste in die Anwendung zu integrieren, ist in Blazor die Verwendung von „Dependency Injection“ (DI). Dabei handelt es sich grundlegend um ein Design Pattern, welches zunächst nur die Abhängigkeiten zwischen Klassen und Objekten beschreibt. Eine Klasse gibt also an, welche Funktionalitäten sie von außen benötigt, um ihre Aufgabe erfüllen zu können. Das Framework stellt dann sicher, dass diese Abhängigkeiten zur Laufzeit passend aufgelöst werden und durch tatsächliche Instanzen ersetzt werden. Für das Design von Bibliotheken hat das den großen Vorteil, dass einzelne Teile viel modularer und damit auch einfacher zu testen und auszutauschen sind. Man ist also nicht mehr an eine bestimmte Implementierung gebunden, sondern fordert nur das Versprechen ein, zu gegebenen Zeitpunkt eine bestimmte Funktionalität zu erhalten. In Blazor kann man so viele Dienste direkt in den verschiedenen Razor-Komponenten anfordern (siehe Listing 21 & 22) und verwenden, als wären sie direkt in der Komponente definiert. In der Startup-Logik der Anwendung, die wir im Kapitel 4 Server Side bereits betrachtet haben, können wir diese Dienste dann registrieren und konfigurieren (siehe Listing 20). Wir werden später im Kapitel 7.3 Plattform abhängiger Code auch einen selbst definierten Dienst kennenlernen, der dieses Pattern verfolgt.

```

1  ...
2
3  //Add Dependency Injection Services here
4  builder.Services.AddScoped(sp => new HttpClient {
5      BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
6  builder.Services.AddSingleton<IPlatformInfo, WebPlatformInfo>();
7  builder.Services.AddBlazoredLocalStorage();
8  builder.Services.AddLocalization();
9  builder.Services.AddScoped<AuthenticationStateProvider, EdubaiAuthStateProvider>();
10
11  ...

```

Listing 20: BlazorWASM Program Startup

```

1  public class LoginBase : ComponentBase
2  {
3      [Inject]
4      public IPlatformInfo PlatformInfo { get; set; } = default!;
5      [Inject]
6      public HttpClient Http { get; set; } = default!;
7      [Inject]
8      public IJSRuntime JS { get; set; } = default!;
9      [Inject]
10     public NavigationManager NavigationManager { get; set; } = default!;
11     [Inject]
12     public ILocalStorageService LocalStorage { get; set; } = default!;
13     [Inject]
14     public AuthenticationStateProvider AuthenticationStateProvider { get; set; } = default!;
15     [Inject]
16     public IStringLocalizer<SharedComponents.Localization.Localization> Localizer {
17         get; set; } = default!;
18     ...
19 }

```

Listing 21: DI in Login Model

```
1 @page "/login"
2 @attribute [AllowAnonymous]
3
4 @layout LoginLayout
5
6 @using SharedComponents.Pages.Login.Model
7 @inherits LoginBase
8
9 @inject IPlatformInfo PlatformInfo
10 @inject HttpClient Http
11 @inject IJSRuntime JS
12 @inject NavigationManager NavigationManager
13 @inject ILocalStorageService LocalStorage
14 @inject AuthenticationStateProvider AuthenticationStateProvider
15 @inject IStringLocalizer<SharedComponents.Localization.Localization> Localizer
16
17
18 <PageTitle>@Localizer[Localization.login]</PageTitle>
19 <div class="screen">
20     <div class="screen__image"></div>
21     <div class="screen__overlay"></div>
22     <div class="screen__content">
23         ...
24     </div>
25 </div>
26
27 <div id="blob"></div>
28 <div id="blur"></div>
29
30
```

Listing 22: Alternativ: DI direkt in Login Razor Komponente

6.5 JS Interop

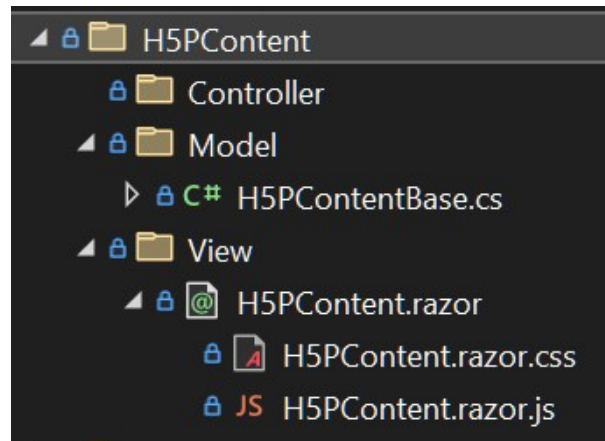


Abbildung 17: H5P Modul Struktur

Wir werden uns hier, beispielhaft für die Verwendung des *JavaScript Interop*, die Verwendung der JS-Bibliothek für die, im späteren Kapitel H5P-Content beschriebenen, H5P-Inhalte ansehen (siehe Listing 23). Wir haben unseren JavaScript-Code wie in Abbildung 17 gezeigt, geeignet zur dazugehörigen Razor-Komponente platziert. Visual Studio arbeitet hier mit sog. *Isolutions-Dateien*, also Dateien, die jeweils nur für die übergeordnete Komponente gelten. Durch die geschickte Benennung, mit dem Namen der Komponente als Präfix, geben wir Visual Studio so zu verstehen, dass die Datei *H5PContent.razor.js* zur Komponente *H5PContent.razor* gehört. In dieser Datei definieren wir eine einzige Funktion, die dem DOM den ausgewählten H5P-Inhalt hinzufügt.

```

1  let H5P = null;
2
3  export async function addh5pcontent(H5PID) {
4      const el = document.getElementById('h5p-container');
5      const options = {
6          h5pJsonPath: '/_content/SharedComponents/H5PContent/public/' + H5PID,
7          frameJs: '/_content/SharedComponents/js/h5p-standalone/frame.bundle.js',
8          frameCss: '/_content/SharedComponents/js/h5p-standalone/styles/h5p.css',
9          //optional:
10         frame: true,
11         copyright: true,
12         export: true,
13         downloadUrl: '/_content/SharedComponents/H5PContent/download/' + H5PID + '.h5p',
14         fullscreen: true,
15     }
16 }
17
18 if (H5P === null) {
19     // save H5P to variable, because main.bundle.js is loaded only once,
20     // and not again after navigating to another page and back
21     H5P = H5PStandalone.H5P; //H5PStandalone.H5P taken from
22     // wwwroot/js/h5p-standalone/main.bundle.js loaded in index.html
23 }
24 await new H5P(el, options);
25 }

```

Listing 23: H5PContent JavaScript Datei

Wir wollen nun Zugriff auf diese Datei erlangen, um diese Funktion an passender Stelle durch unseren C# Code aufzurufen (siehe Listing 24). Dafür benötigen wir den *IJSRuntime* Dienst, den wir mittels Dependency Injection in unsere Razor-Komponente einbinden können. Dieser Dienst erlaubt es uns, JavaScript-Dateien als Module zu laden und Funktionen aus diesen Modulen aufzurufen. Wir laden also zum Erstellzeitpunkt der Komponente dieses Modul und rufen die Funktion „addh5pcontent()“ auf, um den H5P-Inhalt in die Komponente einzufügen. So können wir jede Funktionalität, die wir nicht nativ in Blazor abbilden können oder wollen, in JavaScript auslagern und über den IJSRuntime Dienst aufrufen. Dadurch ist auch die Kompatibilität mit bestehenden beliebten JavaScript-Bibliotheken gegeben, die wir so in unsere Blazor-Anwendung integrieren können.

```
1 [Inject]
2 public IJSRuntime JS { get; set; } = default!;
3 public IJSObjectReference LearningAppsJSMModule { get; set; } = default!;
4
5 ...
6
7 protected override async Task OnInitializedAsync()
8 {
9     LearningAppsJSMModule = await JS.InvokeAsync<IJSObjectReference>("import",
10         "/_content/SharedComponents/Pages/H5PContent/View/H5PContent.razor.js");
11
12     ...
13
14     await LearningAppsJSMModule.InvokeVoidAsync("addh5pcontent", LearningAppData.ID);
15     await base.OnInitializedAsync();
16 }
```

Listing 24: H5PContent Model

6.6 File Management

Wir haben bereits kurz über eine Möglichkeit gesprochen, wie wir Daten in Blazor zumindest kurzzeitig speichern können, mit dem *Local Storage*. Hier den Anwendungsspeicher des Browsers zu verwenden, ist eine der einfachsten Möglichkeiten, um Daten zwischen den Sitzungen zu speichern. Man ist aber durch dessen Aufbau auf das Sichern von primitiven Datentypen (siehe Listing 25 Zeile 2) limitiert, was für z.B. die in Kapitel Authentifizierung und Autorisierung erwähnten *Session Tokens* kein Problem ist. Für größere Dateien oder besondere Dateiformate aber einen hohen Aufwand an Kodierung, Serialisierung und Deserialisierung erfordert.

```
1 public async Task<bool> SetAsync(  
2     string key, string value, IPlatformInfo PlatformInfo,  
3     ILocalStorageService LocalStorage = default!)  
4 {  
5     try  
6     {  
7         if (PlatformInfo.Platform == Platform.Web)  
8         {  
9             if (LocalStorage != null)  
10            {  
11                await LocalStorage.SetItemAsync(key, value);  
12                return true;  
13            }  
14            else  
15            {  
16                ...  
17            }  
18        }  
19        else if (PlatformInfo.Platform == Platform.Maui)  
20        {  
21            Preferences.Default.Set(key, value);  
22            return true;  
23        }  
24        else  
25        {  
26            ...  
27        }  
28    }  
29 }  
30 catch (Exception ex)  
31 {  
32     ...  
33 }  
34 }
```

Listing 25: Local Storage Key-Value Pair setzen

Für alle anderen Dateien müssen wir einen Ort finden, diese abzuspeichern, der für den Browser zugänglich ist. In Blazor haben wir dafür den *wwwroot*-Ordner, der für statische Dateien vorgesehen ist. Dieser Ordner wird mit der Standard-Vorlage mitgeliefert und enthält bereits einige Dateien, wie Stylesheets oder Icons. Dieser Ordner ist für alle statischen Dateien gedacht und wird beim Erstellen der Anwendung in den Anwendungsspeicher des Browsers kopiert. Von dort aus haben wir dann Zugriff auf alle darin enthaltenen Ressourcen. Das *wwwroot*-Verzeichnis fungiert also, wie der Name schon ahnen lässt, als Ausgangspunkt für das Referenzieren dieser statischen Dateien. Wir haben aber

beim JS-Interop bereits einen Spezialfall gesehen, bei dem wir auf die JavaScript-Datei *H5PContent.razor.js* zugegriffen haben, die sich aber nicht im *wwwroot*-Verzeichnis befindet. Wenn man ein paar Nachforschungen anstellt, findet man heraus, was das tatsächliche Stammverzeichnis der gesamten Anwendung ist. Wir können auf dieses mit dem *_content* Präfix zugreifen. Daraufhin können wir anfangen bei der Projektmappe (siehe Abbildung 18) unsere Struktur durchlaufen:

```
/_content/SharedComponents/Pages/H5PContent/View/H5PContent.razor.js
```

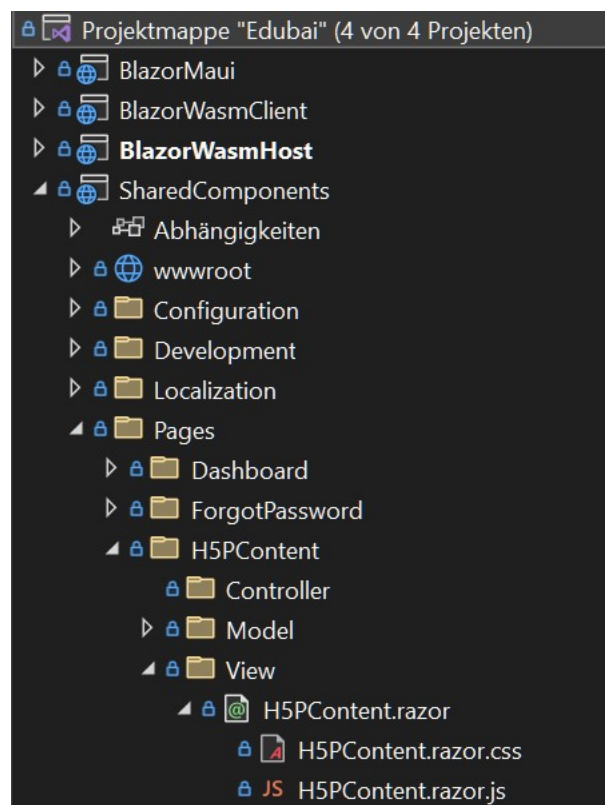


Abbildung 18: Content Ausgangsverzeichnis

Wie immer dürfen wir jetzt noch einen Sonderfall betrachten, für den dem Browser die Berechtigung fehlt. Da wir nicht wissen, wie viele H5P-Inhalte wir zur Verfügung haben, müssen wir diese Anzahl erst herausfinden. Es wäre natürlich denkbar, im Code diese Anzahl zu hinterlegen, aber das wäre nicht besonders flexibel. Stattdessen wollen wir das Verzeichnis, in dem die Lernapps liegen, auslesen und daraus die Anzahl ermitteln.

Auch wenn es dem Browser erlaubt ist, Ressourcen aus dem Verzeichnis zu laden, hat er trotzdem nicht die Berechtigung, das Dateisystem selbstständig zu durchlaufen. Das ist eine Sicherheitsmaßnahme, die verhindern soll, dass böartige Websites auf das Dateisystem des Nutzers zugreifen können. Wir müssen auch hier wieder auf den Server ausweichen. Die Funktionsweise ist hier genau gleich wie bei allen anderen besprochenen API-Aufrufen im Kapitel 4 Server Side. Wir müssen nun die *System.Collections.Generic.Dictionary*-Klasse dazu bringen, das richtige Verzeichnis auszulesen (siehe 26). Blazor macht uns hier einen letzten kleinen Strich durch die Rechnung, da sie der Aufbau des Projektes zwischen Entwicklung und Veröffentlichung minimal ändert. Zudem wechseln wir ja auch noch das Betriebssystem, was die Pfadangaben noch einmal ändert. Wir lösen dieses Problem einfach, indem wir alle möglichen Pfade hinterlegen und so lange auf ihre Existenz überprüfen, bis wir einen gültigen Pfad gefunden haben. Das ist vermutlich zwar nicht die eleganteste Lösung, aber sie funktioniert in allen Fällen und ist für unsere Zwecke ausreichend.

```

1 List<LearningAppData>? apps = null;
2 try
3 {
4     Dictionary<string, string> HostingPaths = new Dictionary<string, string>()
5     {
6         { "/var/www/edubai",
7           "../wwwroot/_content/SharedComponents/H5PContent/" + section },
8         { "C:/GitHub/Personal/Edubai/BlazorWasmASPNetHosted/Server",
9           "../../SharedComponents/wwwroot/H5PContent/" + section},
10        { "C:/GitHub/Personal/Edubai/BlazorMaui",
11          "../SharedComponents/wwwroot/H5PContent/" + section}
12    };
13    ...
14    var relativePaths = HostingPaths.Values;
15    foreach(var rp in relativePaths)
16    {
17        //to allow the different Hostingpaths to be moved around relative to their environment,
18        //we only try the relative paths and look for any "DirectoryNotFoundException"
19        //the first Directory that is found will be used
20        try
21        {
22            //Get all names of folders in the directory (= LearningApps)
23            string path = Path.Combine(Directory.GetCurrentDirectory(), rp);
24            DirectoryInfo[] dirs = new DirectoryInfo(path).GetDirectories();
25
26            apps = new List<LearningAppData>();
27            foreach (DirectoryInfo dir in dirs)
28            {
29                ...
30            }
31            break; //if no Exception accured up to this point, the directory is valid
32        }
33        catch (DirectoryNotFoundException ex)
34        {
35            // skip Exception logging, since this is somewhat expected
36            ...
37        }
38    }
39 }
40 catch (Exception ex)
41 {
42     ...
43 }

```

Listing 26: Verzeichnis auslesen

6.7 Localization

Für EduBai habe ich mich entschieden eine weitere Herausforderung anzunehmen, die für eine Webanwendung nicht unbedingt notwendig ist, jedoch von einer zeitgemäßen Webanwendung erwartet wird: die Mehrsprachigkeit. Sobald wir uns an irgendeine Art von Benutzeroberfläche wagen, stellt sich die Frage, welche Sprache wir für den abgebildeten Text verwenden wollen. Derzeit hätte es mit dem Wissen, dass wir uns im deutschen Sprachraum befinden, natürlich gereicht, die gesamte Anwendung auf Deutsch zu schreiben. Gleichzeitig wäre es auch denkbar international zu denken und Englisch als meistverbreitetste Sprache zu verwenden. Da es sich bei EduBai aber um eine Lernplattform handelt, habe ich mir den Anspruch gesetzt, so zugänglich wie möglich zu sein. Technisch sind hier natürlich einige Schritte notwendig, aber mit Blazor als modernes Framework waren diese ein nicht zu großes Problem.

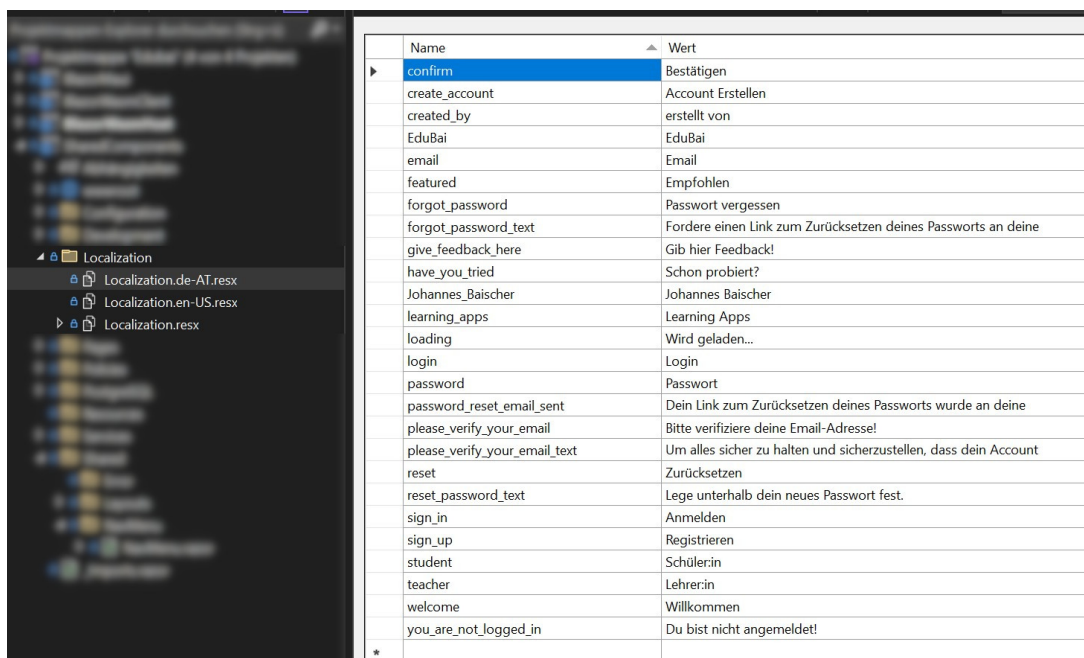


Abbildung 19: Ressourcen Datei

Blazor verwendet für die Mehrsprachigkeit die sog. „Ressourcen-Dateien“ (siehe Abbildung 19), die es ermöglichen, Texte in verschiedenen Sprachen zu hinterlegen und bei Bedarf auszulesen [13]. Jede Sprache bekommt hier eine eigene Datei, welche die verschiedenen Texte in der jeweiligen Sprache enthält. Für den Aufruf im Programmcode müssen alle Texte auch mit einem eindeutigen Schlüssel versehen werden, der sich auch in allen anderen Ressourcen-Dateien wiederfindet. Zusätzlich haben wir für die Kompatibilität

und Organisation auch eine *Default-Datei* die nur die Schlüssel enthält. Wir müssen die jeweilige Sprache jetzt noch auswählen, in Blazor geschieht das über die *CultureInfo*-Klasse, die regionale Einstellungen wie eben die Sprache aber auch z.B. das Datumsformat festlegt. Fügen wir also zum *Startup* die Localization hinzu, und setzen auch gleich die Sprache. Der Einfachheit halber setzen wir die Sprache auf die bevorzugte Sprache des Browsers.

```

1  var builder = WebAssemblyHostBuilder.CreateDefault(args);
2  ...
3  builder.Services.AddLocalization();
4  ...
5
6  //Build the host seperately from running it
7  //to access the services for initialization (e.g. LocalStorage)
8  var host = builder.Build();
9
10 await LocalizationInfo.Instance.SetAllCultureInfosAsync(
11     new WebPlatformInfo(),
12     host.Services.GetRequiredService<ILocalStorageService>());
13
14 await host.RunAsync();
15
16 // -----
17 public async Task<bool> SetAllCultureInfosAsync(
18     IPlatformInfo PlatformInfo, ILocalStorageService LocalStorage = default!)
19 {
20     string cultureCode;
21
22     //Get the language code from local storage/secure storage depending on the platform
23     cultureCode = await EdubaiLocalStorage.Instance.GetAsync(StorageTags.Culture.ToString(),
24         LanguageToLanguageCode(Languages.en_US), PlatformInfo, LocalStorage);
25
26     return await SetAllCultureInfosAsync(LanguageCodeToLanguage(cultureCode),
27         PlatformInfo, LocalStorage);
28 }

```

Listing 27: Localization Startup

Neben dem Standardwert für die Sprache, nämlich *en-US*, sieht man in Listing 27 auch die Tatsache, dass wir die Sprache einerseits aus dem *Local-Storage* auslesen, andererseits dort auch wieder speichern, sobald der Nutzer die Sprache ändert. Will man also in späterer Folge die Möglichkeit bieten, die Sprache aktiv in der Anwendung zu wechseln, kann man das hiermit sehr einfach umsetzen. Wir müssen uns jetzt nur noch ansehen, wie wir die Texte in der Razor-Komponente verwenden können. Dafür gibt es in Blazor den *IStringLocalizer*-Dienst, welchen wir wieder mittels Dependency Injection anfordern

können. Wir binden nun also diesen Dienst unter dem Namen *Localizer* ein und können ihn dann mittels Razor-Syntax wie ein C#-Dictionary verwenden. Wir geben also an passender Stelle den Schlüssel des Textes an, und der Localizer liefert uns den Text in der aktuell eingestellten Sprache zurück. Dadurch, dass wir eine Default-Ressourcendatei erstellt haben, müssen wir uns keine Sorgen machen, dass wir einen ungültigen Schlüssel verwenden. Wir können wie in Listing 28 auf die verschiedenen Schlüssel als Eigenschaften der Datei zugreifen mit dem „-Operator“.

```

1 <PageTitle>@Localizer[Localization.login]</PageTitle>
2 <div class="screen">
3     <div class="screen__content">
4         <div class="screen__title">
5             <span ...>@Localizer[Localization.sign_in]</span>
6             <a class="screen__title-link" href="/" target="_self">EduBai</a>
7         </div>
8         <div class="screen__inputOptions">
9             <form class="screen__form">
10                <input ... placeholder=@Localizer[Localization.email] @bind=@Email>
11                <input ... placeholder=@Localizer[Localization.password] @bind=@Password>
12                <input ... value=@Localizer[Localization.login]>
13            </form>
14            <div class="screen__advancedOptions">
15                <a ...>@Localizer[Localization.forgot_password]</a>
16                <a ...>@Localizer[Localization.sign_up]</a>
17            </div>
18        </div>
19    </div>
20 </div>

```

Listing 28: Localization in Razor Komponente

6.8 Benutzeroberfläche

Wir haben nun die gesamte Vorarbeit geleistet und sind bereit, die Benutzeroberfläche (Abbildungen 20 bis 30) zu erstellen. Dafür verwenden wir die Razor-Syntax, mit passend dazu erstellten Stylesheets, die wir als *.razor.css*-Dateien im selben Verzeichnis wie die Razor-Komponente ablegen. Dadurch wird sichergestellt, dass die Stylesheets nur für die jeweilige Komponente gelten, was uns unerwünschte Seiteneffekte durch andere Stylesheets erspart. Auch wenn die Blazor Standardvorlage bereits Bootstrap als CSS-Framework mitliefert, habe ich mich bewusst dagegen entschieden, dieses zu verwenden, um auch mein Wissen über CSS und die Gestaltung von Benutzeroberflächen zu erweitern. Da vor allem die Stylesheets einer solchen Anwendung sehr schnell sehr umfangreich werden können, werden

wir hier nicht alle speziellen Implementierungen betrachten. Bevor wir uns gleich einige Screenshots der Anwendung ansehen, wollen wir jedoch noch kurz über einen wichtigen technischen Aspekt jeder modernen Webanwendung sprechen: das *Responsive Design*. Dabei handelt es sich um die Fähigkeit, die Benutzeroberfläche an verschiedene Bildschirmgrößen und -auflösungen anzupassen. Da viele Nutzer:innen heutzutage vorrangig mobile Endgeräte verwenden, ist es wichtig, dass die Anwendung auf Smartphones und Tablets genauso gut aussieht und funktioniert wie auf Desktop-Computern. Hier unterscheidet sich Blazor nicht von anderen Web-Technologien. Wir können also altbekannte Techniken wie Media Queries und flexible Layouts verwenden, um die Benutzeroberfläche an verschiedene Bildschirmgrößen anzupassen.

```
1  /*media query for small screens*/
2  @media (max-width: 700px), (max-height: 700px) {
3      .screen {
4          width: 90%;
5          aspect-ratio: revert;
6          max-width: revert;
7      }
8
9      .screen__title-name {
10         font-size: 3rem;
11     }
12
13     .screen__title-link {
14         font-size: 1rem;
15     }
16
17     .screen__form {
18         gap: 1rem;
19         bottom: 10%;
20     }
21 }
```

Listing 29: Media Query Beispiel

In Listing 29 sehen wir das Stylesheet für die Login-Seite. Sollte es sich um einen Bildschirm handeln der entweder in der Länge oder der Breite kleiner als 700 Pixel ist, passen wir das Loginfeld an, indem wir die Schaltfläche vergrößern, und das Hochkant-Format des Feldes zurücksetzen. Dazu passen wir auch noch die Schriftgröße an. Wir müssen hier nicht auf viel mehr reagieren, da wir das Layout grundsätzlich bereits so dynamisch wie möglich gestaltet haben, ohne feste Breiten oder Höhen zu verwenden.

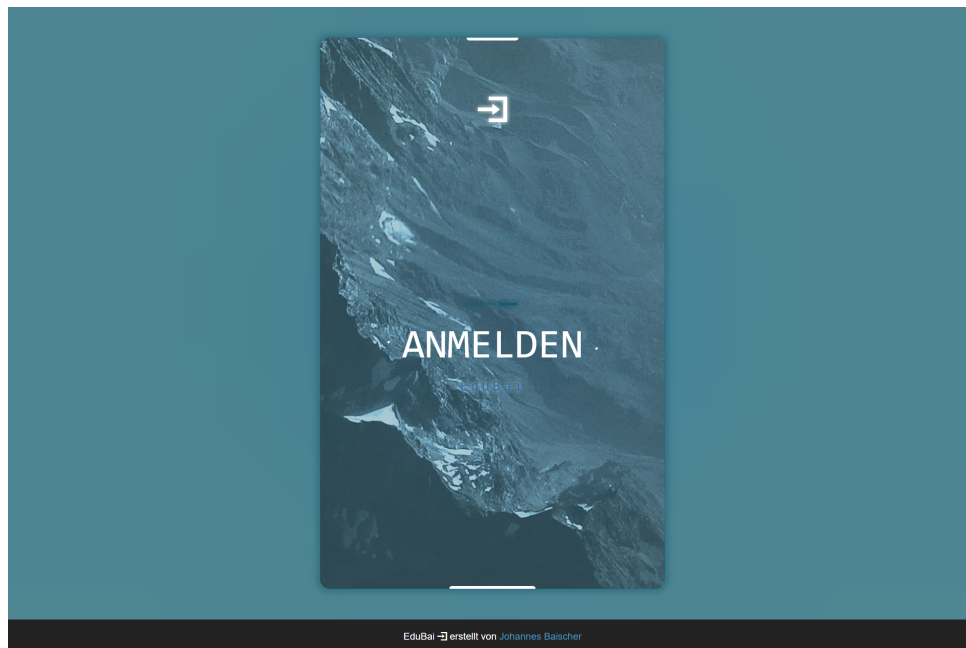


Abbildung 20: Login Seite 1/3

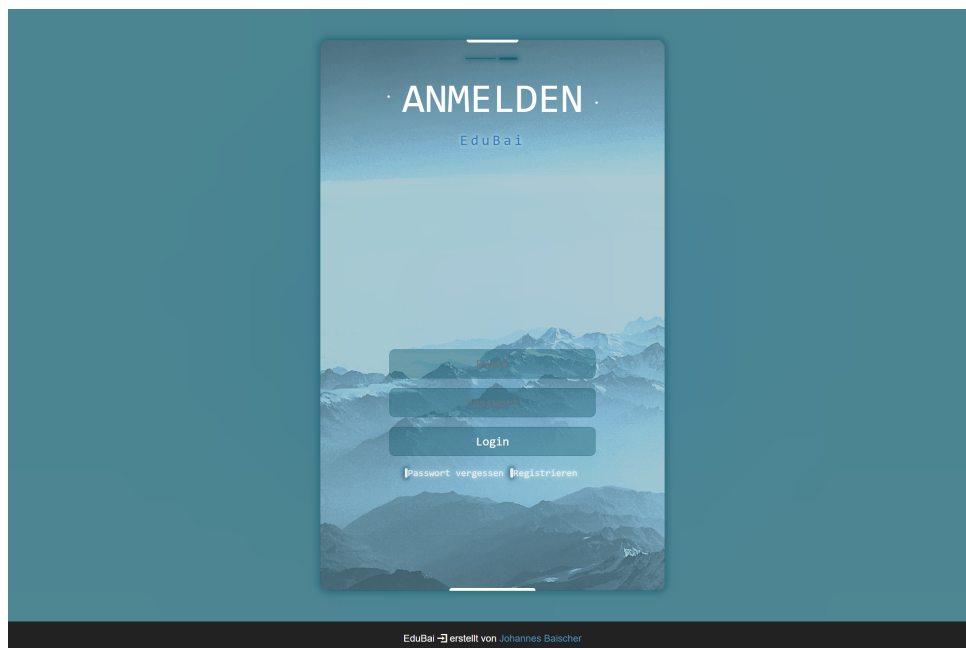


Abbildung 21: Login Seite 2/2

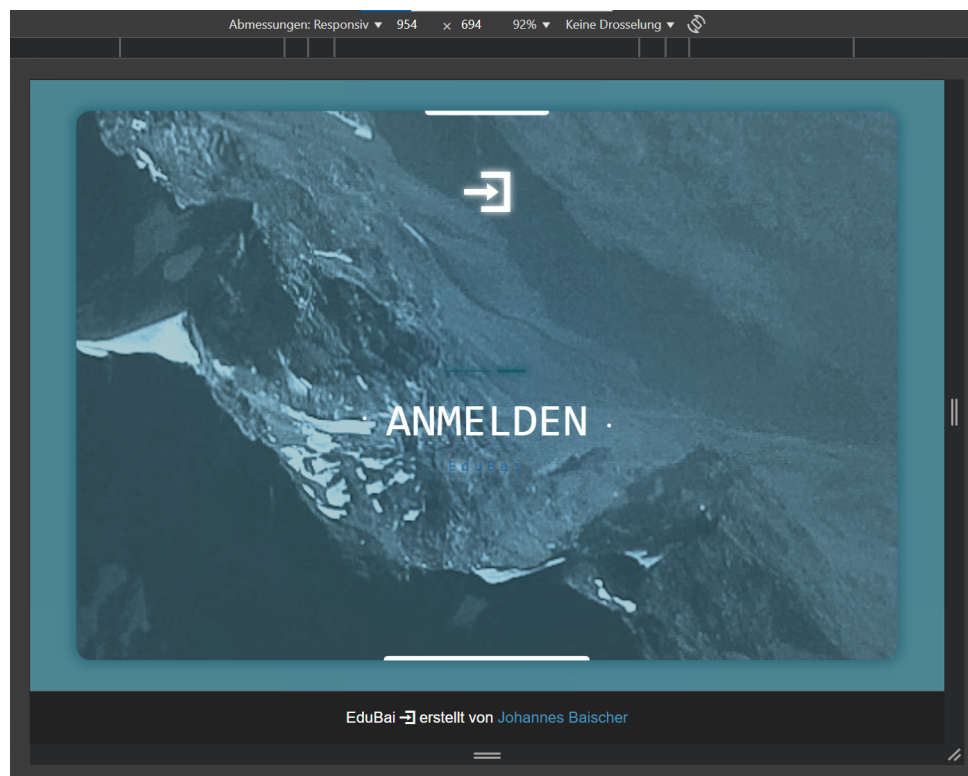


Abbildung 22: Login Seite 3/3



Abbildung 23: Registrierung

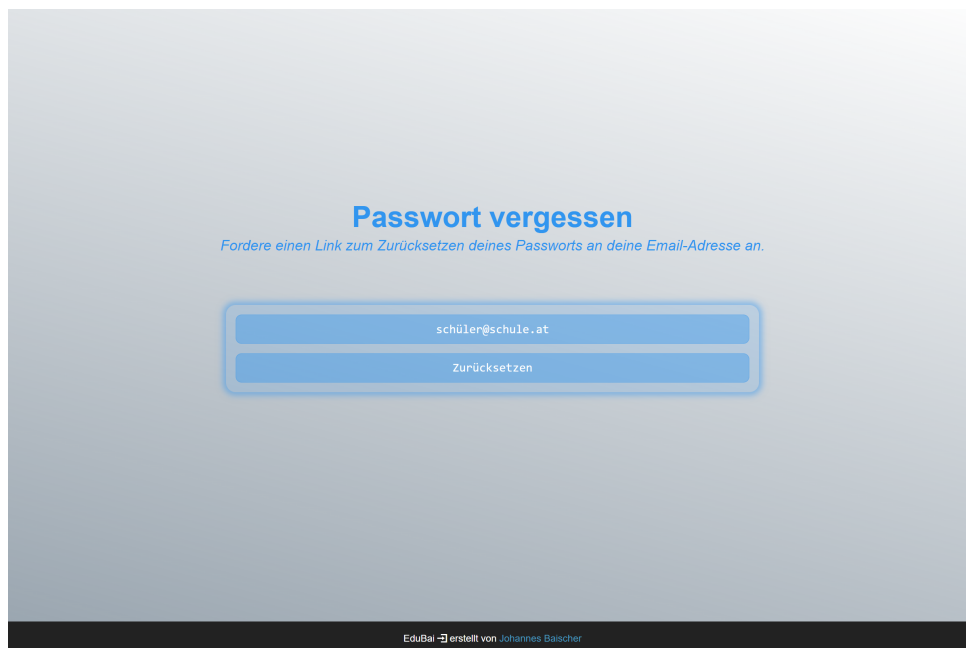


Abbildung 24: Passwort vergessen



Abbildung 25: Lernapps 1/4

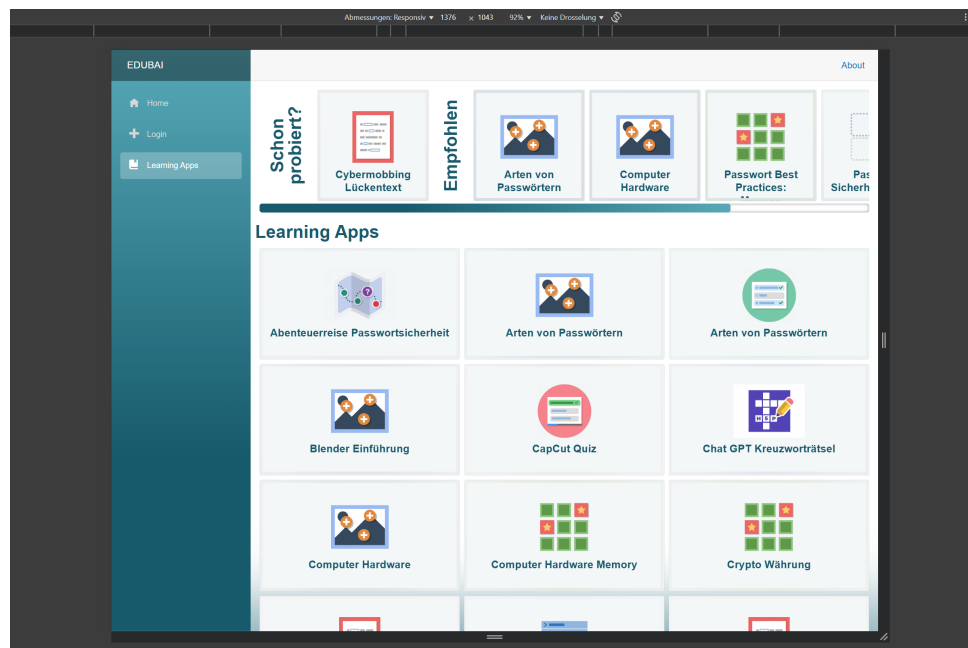


Abbildung 26: Lernapps 2/4

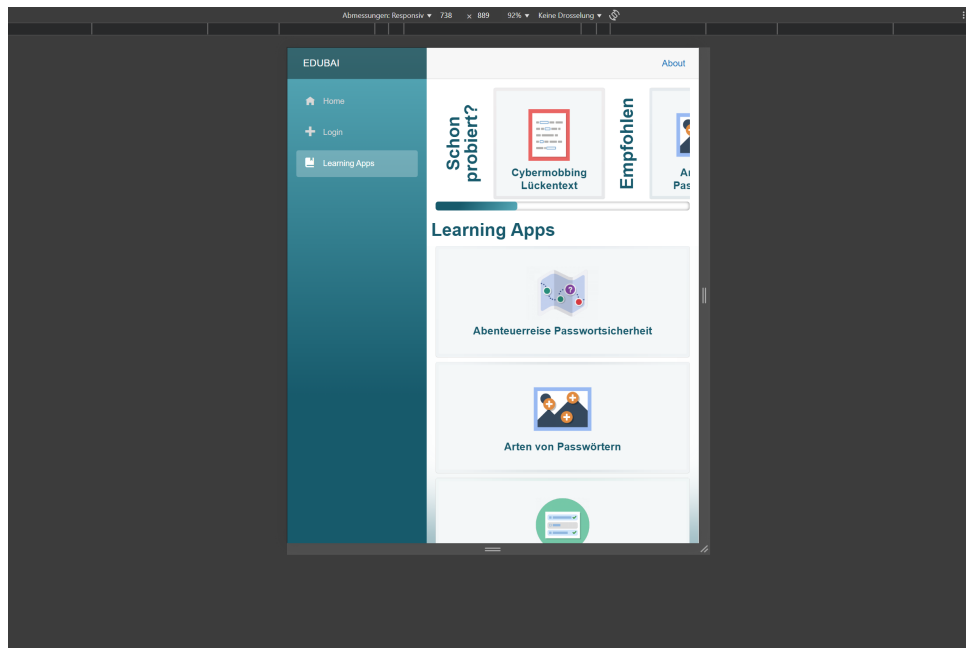


Abbildung 27: Lernapps 3/4

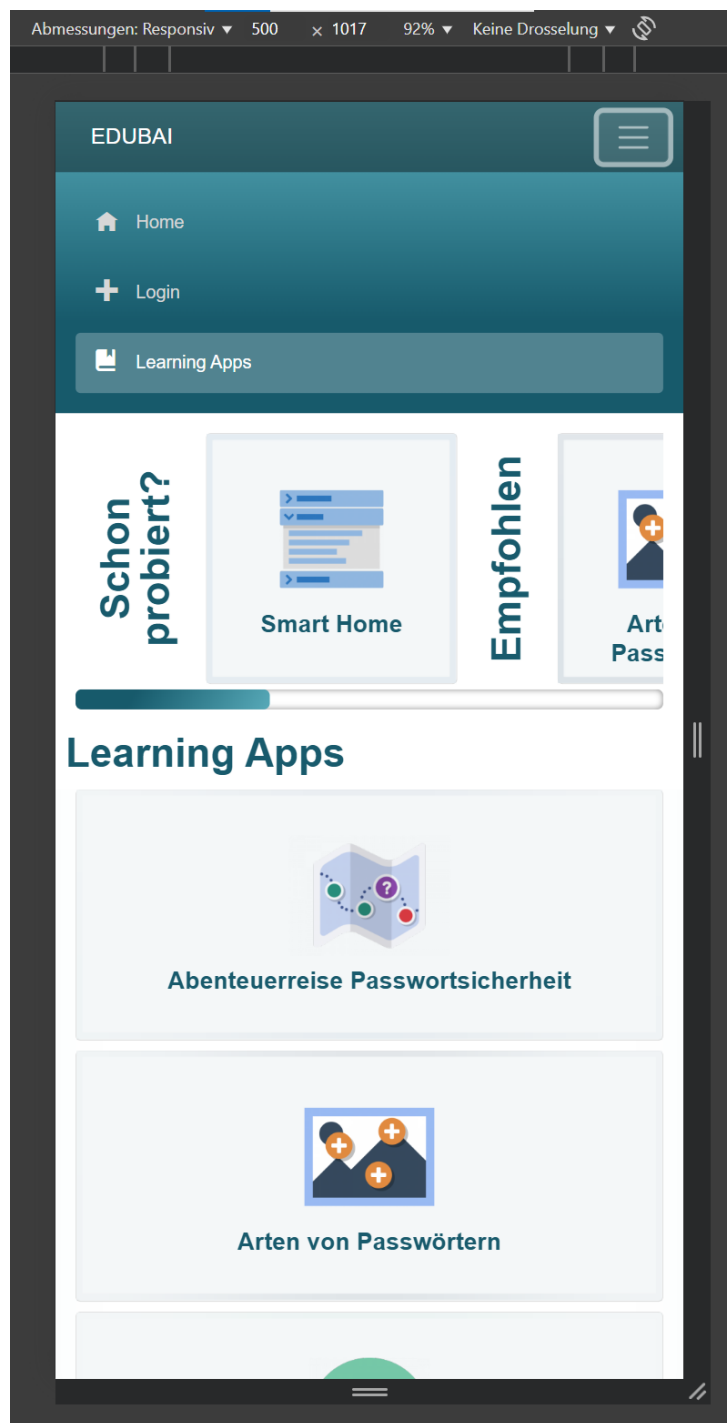


Abbildung 28: Lernapps 4/4

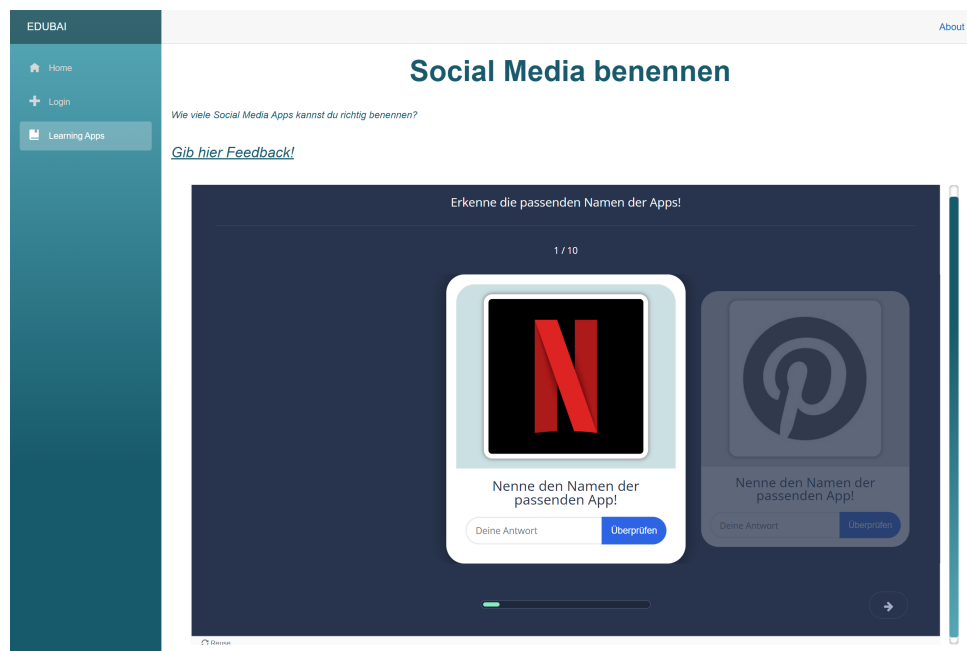


Abbildung 29: Beispiel App 1/2



Abbildung 30: Beispiel App 2/2

7 Cross Platform Development

In einer Zeit, in der das Smartphone längst den Desktop-PC als primäres Endgerät abgelöst hat, ist es für Webanwendungen unerlässlich, auf verschiedenen Plattformen und Geräten lauffähig zu sein. Bei einer Webanwendung hat man einen gewissen Vorteil was die Plattformunabhängigkeit betrifft, da alle gängigen Endgeräte und Betriebssysteme Webbrowser unterstützen. Neben *Responsive Design* wie im Kapitel 6.8 Benutzeroberfläche beschrieben, gibt es aber noch weitere Möglichkeiten, die App für die Endnutzer:innen so komfortabel wie möglich zu gestalten. Blazor hat sich hier über die Jahre zu einem leistungsstarken plattformübergreifenden Framework entwickelt, das es ermöglicht, den WebAssembly-Bytecode auf den verschiedenen Plattformen fast nativ auszuführen. Warum nur „fast“ nativ, werden wir im Kapitel 7.2 Blazor MAUI sehen. Zunächst wollen wir aber betrachten, wie wir in Blazor eine gemeinsame Codebasis für die verschiedenen Plattformen schaffen können.

7.1 Razor Class Library

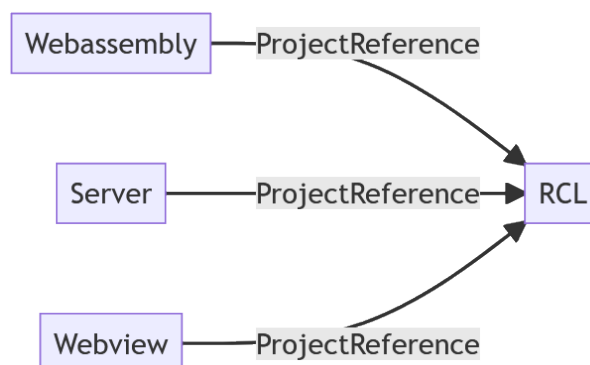


Abbildung 31: RCL Projektstruktur

Wir haben in Visual Studio die Möglichkeit, zu einer bestehenden Projektmappe weitere Projekte hinzuzufügen, um sie besser zu organisieren und Komponenten, die eng miteinander zusammenarbeiten, in einem gemeinsamen Projekt zu bündeln. Für Blazor haben wir hier eine besondere Art von Projekt zur Verfügung, die sog. „Razor Class Library“ (RCL). Eine RCL ist ein Projekt, welches es ermöglicht, Razor-Komponenten, CSS-Dateien und andere Ressourcen zu erstellen und in die Bibliothek zu bündeln. Mittels einer *Projektreferenz* (siehe Abbildung 31) kann die RCL dann mit anderen (Blazor-)Projekten verknüpft und auf dessen Klassen und Namespaces verwiesen werden. Das ermöglicht

es uns einerseits, Code zwischen Server und Client zu teilen. Im speziellen können wir so aber auch andere Projekte erstellen, die nativ auf anderen Geräten mit Windows, Android iOS etc. laufen und dennoch auf die RCL zugreifen können. Organisiert man so die Anwendungslogik, kann man ohne viel Mehraufwand die Entwicklung für verschiedenen Plattformen gleichzeitig bewerkstelligen. Wir teilen für EduBai die Anwendungslogik mit drei verschiedenen Projekten: BlazorWASMHost, BlazorWASMClient und BlazorMaui (siehe Abbildung 32).



Abbildung 32: Projektmappe mit RCL

Die einzelnen Projekte beinhalten jeweils nur für sich selbst spezifische Logik, wie etwa die Startup-Logik des Servers, oder die API-Schnittstelle für Datenbankabfragen durch den Client. Ein Detail, welches wir in Kapitel 4 Server Side zur API noch etwas verschwiegen haben, ist die Tatsache, dass der Server gar keine Logik für Datenbankabfragen selbst hält (siehe Abbildung 30). Auch diese Logik befindet sich in der RCL, die wir in der Server-Anwendung lediglich referenzieren. Sobald die API eine Anfrage erhält, ruft sie die entsprechende Methode der RCL auf, die dann die Datenbankabfrage durchführt und das Ergebnis zurückliefert.

```

1 private SharedComponents.PostgreSQL.UserCredentialsController GetSharedController(
2     JsonElement services)
3 {
4     IPlatformInfo platformInfo = services[0].Deserialize<WebPlatformInfo>();
5     HttpClient httpClient = services[1].Deserialize<HttpClient>();
6
7     string token = HttpContext.Request.Headers["Authorization"]
8         .ToString().Replace("Bearer ", "");
9     httpClient.DefaultRequestHeaders.Authorization
10         = new AuthenticationHeaderValue("Bearer", token);
11
12     return new SharedComponents.PostgreSQL.UserCredentialsController(
13         platformInfo, httpClient, null);
14 }
15
16 //...
17
18 [HttpPost]
19 [Authorize(Roles = "System, Student, Teacher")]
20 [ActionName("Read")]
21 [ProducesResponseType(typeof(UserCredential), 200)]
22 public async Task<IActionResult> Internal_UserCredentials_Read_Post(
23     [FromBody] JsonElement body)
24 {
25     var services = body[0];
26     var args = body[1];
27
28     //calls SharedComponents Controller
29     SharedComponents.PostgreSQL.UserCredentialsController ucc
30         = GetSharedController(services);
31
32     //calls Methode as Server (callerIsServer = true)
33     UserCredential? uc = null;
34     HttpStatusCode? statusCode = HttpStatusCode.OK;
35
36     try
37     {
38         uc = await ucc.Read(args[0].Deserialize<string>(),
39             args[1].Deserialize<string>(), true);
40     }
41     catch (HttpRequestException ex)
42     {
43         statusCode = ex.StatusCode;
44     }
45
46     return StatusCode((int)statusCode, uc);
47 }

```

Listing 30: API RCL Aufruf

Wir erstellen hier mit den überlieferten Service- und Parameter-Daten eine Instanz der RCL-Klasse „UserCredentialsController“ mit dem expliziten Hinweis, die Klasse aus dem RCL-Namespace „SharedComponents.PostgreSQL“ zu verwenden. Hier ist die eigentliche Logik der Datenbankabfrage implementiert, die wir über eine gleichnamige Methode zum API-Aufruf verwenden können (in diesem Fall die Methode *ucc.READ()*). Diese besondere Trennung der Zuständigkeiten ermöglicht es uns jetzt auch, von anderer Stelle aus, dieselbe Logik für dieselbe Datenbankabfrage zu verwenden. Wir haben bereits kurz das Projekt *BlazorMaui* erwähnt. Dieses ermöglicht es, für verschiedene Plattformen native Apps zu erstellen und hat, anders als der BlazorWASM-Client, durchaus die Berechtigung, eigenständig Datenbankabfragen durchzuführen. Wir benötigen also hier keinen Umweg über eine API-Schnittstelle, sondern können durch die Aufteilung der Anwendungslogik in die RCL direkt auf die entsprechenden Methoden zugreifen.

7.2 Blazor MAUI

Wir wollen nun kurz betrachten, was „BlazorMAUI“ [16] (bzw. offiziell „NET MAUI Blazor Hybrid“) genau ist, und wie wir es verwenden können, um eine native App für verschiedene Plattformen zu erstellen. DOTNET MAUI (ohne Blazor) ist ein von Microsoft entwickeltes Framework, welches es ermöglicht, plattformübergreifende Anwendungen für Windows, macOS, iOS und Android zu erstellen. Es ist die Weiterentwicklung von Xamarin, einem in 2011 veröffentlichtem Projekt, welches erstmals Cross-Platform Development für Microsoft Anwendungen ermöglichen sollte. 2024 wurde der Support für Xamarin offiziell eingestellt und durch .NET MAUI ersetzt [20]. Die in C# und XAML entwickelten Anwendungen können so durch die MAUI Plattform auf verschiedenen Geräten (iOS, Android, macOS, Windows (siehe Abbildung 33)) nativ ausgeführt werden. DOTNET MAUI Blazor Hybrid erweitert dieses Konzept, indem es erlaubt, mit der Blazor-Syntax entwickelte Webanwendungen auch über die MAUI Engine als native App auszuführen. Dafür verwendet BlazorMAUI *Microsoft Edge WebView2* um ein natives, eingebettetes Browserfenster zu erstellen, in dem die Blazor WebAssembly Anwendung ausgeführt wird. Dadurch können wir die Vorteile von Blazor WebAssembly (Wiederverwendbarkeit von Code und die einfache Entwicklung von Webanwendungen) mit den Vorteilen von .NET MAUI (Zugriff auf native APIs und die Möglichkeit, plattformübergreifende Anwendungen zu erstellen) kombinieren.

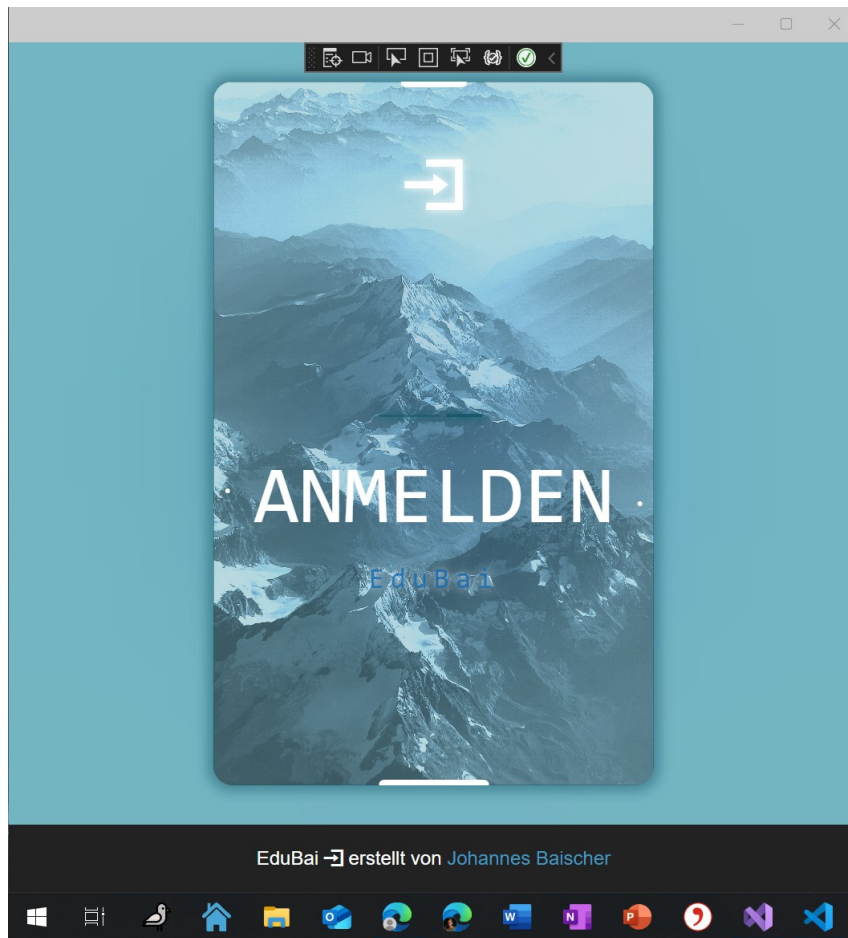


Abbildung 33: EduBai Windows App

7.3 Plattform abhängiger Code

So schön die Idee auch ist, für alle möglichen Geräte und Plattformen nur einmal Code schreiben zu müssen, so gibt es leider doch immer wieder Teile der Anwendung, die sich auf verschiedenen Plattformen einfach nicht auf dieselbe Art implementieren lassen. Derzeit gibt es zwei große Knackpunkte, an denen es wichtig ist zu wissen, auf welcher Plattform die Anwendung läuft. Zum einen ist das ein Punkt, den wir schon mehrere Male angesprochen haben: die Kommunikation mit der Datenbank. Da wir die Logik für die Datenbankabfrage übergreifend in einer RCL haben, würde ohne Unterscheidung zwischen verschiedenen Plattformen einer von zwei Fällen eintreten:

1. Wir könnten den Zugriff ohne API-Anfrage implementieren, wodurch die BlazorWASM-Anwendung versuchen würde, direkt auf die Datenbank zuzugreifen, was in einem

Laufzeitfehler enden würde

2. Wir könnten den Zugriff immer über eine API-Anfrage laufen lassen, wodurch wir auch für die native Windows Anwendung einen Server zur Verfügung stellen müssten

Beide Optionen sind nicht optimal. Unsere beste Chance ist es, herauszufinden auf welcher Plattform die Anwendung läuft und dafür spezifischen Code zu verwenden. Bevor wir betrachten, wie wir das bewerkstelligen könnten, wollen wir uns das zweite große plattformspezifische Problem ansehen: den Zugriff auf lokalen Gerätespeicher. Damit ist einerseits klassisches Dateimanagement gemeint, das von Betriebssystem zu Betriebssystem unterschiedlich ist, aber auch die Verwendung von „Local Storage“ bzw. „Session Storage“. Hier gibt es jeweils verschiedene Bibliotheken, die die verschiedenen Plattformen unterstützen, um temporäre Nutzerdaten oder Sitzungsinformationen speichern können. Ein Beispiel für dessen Nutzung in EduBai wäre das Speichern des aktuellen Session Tokens, damit sich der Nutzer nicht bei jedem Seitenaufruf neu anmelden muss.

Zum Glück bietet auch hier Blazor elegante Lösungen, einen solchen *Dienst* zu erstellen der uns sagen kann, auf welcher Plattform wir uns befinden. Wir erstellen dafür ein Interface, welches wir je nach Plattform passend implementieren. Diesen Dienst können wir zum Erstellungszeitpunkt der Anwendung registrieren und wie in Kapitel 6.4 Dependency Injection als Dependency Injection in die Komponenten einbinden. Dadurch können wir in der Komponente abfragen, auf welcher Plattform wir uns befinden und den entsprechenden Code ausführen. Listings 31 bis 33 zeigt diese Logik am Beispiel des Zugriffs auf den Gerätespeicher zum Speichern des Session Tokens.

```

1 namespace SharedComponents.Services
2 {
3     public class EdubaiAuthstateProvider : AuthenticationStateProvider
4     {
5         private IPlatformInfo PlatformInfo { get; set; } = default!;
6         private ILocalStorageService LocalStorage { get; set; } = default!;
7         private HttpClient Http { get; set; } = default!;
8
9         /// <summary>
10        /// Constructor that gets calles by the Engine,
11        /// resolving all parameters to registered Services of the same Type.
12        /// </summary>
13        /// <param name="localStorage"></param>
14        /// <param name="http"></param>
15        /// <param name="platformInfo"></param>
16        public EdubaiAuthstateProvider(
17            ILocalStorageService localStorage,
18            HttpClient http,
19            IPlatformInfo platformInfo)
20        {
21            this.LocalStorage = localStorage;
22            this.Http = http;
23            this.PlatformInfo = platformInfo;
24        }
25
26        public async Task<AuthenticationState> GetAuthenticationStateAsync(
27            IEnumerable<Claim>? claims, string passwordHash)
28        {
29            string token = null;
30
31            if(claims == null)
32            {
33                if (PlatformInfo.Platform == Platform.Maui)
34                {
35                    token = EdubaiLocalStorage.Instance.Get(
36                        "edubaiauthstatetoken", "", PlatformInfo);
37                }
38                else
39                {
40                    token = await EdubaiLocalStorage.Instance.GetAsync(
41                        "edubaiauthstatetoken", "", PlatformInfo, LocalStorage);
42                }
43                //...
44            }
45            //...
46        }
47        //...
48    }
49 }

```

Listing 31: AuthstateProvider mit Plattform Überprüfung

```
1 namespace BlazorMaui
2 {
3     public static class MauiProgram
4     {
5         public static MauiApp CreateMauiApp()
6         {
7             //...
8             //Add your services here
9             builder.Services.AddScoped(sp => new HttpClient {
10                 BaseAddress = new Uri("https://localhost:7226/")
11             });
12             builder.Services.AddSingleton<IPlatformInfo, MauiPlatformInfo>();
13             builder.Services.AddBlazoredLocalStorage(); //Not used as a service,
14                 // but added for compatibility with Shared Components with Blazor
15             builder.Services.AddLocalization();
16             builder.Services.AddScoped<AuthenticationStateProvider,
17                 EdubaiAuthStateProvider>();
18             //...
19             //Set Culture Info of the entire application from secure storage
20             LocalizationInfo.Instance.SetAllCultureInfos(new MauiPlatformInfo());
21
22             return builder.Build();
23         }
24     }
25 }
```

Listing 32: BlazorMAUI Service Registrierung

```
1  //...
2  //Add Dependency Injection Services here
3  builder.Services.AddScoped(sp => new HttpClient
4  {
5      BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
6  });
7  builder.Services.AddSingleton<IPlatformInfo, WebPlatformInfo>();
8  builder.Services.AddBlazoredLocalStorage();
9  builder.Services.AddLocalization();
10 builder.Services.AddScoped<AuthenticationStateProvider, EdubaiAuthStateProvider>();
11 //...
12 //Build the host seperately from running it
13 // to access the services for initialization (e.g. LocalStorage)
14 var host = builder.Build();
15 await LocalizationInfo.Instance.SetAllCultureInfosAsync(new WebPlatformInfo(),
16     host.Services.GetRequiredService<ILocalStorageService>());
17
18 await host.RunAsync();
```

Listing 33: BlazorWASM Service Registrierung

Wir sehen hier wie der selbe Dienst, nämlich *IPlatformInfo*, in der BlazorWASM Anwendung und der BlazorMAUI Anwendung mit verschiedenen Klassen implementiert werden. (*WebPlatformInfo()* bzw. *MauiPlatformInfo()*). Im Listing 31 können wir dann den, in diesem Fall durch den Konstruktor überlieferten, Dienst verwenden, um basierend auf der jeweiligen Implementierung herauszufinden, auf welcher Plattform wir uns befinden. Der Dienst selbst ist dabei sehr einfach gehalten und besitzt nur ein Feld *Platform*, welches für jede Plattform den entsprechenden Identifier zurückgibt. An jeder Stelle, an der wir nun plattformspezifischen Code benötigen, können wir diesen Dienst (siehe Listings 34 bis 36) verwenden, um die Logik entsprechend anzupassen.

```
1 namespace SharedComponents.Configuration
2 {
3     /// <summary>
4     /// Information about the platform the application is running on.
5     /// Used for Dependency Injection.
6     /// </summary>
7     public interface IPlatformInfo
8     {
9         Platform Platform { get; }
10    }
11
12    /// <summary>
13    /// List of supported platforms.
14    /// </summary>
15    public enum Platform
16    {
17        None,
18        Maui,
19        Web
20    }
21 }
```

Listing 34: IPlatformInfo Interface

```
1 namespace SharedComponents.Configuration
2 {
3     /// <summary>
4     /// Information about Maui platform.
5     /// </summary>
6     public class MauiPlatformInfo : IPlatformInfo
7     {
8         public Platform Platform => Platform.Maui;
9     }
10 }
```

Listing 35: MauiPlatformInfo Implementation

```
1 namespace SharedComponents.Configuration
2 {
3     /// <summary>
4     /// Information about Web platform.
5     /// </summary>
6     public class WebPlatformInfo : IPlatformInfo
7     {
8         public Platform Platform => Platform.Web;
9     }
10 }
```

Listing 36: WebPlatformInfo Implementation

8 Gamification mit H5P-Content

Bevor wir noch einmal genauer auf die technische Umsetzung der EduBai Lernapps eingehen, wollen wir uns kurz mit dem Hintergrund beschäftigen, warum Spiele/Gamification von Lerninhalten gewinnbringend für den Schulunterricht und das Lernen sein kann. Zwei großartige Ressourcen zu dem Thema sind das Buch *Let's play! Mehr Erfolg mit Seminaren und Workshops* [2] mit Fokus auf Seminare und Spiele in der realen Welt, und drei technische Anwendungen von Gamifikation in *HCI in Games* [4] mit den Kapiteln „*Exploring Virtual Reality (VR) to Foster Attention in Math Practice - Comparing a VR to a Non-VR Game*“, „*Students' Learning Outcomes Influenced by Textbook Selection: A Gamification Method Using Eye-Tracking Technology*“ und „*Skull Hunt: An Educational Game for Teaching Biology*“.

Nicht nur für Kinder sind Spiele sehr interessant, sie sind auch im Erwachsenenalter ein wichtiger Teil vieler Kulturen und Gesellschaften. Kinder lernen durch Spiele ihre Umwelt zu erkunden, nach bestimmten Regeln zu handeln und sozial zusammenzuarbeiten. Diese Vorgehensweisen spiegeln aber auch das allgemeine Zusammenleben in gewisser Weise wider. Für Erwachsene bieten Spiele zudem die Möglichkeit, neue Konzepte ohne Ängste oder Druck auszuprobieren, weil sie sich in Rollen und Szenarien bewegen, die nicht unbedingt direkten Zusammenhang mit ihrem Leben haben und deswegen oft auch keine Auswirkungen. Fang [4] hat sich hier im Kontext von erlebnispädagogischem Training von Jugendlichen aber auch Erwachsenen damit beschäftigt, Übungen, Energizer und spielerische Methoden zu sammeln, um die genannten Aspekte bestmöglich für den Transfer von Lerninhalten zu nutzen. Wichtig dabei ist auch die Haltung der Spielleitung. Nur wenn sie selbst von der Methode/dem Spiel überzeugt ist und den Bezug zum Lernziel herstellen kann, können die Teilnehmenden den Nutzen erkennen und sich auf das Spielerische einlassen.

8.1 Gamification Konzept

Wir wollen jetzt eine Möglichkeit finden, ein Gamification Konzept digital zu gestalten, welches wir dann über EduBai zur Verfügung stellen können. „Gamification“ bezeichnet den Einsatz von spieltypischen Elementen in eigentlich spielfremden Kontexten. Ziel ist es die Motivation, das Engagement und den Lernerfolg der Nutzen:innen, in diesem Fall Schüler:innen, zu steigern. Eine bekannte, frei zugängliche Technologie, die es ermöglicht, solche interaktiven spielerischen Lerninhalte zu erstellen, ist *H5P* [6]. Es handelt sich hier um einen Open-Source-Standard, der es ermöglicht, vorgefertigte Templates für z.B. Quizzes, Memory-Spiele (wie in Abbildung 34) oder Drag-and-Drop-Aufgaben mit eigenen Lerninhalten zu füllen. Mit den Lerninhalten gehen wir auf alle genannten Aspekte gut ein.

Lernende haben die Möglichkeit in einem geschützten Rahmen zu experimentieren, erhalten automatisiertes, nicht wertendes Feedback und entwickeln im besten Fall Motivation dafür ihr bestes zu geben. Viele an Schulen gängige Lernplattformen wie Blackboard oder Moodle unterstützen H5P-Inhalte direkt, sowohl die Erstellung als auch die Einbettung in bestehende Kurse. So können Lehrpersonen die interaktiven Inhalte direkt in ihre vorhandenen Unterrichtsmaterialien integrieren. Die Idee von EduBai ist es, diese H5P-Inhalte aber selbstverwaltet und unabhängig von bestehenden Lernplattformen für die Lernenden zur Verfügung zu stellen. Wie diese Lerninhalte technisch umgesetzt werden, wollen wir im nächsten Abschnitt betrachten.

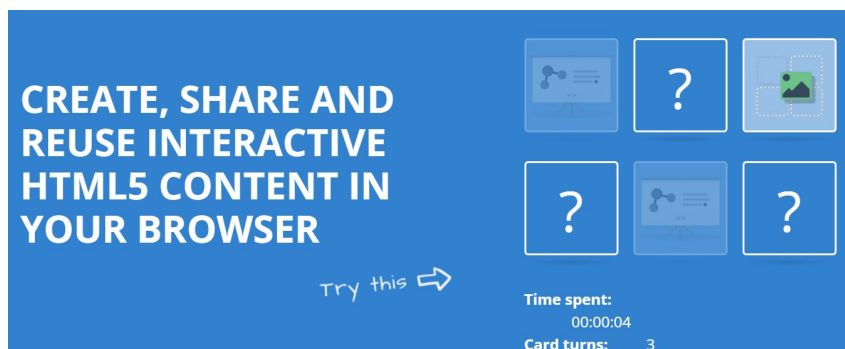


Abbildung 34: H5P Website Beispiel

8.2 H5P-Content

Wie der Name unschwer erkennen lässt, handelt es sich bei H5P um ein Projekt das auf HTML5 basiert, also rein durch Webtechnologien wie HTML, CSS und JavaScript umgesetzt ist. Das macht es besonders einfach in bestehende Webanwendungen zu integrieren, da es keine speziellen Server- oder Client-Anforderungen gibt. H5P-Inhalte sind in der Regel in Form von „H5P-Paketen“ gebündelt. Darin befindet sich der Quellcode für den jeweiligen Lerntypen, aber auch alle benötigten Assets wie Bilder oder Videos, die für die Erstellung verwendet wurden. Auch wenn die Dateiendung *.h5p* auf eine besondere Codierung, oder Dateistruktur schließen lässt, handelt es sich bei H5P-Paketen eigentlich um eine ZIP-Datei, die alle benötigten Dateien enthält. Will man den Source-Code außerhalb eines speziellen Editors, aus welchem Grund auch immer, bearbeiten, hat man also hier direkten Zugang zu den Quelldateien (siehe Abbildung 35).

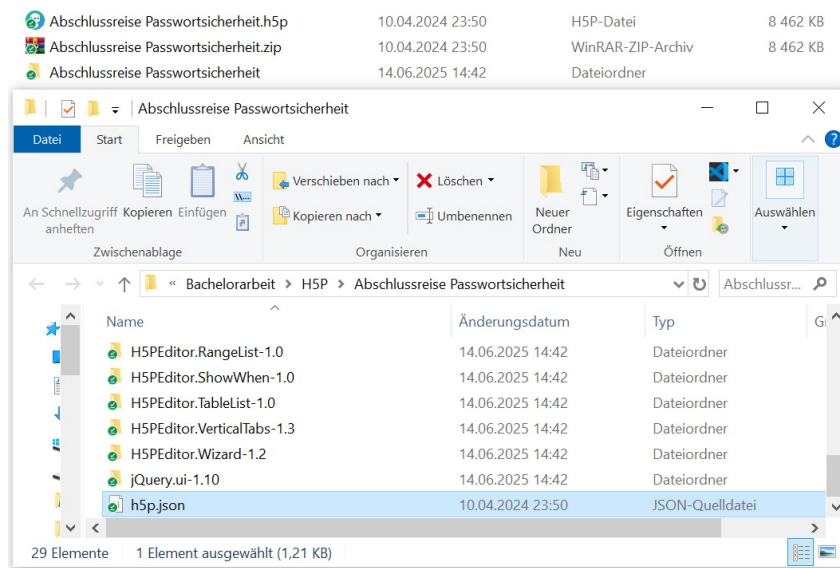


Abbildung 35: H5P Dateiformat

Neben dem Quellcode befinden sich in diesem Ordner vor allem zwei wichtige Teile, die den erstellten Lerninhalt beschreiben. Zum einen ist das die *h5p.json* Datei, welche einige Metadaten und vor allem für die Anwendung wichtig, den gewählten Typ des Lerninhalts enthält. Als eigens entwickelte Maßnahme verwenden wir für EduBai diese Datei auch, um alle anderen Metadaten, wie etwa den/die Autor:in, eine Beschreibung und die Veröffentlichungslizenz gemeinsam mit eigenen Informationen, wie etwa die passende Schulstufe oder das Unterrichtsfach zu vermerken. Auf diese Informationen können wir dann in weiterer Folge in der Anwendung zugreifen, um die Lerninhalte entsprechend zu filtern und anzuzeigen. Zum anderen ist das der *content* Ordner, in welchem alle benötigten Dateien wie Bilder, Videos oder Audiodateien liegen, die für die Erstellung des Lerninhalts verwendet wurden. Am wichtigsten ist darin der gesamte Text und die getätigte Konfiguration der Lerninhalte (siehe Listing 37).

```

1  {
2  ...
3  "gamemap": {
4  "elements": [
5  {
6  "id": ...,
7  "label": "Arten von Passwörtern",
8  ...
9  "contentType": {
10 "params": {
11 "media": {...},
12 "answers": [
13   {
14     "correct": false,
15     "tipsAndFeedback": {
16       "chosenFeedback": "<div>Bei Gesichtserkennung und Fingerabdruckerkennung ...",
17       "tip": "",
18       "notChosenFeedback": ""
19     },
20     "text": "<div>Login mit Gesichtserkennung und Fingerabdruck birgt keine Risiken, ..."
21   },
22   {
23     "correct": true,
24     "tipsAndFeedback": {
25       "notChosenFeedback": "<div>OT-Links und OT-Passwörter verlieren nach ...",
26       "tip": "",
27       "chosenFeedback": ""
28     },
29     "text": "<div>Bei OT-Links bzw. OT-Passwörtern ist es kein Problem, ..."
30   },
31   {
32     "correct": true,
33     "tipsAndFeedback": {
34       "notChosenFeedback": "<div>57% gaben an keinen Passwort-Manager zu ...",
35       "tip": "",
36       "chosenFeedback": ""
37     },
38     "text": "<div>Mehr als die Hälfte aller Befragten einer Studie benutzen ..."
39   },
40   {
41     "correct": true,
42     "tipsAndFeedback": {
43       "notChosenFeedback": "<div>Es wird auch noch lange Zeit in die Zukunft ...",
44       "tip": "",
45       "chosenFeedback": ""
46     },
47     "text": "<div>Auch wenn der Login mit Email / Passwort unsicherer ist als ..."
48   }
49 ]...
50 }...
51 }...
52 }
53 ]
54 }
55 }

```

Durch den Open-Source Ansatz des H5P Projektes gibt es einige Tools, die es einem erleichtern, Inhalte zu erstellen und bereitzustellen. Hat man Zugriff auf ein H5P-kompatibles System, ist es am einfachsten, die Inhalte direkt mit den dort mitgelieferten Editoren zu erstellen und direkt an passender Stelle einbinden zu lassen. Das setzt natürlich an einer Schule voraus, dass jemand dieses System installiert, konfiguriert und betreut. Da wir aber eine selbstverwaltete Lösung für EduBai entwickeln wollen, müssen wir eine Möglichkeit finden, die H5P-Inhalte systemunabhängig in die Anwendung einzubinden. Es gibt einige Anbieter, die einen Editor und auch das Hosting der H5P-Inhalte anbieten, welcher wir dann direkt mittels eines *iframe*-Tags in die Anwendung einbinden können. Diese Anbieter verlangen aber für diesen Service in der Regel eine entsprechende Vergütung. Da es unser Ziel ist so unabhängig wie möglich zu bleiben, werden wir uns aber weiterhin zwei Community-Projekte bedienen. Wir werden hier unsere zwei Probleme mit folgenden Lösungen angehen:

1. **H5P-Inhalte erstellen:** Hier verwenden wir den Lumi H5P Editor von der Lumi Education UG [24].
2. **H5P-Inhalte bereitstellen:** Hier verwenden wir eine Bibliothek des Github-Nutzers tunapanda namens „h5p-standalone“ [23].

8.3 Lumi H5P Editor

Wir wollen uns hier den Lumi Editor mit einigen Beispielbildern (Abbildungen 36 bis 38) genauer ansehen. Früher begrüßte uns der Editor mit einem Willkommensbildschirm, in dem er die Auswahl zwischen dem eigentlichen Editor, einem Analysetool und einer Verbindung zu einer Online-Hosting Möglichkeit von Lumi anbot. In der neusten Version wurde der Editor für die Anwendung vereinfacht, sodass wir nun direkt mit dem Erstellen von Inhalten beginnen können.

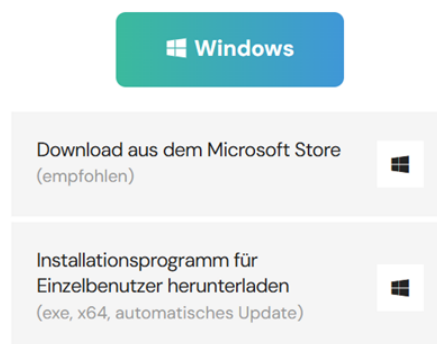


Abbildung 36: Lumi Download

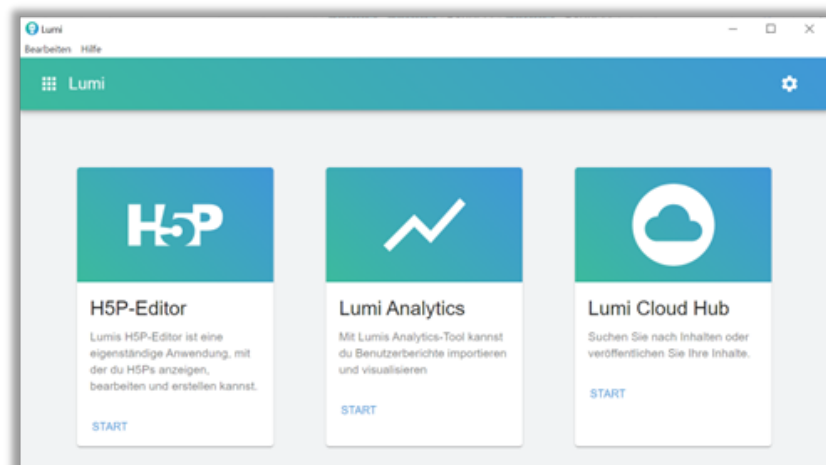


Abbildung 37: Alter Lumi Startbildschirm

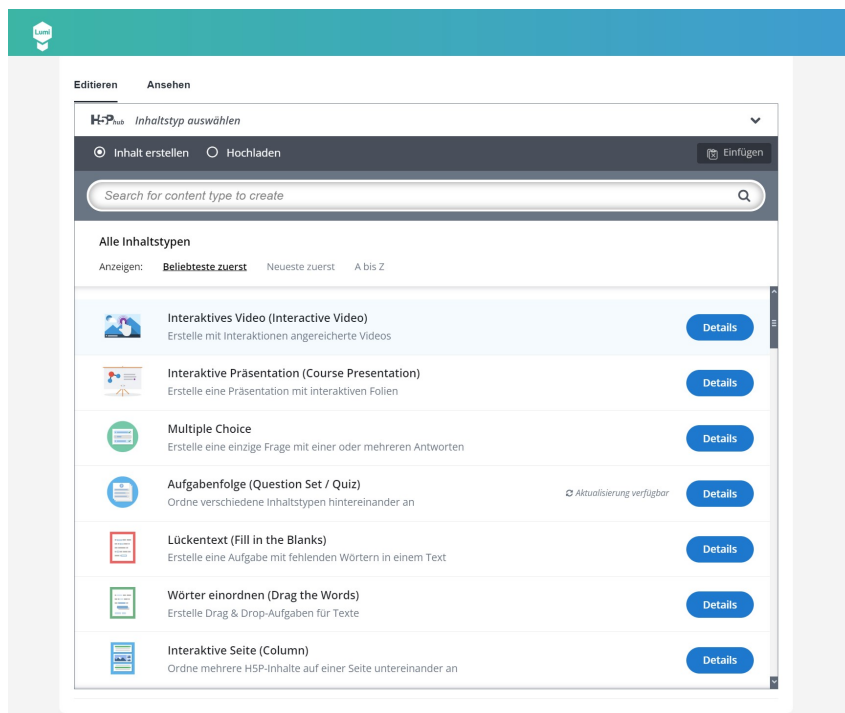


Abbildung 38: Neuer Lumi Startbildschirm

Hier muss man dann zunächst den gewünschten Lerntypen auswählen (siehe Abbildung 39). Viele der gängigsten Methoden kann man bereits mit der Installation des Editors zusammen installieren lassen. Will man einen der neueren oder nicht vorinstallierten

Lerntypen verwenden, muss man diese mit einem Knopfdruck einmalig herunterladen. Je nach Stand des Editors im Vergleich zur H5P Version, kann es sein, dass manche Lerntypen noch nicht verfügbar sind. Es wird noch immer aktiv am Editor gearbeitet und regelmäßig auf die Unterstützung aller neuen Content-Typs geachtet. Wir erstellen jetzt zum Testen einen einfachen Lückentext.

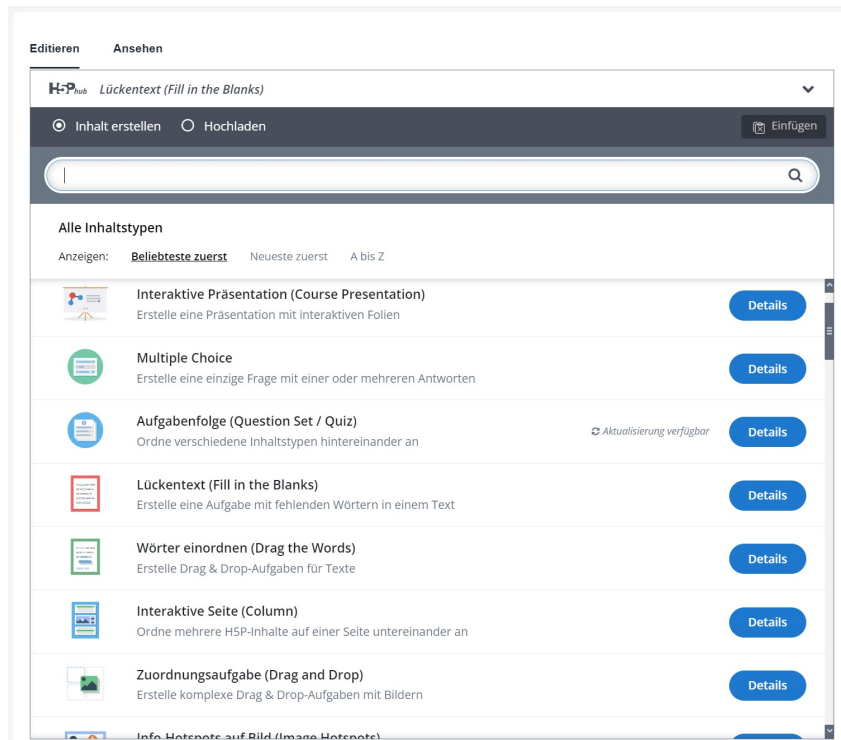


Abbildung 39: Lumi Lerntypen Auswahl

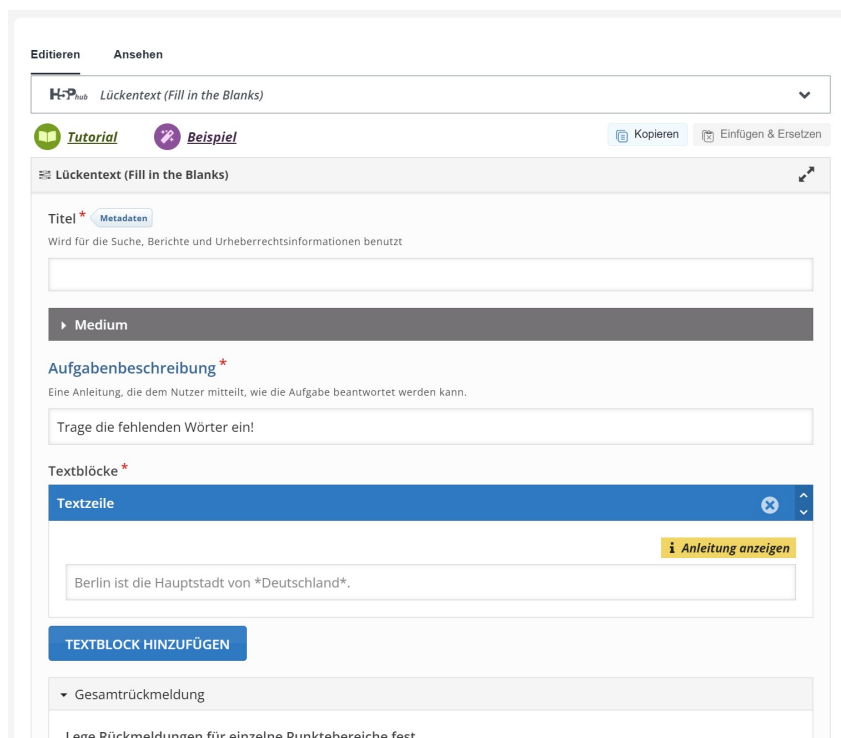


Abbildung 40: Lumi Editor

Der Editor (siehe Abbildung 40) ist sehr intuitiv aufgebaut, in der neusten Version sind ganz zu Beginn auch immer zwei Schaltflächen eingebettet, die auf die offizielle Dokumentation und Beispiele zum jeweiligen Lerntypen verweisen. So kann man sich vor dem Erstellen schnell einen Überblick verschaffen. Alle benötigten Schaltflächen sind direkt im Editor erklärt, für besondere Syntax, wie hier beim Lückentext, stehen auch immer kleine Hilfetexte (siehe Abbildung 41) zur Verfügung.

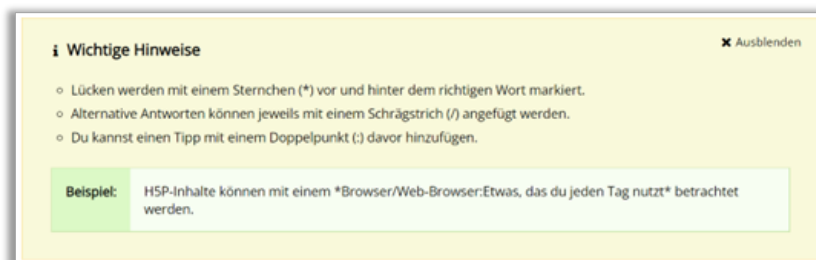


Abbildung 41: Lumi Hinweise

Der Editor erlaubt es auch, die erstellten Inhalte direkt zu testen, indem man auf die

Schaltfläche „Ansehen“ klickt (siehe Abbildung 42). Sollte man bei der Syntax irgendwelche Fehler gemacht haben, würde man das auch spätestens jetzt merken. Ist man zufrieden mit dem Ergebnis, kann man die erstellten Inhalte entweder eingebettet in eine HTML-Seite exportieren (siehe Abbildung 43), um sie lokal zu verwenden, direkt zu veröffentlichen, oder man speichert die gebündelte H5P-Datei, um die Lernapp auf einer Lernplattform hochzuladen. In unserem Fall binden wir sie in eine eigene Anwendung ein.

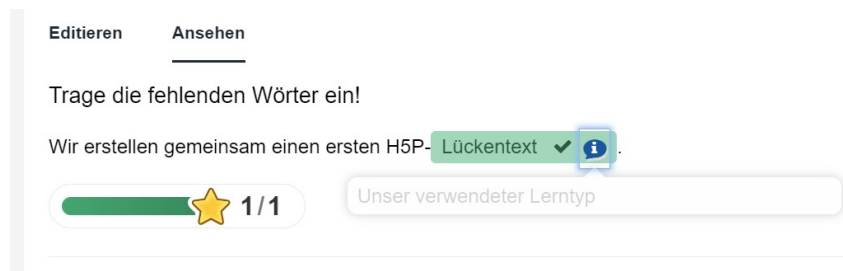


Abbildung 42: Lumi Viewer

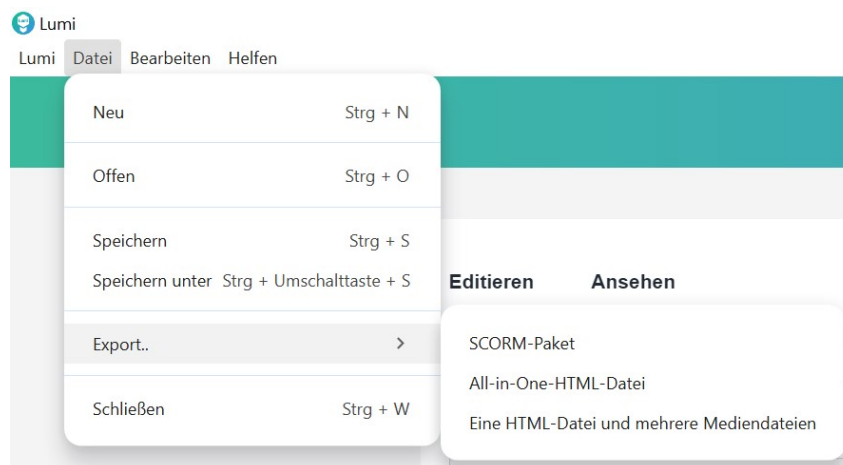


Abbildung 43: Lumi Export

8.4 H5P-Standalone

Um nun endlich unsere erstellten H5P-Inhalte in EduBai einzubinden, benötigen wir eine Möglichkeit, den H5P-Content zu interpretieren und richtig darzustellen. Zum Glück hat sich auch hier schon jemand die Mühe gemacht, eine einfache Lösung zu entwickeln, die es ermöglicht, H5P-Inhalte ohne großen Aufwand in eine Webanwendung einzubinden.

Die Bibliothek „h5p-standalone“ vom Github-Nutzer tunapanda [23] ist eine JavaScript-Bibliothek, die es ermöglicht, H5P-Inhalte direkt in einer Webanwendung darzustellen. Die Bibliothek ist grundsätzlich sehr einfach zu verwenden, sobald man die benötigten Dateien eingebunden hat. Man muss letztendlich nur einen HTML-Container mit der richtigen ID erstellen, und diesen, zusammen mit dem Speicherpfad zum extrahierten H5P-Paket an das Skript übergeben (siehe Listing 38). Die Bibliothek kümmert sich dann um die Darstellung des Inhalts und die Interaktion mit dem Nutzer. Die Bibliothek ist seit Kurzem auch über CDN (Content Delivery Network) verfügbar, was die Einbindung in die Anwendung noch einfacher macht. Wir müssen lediglich den Link zur Bibliothek in unsere HTML-Datei einfügen und können dann die H5P-Inhalte direkt in der Anwendung verwenden.

```

1  <html>
2  <head>
3      <link rel="stylesheet"
4          href="https://cdn.jsdelivr.net/npm/h5p-standalone@latest/dist/styles/h5p.css">
5  </head>
6  <body>
7  <div id="h5p-container"></div>
8
9  <script
10     src="https://cdn.jsdelivr.net/npm/h5p-standalone@latest/dist/main.bundle.js">
11 </script>
12 <script>
13     document.addEventListener('DOMContentLoaded', function () {
14
15         const options = {
16             h5pJsonPath: '/path/to/your/h5p-folder', // Path to H5P content directory
17             frameJs: 'https://cdn.jsdelivr.net/npm/h5p-standalone@latest/dist/frame.bundle.js',
18             frameCss: 'https://cdn.jsdelivr.net/npm/h5p-standalone@latest/dist/styles/h5p.css',
19         };
20
21         const el = document.getElementById('h5p-container');
22
23         // H5PStandalone is globally available when main.bundle.js is included via a script tag
24         new H5PStandalone.H5P(el, options);
25     });
26 </script>
27 </body>
28 </html>

```

Listing 38: H5P-Standalone Beispiel

Es besteht aber auch die Möglichkeit, die Bibliothek lokal zum Projekt dazuzulegen, und auf diese Art zu referenzieren (siehe Listing 39). Dafür müssen wir den aktuellsten Release

der Bibliothek von Github herunterladen und im Projektverzeichnis ablegen. Danach müssen wir bei den übergebenen Optionen den passenden Pfad angeben.

```
1  const options = {  
2    h5pJsonPath: '/path/to/your/h5p-folder', // Path to the extracted H5P content  
3    frameJs: 'assets/h5p-player/frame.bundle.js', // Path to player's frame.bundle.js  
4    frameCss: 'assets/h5p-player/styles/h5p.css', // Path to player's h5p.css  
5  };
```

Listing 39: H5P-Standalone Beispiel mit lokaler Bibliothek

Damit haben wir nun alle Tools kennengelernt, die wir benötigen, um interaktive Lerninhalte für Edubai zu erstellen und wie in Kapitel 6 Client Side für die Lernenden bereitzustellen. Damit haben wir unser großes Ziel einer selbstverwalteten, zeitgemäßen Webanwendung für interaktiven Gamification-Content erreicht und können ab sofort die Lernenden mit spannenden und motivierenden Inhalten versorgen.

9 Fazit

Ich habe mir mit EduBai als Projekt selbst eine Herausforderung gestellt. Mit der Idee, H5P-Inhalte für meine (zukünftigen) Schüler:innen bereitzustellen hatte ich eine klare Vorstellung davon, was ich erreichen will und immer ein Ziel vor Augen. Aber wie so oft in der Softwareentwicklung, sind die Umsetzung einer Idee und die Idee selbst oft weiter voneinander entfernt als gedacht. Ich wollte aber keine Abkürzungen nehmen, kein *gut genug* fabrizieren, sondern mich selbst fordern und alles mitnehmen, was es in der Webentwicklung mitzunehmen gibt. Mit Blazor als Framework bin ich hier am Zahn der Zeit, bin in einem etablierten Ökosystem und doch mit „relativ“ neuen Herausforderungen wie Cross-Platform-Development beschäftigt. Auch habe ich versucht, mein gesamtes Wissen zu Softwarearchitektur, Design-Patterns und Syntax-Konventionen so genau als möglich anzuwenden und gleichzeitig neue Konzepte, wie die Besonderheiten bei geteiltem und plattformabhängigem Code, zu erforschen. In Summe bin ich mit den erbrachten Leistungen mehr als zufrieden, weil ich wie geplant eine erste Version meiner Lernplattform für meinen Unterricht fertigstellen konnte, ein neues Framework besser kennenlernen und mich selbst in der Webentwicklung weiterbilden konnte.

Abbildungsverzeichnis

1	Erstellen der WebAssembly-App	3
2	WebAssembly-App Konfiguration	3
3	Blazor WebAssembly Projektvorlage	4
4	SignalR Architektur [11]	8
5	SignalR Architektur [22]	9
6	HP Elitedesk Mini-PC [8]	24
7	Projekt veröffentlichen	25
8	DuckDNS Website	28
9	pgAdmin Logo	30
10	pgAdmin Menü	30
11	pgAdmin Abfrage	31
12	WebAssembly Flowchart [21]	37
13	WebAssembly PWA Installation	37
14	MVC Pattern [9]	39
15	Dateistruktur	39
16	Pages MVC Teilung	40
17	H5P Modul Struktur	46
18	Content Ausgangsverzeichnis	50
19	Ressourcen Datei	53
20	Login Seite 1/3	57
21	Login Seite 2/2	57
22	Login Seite 3/3	58
23	Registrierung	59
24	Passwort vergessen	59
25	Lernapps 1/4	60
26	Lernapps 2/4	60
27	Lernapps 3/4	61
28	Lernapps 4/4	62
29	Beispiel App 1/2	63
30	Beispiel App 2/2	63
31	RCL Projektstruktur	64
32	Projektmappe mit RCL	65
33	EduBai Windows App	68
34	H5P Website Beispiel	76
35	H5P Dateiformat	77
36	Lumi Download	79
37	Alter Lumi Startbildschirm	80
38	Neuer Lumi Startbildschirm	80

39	Lumi Lerntypen Auswahl	81
40	Lumi Editor	82
41	Lumi Hinweise	82
42	Lumi Viewer	83
43	Lumi Export	83

Code Listings

1	Startup-Logik (Server): program.cs 1/2	6
2	Startup-Logik (Server): program.cs 2/2	7
3	Skeleton des User Credential Controllers	10
4	User Credential Controller	11
5	Read Endpunkt des User Credential Controllers	12
6	AuthstateProvider	15
7	GetAuthenticationStateAsync	17
8	LearningApps Component	19
9	EmailVerifiedRequirement	20
10	Email Verification Service	22
11	Symmetrische Token Verschlüsselung	23
12	edubaihost.service	26
13	edubai.conf	27
14	DbContext	33
15	UserCredentials Entity Framework Klasse	34
16	UserCredentials Upsert Methode	35
17	Sidebar Razor Komponente	41
18	LearningApps Razor Komponente	42
19	Razor Syntax Beispiele	43
20	BlazorWASM Program Startup	44
21	DI in Login Model	44
22	Alternativ: DI direkt in Login Razor Komponente	45
23	H5PContent JavaScript Datei	47
24	H5PContent Model	48
25	Local Storage Key-Value Pair setzen	49
26	Verzeichnis auslesen	52
27	Localization Startup	54
28	Localization in Razor Komponente	55
29	Media Query Beispiel	56
30	API RCL Aufruf	66
31	AuthstateProvider mit Plattform Überprüfung	70
32	BlazorMAUI Service Registrierung	71
33	BlazorWASM Service Registrierung	72
34	IPlatformInfo Interface	73
35	MauiPlatformInfo Implementation	73
36	WebPlatformInfo Implementation	74
37	content.json	78
38	H5P-Standalone Beispiel	84

39	H5P-Standalone Beispiel mit lokaler Bibliothek	85
----	--	----

Tabellenverzeichnis

1	Felder der Tabelle <code>user_credentials</code>	31
---	--	----

Literatur

- [1] Alexander Bartz. *Was ist Blazor? Das Web-Framework für moderne .NET und CSharp Anwendungen*. <https://crispycode.net/de/was-ist-blazor/>. Accessed: 2025-06-07. 2023.
- [2] Ariane Bertz u. a. *Let's play! Mehr Erfolg mit Seminaren und Workshops: 64 Spiele für wirkungsvolle Präsenz- und Online-Trainings*. de. Freiburg, Germany: Haufe-Lexware, Okt. 2021.
- [3] DuckDNS. *DuckDNS - Install*. <https://www.duckdns.org/install.jsp>. Accessed: 2025-06-12. 2025.
- [4] Xiaowen Fang. *HCI in games: 5th International Conference, HCI-games 2023, held as Part of the 25th HCI International Conference, HCII 2023, Copenhagen, Denmark, July 23–28, 2023, proceedings, Part II*. en. Cham, Switzerland: Springer Nature, Juli 2023.
- [5] Electronic Frontier Foundation. *Certbot*. <https://certbot.eff.org/>. Accessed: 2025-06-12. 2025.
- [6] H5P Group. *H5P - Create and Share Rich HTML5 Content and Applications*. <https://h5p.org/>. Accessed: 2025-06-14. 2025.
- [7] WebAssembly Community Group. *Feature Status - WebAssembly*. <https://webassembly.org/features/>. Accessed: 2025-06-15. 2025.
- [8] Inc. HP. *HP EliteDesk 800 G6 Desktop Mini PC*. <https://support.hp.com/at-de/product/details/hp-elitedesk-800-g5-desktop-mini-pc/27066639>. Accessed: 2025-06-12. 2025.
- [9] Yuanchun Liu und Heyi Zhu. „System architecture diagram“. In: *J. Phys. Conf. Ser.* 1746.1 (Jan. 2021), S. 012034.
- [10] Yuanchun Liu und Heyi Zhu. „Technology application base on ASP.NET model view controller“. In: *J. Phys. Conf. Ser.* 1746.1 (Jan. 2021), S. 012034.
- [11] Inc Microsoft. *ASP.NET Core Blazor*. https://learn.microsoft.com/de-de/aspnet/core/blazor/index/_static/blazor-server.png?view=aspnetcore-9.0. Accessed: 2025-06-09. 2024.
- [12] Inc Microsoft. *ASP.NET Core Blazor*. https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-9.0&WT.mc_id=dotnet-35129-website. Accessed: 2025-06-07. 2025.
- [13] Inc Microsoft. *ASP.NET Core Blazor globalization and localization*. <https://learn.microsoft.com/en-us/aspnet/core/blazor/globalization-localization?view=aspnetcore-9.0>. Accessed: 2025-06-16. 2024.

- [14] Inc Microsoft. *ASP.NET Core-Razor-Komponenten*. <https://learn.microsoft.com/de-de/aspnet/core/blazor/components/?view=aspnetcore-9.0>. Accessed: 2025-06-15. 2024.
- [15] Inc Microsoft. *Build your first web app with ASP.NET Core using Blazor*. <https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro>. Accessed: 2025-06-07. 2025.
- [16] Inc Microsoft. *Erstellen einer .NET MAUIBlazor Hybrid-App*. <https://learn.microsoft.com/de-de/aspnet/core/blazor/hybrid/tutorials/maui?view=aspnetcore-9.0>. Accessed: 2025-06-13. 2024.
- [17] Inc Microsoft. *Introduction to Identity on ASP.NET Core*. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-9.0&tabs=visual-studio>. Accessed: 2025-06-11. 2025.
- [18] Inc Microsoft. *Introduction to Razor Pages in ASP.NET Core*. https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-9.0&WT.mc_id=dotnet-35129-website&tabs=visual-studio. Accessed: 2025-06-07. 2025.
- [19] Inc Microsoft. *Verwenden von ASP.NET Core SignalR mit Blazor*. <https://learn.microsoft.com/de-de/aspnet/core/blazor/tutorials/signalr-blazor?view=aspnetcore-9.0&tabs=visual-studio>. Accessed: 2025-06-09. 2025.
- [20] Inc Microsoft. *Was ist .NET MAUI?* <https://learn.microsoft.com/de-de/dotnet/maui/what-is-maui?view=net-maui-9.0>. Accessed: 2025-06-13. 2025.
- [21] Partha Ray. *Webassembly Data Flow Architecture*. https://www.researchgate.net/figure/WebAssembly-data-flow-architecture_fig2_373229823. Accessed: 2025-06-15. 2023.
- [22] SergeyZ. *Blazor: Server and WebAssembly at the same time in one application*. <https://habr.com/ru/articles/546414/>. Accessed: 2025-06-09. 2021.
- [23] tunapanda. *h5p-standalone*. <https://github.com/tunapanda/h5p-standalone>. Accessed: 2025-06-14. 2015.
- [24] Lumi Education UG. *Erstellen sie H5P und hosten sie Ihre Inhalte auf Lumi - Lumi Education*. <https://lumi.education/de/>. Accessed: 2025-06-14. 2025.
- [25] W3Schools. *PostgreSQL: Install Introduction*. https://www.w3schools.com/postgresql/postgresql_install.php. Accessed: 2025-06-12. 2025.