

To-Do-App "To-Dino" in Blazor

Leo - Jonathan - Sebastian

16. Juni 2024

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Glossar	VI
Abkürzungsverzeichnis	VII
1 Einleitung	1
Projektumfeld	1
Projektziel	1
Projektbegründung	2
Projektschnittstellen	2
Personelle Schnittstellen	2
Technische Schnittstellen	2
2 Projektplanung	2
Projektphasen	2
Ressourcenplanung	3
Entwicklungsprozess	3
3 Analysephase	4
Ist-Analyse	4
Wirtschaftlichkeitsanalyse	4
Projektkosten	4
Amortisationsrechnung	5
Qualitätsanforderung	5
4 Entwurfsphase	6
Zielplattform	6
Browser	6

Entscheidung der Programmiersprache	6
Umsetzung der grafischen Oberfläche	6
Architekturdesign	6
Entscheidung der Bibliotheken	6
Strukturelles Design	7
Entwurf der Benutzeroberfläche	7
Maßnahmen zur Qualitätssicherung	7
5 Implementierungsphase	7
Implementierung der Blazorpage	7
Implementierung der To-Do Liste	8
Implementierung der To-Do Items	9
Implementierung der MS SQL Datenbank	9
6 Abnahmephase	10
7 Dokumentation	10
8 Fazit	11
Soll-/Ist-Vergleich	11
Reflexion	11
Ausblick	11
9 Quellenverzeichnis	i
10 Anhang	i
A1 Wasserfall-Modell	i
A2 To-Dino Applikation	ii
A3 ApplicationDbContext.cs	iii
A4 Datenbank Init File	iv
A5 Datenbank Migration Snapshot	iv
A6 LoginModal.cs	iv
A7 ToDoItem.cs	iv
A8 ToDoItemModal.cs	iv
A9 ToDoListComponent.cs	iv

A10 Userclass.cs	iv
----------------------------	----

Abbildungsverzeichnis

1	Darstellung des geplanten Wasserfall-Modells	i
2	To-Dino Applikation	ii
3	ApplicationDbContext.cs	iii
4	Generierte Init-File für die Datenbankerstellung	v
5	Generierter Datenbank-Migration-Snapshot	vi
6	LoginModal.cs	vii
7	ToDoItem.cs	viii
8	ApplicationDbContext.cs	ix
9	ToDoListComponent.cs	x
10	Userclass.cs	xi

Tabellenverzeichnis

1	Grobe Zeitplanung	3
2	Projektkosten	5

Glossar

Begriff	Beschreibung / Ergänzung
To-Dino	To-Dino ist eine für Webbrowser Optimierte To-Do-Listen Applikation zum erstellen und verwalten von To-Do Listen
Radzen	Trademark der Firma Radzen
Git	Git ist ein Tool zur Versionskontrolle in Softwareprojekten
C#	CSharp (abgekürzt C#) ist eine objektorientierte Programmiersprache
Blazor	Blazor ist ein Webframework welches das Entwickeln von Webapplikationen in C# und HTML ermöglicht
Visual Studio	Entwicklungsumgebung von Microsoft
Core-Review	Manuelle Überprüfung von Programmcode, um Fehler zu verhindern und das Einhalten von Standards zu garantieren
Mockup	Vorläufiger Entwurf einer geplanten Benutzeroberfläche
Debugger	Software-Tool, welches Entwicklern hilft Code eines Programmes zu analysieren und Fehler zu finden

Abkürzungsverzeichnis

Abkürzung	Bedeutung
BBZ	Berufsbildungs Zentrum
Core-Review	Manuelle Überprüfung von Programmcode, um Fehler zu verhindern und das Einhalten von Standards zu garantieren
CSS	Cascading Style Sheets
etc	"et cetera" im Sinne von "und so weiter"
GmbH	Gesellschaft mit beschränkter Haftung
GUI	Graphical User Interface
h	Stunde(n)
HTML	Hyper Text Markup Language
MS	Microsoft
PC	Personal Computer

1 Einleitung

Die folgende Projektdokumentation beschreibt das Ersatzleistungsprojekt, welches statt einer schriftlichen Klausur in Lernfeld 8 geleistet wird. Das Projekt wurde durch die Gruppe Leo M., Jonathan G. und Sebastian K. im Zeitraum vom 15.05.2024 bis 19.06.2024 absolviert. Die beteiligten an diesem Projekt absolvieren je eine Ausbildung zum Fachinformatiker in den Fachgebieten Digitale Vernetzung und Anwendungsentwicklung im Ausbildungsbetrieb inray Industriesoftware GmbH.

Projektumfeld

Für die Durchführung des Projektes wird von dem Berufsbildungszentrum Rendsburg, folgend BBZ genannt, während des wöchentlichen Unterrichtes für 90 Minuten, ein Arbeitsplatz zur Verfügung gestellt, welcher mit einem Computer mit geeigneter Hardware für die Softwareentwicklung und einer Internetverbindung ausgestattet war. Alternativ durften auch Geräte des Ausbildungsbetriebes oder Privatgeräte zur Entwicklung genutzt werden.

Der Auftraggeber des Projektes ist das BBZ, vertreten durch den Klassenlehrer Till Gades.

Projektziel

Ziel des Projektes ist es, eine To-Do Applikation zu entwickeln, mit welcher Nutzer sich Anmelden können und Listen mit Aufgaben erstellen, bearbeiten, als erledigt kennzeichnen, teilen oder löschen können. Die Daten sollen dabei in einer SQL-Datenbank gesichert werden.

Die Applikation soll auch das Teilen von Listen oder Aufgaben zwischen Nutzern unterstützen, ähnlich eines Ticketsystems.

Projektbegründung

Viele To-Do Applikationen versprechen meist wenig Umfang oder sind mit Kosten verbunden. Eine solche To-Do-App zu entwickeln ist nicht nur ein beliebtes Anfängerprojekt in der Objektorientierten Programmierung, sondern lässt sich auch gut skalieren, wie zum Beispiel durch das Managen von mehreren Nutzern oder das Teilen von Listen mit anderen, wodurch die Applikation auch für die Planung von Gruppenaktivitäten mit mehreren Personen gut geeignet ist.

Projektschnittstellen

Personelle Schnittstellen

Als Gruppe werden die fachlichen und technischen Anforderungen definiert sowie Mockups der Benutzeroberfläche besprochen. Bei Problemen oder Fragen während des Projektes steht den Autoren auch der Klassenlehrer der IT22B des BBZ zur Verfügung.

Technische Schnittstellen

Das Projekt ist eine alleinstehende Applikation, welche von der Gruppe in Zusammenarbeit entwickelt wurde. Genauere Erläuterungen zur Funktionsweise und technischen Details werden in folgenden Kapiteln behandelt.

2 Projektplanung

Projektphasen

Durchgeführt wurde die Projektarbeit in dem Zeitraum vom 15.05.2024 bis zum 19.06.2024 während der regulären Schulzeiten, sowie freiwillig in der Freizeit der Projektteilnehmer. Für das Projekt waren insgesamt 9 Stunden (6 Unterrichtsstunden zu je 90 Minuten) Bearbeitungszeit geplant. Die grobe Zeitplanung ist in der Untenstehenden Tabelle 1 zu sehen und die genaue Zeitplanung mit Differenz befindet sich im Anhang A1. 2

Projektphase	Geplante Zeit
Gruppen- und Projektfindung	1.5 Stunden
Ausarbeitung des Grobkonzeptes	1.5 Stunden
Ausarbeitung der Umsetzungsphase	1.5 Stunden
Durchführen der Umsetzungsphase	3.0 Stunden
Abschließen der Dokumentation und Vorbereitung der Präsentation	1.5 Stunden
Gesamt	9.0 Stunden

Tabelle 1: Grobe Zeitplanung

Ressourcenplanung

Alle der notwendigen Ressourcen waren vor dem Projektbeginn bereits vorhanden, wie beispielsweise Notebooks, Internetzugänge, Entwicklungsumgebung (Visual Studio 2022), Git etc.

Der Personalbedarf beschränkt sich auf die Autoren.

Zur optimalen Zeit-/Kosten Nutzung werden neben Microsofts Front-End Web Framework Blazor, sowie die öffentlich zugängliche Bibliothek Radzen.Blazor genutzt. Für das Backend wird eine lokale MS SQL Datenbank aufgesetzt.

Entwicklungsprozess

Für den Entwicklungsprozess des Projektes wurde sich für ein Wasserfall-Modell entschieden, da dies das einfachste und effektivste Planungsmodell für ein Projekt in diesem Umfang ist. Eine agile Projektmanagementmethode wie Scrum wurde bewusst nicht gewählt, da das Projekt klare und definierte Anforderungen hat.

Scrum bietet Flexibilität bei sich ändernden Anforderungen im Projektverlauf, da iterativ in Sprints gearbeitet wird und somit Änderungen möglich sind.

Das Wasserfall-Modell bietet eine klare Struktur mit definierten Phasen, was die Planung und Durchführung mit klaren Zielen erleichtert. Diese klare Struktur unterstützt die Entwicklung der Applikation, da die Anforderungen für das Projekt fest gegeben sind.

In Abbildung 1 im Anhang A2 sieht man die Planung in einem Wasserfall-Modell.

3 Analysephase

Ist-Analyse

Die To-Dino App soll ein digitales Ticket- und To-Do-Management-System darstellen, das es Benutzern ermöglicht, Aufgaben zu erstellen, zu verwalten und zu verfolgen. Der aktuelle Zustand beinhaltet keine automatisierte oder digitalisierte Lösung, sondern basiert möglicherweise auf manuellen oder weniger effizienten Methoden.

Wirtschaftlichkeitsanalyse

Projektkosten

Die Kosten für die Durchführung des Projekts setzen sich sowohl aus Personal-, als auch aus Ressourcenkosten zusammen. Laut Ausbildungsvertrag verdient ein Auszubildender im zweiten Lehrjahr bei inray monatlich 950 € (brutto).

$$\begin{aligned}8 \frac{\text{h}}{\text{Tag}} \times 220 \frac{\text{Tage}}{\text{Jahr}} &= 1,760 \frac{\text{h}}{\text{Jahr}} \\950 \frac{\text{€}}{\text{Monat}} \times 12 \frac{\text{Monate}}{\text{Jahr}} &= 11,400 \frac{\text{€}}{\text{Jahr}} \\ \frac{11,400 \frac{\text{€}}{\text{Jahr}}}{1,760 \frac{\text{h}}{\text{Jahr}}} &\approx 6,48 \frac{\text{€}}{\text{h}}\end{aligned}$$

Daraus ergibt sich ein Stundensatz von 6,48 EUR je Gruppenmitglied. Die Dauer der Projektdurchführung beträgt 9 Stunden. Für die Nutzung von Ressourcen wird ein pauschaler Stundensatz von 8 EUR angenommen. Die Kosten für das Projekt sind in Tabelle 2 dargestellt und belaufen sich auf insgesamt 390,96 EUR.

Vorgang	Zeit (h)	Kosten / Stunde	Kosten
Entwicklung (pro Person)	9	6,48 € + 8 € = 14,48 €	130,32 €
Gesamt	27	43,44 €	390,96 €

Tabelle 2: Projektkosten

Amortisationsrechnung

Zur Berechnung der Amortisation des Projektes wurde eine Schnittpunktanalyse zwischen den Gesamtkosten des Projektes und dem vorraussichtlichen Preis von 20 EUR pro verkaufter Lizenz durchgeführt

$$\text{Anzahl verkaufter Lizenzen bis Amortisation} = \frac{\text{Gesamtprojektkosten}}{\text{Verkaufspreis pro Lizenz}}$$

$$\text{Anzahl verkaufter Lizenzen bis Amortisation} = \frac{390,96}{20}$$

$$\text{Anzahl verkaufter Lizenzen bis Amortisation} \approx 19,5$$

Sobald 20 Lizenzen verkauft werden, amortisiert sich das Projekt.

Qualitätsanforderung

Die Qualitätsanforderungen für die Ergebnisse dieses Projektes orientieren sich an dern Qualitätskriterien für Software nach ISO/IEC 9126-1 [2001].

4 Entwurfsphase

Zielformat

Browser

Durch das in diesem Projekt genutzte Framework Blazor, entwickelt von Microsoft, lässt sich die Applikation auf so gut wie jedem System nutzen, da man den Dienst auf einem Server hosten kann und nach Belieben im eigenen Netzwerk oder über das Internet mit jedem Browser erreichen kann.

Entscheidung der Programmiersprache

Das Blazor-Framework benutzt als Programmiersprache C#. Dank bereits vorhandenen Kenntnissen und Fähigkeiten in dieser Sprache, fiel die Wahl der Umsetzung in Blazor schnell und die Gruppe kann von den Erfahrungen aus dem Ausbildungsbetrieb profitieren. Zusätzlich bietet sich so die Möglichkeit, erfahrenere Kollegen bei Problemen befragen zu können.

Umsetzung der grafischen Oberfläche

Bei der To-Do-App handelt es sich um eine lokal installierte Webanwendung, die das Framework Blazor in C# zur Darstellung von GUI-Elementen verwendet. Dabei werden Radzen.Blazor- und Bootstrap-Komponenten genutzt, um ein modernes und einheitliches Erscheinungsbild zu gewährleisten.

Architekturdesign

Entscheidung der Bibliotheken

Aufgrund der bisherigen Erfahrungen des Teams, durch die im Ausbildungsbetriebes genutzten Bibliotheken, wurde sich für die Komponentenbibliothek **Blazor.Radzen** der Firma Radzen entschieden. Diese bietet unter der MIT-Lizenz diverse Komponenten zur Nutzung in Blazor-Projekten an. Diese sind für die Umsetzung des Projektes zwar nicht nötig, da Blazor von Haus aus bereits viele Komponenten anbietet und man auch eigene Komponenten, maßgeschneidert an die eigenen Bedürfnisse und die des Teams, erstellen kann,

aber durch den Zeitlich begrenzten Faktor und durch eine ausführliche Dokumentation zu den Radzen-Komponenten, fiel die Wahl auf die Blazor.Radzen Komponentenbibliothek.

Strukturelles Design

Um ein strukturelles Design zu entwerfen und die Planung fortzuführen, hat sich Sebastian K. zunächst mit der Dokumentation und der Funktionsweise der Bibliotheken vertraut gemacht und überprüft, ob die gewollten Funktionen unterstützt werden.

Entwurf der Benutzeroberfläche

Zur Gestaltung der Benutzeroberfläche wurde zunächst ein Mockup erstellt (siehe Anhang 11, Abbildung 10).

Maßnahmen zur Qualitätssicherung

Um die Funktionsweise und somit die Qualität des Projektes sicherzustellen, wurde am Ende des Zeitraumes zur Bearbeitung bereits früh Zeit eingeplant, sodass die geforderten Funktionen ausgiebig getestet werden können.

5 Implementierungsphase

Implementierung der Blazorpage

Es wurde mit der Implementierung der Blazor-Page begonnen, welche bei der Planung entworfen wurde. Dazu gehört das erstellen einer .razor-Datei im Projekt (siehe Anhang A3 in Abbildung 2), wobei eine Klasse mit dem Namen der Datei entsteht. In diesen Dateien können wir nicht nur den C# Code abbilden sondern auch die für Blazor typischen Komponenten, mit denen das Frontend bearbeitet wird. Für unsere Applikation haben wir uns dazu entschieden, eine Komponente für die verschiedenen zu implementierenden Listenabschnitte erstellen (siehe Anhang A4, Abbildung 3). Für die Listeneinträge haben wir eine eigene Komponente angelegt, welche wir dann mit einem Button, in der Liste mit dem Titel "Open", über den Aufruf des asynchronen Tasks `NewToDoItem` zuerst konfigurieren können und abschließend der To-Do Liste hinzufügen können.

Implementierung der To-Do Liste

Für die einzelnen Listenabschnitte, in welchen die `ToDoItems` abgebildet werden, haben wir uns der Komponentensbibliothek `Blazor.Radzen` (ff. `Radzen`) bedient. Mit der Komponente `RadzenDropZone` können wir Bereiche definieren, in denen wir mit Drag-and-Drop später die Listeneinträge ablegen können (siehe Anhang A5, Abbildung 4). Als Parameter wird ein `Value` mit dem Datentyp `enum` erwartet, über den wir dem eigentlichen Item automatisch mitteilen, in welchem Zustand es sich befindet. Über die allgemeine Komponente `ChildContent` definieren wir zusätzliche Inhalte, welche wir in der Live-Ansicht rendern möchten. Für die Dropzonen sind dies die Titel. Diese fügen wir mit einer weiteren `Radzen`-Komponente, `RadzenText` ein. Für die erste Spalte benutzen wir zusätzlich eine `Footer-Komponente`, in welchem sich der `RadzenButton` befindet, um ein neues Item in die Liste einzufügen. Nachdem wir die anderen Dropzonen definiert haben, werden diese von einem `RadzenDropZoneContainer` umschlossen. Als Pflicht-Parameter geben wir zuerst ein Typen des Objektes an, mit welchem später gearbeitet werden soll. In unserem Fall ist es unsere eigene Klasse `ToDoItem`. Für den Parameter `Data` haben wir im Code-Block dieser Seite eine generische Liste mit dem Typenparameter `T` (`IList<T>`) verwendet und dieser den Namen 'data' und als Typ unsere Klasse `ToDoItem` angegeben (siehe Anhang A6, Abbildung 5). Als nächstes brauchen wir einen `ItemSelector`. Für diesen haben wir im Code-Block einen Delegaten mit dem Namen `ItemSelector` erstellt. Dieser stellt eine vordefinierte Methode dar, welche zwei Parameter (`T1` und `T2`) erwartet und einen Typ 'TResult' zurück gibt. In unserem Fall sind das `T1 = ToDoItem`, `T2 = RadzenDropZone<ToDoItem>` und `TResult = bool`. Danach führen wir eine Lambda-Funktion aus, mit der Parameterliste `(item, zone)`. In der Lambda-Funktion überprüfen wir schließlich ob der `item.Status` gleich mit dem Status von `zone.Value` ist und gleichzeitig ob `item.Status` ungleich dem Status `Status.Deleted` ist. Sind beide Bedingungen erfüllt, gibt diese Funktion 'true' zurück, andernfalls 'false'.

Als nächstes übergeben wir den Parameter `ItemRender` unsere Methode `OnItemRender` welche als Argument den Typ `RadzenDropZoneItemRenderEventArgs<ToDoItem>` erwartet. Im Methodenkörper implementieren wir, dass wir Items, welche den Status `Status.Deleted` haben, nicht länger gerendert werden, indem wir den Wert `args.Value` das Ergebnis des Vergleiches zwischen `args.Item.Status` ist ungleich `Status.Deleted` ge-

ben. Für den Parameter `CanDrop` haben wir wie schon für den `ItemSelector` einen gleichnamigen Delegate mit einer Lambda-Funktion erstellt. In diesem legen wir fest, dass die Items von ihrer Dropzone nur in die nächste oder vorherige bzw. in die "Delete"-Zone abgelegt werden können. `Drop` ist der letzte wichtige Parameter für die `RadzenDropZoneContainer`-Komponente. Hier übergeben wir den asynchronen Task `OnDrop`. In diesem Task überprüfen wir die aktuelle Zone des Items und die Zone in welcher versucht wird, das Item wieder abzulegen.

Implementierung der To-Do Items

Für die Klasse `ToDoItem` müssen wir, durch die Besonderheit der Komponenten der Radzen Bibliothek, benötigen wir keinen ausführbaren Code (siehe Anhang A7, Abbildung 6). Wir definieren eigene Parameter, welche wir benutzen um die Daten abzulegen und von wo aus diese wieder aufgerufen werden können. Der erste und wichtigste Parameter ist der `enum Status` welchen wir gleich benennen. Dieser legt fest, in welchem Zustand sich das Item später befinden wird und ob es noch gerendert werden muss. Der nächste Parameter ist `Title` von Typ `string?`. Das '?' am Ende des Typs gibt uns an, dass dieser Parameter auch `null` sein darf. `Id` vom Typ `int` ist eine Vorbereitung um das Item später in eine Datenbank schreiben zu können. `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]` wird im nächsten Abschnitt erklärt.

Desweiteren haben wir uns für die Parameter `CreationDate`, `DueDate`, `Category`, `Owner` sowie `Description` entschlossen. `CreationDate` (Datum der Erstellung) und `DueDate` (Datum zu wann dieses To-Do abgeschlossen werden soll) sind von dem Typ `DateTime`, die restlichen vom Typ `string?`. Zum Schluss definieren wir noch einen leeren `Konstruktor` der Klasse, welcher von Blazor gefordert ist, um die Klasse bei Bedarf zu initialisieren.

Implementierung der MS SQL Datenbank

Da der Fokus dieses Projektes auf der Planung und Umsetzung einer Anwendung ist und Datenbanken in einem früheren Projekt bereits ausführlich behandelt wurde, haben wir uns dazu entschlossen die integrierte Funktion von Visual Studio zu nutzen und lassen eine MS SQL Datenbank automatisch generieren. Als erstes erstellten wir die `ApplicationDbContext.cs`-Klasse, welche von der Klasse `DbContext` aus dem Name-

space `Microsoft.EntityFrameworkCore` (siehe Anhang 10, Abbildung 9). In dieser Klasse definieren wir, welche Tabellen für uns erstellt werden sollen. Dazu nutzen wir die Klasse `DbSet<T>`, wobei `T` eine unserer gewünschten Klassen ist, zum Beispiel `ToDoItem`, gefolgt von dem Namen den die Tabelle bekommen soll sowie `Getter` und `Setter`.

Danach öffnen wir die `Pakte-Manager-Konsole` und geben den Befehl `"Add-Migration init"` ein. Dieser Befehl erstellt nun auf Basis der Parameter der Klassen, die wir als `DbSet<T>` definiert haben, zwei neue Klassen an und legt diese im Ordner mit dem Namen `Migration` (siehe Anhang A8, Abbildung 7) ab.

Geben wir in der Konsole nun den Befehl `"Update-Database"` ein und bestätigen die Eingabe, wird im Hintergrund eine MS SQL Datenbank auf dem Hostsystem erstellt und veröffentlicht. Mit dem in Visual Studio integrierten

`SQL Server-Objekt-Explorer` können wir den Aufbau der Datenbank kontrollieren und sehen dass wir 3 Tabellen haben, für unsere Klassen `ToDoItem`, `ToDoList` und `Userclass` erstellt wurden. In der `appsettings.json`-Datei müssen wir nur noch den `DbConnectionString` angeben und in der Datei `Startup.cs` eine neue Methode vom Typ `void`, dem Namen `ConfigureServices` und den Argument `services` mit dem Typ `IServiceCollection` erstellen. Dort führen wir von der `ServiceCollection` die integrierte Methode `AddDbContext<ApplicationDbContext>` mit einem Lambda-Aufruf aus, mit dem der zuvor definierte `ConnectionString` angegeben wird (Anhang A9, Abbildung 8).

6 Abnahmephase

Um die grundlegenden Funktionen der geplanten Applikation sicherzustellen. Wurde nach der Implementierung dieser stets getestet, ob diese auch den Anforderungen entsprechen. Dazu zählen das Erstellen eines neuen `ToDoItems`, das Verschieben zwischen mehreren Zuständen und das Löschen eines erfüllten `To-Do Tasks`.

7 Dokumentation

Für die Dokumentation wurde eine PowerPoint Präsentation, sowie diese Dokumentation angefertigt. Neben der PowerPoint Präsentation, wurde diese Dokumentation nach Vorgaben der IHK Kiel erstellt. Hierzu wurde die Programmiersprache `TeX` benutzt. Dieser

Umstand war nicht gefordert und wurde auf freiwilliger Basis umgesetzt.

8 Fazit

Soll-/Ist-Vergleich

In der abschließenden Projektbetrachtung lässt sich feststellen, dass nicht alle definierten Projektziele erfüllt wurden. Der zeitliche Rahmen wurde eingehalten, lässt man die freiwilligen Dienste (Erstellung der Dokumentation nach IHK Anforderung) außer acht.

Reflexion

Das Projekt ermöglichte es dem Team, alle bisher erlangten Fähigkeiten anzuwenden und zu erweitern, die während der Bearbeitung dieses Projektes vonnöten waren. Es konnte praktische Erfahrung in dem Bereich des Projektmanagements, sowie dem Erstellen von Software, gesammelt werden. Auch wenn die definierten Ziele am Ende nicht erfüllt werden konnten, so konnte die Erfahrung gewonnen werden, wie viel Arbeit sich hinter scheinbar einfachen Features verstecken kann (zum Beispiel User-Management).

Im Laufe des Projektes hat sich gezeigt, dass eine gründliche Planung inklusive bereits geplanter Entwürfe sich positiv auf die Implementierungszeit auswirken.

Ausblick

Das Projekt ist abgeschlossen. Die Anwendung ist, per definition "To-Do Liste" voll funktionsfähig, auch wenn nicht alle gewünschten Features eingebaut werden konnten. Da der Code gut dokumentiert ist, sollten andere Entwickler keine Probleme haben, ihn weiterzuentwickeln oder anzupassen. Zusammenfassend kann festgestellt werden, dass die grundlegende Funktion erfüllt wurde und dass darauf geachtet wurde, Erweiterungen im späteren Verlauf zu ermöglichen.

9 Quellenverzeichnis

Radzen Blazor.Radzen Komponenten Bibliothek, 2024. [Online]
Available at: <https://blazor.radzen.com/docs/guides/index.html>
(Zugriff am 11.06.2024).

10 Anhang

A1 Wasserfall-Modell

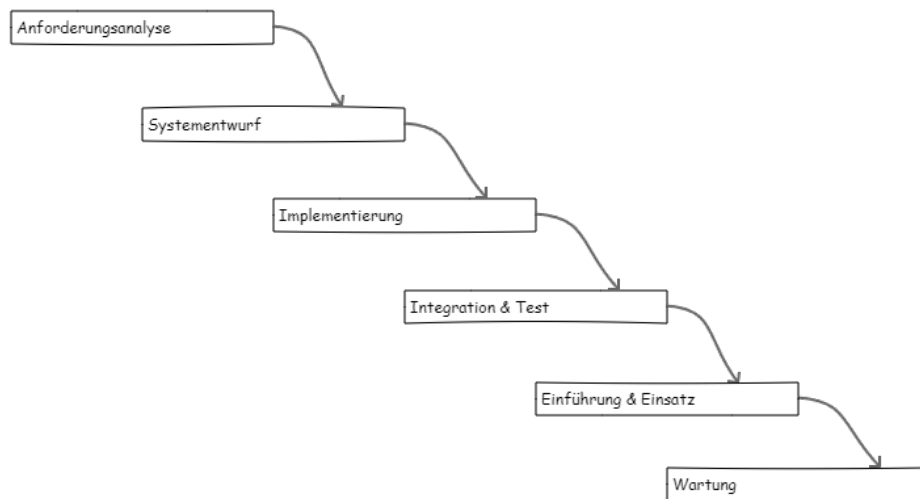


Abbildung 1: Darstellung des geplanten Wasserfall-Modells

Wasserfall-Modell

A2 To-Dino Applikation

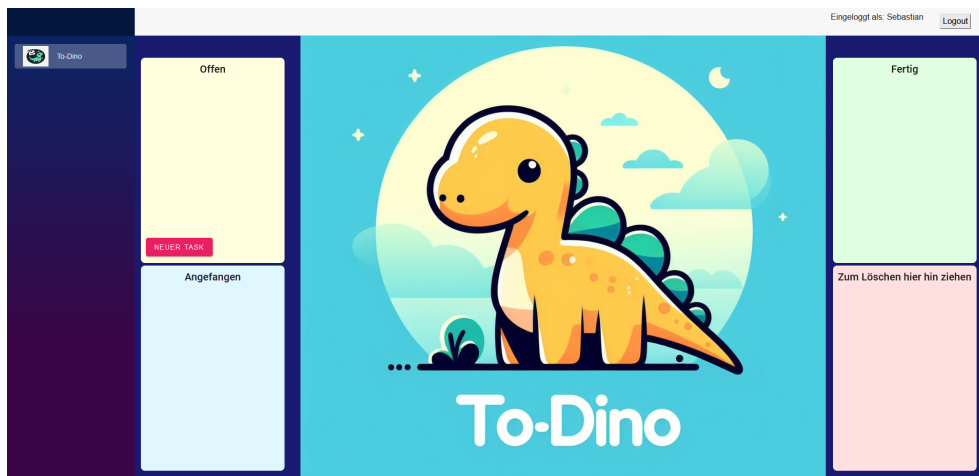


Abbildung 2: To-Dino Applikation

To-Dino Applikation

A3 ApplicationDbContext.cs

```
1  using Microsoft.EntityFrameworkCore;
2  using ToDoApp.Shared;
3
4  namespace ToDoApp.Klassen
5  {
6      10 Verweise
7      public class ApplicationDbContext : DbContext
8      {
9          #region Member
10         protected readonly IConfiguration Configuration;
11         #endregion
12
13         #region Konstruktor
14         0 Verweise
15         public ApplicationDbContext(IConfiguration configuration)
16         {
17             Configuration = configuration;
18         }
19         #endregion
20
21         #region DbSets
22         // Definieren die Tabellen, welche auf Basis der angegebenen Klassen erstellt werden sollen
23         3 Verweise
24         public DbSet<Userclass> Userclasses { get; set; }
25         6 Verweise
26         public DbSet<ToDoItem> ToDoItems { get; set; }
27         #endregion
28
29         #region OnConfiguring
30         /// <summary>
31         /// Holt sich den in den appsettings.json definierten DbConnectionString
32         /// </summary>
33         /// <param name="optionsBuilder"></param>
34         0 Verweise
35         protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
36         {
37             optionsBuilder.UseSqlServer(Configuration.GetConnectionString("DbConnectionString"));
38         }
39         #endregion
40     }
41 }
```

Abbildung 3: ApplicationDbContext.cs

ApplicationDbContext

A4 Datenbank Init File

Datenbank Init-File

A5 Datenbank Migration Snapshot

Datenbank Migration Snapshot

A6 LoginModal.cs

Login Modal Dialog

A7 ToDoItem.cs

ToDoItem-Klasse

A8 ToDoItemModal.cs

ApplicationDbContext

A9 ToDoListComponent.cs

ToDoListComponent-Komponente

A10 Userclass.cs

Userclass-Klasse

```
1 using System;
2 using Microsoft.EntityFrameworkCore.Migrations;
3
4 #nullable disable
5
6 namespace ToDoApp.Migrations
7 {
8     /// <inheritdoc />
9     1 Verweis
10     public partial class init : Migration
11     {
12         /// <inheritdoc />
13         0 Verweise
14         protected override void Up(MigrationBuilder migrationBuilder)
15         {
16             migrationBuilder.CreateTable(
17                 name: "ToDoItems",
18                 columns: table => new
19                 {
20                     Id = table.Column<int>(type: "int", nullable: false)
21                         .Annotation("SqlServer:Identity", "1, 1"),
22                     Title = table.Column<string>(type: "nvarchar(max)", nullable: true),
23                     Status = table.Column<int>(type: "int", nullable: false),
24                     CreationDate = table.Column<DateTime>(type: "datetime2", nullable: true),
25                     DueDate = table.Column<DateTime>(type: "datetime2", nullable: true),
26                     Category = table.Column<string>(type: "nvarchar(max)", nullable: true),
27                     Owner = table.Column<string>(type: "nvarchar(max)", nullable: true),
28                     Description = table.Column<string>(type: "nvarchar(max)", nullable: true)
29                 },
30             constraints: table =>
31             {
32                 table.PrimaryKey("PK_ToDoItems", x => x.Id);
33             });
34         }
35     }
36 }
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
```

Abbildung 4: Generierte Init-File für die Datenbankerstellung


```

1  // <auto-generated />
2  using System;
3  using Microsoft.EntityFrameworkCore;
4  using Microsoft.EntityFrameworkCore.Infrastructure;
5  using Microsoft.EntityFrameworkCore.Metadata;
6  using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
7  using ToDoApp.Klassen;
8
9  #nullable disable
10
11 namespace ToDoApp.Migrations
12 {
13     [DbContext(typeof(ApplicationDbContext))]
14     partial class ApplicationDbContextModelSnapshot : ModelSnapshot
15     {
16         protected override void BuildModel(ModelBuilder modelBuilder)
17         {
18             #pragma warning disable 612, 618
19             modelBuilder
20                 .HasAnnotation("ProductVersion", "8.0.6")
21                 .HasAnnotation("Relational:MaxIdentifierLength", 128);
22
23             SqlServerModelBuilderExtensions.UseIdentityColumns(modelBuilder);
24
25             modelBuilder.Entity("ToDoApp.Klassen.Userclass", b =>
26             {
27                 b.Property<int>("Id")
28                     .ValueGeneratedOnAdd()
29                     .HasColumnType("int");
30
31                 SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("Id"));
32
33                 b.Property<string>("Password")
34                     .IsRequired()
35                     .HasMaxLength(50)
36                     .HasColumnType("nvarchar(50)");
37
38                 b.Property<string>("Username")
39                     .IsRequired()
40                     .HasMaxLength(50)
41                     .HasColumnType("nvarchar(50)");
42
43                 b.HasKey("Id");
44
45                 b.ToTable("Userclasses");
46             });
47
48             modelBuilder.Entity("ToDoApp.Shared.ToDoItem", b =>
49             {
50                 b.Property<int>("Id")
51                     .ValueGeneratedOnAdd()
52                     .HasColumnType("int");
53
54                 SqlServerPropertyBuilderExtensions.UseIdentityColumn(b.Property<int>("Id"));
55
56                 b.Property<string>("Category")
57                     .HasColumnType("nvarchar(max)");
58
59                 b.Property<DateTime?>("CreationDate")
60                     .HasColumnType("datetime2");
61
62                 b.Property<string>("Description")
63                     .HasColumnType("nvarchar(max)");
64
65                 b.Property<DateTime?>("DueDate")

```

To-Do-App "To-Dino" in Blazor

```
1 @page "/login"
2 using ToDoApp.Klassen
3 using Microsoft.EntityFrameworkCore
4 [Inject] Radzen.DialogService dialogService
5 [Inject] Radzen.NotificationService notificationService
6 [Inject] Radzen.ContextMenuService contextMenuService
7 [Inject] Radzen.NotificationService radzenNotification
8 [Inject] Radzen.TooltipService tooltipService
9 [Inject] ApplicationDbContext DbContext
10 [Inject] Userclass Userclass
11 [Inject] NavigationManager NavigationManager;
12
13 <RadzenRow JustifyContent="@JustifyContent" Gap="@Gap">
14     <RadzenColumn>
15         <RadzenText> Nutzername</RadzenText>
16     </RadzenColumn>
17     <RadzenColumn>
18         <RadzenTextBox Name="UserName" @bind-Value="@Userclass.UserName" ReadOnly="false" />
19     </RadzenColumn>
20 </RadzenRow>
21 <br />
22 <RadzenRow JustifyContent="@JustifyContent" Gap="@Gap">
23     <RadzenColumn>
24         <RadzenText> Passwort</RadzenText>
25     </RadzenColumn>
26     <RadzenColumn>
27         <RadzenPassword Name="UserPassword" @bind-Value="@Userclass.Password" ReadOnly="false" />
28     </RadzenColumn>
29 </RadzenRow>
30 <br />
31 <RadzenStack>
32     <RadzenRow>
33         <RadzenColumn>
34             <RadzenButton Click="@{(args) => LoginTask()}" Variant="Variant.Flat" Text="Login" Style="width: auto" />
35         </RadzenColumn>
36         <RadzenColumn>
37             <RadzenButton Click="@{(args) => RegisterAsync()}" Variant="Variant.Flat" Text="Registrieren" Style="width: auto" />
38         </RadzenColumn>
39     </RadzenRow>
40     <RadzenText Text="@loginMessage"></RadzenText>
41 </RadzenStack>
42
43 @code {
44     #region Parameter
45     [Parameter]
46     public Userclass User { get; set; }
47     #endregion
48
49     #region Member
50     string loginMessage = "";
51     string gap = "1em";
52     JustifyContent JustifyContent = JustifyContent.Normal;
53     #endregion
54
55     #region Tasks
56     #region RegisterAsync
57     /// <summary>
58     /// Asynchroner Tasks, welcher prüft ob ein Nutzername bereits in der Datenbank vorhanden ist.
59     /// Ist der Nutzername noch Verfügbar, wird der Nutzer mit dem Passwort in der Datenbank angelegt und der Nutzer wird eingeloggt
60     /// Anderfalls wird der Nutzer darauf hingewiesen, dass der Nutzer bereits existiert.
61     /// </summary>
62     /// <returns></returns>
63     private async Task RegisterAsync()
64     {
65         Userclass? existingUser = DbContext.Userclasses.Where(dbData => dbData.UserName == Userclass.UserName).FirstOrDefault();
66
67         if (existingUser == null)
68         {
69             DbContext.Userclasses.Add(Userclass);
70             loginMessage = "User created!";
71             await DbContext.SaveChangesAsync();
72
73             // Instant Login new User
74             await LoginTask();
75         }
76         else
77         {
78             loginMessage = $"Es existiert bereits ein Nutzer mit dem Namen: \"{Userclass.UserName}\"";
79         }
80     }
81     #endregion
82
83     #region LoginTask
84     /// <summary>
85     /// Überprüft ob ein Nutzer in der Datenbank hinterlegt ist und ob das Passwort mit dem in der Datenbank übereinstimmt.
86     /// Wenn ja, wird der Nutzer angemeldet.
87     /// Wenn entweder das Passwort falsch ist oder der Nutzer nicht existiert, wird dem Nutzer die Nachricht "Username oder Passwort sind falsch" angezeigt.
88     /// </summary>
89     /// <returns></returns>
90     private Task LoginTask()
91     {
92         Userclass? existingUser = DbContext.Userclasses.Where(dbData => dbData.UserName == Userclass.UserName).FirstOrDefault();
93
94         if (existingUser != null)
95         {
96             if (existingUser.Password == Userclass.Password)
97             {
98                 Userclass.IsLoggedIn = true;
99                 dialogService.Close(true);
100                 NavigationManager.NavigateTo("/", true);
101             }
102             else
103             {
104                 // Passwort ist falsch
105             }
106         }
107         else
108         {
109             // Kein Nutzer gefunden
110         }
111         loginMessage = "Username oder Passwort sind falsch";
112         return Task.CompletedTask;
113     }
114     #endregion
115     #endregion
116 }
117
118 }
```

```
1  using System.ComponentModel.DataAnnotations.Schema;
2  using static ToDoApp.Pages.ToDoListComponent;
3
4  namespace ToDoApp.Shared
5  {
6      /// <summary>
7      /// Repräsentiert eine Aufgabe in der To-Do Liste. Hat 4 verschiedene Stati (enum Status)
8      /// </summary>
9      public class ToDoItem
10     {
11         #region Properties
12         /// <summary>
13         /// Titel der Aufgabe
14         /// </summary>
15         public string? Title { get; set; }
16
17         /// <summary>
18         /// Zustand in dem sich die Aufgabe befindet
19         /// </summary>
20         public Status Status { get; set; }
21
22         /// <summary>
23         /// Primary Key in der Tabelle ToDoItems in der Datenbank
24         /// </summary>
25         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
26         public int Id { get; set; }
27
28         /// <summary>
29         /// Gibt an, wann die Aufgabe erstellt wurde
30         /// </summary>
31         public DateTime? CreationDate { get; set; } = DateTime.Now;
32
33         /// <summary>
34         /// Gibt an, bis wann die Aufgabe erledigt sein soll
35         /// </summary>
36         public DateTime? DueDate { get; set; }
37
38         /// <summary>
39         /// Möglichkeit die Aufgabe einer Kategorie (zum Beispiel: Haushalt) zuzuordnen
40         /// </summary>
41         public string? Category { get; set; }
42
43         #region Konstruktor
44         1 Verweis
45         57
46         }
47         #endregion
48     }
49 }
```

Abbildung 7: ToDoItem.cs

```
1  using Microsoft.EntityFrameworkCore;
2  using ToDoApp.Shared;
3
4  namespace ToDoApp.Klassen
5  {
6      10 Verweise
7      public class ApplicationDbContext : DbContext
8      {
9          #region Member
10         protected readonly IConfiguration Configuration;
11         #endregion
12
13         #region Konstruktor
14         0 Verweise
15         public ApplicationDbContext(IConfiguration configuration)
16         {
17             Configuration = configuration;
18         }
19         #endregion
20
21         #region DbSets
22         // Definieren die Tabellen, welche auf Basis der angegebenen Klassen erstellt werden sollen
23         3 Verweise
24         public DbSet<Userclass> Userclasses { get; set; }
25         6 Verweise
26         public DbSet<ToDoItem> ToDoItems { get; set; }
27         #endregion
28
29         #region OnConfiguring
30         /// <summary>
31         /// Holt sich den in den appsettings.json definierten DbConnectionString
32         /// </summary>
33         /// <param name="optionsBuilder"></param>
34         0 Verweise
35         protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
36         {
37             optionsBuilder.UseSqlServer(Configuration.GetConnectionString("DbConnectionString"));
38         }
39         #endregion
40     }
41 }
```

Abbildung 8: ApplicationDbContext.cs

To-Do-App "To-Dino" in Blazor

```
1 @page "/"
2 using Microsoft.EntityFrameworkCore
3 using ToDoApp.Klassen
4 using Radzen.Blazor
5 using ToDoApp.Modal
6 [inject DialogService DialogService]
7 [inject ApplicationDbContext DbContext]
8 [inject Userclass Userclass]
9
10 /* Durch den Singleton "Userclass" prüfen wir, ob ein Nutzer angemeldet ist, wenn dem nicht der Fall ist, wird die Anmeldemaske angezeigt */
11 if (!Userclass.IsLoggedIn)
12 {
13     ShowLoginModal();
14 }
15
16 /*
17 Die Hauptseite der Anwendung
18 RadzenDropZoneContainer => Eine Komponente, welche als Unterlemite (ChildContent) RadzenDropZones akzeptiert
19 RadzenDropZones => Die eigentlichen Dropzonen für die einzelnen Items, welche UI-Komponenten als Unterlemite akzeptieren, wie zum Beispiel RadzenText
20 RadzenCard => Mit diesen werden hier die ToDoItems in den Dropzonen definiert. Wegen Renderfehlern (Items werden Doppelt gelistet) wird diese Komponente versteckt
21 Sie ist jedoch nötig, um den Bearbeitungs-Button hinzuzufügen
22 */
23 @
24 <div class="board">
25     <RadzenDropZoneContainer Title="ToDoItem" Data="data"
26         ItemSelector="@ItemSelector"
27         ItemRender="@OnItemRender"
28         CanDrop="@CanDrop"
29         Drop="OnDrop">
30         <ChildContent>
31             <div class="left-container">
32                 <div style="position: relative;">
33                     <RadzenDropZone Value="Status.NotStarted" class="column not-started" Style="flex: 1; gap: 1rem;">
34                         <ChildContent>
35                             <RadzenText Text="Offen" TextStyle="TextStyle.DisplayH6" Style="text-align: center;" />
36                             @foreach (ToDoItem item in data.Where(i => i.Status == Status.NotStarted))
37                             {
38                                 <RadzenCard Style="margin: 1rem; cursor: grab;" @key="item.Id" hidden="true">
39                                     <div class="task-content">
40                                         <strong>@item.Title</strong>
41                                         <RadzenButton Icon="edit" ButtonStyle="ButtonStyle.Secondary" Click="@{(args => EditItem(item))}" Style="margin-left: 1rem;" />
42                                     </div>
43                                 </RadzenCard>
44                             }
45                         </ChildContent>
46                         <Footer>
47                             <div style="position: absolute; bottom: 1em;">
48                                 <RadzenButton Text="Neuer Task" ButtonStyle="ButtonStyle.Secondary" Click="NewToDoItem" />
49                             </div>
50                         </Footer>
51                     </div>
52                     <div style="position: absolute; bottom: 1em;">
53                         <RadzenButton Text="Neuer Task" ButtonStyle="ButtonStyle.Secondary" Click="NewToDoItem" />
54                     </div>
55                 </RadzenDropZone>
56             </div>
57             <div>
58                 <RadzenDropZone Value="Status.Started" class="column started" Style="flex: 1; gap: 1rem; min-width: fit-content;">
59                     <RadzenText Text="Angefangen" TextStyle="TextStyle.DisplayH6" Style="text-align: center;" />
60                     @foreach (ToDoItem item in data.Where(i => i.Status == Status.Started))
61                     {
62                         <RadzenCard Style="margin: 1rem; cursor: grab;" @key="item.Id" hidden="true">
63                             <div class="task-content">
64                                 <strong>@item.Title</strong>
65                                 <RadzenButton Icon="edit" ButtonStyle="ButtonStyle.Secondary" Click="@{(args => EditItem(item))}" Style="margin-left: 1rem;" />
66                             </div>
67                         </RadzenCard>
68                     }
69                 </RadzenDropZone>
70             </div>
71             <div>
72                 <RadzenDropZone Value="Status.Completed" class="column completed" Style="flex: 1; gap: 1rem; min-width: fit-content;">
73                     <RadzenText Text="Fertig" TextStyle="TextStyle.DisplayH6" Style="text-align: center;" />
74                     @foreach (ToDoItem item in data.Where(i => i.Status == Status.Completed))
75                     {
76                         <RadzenCard Style="margin: 1rem; cursor: grab;" @key="item.Id" hidden="true">
77                             <div class="task-content">
78                                 <strong>@item.Title</strong>
79                                 <RadzenButton Icon="edit" ButtonStyle="ButtonStyle.Secondary" Click="@{(args => EditItem(item))}" Style="margin-left: 1rem;" />
80                             </div>
81                         </RadzenCard>
82                     }
83                 </RadzenDropZone>
84             </div>
85             <div>
86                 <RadzenDropZone Value="Status.Deleted" class="column delete" Style="flex: 1; gap: 1rem; min-width: fit-content;">
87                     <RadzenText Text="Zum Löschen hier hin ziehen" TextStyle="TextStyle.DisplayH6" Style="text-align: center;" />
88                     </RadzenDropZone>
89                 </div>
90             </ChildContent>
91         </div>
92
93 /*
94 Stellt die Items in den Dropzones dar, mit denen wir am Ende interagieren
95 Hier sagen wir, was als Titel der Karte angezeigt werden soll
96 */
97 @
98 <Template>
99     <div class="task-content">
100         <strong>@context.Title</strong>
101         <RadzenButton Icon="edit" ButtonStyle="ButtonStyle.Secondary" Click="@{(args => EditItem(context))}" Style="margin-left: 1rem;" />
102     </div>
103 </Template>
104 </RadzenDropZoneContainer>
105 </div>
106
107 @code {
108     #region Member
109     // Theoretisch könnten wir hier auch mit dem Datatype List arbeiten, wären dann jedoch auch auf Listen beschränkt.
110     // So könnten hier Listen, aber auch Arrays zum Einsatz kommen, was das Programm flexibler in der Gestaltung macht
111     IList<ToDoItem>? data;
112     #endregion
113
114     #region Delegates
115     // Ein Delegate ist ein Typ, der einen Verweis auf eine Methode speichert. Sie werden verwendet um Methoden als Parameter zu übergeben und
116     // Rückfunktionen zu definieren
117     // Sie sind besonders nützlich für Ereignisbehandlung und Situationsabhängige Methodenaufrufe
118 }
```

```
1  using System.Collections.Generic;
2  using System.ComponentModel.DataAnnotations;
3  using System.ComponentModel.DataAnnotations.Schema;
4  using System.Threading.Tasks;
5  using Microsoft.EntityFrameworkCore;
6
7  namespace ToDoApp Klassen
8  {
9      /// <summary>
10     /// Repräsentiert einen Benutzer der Anwendung. Hat die Möglichkeit sich bei der Anwendung mit seinem Username und Password anzumelden.
11     /// </summary>
12     /// 13 Verweise
13     public class Userclass
14     {
15         /// <summary>
16         /// Primary Key in der Datenbanktabelle Userclasses
17         /// </summary>
18         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
19         public int Id { get; set; }
20
21         [Required]
22         [MaxLength(50)]
23         public string Username { get; set; }
24
25         [Required]
26         [MaxLength(50)]
27         public string Password { get; set; }
28
29         [NotMapped] // Wird von der Datenbank ignoriert, da nicht wichtig
30         public bool IsLoggedIn { get; set; } = false;
31     }
32 }
```

Abbildung 10: Userclass.cs