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

Leo - Jonathan - Sebastian

12. Juni 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Begriff	Beschreibung / Ergänzung
To-Dino	TTo-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 garantierenäufiger Entwurf einer geplanten Benutzeroberfläche
Debuggerp	Vorläufiger Entwurf einer geplanten Benutzeroberfläche
Debugger	Software-Tool, welches Entwicklern hilft Code eines Programmes zu analysieren und Fehler zu finden

Abkürzungsverzeichnis

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

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 Appliklation 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 Unterrichtstunden 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 des 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, Internetzuugänge, Entwicklungsumgebung (Visual Studio 2022), Git etc.

Der Personalbedarf beschränlt 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 Sprint 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 Anfoderungen 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 Lehrjar bei inray monatlich 950 € (brutto).

$$8\frac{\text{h}}{\text{Tag}} \times 220\frac{\text{Tage}}{\text{Jahr}} = 1{,}760\frac{\text{h}}{\text{Jahr}}$$

$$950\frac{€}{\text{Monat}} \times 12\frac{\text{Monate}}{\text{Jahr}} = 11{,}400\frac{€}{\text{Jahr}}$$

$$\frac{11{,}400\frac{€}{\text{Jahr}}}{1{,}760\frac{\text{h}}{\text{Jahr}}} \approx 6{,}48\frac{€}{\text{h}}$$

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

Zielplattform

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 eigenem Netzwerk oder über das Internet mit jedem Browser erreichen kann.

Entscheidung der Programmiersprache

Das Blazor-Framework benutzt als Programmiersprache CSharp. 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 CSharp 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 ausfürhliche 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 Mock-up 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 Komponentenbibliothek 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 `ChildContet` 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 zusätlich 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 schießlich 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. `[DatabaseGereated(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 Agrument 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 stehts getestet, ob diese auch den Anforderungen entsprechen. Dazu zählen dass 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. Nebden 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 von nöten 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 Featuren 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

Detaillierte Zeitplanung Hier könnte ihre Tabelle abgebildet sein

A2

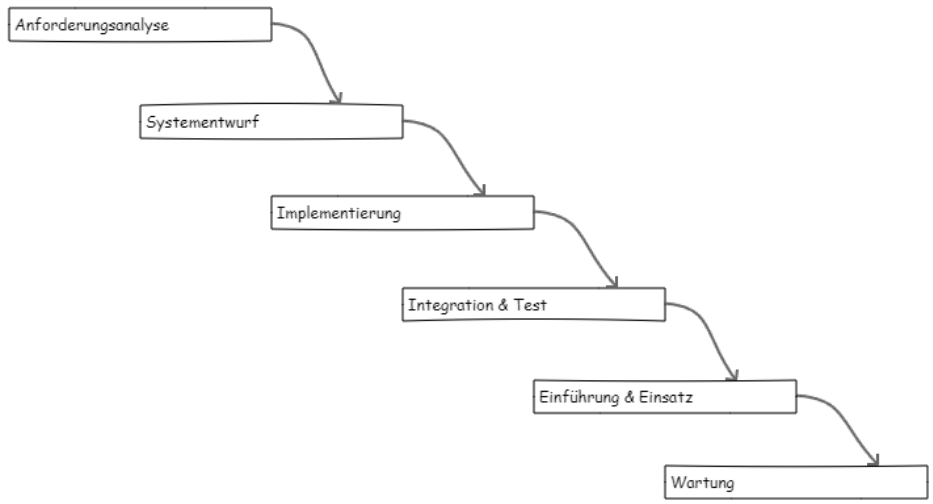


Abbildung 1: Darstellung des geplanten Wasserfall-Modells

Wasserfall-Modell

A3

Programmstruktur

A4

To-Dino Seite

A5

Komponenten der Hauptseite

A6

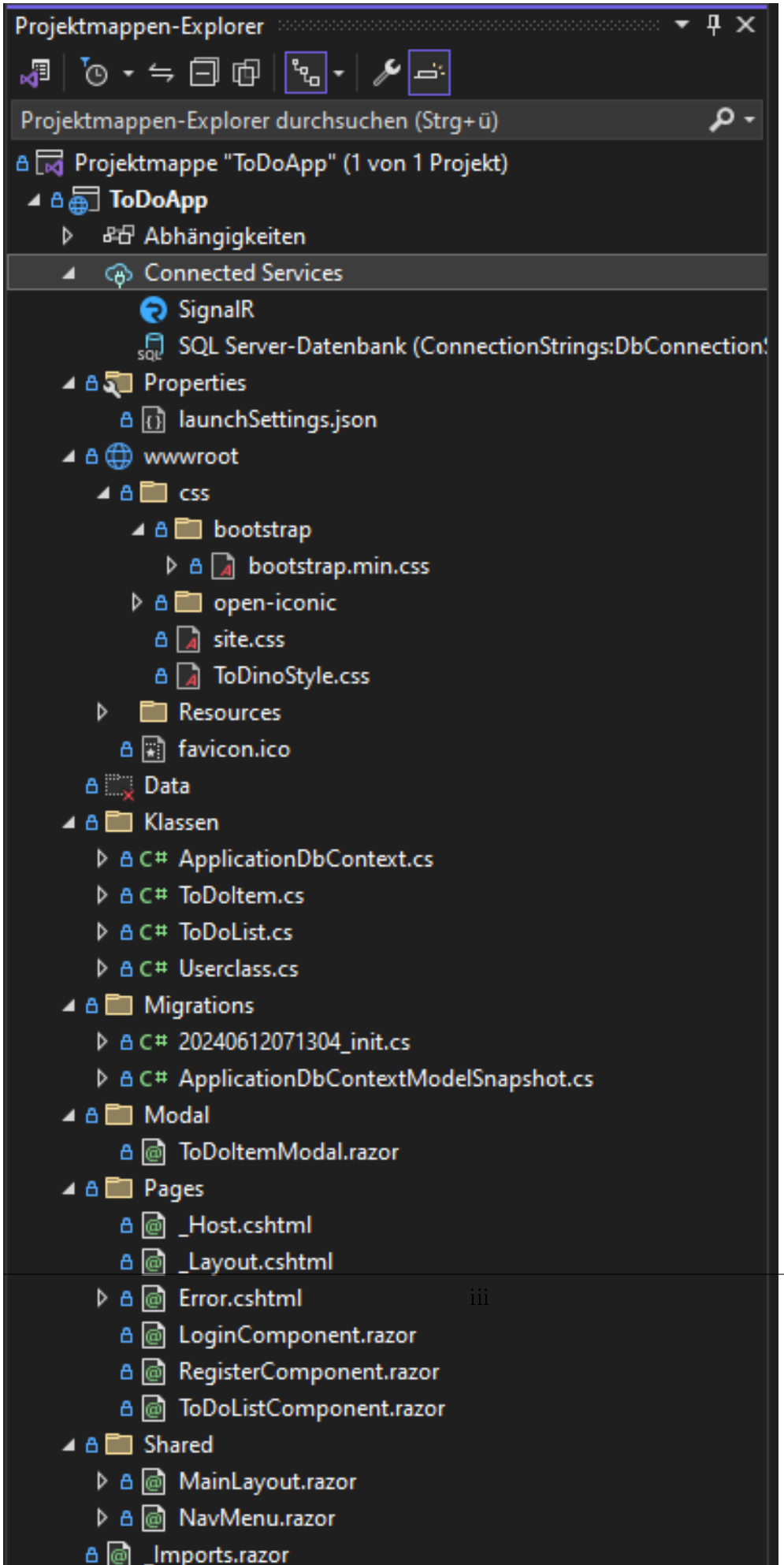
Hauptseiten Code-Block

A8

Programmstruktur

A10

ApplicationDbContext-Klasse


Projektmappen-Explorer
Projektmappen-Explorer durchsuchen (Strg+ü)
Projektmappe "ToDoApp" (1 von 1 Projekt)
ToDoApp
Abhängigkeiten
Connected Services
SignalR
SQL Server-Datenbank (ConnectionStrings:DbConnection
Properties
launchSettings.json
wwwroot
css
bootstrap
bootstrap.min.css
open-iconic
site.css
ToDinoStyle.css
Resources
favicon.ico
Data
Klassen
ApplicationDbContext.cs
ToDoItem.cs
ToDoList.cs
Userclass.cs
Migrations
20240612071304_init.cs
ApplicationDbContextModelSnapshot.cs
Modal
ToDoItemModal.razor
Pages
_Host.cshtml
_Layout.cshtml
Error.cshtml
LoginComponent.razor
RegisterComponent.razor
ToDoListComponent.razor
Shared
MainLayout.razor
NavMenu.razor
_Imports.razor

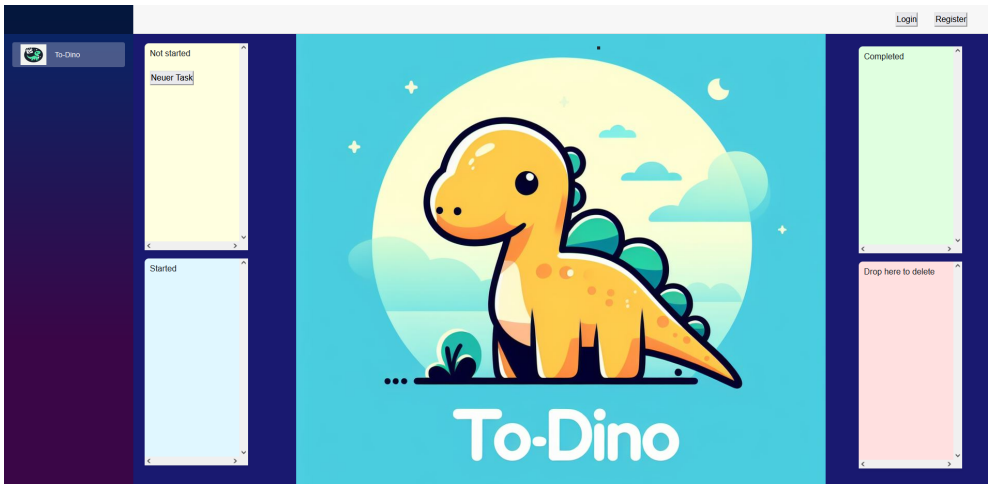
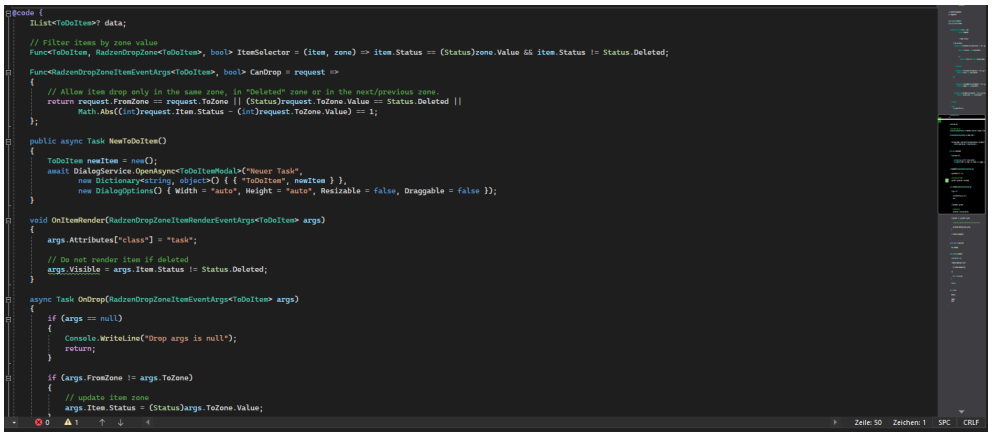


Abbildung 3: Darstellung der implementierten To-Dino Seite

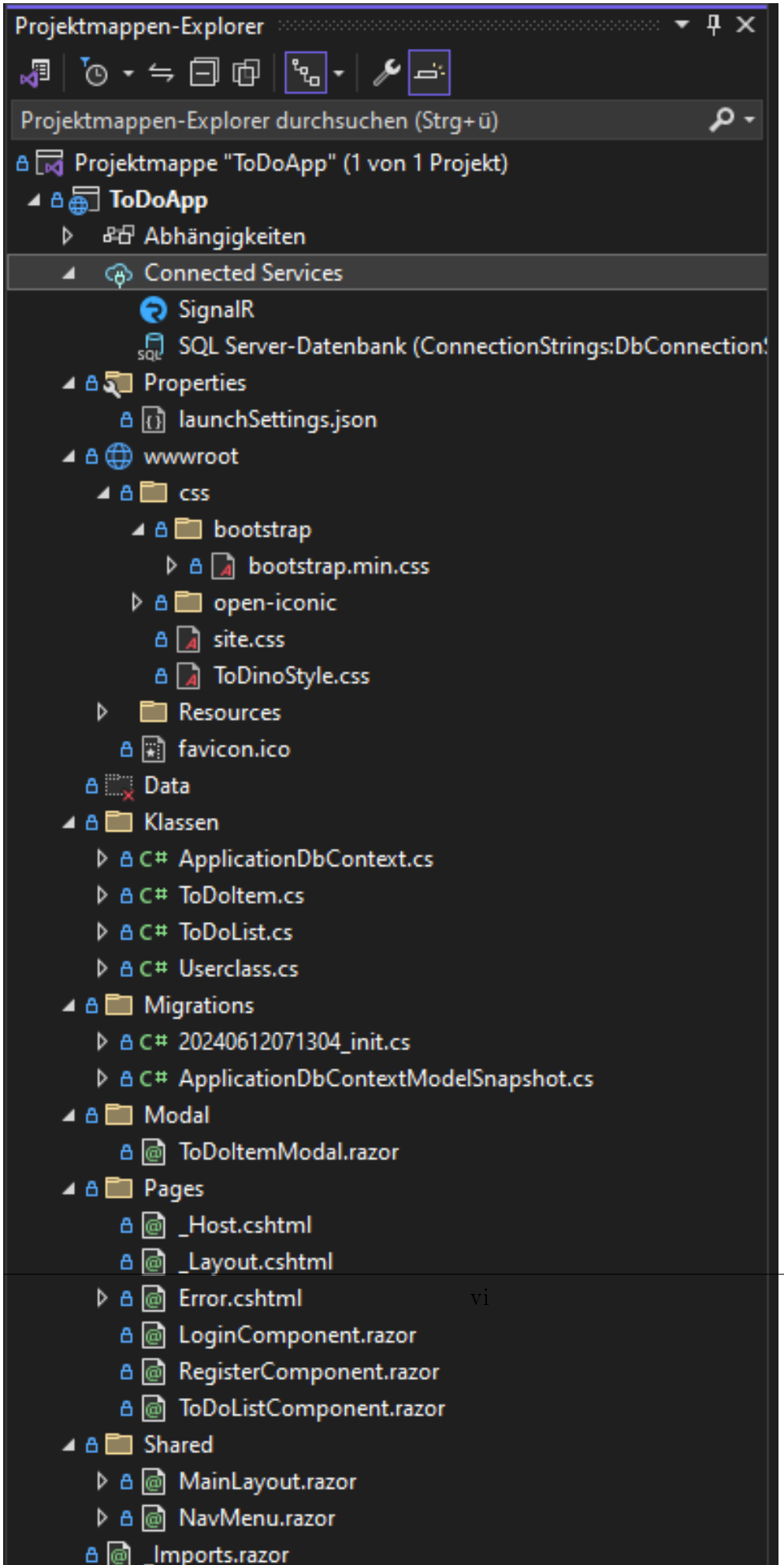
```
<div class="board">
    <RadzenDropZoneContainer TItem="ToDoItem" Data="data"
                             ItemSelector="@ItemSelector"
                             ItemRender="@OnItemRender"
                             CanDrop="@CanDrop"
                             Drop="OnDrop" class="board">
        <ChildContent>
            <div class="left-container">
                <RadzenDropZone Value="Status.NotStarted" class="column not-started" Style="flex: 1; gap: 1rem;">
                        <ChildContent>
                            <RadzenText Text="Not started" TextStyle="TextStyle.Subtitle2" />
                        </ChildContent>
                        <Footer>
                            <div>
                                <RadzenButton Text=@($"Neuer Task") ButtonStyle="ButtonStyle.Secondary" Click=@NewToDoItem />
                            </div>
                        </Footer>
                    </RadzenDropZone>

                    <RadzenDropZone Value="Status.Started" class="column started" Style="flex: 1; gap: 1rem; min-width:fit-content;">
                        <RadzenText Text="Started" TextStyle="TextStyle.Subtitle2" />
                    </RadzenDropZone>
            </div>

            <div class="right-container">
                    <RadzenDropZone Value="Status.Completed" class="column completed" Style="flex: 1; gap: 1rem; min-width:fit-content;">
                        <RadzenText Text="Completed" TextStyle="TextStyle.Subtitle2" />
                    </RadzenDropZone>

                <RadzenDropZone Value="Status.Deleted" class="column delete" Style="flex: 1; gap: 1rem; min-width:fit-content;">
                        <RadzenText Text="Drop here to delete" TextStyle="TextStyle.Subtitle2" />
                    </RadzenDropZone>
            </div>
        </ChildContent>

        <Template>
            <strong>@context.Title</strong>
        </Template>

    </RadzenDropZoneContainer>
</div>
```

Abbildung 4: Die integrierten Blazor.Radzen Komponenten



```
@code {
    IList<ToDoItem>? data;

    // Filter items by zone value
    Func<ToDoItem, RadzenDropZone<ToDoItem>, bool> ItemSelector = (item, zone) => item.Status == (Status)zone.Value && item.Status != Status.Deleted;

    Func<RadzenDropZoneItemEventArgs<ToDoItem>, bool> CanDrop = request =>
    {
        // Allow item drop only in the same zone, in "Deleted" zone or in the next/previous zone.
        return request.FromZone == request.ToZone || (Status)request.ToZone.Value == Status.Deleted ||
               Math.Abs((int)request.Item.Status - (int)request.ToZone.Value) == 1;
    };

    public async Task NewToDoItem()
    {
        ToDoItem newItem = new();
        await DialogService.OpenAsync<ToDoItemModal>("Neuer Task",
              new Dictionary<string, object>() { { "ToDoItem", newItem } },
              new DialogOptions() { Width = "auto", Height = "auto", Resizable = false, Draggable = false });
    }

    void OnItemRender(RadzenDropZoneItemRenderEventArgs<ToDoItem> args)
    {
        args.Attributes["class"] = "task";

        // Do not render item if deleted
        args.Visible = args.Item.Status != Status.Deleted;
    }

    async Task OnDrop(RadzenDropZoneItemEventArgs<ToDoItem> args)
    {
        if (args == null)
        {
            Console.WriteLine("Drop args is null");
            return;
        }

        if (args.FromZone != args.ToZone)
        {
            // update item zone
            args.Item.Status = (Status)args.ToZone.Value;
        }
```

Abbildung 5: Darstellung des geplanten Wasserfall-Modells


Projektmappen-Explorer
Projektmappen-Explorer durchsuchen (Strg+ü)
Projektmappe "ToDoApp" (1 von 1 Projekt)
ToDoApp
Abhängigkeiten
Connected Services
SignalR
SQL Server-Datenbank (ConnectionStrings:DbConnection
Properties
launchSettings.json
wwwroot
css
bootstrap
bootstrap.min.css
open-iconic
site.css
ToDinoStyle.css
Resources
favicon.ico
Data
Klassen
ApplicationDbContext.cs
ToDoItem.cs
ToDoList.cs
Userclass.cs
Migrations
20240612071304_init.cs
ApplicationDbContextModelSnapshot.cs
Modal
ToDoItemModal.razor
Pages
_Host.cshtml
_Layout.cshtml
Error.cshtml
LoginComponent.razor
RegisterComponent.razor
ToDoListComponent.razor
Shared
MainLayout.razor
NavMenu.razor
_Imports.razor

```
12 Verweise
public class ApplicationDbContext : DbContext
{
    #region Member
    protected readonly IConfiguration Configuration;
    #endregion

    #region Konstruktor
    0 Verweise
    public ApplicationDbContext(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    #endregion

    #region DbSets
    // Definieren die Tabellen, welche auf Basis der angegebenen Klassen erstellt werden sollen
    3 Verweise
    public DbSet<Userclass> Userclasses { get; set; }
    0 Verweise
    public DbSet<ToDoList> ToDoLists { get; set; }
    3 Verweise
    public DbSet<ToDoItem> ToDoItems { get; set; }
    #endregion

    #region OnConfiguring
    /// <summary>
    /// Holt sich den in den appsettings.json definierten DbConnectionstring
    /// </summary>
    /// <param name="optionsBuilder"></param>
    0 Verweise
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(Configuration.GetConnectionString("DbConnectionString"));
    }
    #endregion
}
```

Abbildung 7: Implementierung der Klasse 'ApplicationDbContext'