# Laboration

## Info om laboration

Den här labben vägleder dig genom Blazor och kontrollerar kunskaper om följande:

* Skapa en Blazor-webbapp.
* Hantera tillstånd för din app.
* Anpassa utseende med hjälp av CSS.
* Skapa ett ”Connect Four”-spel.

Vi uppnår ovanstående mål genom att skriva ett klassiskt fyra-i-rad "Connect Four"-spel som körs i din webbläsare.

Titta denna för mer info [Connect 4 - Play online game - 2 player papergames.io](https://papergames.io/en/connect4)

Efter en del av övningar finns frågor, svara på de under respektive fråga. Frågor ska svaras självständigt. De är markerad med Fn(där n står för nummer).

Uppgifter i slutet av labben kan diskuteras i studiegrupp för att få tips /idéer men ska också göras självständigt.

### Scenario - skapa ett "Connect Four"-spel

Du är utvecklare på ett företag som bygger datorspel. Du bestämmer dig för att utvärdera Blazor genom att bygga ett spel med det.

I det här spelet, två spelare alternativa turas om att placera en spelpjäs (vanligtvis en checker) i toppen av brädet. Spelstycken faller till den lägsta raden i en kolumn och spelaren som placerar fyra speldelar för att göra en linje horisontellt, vertikalt eller diagonalt vinner.

## Blazor

Det finns många sätt att skapa webbappar. Om du vill skapa webbappar som är mycket interaktiva och snabba använder du vanligtvis mycket JavaScript.

En annan metod är användningen av Blazor-ramverket från Microsoft, som gör att du kan använda C# och .NET för att hantera alla dina problem med webbutveckling.

## Vad är Blazor?

Blazor är ett ramverk för att skapa webbsidor med HTML, CSS och C#. Vi kan definiera webbplatsens layout och design med standard-HTML och CSS. De interaktiva komponenterna på webbsidorna kan sedan hanteras med C#-kod som körs på en server eller i webbläsaren med hjälp av en webbstandardteknik som heter WebAssembly. Med Blazor kan vi definiera våra webbsidor och komponenter med hjälp av Razor-syntax, en bekväm blandning av HTML och C#. Du kan enkelt återanvända Blazor-komponenter på andra sidor och komponenter. Den här funktionen innebär att vi enkelt kan skapa och återanvända delar av vår app.

## Vad är WebAssembly?

WebAssembly är en standardteknik som är tillgänglig i varje modern webbläsare som gör att kod kan köras, ungefär som JavaScript, i en webbläsare. Vi kan använda verktyg för att förbereda vår C#-kod för användning i webbläsaren som en WebAssembly-app, och dessa verktyg ingår i .NET SDK.

## Uppdatera din app omedelbart med Hot Reload

När du utvecklar en app vill du se till att utvecklarflödet är snabbt så att du kan se hur ändringar påverkar din app. Det kan vara tidskrävande att göra en ändring, spara den, återskapa alla resurser och distribuera om appen i webbläsaren.

Du kan förbättra arbetsflödet med frekvent inläsning. Med Hot Reload kan du tillämpa en ändring på appen som körs utan att behöva starta om den.

Under utveckling ska du testa alla funktioner.

## Blazor-komponenter

När du skapar appar med Blazor består din app av många komponenter, var och en med sitt eget ansvarsområde. För att skapa komponenter använder du Razor-filer med ett *.razor-tillägg* . Så, vad är Razor?

Razor är en programmeringssyntax som blandar HTML-syntax med C#-kod. Här är ett exempel:

En bild som visar text, Teckensnitt, skärmbild, linje

Automatiskt genererad beskrivning

I det här exemplet @product.Name matchas till strängen "Blazor", som sedan återges inuti taggen div . De renderade utdata ser ut så här:



Den översta delen av Razor-exemplet består av HTML-kod som komponenten ska återge när den körs. Du kan återge värdet för C#-uttryck med hjälp av @ tecknet . Razor fortsätter att tolka C#-kod efter tills den identifierar en tagg och återupptar sedan återgivningen @ av HTML. Du kan vara tydlig med när C#-uttrycket börjar och slutar med parens: @(...).

Allt inom @code {} innehåller C#-kod för att definiera medlemmar i din genererade komponentklass. Du kan se en Razor-fil som ett praktiskt sätt att definiera en C#-klass som definierar HTML-renderingslogik. Du använder @code blocket för att definiera C#-medlemmar för din komponenttyp, till exempel fält, egenskaper och metoder.

## Styla en Blazor-komponent

Blazor-komponenter renderar HTML så att du kan formatera Blazor-komponenter med vanliga CSS-formatmallar. Alternativt har Blazor-komponenter en funktion som kallas CSS-isolering som gör att du kan skapa formatregler som endast gäller för innehållet i komponenten. Genom att skapa en fil med samma namn som komponenten och lägga till filnamnstillägget .css identifierar Blazor den här namngivningen som de formatmallar som ENDAST ska tillämpas på HTML-innehåll i motsvarande komponent.

Blazor-komponenter kan också definiera innehåll som ska läggas till i HTML-header på sidan med hjälp av en särskild HeadContent tagg:

En bild som visar text, Teckensnitt, vit, design

Automatiskt genererad beskrivning

Den här style taggen och innehållet återges i taggen head på sidan.

## Routable komponenter

Filen Home.razor är en komponent som kan navigeras till från en webbläsare. Den innehåller HTML, C# och referenser till andra Blazor-komponenter. Vi kan identifiera den här filen som en sida på grund av @page "/" att direktivet finns på den första raden. Det här direktivet tilldelar "/"-vägen till komponenten och instruerar Blazor att svara med innehållet i den här filen när standardsidan på "/"-adressen begärs.

## Övning - Blazor

I den här övningen skapar vi ett Blazor-projekt och vi börjar skapa en app som blir ett "Connect four"-spel under hela modulen.

### Skapa ett nytt Blazor-projekt

Först ska vi skapa ett nytt projekt för vårt spel med Visual Studio 2022.

1. Skapa en ny Blazor-app i Visual Studio 2022 genom att välja **menyn File > New > Project**.
2. Välj "**Blazor Web App**" i listan med mallar och ge den namnet ”Connect Four”. Välj **Next**.
3. Välj **.NET 8** som ramverksversion. **Authentication type** ska vara inställd på **None**, **Interactive render mode** ska vara inställt på **Server** och **Interactivity location** ska anges till **Per page/component**.

Include sample pages och configure for HTTPS ska vara bockad.

Den här åtgärden bör skapa en Connect Four-katalog som innehåller vår app.

1. Kör appen genom att trycka på F5 i Visual Studio 2022.

Nu bör du se att Blazor-appen körs i webbläsaren:

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

### Skapa en brädkomponent/Board

Nu ska vi skapa en Board komponents som ska användas av spelare i vårt spel. Komponenten definieras med Razor-syntax, vilket är en blandning av HTML och C#.

1. Högerklicka på mappen Komponenter i Solution Explorer i Visual Studio. Välj Lägg till > Razor-komponent på snabbmenyn och ge filen namnet Board.razor.

Vi använder den här komponenten för att lagra allt som behövs för spelkortslayouten och hantera interaktioner med den. Det första innehållet i den här nya komponenten är en h3 tagg och ett @code block som anger var C#-kod ska skrivas:

En bild som visar text, Teckensnitt, handskrift, vit

Automatiskt genererad beskrivning

1. **Home** Förbered sidan genom att öppna filen Components/Pages/Home.razor och rensa ut allt efter den tredje raden med taggen PageTitle.

En bild som visar text, Teckensnitt, vit, skärmbild

Automatiskt genererad beskrivning

1. Lägg till vår Board komponent på Home sidan genom att lägga till en <Board /> tagg som matchar filnamnet för vår nya komponent.

En bild som visar text, Teckensnitt, vit, design

Automatiskt genererad beskrivning

1. Kör appen med F5 för att se ändringarna. Om appen redan körs trycker du på knappen Hot Reload bredvid knappen Kör/Fortsätt för att tillämpa ändringarna på appen som körs.

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

Tips - Välj alternativet Hot Reload on File Save på menyn Hot Reload för att tillämpa ändringar i appen som körs när du ändrar en fil.

### Lägga till innehåll och stil i spelplanen

Nu ska vi börja definiera en spelplan med de sju kolumnerna och sex raderna. Vi lägger till lite stil för att väcka vår styrelse till liv.

1. I board.razor-filen tar du bort HTML-koden överst och lägger till följande innehåll för att definiera en tavla med 42 platser för spelbitar.

Vi kan använda en C#- for loop för att generera 42 board positions. Container taggen **span** plockas upp och upprepas med dess innehåll 42 gånger för att representera vår tavla.

En bild som visar text, Teckensnitt, handskrift, skärmbild

Automatiskt genererad beskrivning

När vi sparar brädkomponenten uppdateras vår app och den visas som en tom sida tack vare funktionerna för frekvent omläsning som återskapar och startar den uppdaterade appen.

OBS! Du kan uppmanas av Visual Studio att starta om appen när filerna ändras. Bekräfta att appen ska återskapas vid kodredigeringar och att appen automatiskt startar om och uppdaterar webbläsaren när du lägger till funktioner.

### Utforma komponenten

Nu ska vi lägga till lite stil i komponenten Board genom att definiera några färger för brädramen och spelarna ovanför den första div taggen i Board.razor-filen :

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

Dessa CSS-variabler --board-bg, --player1: red, --player2: blue hämtas och används i resten av vår formatmall för den här komponenten.

Nu ska vi lägga till en färdig formatmall för spelet i komponenten Board .

1. Högerklicka i Solution Explorer i mappen Komponenter och skapa en ny CSS-fil med namnet Board.razor.css.
2. Kopiera följande innehåll till den nya Board.razor.css-filen :

div{position:relative}nav{top:4em;width:30em;display:inline-flex;flex-direction:row;margin-left:10px}nav span{width:4em;text-align:center;cursor:pointer;font-size:1em}div.board{margin-top:1em;flex-wrap:wrap;width:30em;height:24em;overflow:hidden;display:inline-flex;flex-direction:row;flex-wrap:wrap;z-index:-5;row-gap:0;pointer-events:none;border-left:10px solid var(--board-bg)}span.container{width:4em;height:4em;margin:0;padding:4px;overflow:hidden;background-color:transparent;position:relative;z-index:-2;pointer-events:none}.container span{width:3.5em;height:3.5em;border-radius:50%;box-shadow:0 0 0 3em var(--board-bg);left:0;position:absolute;display:block;z-index:5;pointer-events:none}.player1,.player2{width:3.5em;height:3.5em;border-radius:50%;left:0;top:0;position:absolute;display:block;z-index:-8}.player1{background-color:var(--player1);animation-timing-function:cubic-bezier(.5,.05,1,.5);animation-iteration-count:1;animation-fill-mode:forwards;box-shadow:0 0 0 4px var(--player1)}.player2{background-color:var(--player2);animation-timing-function:cubic-bezier(.5,.05,1,.5);animation-iteration-count:1;animation-fill-mode:forwards;box-shadow:0 0 0 4px var(--player2)}.col0{left:calc(0em + 9px)}.col1{left:calc(4em + 9px)}.col2{left:calc(8em + 9px)}.col3{left:calc(12em + 9px)}.col4{left:calc(16em + 9px)}.col5{left:calc(20em + 9px)}.col6{left:calc(24em + 9px)}.drop1{animation-duration:1s;animation-name:drop1}.drop2{animation-duration:1.5s;animation-name:drop2}.drop3{animation-duration:1.6s;animation-name:drop3}.drop4{animation-duration:1.7s;animation-name:drop4}.drop5{animation-duration:1.8s;animation-name:drop5}.drop6{animation-duration:1.9s;animation-name:drop6}@keyframes drop1{100%,75%,90%,97%{transform:translateY(1.27em)}80%{transform:translateY(.4em)}95%{transform:translateY(.8em)}99%{transform:translateY(1em)}}@keyframes drop2{100%,75%,90%,97%{transform:translateY(5.27em)}80%{transform:translateY(3.8em)}95%{transform:translateY(4.6em)}99%{transform:translateY(4.9em)}}@keyframes drop3{100%,75%,90%,97%{transform:translateY(9.27em)}80%{transform:translateY(7.2em)}95%{transform:translateY(8.3em)}99%{transform:translateY(8.8em)}}@keyframes drop4{100%,75%,90%,97%{transform:translateY(13.27em)}80%{transform:translateY(10.6em)}95%{transform:translateY(12em)}99%{transform:translateY(12.7em)}}@keyframes drop5{100%,75%,90%,97%{transform:translateY(17.27em)}80%{transform:translateY(14em)}95%{transform:translateY(15.7em)}99%{transform:translateY(16.5em)}}@keyframes drop6{100%,75%,90%,97%{transform:translateY(21.27em)}80%{transform:translateY(17.4em)}95%{transform:translateY(19.4em)}99%{transform:translateY(20.4em)}}

Här är några av de CSS som används för att formatera brädet och "hålslag" för vart och ett av utrymmena. Det finns mer tillgängligt innehåll än vad som visas nedan i CSS-filen för speldelarna och deras animeringar på skärmen.

En bild som visar text, skärmbild, Teckensnitt, dokument

Automatiskt genererad beskrivning

Webbläsaren bör uppdateras åt dig (om inte, kan du manuellt uppdatera webbläsaren med F5), och du bör hälsas med en korrekt gul Connect Four board:

Grattis, nu har du början av spelet. I senare sektion lägger vi till spellogik i den.

En bild som visar text, skärmbild, Operativsystem, design

Automatiskt genererad beskrivning

## Spellogik

I den här sektion går vi igenom begrepp som hur ”Connect Four”-spelet fungerar och vilka konstruktioner av Blazor du behöver veta om att producera ett spel.

### Connect Four

Tanken med spelet "Connect Four" är att du ska lägga ut fyra spelbitar i rad vågrätt, vertikalt eller diagonalt innan motståndaren gör det. Saker du behöver tänka på när du implementerar det här spelet är att hålla reda på ditt nuvarande tillstånd av spel bitar, dina motståndare och att söka efter en vinnare. Det är användbart när du programmerar för att tänka på en spelslinga, en uppsättning åtgärder som du måste upprepa tills du kan deklarera en vinnare. Ungefär så här:

1. Börja med ett "reset state", dvs ett rent bräde utan spelstycken.
2. Användaren placerar en spelpjäs.
3. Motståndaren placerar en spelpjäs.
4. Sök efter vinnare:
   1. Om det finns en vinnare, deklarera vinnare och slutspel, eller starta om spelet.
   2. Upprepa steg 2 om ingen vinnare.

### Representerar tillstånd/state i kod

Först och främst, vad är tillstånd? Tillstånd i spel är vad som har ägt rum i spelet, hur många poäng du har, var dina spelplatser placeras och så vidare.

En viktig vägledning när det gäller tillstånd i spelutveckling är att hålla staten åtskild från användargränssnittet eftersom det gör ändringar enklare och din kod enklare att läsa bland andra fördelar.

När det gäller Blazor innebär det att tillstånd och logik kring tillstånd ska vara i sin egen C#-klass, så här:

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

#### F1. Vad innehåller klassen State?

* Inledande variabler som array med spelare, int för antalet rundor spelade och en bool som avgör om spelet är över eller inte
* State-konstruktör som innehåller arrayen med spelare, deras respektive namn samt antal startpoäng. Konstruktören innehåller info för ”start-stadiet” för spelet där ”gameRoundsPlayed” är satt på 0 då inga spel har körts och ”gameOver” är false då spelet inte är slut.
* Slutligen två metoder, en för att återställa spelet (gameOver = false som är dess inledande stadie), och en för att spara informationen för avslutat spel (gameOver = true och att öka antalet spel körda)

#### F2. Vad används en instans av klassen State för?

* Som tidigare skrivet så innehåller State-klassen information såsom variabler, konstruktör och metoder som är används för att hålla reda på den nuvarande stadiet i spelet. Den håller koll på spelarnas namn, deras poäng, antal rundor spelade och om spelet är över eller inte.

### Reset game state

I Blazor finns det en metod som anropas när komponenten initieras innan något annat händer. Den här metoden är ett bra ställe att placera kod som hjälper dig att placera ditt spel i ett "Reset state", som att skapa brädet, spelarna och återställa eventuella poäng om det behövs från tidigare spelsessioner.

#### F3. Vilken metod är det? Och hur kan det se ut i kod?

* Metoden är ResetBoard i GameState.cs som tar listan ”TheBoard” och återskapar en ny lista när metoden kallas.
* Så här ser den ut: public void ResetBoard() {

TheBoard = new List<int>(new int[42]); }

### Hantera spelarinteraktioner med hjälp av händelser

När du eller dina motståndare gör ett drag måste du fånga den här interaktionen. Du kodar användarinteraktioner som händelser/events som ditt spel/din app bör svara på.

Du kan till exempel välja en knapp eller utföra en drag & drop-rörelse för att flytta en spelbit.

Så här kan det se ut i kod:



#### F4. Vad gör koden ovan?

* Den innehåller en uppmaning till spelaren att trycka på symbolen för att sätta ut en speldel i spelet. ”@onclick” är ett event som aktiveras när span-symbolen klickas. Metoden PlayPiece verkställs med argumentet ”0”.

### Ändra tillstånd/state

En åtgärd som sker i spelet bör påverka ditt speltillstånd. I föregående exempel där vi anropar PlayPiece()bör vi ändra tillståndet till att säga att den här delen av tavlan nu är upptagen av en bit. Det innebär att med tanke på vår exempelklass State behöver vi ett sätt att representera spelbitar, ungefär så här:

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

## Övning – spellogik

I den här övningen lägger vi till spellogik i vår app för att säkerställa att vi får ett fullt fungerande spel.

För att hålla den här labben på ämnet om Blazor tillhandahåller jag en klass som heter GameState som innehåller logiken för att hantera spelet. (Finns att ladda ner på LearnPoint under Laboration)

OBS! Bekanta dig med innehåll i klassen Game State innan du fortsätter vidare med labben.

### Lägga till speltillstånd

Nu ska vi lägga till klassen GameState i projektet och sedan göra den tillgänglig för komponenter som en singleton-tjänst via dependency injection.

1. Kopiera filen GameState.cs till roten för projektet.
2. Öppna filen Program.cs i projektets rot och lägg till den här instruktionen för att konfigurera GameState som en singleton-tjänst i din app:



1. Lägg till följande @inject direktiv överst i board.razor-filen för att mata in spelets aktuella tillstånd i komponenten:



Nu kan vi börja ansluta vår Board komponent till spelets tillstånd.

### Reset state

Vi börjar med att återställa spelets tillstånd när komponenten Board först målas på skärmen. Vi lägger till kod för att återställa spelets tillstånd när komponenten initieras.

1. Lägg till en OnInitialized metod med ett anrop till ResetBoard i @code blocket längst ned i Board.razor-filen , så här:

En bild som visar text, Teckensnitt, skärmbild, linje

Automatiskt genererad beskrivning

När brädet först visas för en användare återställs tillståndet till början av ett spel.

### Skapa spelbitar

Nu ska vi allokera de möjliga 42 speldelarna som kan spelas. Vi kan representera spelbitarna som en matris som refereras av 42 HTML-element på tavlan. Vi kan flytta och placera dessa delar genom att tilldela en uppsättning CSS-klasser med kolumn- och radpositioner.

1. Definiera ett strängmatrisfält i code blocket för att lagra våra speldelar:



1. Lägg till kod i HTML-avsnittet som skapar 42 span taggar, en för varje speldel, i samma komponent (yttersta div):

En bild som visar text, Teckensnitt, handskrift, linje

Automatiskt genererad beskrivning

Din fullständiga kod bör se ut så här: (… för innersta div)

En bild som visar text, skärmbild, Teckensnitt, linje

Automatiskt genererad beskrivning

(OBS! fel i Pieces /pieces välj en när ni döper arrayen)

Detta tilldelar en tom sträng till CSS-klassen för varje speldelsintervall.

#### F5. Vad händer när inget format tillämpas på spelbitarna, dvs när vi har en tom sträng för CSS class?

* När spelaren trycker på knapparna överst på sidan placeras en av speldelarna i den kolumn spelaren valde. Saknas denna bit av kod fungerar spelet fortfarande men användaren kan inte se var speldelarna är.

### Hantera spelpjäsplacering

Nu ska vi lägga till en metod som ska hanteras när en spelare placerar en bit i en kolumn. Klassen GameState vet hur du tilldelar rätt rad för spelstycket och rapporterar tillbaka raden som den hamnar i. Vi kan använda den här informationen för att tilldela CSS-klasser som representerar spelarens färg, den sista platsen för stycket och en CSS-släppanimering.

Vi anropar den här metoden PlayPieceoch accepterar en indataparameter som anger kolumnen som spelaren har valt.

1. Lägg till den här koden under matrisen pieces vi definierade i föregående steg.

En bild som visar text, skärmbild, Teckensnitt, linje

Automatiskt genererad beskrivning

#### F6. Vad gör PlayPiece metoden?

* Metoden tar inledningsvis parametern col med datatypen byte. Efter detta hämtar den Playerturn från State-objektet, även CurrentTurn hämtas från detta objekt och detta appliceras till ”player” respektive ”turn”. Även PlayPiece hämtas från State-objektet men den tar parametern ”col” som inparameter för att hämta information från objektet. Slutligen skrivs en string med vem som spelar, vilken kolumn det hamnar i och och vilken rad speldelen hamnar på.

Om du tittar i den angivna Board.razor.css hittar du CSS-klasserna matchande kolumn, rad och spelarsväng.

Den resulterande effekten är att spelstycket placeras i kolumnen och animeras för att släppas i den nedre raden när den här metoden anropas.

### Välja en kolumn

Vi måste sedan placera några kontroller som gör det möjligt för spelare att välja en kolumn och anropa vår nya PlayPiece metod. Vi använder tecknet "🔽" för att ange att du kan släppa en del i den här kolumnen.

1. Lägg till en rad med klickbara knappar ovanför första <div> :

En bild som visar text, skärmbild, Teckensnitt, linje

Automatiskt genererad beskrivning

Attributet @onclick anger en händelsehanterare för klickhändelsen. Men för att hantera UI händelse måste en Blazor-komponent renderas med hjälp av ett interaktive render mode. Som standard återges Blazor-komponenter statiskt från servern. Vi kan använda ett interaktivt återgivningsläge för en komponent med hjälp av attributet @rendermode.

1. Uppdatera komponenten Boar på sidan Home för att använda återgivningsläget InteractiveServer.



Återgivningsläget InteractiveServer hanterar UI-händelser för dina komponenter från servern via en WebSocket-anslutning med webbläsaren.

1. Kör appen med dessa ändringar. Det bör se ut så här nu:

En bild som visar text, skärmbild, Färggrann, design

Automatiskt genererad beskrivning

Ännu bättre är att när vi väljer en av släppknapparna längst upp kan följande beteende observeras:

En bild som visar gul, skärmbild, mönster, Färggrann

Automatiskt genererad beskrivning

Bra jobbat! Nu kan vi lägga till bitar på tavlan. Objektet GameState är smart nog att svänga fram och tillbaka mellan de två spelarna. Gå vidare och välj fler släppknappar och titta på resultatet.

### Hantering av vinster och fel

Om du spelar med det spel som du har konfigurerat just nu ser du att det ger upphov till fel när du försöker placera för många bitar i samma kolumn och när en spelare har vunnit spelet.

Nu ska vi lägga till några felhantering och indikatorer i vår styrelse för att göra det aktuella tillståndet tydligt. Vi lägger till ett statusområde ovanför brädet och under släppknapparna.

1. Infoga följande markering efter elementet nav :

En bild som visar text, Teckensnitt, linje, skärmbild

Automatiskt genererad beskrivning

Med den här pålägget kan vi visa indikatorer för:

* + Visa spelvinnare
  + En knapp som gör att vi kan starta om spelet
  + Felmeddelanden
  + Den aktuella spelarens tur

Vi fyller i lite logik för att ange dessa värden.

1. Lägg till följande kod efter matrisen pieces:

En bild som visar text, skärmbild, Teckensnitt, linje

Automatiskt genererad beskrivning

* + Egenskapen **CurrentTurn** beräknas automatiskt baserat på tillståndet för winnerMessage egenskapen och och **PlayerTurn** för **GameState**.
  + **ResetStyle** Beräknas baserat på innehållet i **WinnerMessage**. Om det finns en **winnerMessage** visas Reset på skärmen.

1. Nu ska vi hantera felmeddelandet när en pjäs spelas upp. Lägg till en rad för att rensa felmeddelandet och sedan omsluta koden i PlayPiece metoden med ett try...catch block för att ange errorMessage om ett undantag inträffade:

En bild som visar text, skärmbild, Teckensnitt

Automatiskt genererad beskrivning

Vår felhanterarindikator är enkel och använder Bootstrap CSS-ramverket för att visa ett fel i riskläge.

En bild som visar text, skärmbild, Färggrann, Operativsystem

Automatiskt genererad beskrivning

1. Nu ska vi lägga till den **ResetGame** metod som knappen utlöser för att starta om ett spel. För närvarande är det enda sättet att starta om ett spel att uppdatera sidan. Med den här koden kan vi stanna kvar på samma sida.

En bild som visar text, Teckensnitt, skärmbild, linje

Automatiskt genererad beskrivning

#### F7. Vad ansvarar för ResetGame metoden?

* Metoden ResetGame använder metoden ResetBoard för att ta bort speldelarna och återställa brädet till hur det var för början. winnerMessage och errorMessage återställs till en ny string. Pieces = new string[42] återställer spelet så att spelarna har en ny möjlighet att fylla ut de 42 platserna på brädet.

Vi är fortfarande kvar i en situation där vi inte kan välja återställningsknappen. Nu ska vi lägga till lite logik i PlayPiece metoden för att identifiera slutet av spelet.

1. Nu ska vi se om det finns en vinnare i spelet genom att lägga till ett switch-uttryck efter vårt try...catch block i PlayPiece.

En bild som visar text, Teckensnitt, skärmbild, linje

Automatiskt genererad beskrivning

F6. Vad returnerar Metoden CheckForWin?  
Metoden CheckForWin returnerar en uppräkning som rapporterar vilken spelare, om någon har vunnit spelet eller om spelet är oavgjort. Det här växlingsuttrycket anger fältet winnerMessage på lämpligt sätt om ett spel över tillstånd har inträffat.

Nu när vi spelar och når ett spelslutsscenario visas dessa indikatorer:

En bild som visar text, skärmbild, Färggrann, Operativsystem

Automatiskt genererad beskrivning

### Svara på följande frågor om olika delar i labben

#### F8. Hur svarar du på en klickhändelse?

* Genom att använda en onclick-eventhandler kopplad till kod i @code-blocket

#### F9. I vilken komponentmetod skulle du placera logik som skulle initiera komponenten?

* Jag ser det som lämpligast i OnInitialized-metoden då logiken isåfall kan initieras direkt när programmet startas

#### F10. Hur kan en komponent hantera en specifik väg/path/route?

* Genom att lägga till t.ex. @page ”/counter” överst på razor-sidorna

## Uppgift – Anpassning med hjälp av parametrar

Spelet fungerar, men du kanske inte gillar våra standardfärger. I Blazor kan vi definiera parametrar för våra komponenter som gör att vi kan skicka in värden som ser ut som attribut på en HTML-tagg.

I den här övningen fokuserar vi på anpassning och att få spelet att se bättre ut med hjälp av parametrar.

Är inte det en cool bräda? Gör bräden svart, player1 - green och player 2 - purple via parametrar.

En bild som visar skärmbild, text, Färggrann, design

Automatiskt genererad beskrivning

## Uppgift – Skapa test projekt med xUnit

Testa metoden CheckForWin med två test metoder CheckForWin \_NoWinner\_EmptyBoard() och CheckForWin\_TieGame()

## Utmaningar

Tänk på följande utmaningar:

* Ta bort standardlayouten och extra sidor i appen för att göra den mindre.
* Förbättra parametrarna till komponenten Board så att du kan skicka ett giltigt CSS-färgvärde.
* Förbättra indikatorernas utseende med viss CSS- och HTML-layout.
* Introducera ljudeffekter.
* Lägg till en visuell indikator och förhindra att en släppknapp används när kolumnen är full.
* Lägg till nätverksfunktioner så att du kan spela upp en vän i webbläsaren.
* Infoga spelet i ett .NET MAUI med Blazor-program och spela det på din telefon eller surfplatta.