SchoolShop : de score implementeren

1. Maak een nieuw model aan met de naam “Score”.  
   We maken dus (zometeen) een nieuwe tabel Score aan.  
   Daar zullen we per gebruiker en per artikel bijhouden of een score (waarde van 1 tot 5) door betrokken gebruiker werd toegekend aan een artikel en of er eventueel een comment werd gemaakt.  
   We zullen dus voor elke gebruiker per artikel waar hij score/opmerkingen voor invoert een record toevoegen aan deze tabel. Maakt de bezoeker geen score/opmerking, dan zal er uiteraard ook geen record voor bestaan.

public class Score

{

public int Id { get; set; }

[ForeignKey("User")]

public int UserId { get; set; }

[Display(Name = "Gebruiker")]

public User User { get; set; }

[ForeignKey("Article")]

public int ArticleId { get; set; }

[Display(Name = "Artikel")]

public Article Article { get; set; }

[Display(Name = "Score")]

[Range(1, 5, ErrorMessage = "Kies een waarde tussen 1 en 5")]

public int Stars { get; set; } = 5;

[Display(Name = "Opmerking")]

public string Comment { get; set; }

}

1. We hebben nu uiteraard nog maar enkel een model dat nog geen tabel heeft in de database zelf. Tot nu toe gingen we telkens eerst scaffolding toepassen vooraleer we onze database lieten bijwerken, maar dat hoeft niet noodzakelijk.
   * Open /Data/SchoolShopContext.cs

Voeg onder de bestaande DBSet’s een nieuwe toe met de naam Scores :

public DbSet<SchoolShop.Models.Score> Scores { get; set; }

* + Voeg een nieuwe migratie toe en laat je database bijwerken.

add-migration ScoresImplemented

update-database

1. De score en eventuele opmerkingen zullen we laten gebeuren in de /Pages/Articles/Details pagina.  
   Aan deze pagina zullen we dus heel wat wijzigingen dienen aan te brengen.

Zoals gesteld zal elke gebruiker bij elk artikel een score kunnen invoeren : deze score zal weggeschreven worden als record in de tabel Scores.  
Omdat we dit per gebruiker en per artikel gaan bijhouden is het mogelijk om, wanneer een gebruiker een bepaald artikel terug in detail opent hem de score te laten zien die hij zelf aan het artikel toekende (iets gelijkaardigs gaan we trouwens ook doen met de opmerkingen).

In de tabel Article houden we ook een score bij, maar deze score is dan het gemiddelde van alle scores voor dat artikel die door alle gebruikers (eventueel) werd toegekend.  
Telkens een gebruiker voor een bepaald artikel een score toevoegt/wijzigt/verwijdert zullen we deze gemiddelde score van het artikel aanpassen.

1. De Details.cshtml.cs file

We werken eerst ons code bestand uit en passen hierna de Details.cshtml pagina aan.

Er zal heel wat code hiervoor in deze pagina bijkomen. Wij brengen stuk per stuk aan, maar op het einde van deze paragraaf tonen we de volledige content van de details.cshtml.cs file.

Bijkomende props aanmaken :

public Score Score { get; set; }

public int ScoreCount { get; set; } = 0;

public List<Score> Reviews { get; set; }

Wanneer we op deze pagina terechtkomen, dan zijn we 1 (één) artikel aan het bekijken.  
Via de availability klasse weten we ook met welke bezoeker we (eventueel) te maken hebben.  
Ter info : bezoekers die niet zijn ingelogd zullen geen beoordeling kunnen geven.

Wanneer de pagina geladen wordt, gaan we straks op zoek naar een eventueel record dat overeenstemt met het ID van de huidige gebruiker en het ID van het artikel dat momenteel bekeken wordt. Wordt dit gevonden, dan vullen we de prop **Score** met de info uit dat record. Vinden we het niet, dan wordt de prop Score gevuld met de Null waarde.

Op de details.cshtml pagina willen we straks afbeelden hoeveel bezoekers een score hebben ingediend. We zullen straks bij het laden van de pagina bekijken hoeveel bezoekers op het huidige artikel een score hebben ingediend en we zullen dit bijhouden in de prop **ScoreCount**.

Verderop zullen we ook de commentaren die de verschillende bezoekers hebben geschreven over dit artikel willen afbeelden. We gaan dus alle records uit de tabel Scores bijhouden in de prop **Review** (een List van Score records).

De code om dit allemaal te regelen is behoorlijk lang. Om de OnGet methode overzichtelijk te houden zullen we een groot deel van de (huidige en de nieuw te maken) bewerkingen in afzonderlijke methode onderbrengen, temeer we een deel van de logica straks kunnen hergebruiken in onze OnPost methode.

Vervang de OnGet (of OnGetAsync) methode helemaal door onderstaande code : de methode die opgeroepen worden en die we straks zullen uitwerken worden in het geel gemarkeerd.

public IActionResult OnGet(int? id, int? categoryid, int? brandid)

{

Availability = new Availability(\_context, HttpContext);

if (id == null)

{

return NotFound();

}

Article = GetArticle(id);

if (Article == null)

{

return NotFound();

}

Score = GetPersonalScore();

CalculateNumberOfScores();

Articles = GetArticles(id, categoryid, brandid);

BrandId = brandid;

CategoryId = categoryid;

HandlePagingButtons(id);

Reviews = BuildReviews();

return Page();

}

* De methode GetArticle zoekt in de tabel Article naar het artikel met het meegeleverde ID en retourneert dat artikel (of Null) :

private Article GetArticle(int? id)

{

// vul de PROP Article a.d.h.v. het meegeleverde id

return \_context.Article

.Include(a => a.Brand)

.Include(a => a.Category)

.FirstOrDefault(a => a.Id == id);

}

* De methode GetArticles retourneert alle artikels waarbij rekening wordt gehouden met de filters die in de indexpagina van toepassing zijn. De List van artikels die hier geretourneerd wordt zullen we straks gebruiken om onze 2 navigatieknopjes correct te laten werken.

private List<Article> GetArticles(int? id, int? categoryid, int? brandid)

{

// vraag alle artikels op

// sorteer ze op dezelfde manier als de index pagina

// filter ze op dezelfde manier als de index pagina

IQueryable<Article> artquery = \_context.Article

.OrderBy(a => a.Category.CategoryName)

.ThenBy(a => a.Price);

if (brandid != null && categoryid == null)

{

artquery = artquery.Where(b => b.BrandId.Equals(brandid));

}

if (brandid == null && categoryid != null)

{

artquery = artquery.Where(b => b.CategoryId.Equals(categoryid));

}

if (brandid != null && categoryid != null)

{

artquery = artquery.Where(b => b.BrandId.Equals(brandid) && b.CategoryId.Equals(categoryid));

}

return artquery.ToList();

}

* De methode HandlePagingButtons zal er voor zorgen dat de knoppen “Vorige” en “Volgende” je effectief naar het juiste artikel brengen. Zoals hierboven aangegeven zullen we de zonet aangemaakt List van (eventueel gefilterde) artikels hiervoor gebruiken.  
  De methode vult dat props PreviousId en NextId en deze waarden worden gekoppeld aan de knoppen “Vorige” en “Volgende” :

private void HandlePagingButtons(int? id)

{

PreviousId = null;

NextId = null;

// ga op zoek naar het id van vorige en volgende

for (int i = 0; i < Articles.Count; i++)

{

if (((Article)Articles[i]).Id == id)

{

if (i > 0)

PreviousId = ((Article)Articles[i - 1]).Id;

if (i < Articles.Count - 1)

NextId = ((Article)Articles[i + 1]).Id;

break;

}

}

if (PreviousId == null) PreviousId = id;

if (NextId == null) NextId = id;

}

* De methode GetPersonalScore gaat op zoek naar een eventueel eerder ingevoerde score voor het huidige artikel. Wordt dit gevonden dan wordt de inhoud van dat record in de vorm van een Score opbject geretourneerd. Wordt het niet gevonden (de gebruiker heeft nog niet eerder een score opgegeven voor dit artikel) of is de bezoeker niet ingelogd dan wordt Null geretourneerd.

private Score GetPersonalScore()

{

// vul de prop Score met het record uit de tabel Score

// die door de betrokken gebruiker eventueel eerder al

// al gemaakt heeft (anders gewoon Null)

if (!string.IsNullOrEmpty(Availability.UserId))

{

int userId = int.Parse(Availability.UserId);

Score = \_context.Scores

.FirstOrDefault(s => s.UserId == userId && s.ArticleId == Article.Id);

return Score;

}

else

return null;

}

* De methode BuildReviews retourneert een List van scores waarvoor geldt dat er effectief een commentaar werd geschreven. Deze lijst gaan we straks gebruiken in Details.cshtml zodat de (ingelogde) bezoeker kan lezen wat anderen hiervan vonden.

private List<Score> BuildReviews()

{

// we halen alle records op uit de tabel score

// (met join naar tabel users)

// voor het actieve artikel

// we vullen hiermee de PROP Reviews (List<Score>)

IQueryable<Score> reviewQuery = \_context.Scores

.Include(s => s.User)

.Where(s => s.ArticleId.Equals(Article.Id))

.Where(s => s.Comment.Trim().Length > 0)

;

return reviewQuery.ToList();

}

* De methode CalculateNumberOfScores berekent gewoon hoeveel scores er werden gegeven aan het betrokken artikel en vult de prop ScoreCount :

private void CalculateNumberOfScores()

{

// bereken het aantal scores voor dit ene artikel

ScoreCount = \_context.Scores.Where(s => s.ArticleId == Article.Id).Count();

}

We moeten in onze Detail pagina nu ook een OnPost event voorzien : de gebruiker zal straks immers een score kunnen invoeren (van 1 tot 5 sterren) en eventueel een commentaar invoeren.   
  
Vervang de eventueel reeds aanwezige OnPost (of OnPostAsync) methode met onderstaande code. Terug plaatsen we het grootste deel van de logica in afzonderlijke methoden die we daaronder één voor één afbeelden (nieuwe methode markeren we in geel, methoden die reeds bestaan markeren we in groen) :

public void OnPost(int? star, int? savecomment)

{

Availability = new Availability(\_context, HttpContext);

Article = GetArticle(Article.Id);

if (star != null)

{

HandleStars((int)star);

UpdateAverageScore();

}

string comment = "";

if(savecomment != null)

{

comment = Score.Comment;

}

CalculateNumberOfScores();

Score = GetPersonalScore();

if (savecomment != null)

{

if (savecomment == 1)

UpdateComment(comment);

else

RemoveComment();

}

Reviews = BuildReviews();

}

Zoals je kan zien ontvangt onze OnPost methode 2 argumenten die uiteraard elk een verschillende betekenis hebben. Hoe we er voor zorgen dat deze argumenten hier terechtkomen zien we straks in Details.cshtml.  
In het argument **star** zal een getal van 1 t/m 5 terechtkomen : dit is de score die de bezoeker aan dit artikel geeft.  
In het argument **savecomment** zal ofwel null, ofwel 1 ofwel -1 terechtkomen : is de waarde 1, dan betekent dit dat we het commentaar dienen toe te voegen aan het record dat eerder al gemaakt werd wanneer een score werd toegekend (niet vergeten, commentaar mag pas toegevoegd worden nadat de gebruiker een score heeft toegekend). Is de waarde -1 dan betekent dit dat de gebruiker zijn commentaar wenst te wissen.

* De methode **HandleStars** zorgt er voor dat er in de tabel Scores ofwel een nieuw record wordt toegevoegd, ofwel een bestaand record wordt gewijzigd. Deze methode wordt opgeroepen wanneer de gebruiker een score heeft opgegeven.

private void HandleStars(int stars)

{

// de bezoeker heeft op één van de knoppen met sterretjes geklit

// we kijken na of betrokken bezoeker ooit eerder al een oordeel geveld heeft

int userId = int.Parse(Availability.UserId);

Score = \_context.Scores

.FirstOrDefault(s => s.UserId == userId && s.ArticleId == Article.Id);

// is dat niet zo, dan maken we een nieuw record aan in de tabel Score

if (Score == null)

{

Score = new Score();

Score.UserId = userId;

Score.ArticleId = Article.Id;

Score.Stars = stars;

Score.Comment = "";

\_context.Scores.Add(Score);

\_context.SaveChanges();

}

// is dat wel zo, dan zoeken we dat record op en passen we dat aan

else

{

Score.Stars = stars;

\_context.Attach(Score).State = EntityState.Modified;

\_context.SaveChanges();

}

}

* In de methode **UpdateAverageScore** gaan we van het betrokken record de som laten bereken van alle toegekende scores en het aantal toegekende scores. Hiermee berekenen we het gemiddelde van alle toekenningen en passen de waarde van het veld score aan in het betrokken artikel :

private void UpdateAverageScore()

{

// bereken de som van alle scores voor dit ene artikel

int total = \_context.Scores.Where(s => s.ArticleId == Article.Id).Sum(s => s.Stars);

// bereken het aantal scores voor dit ene artikel

int count = \_context.Scores.Where(s => s.ArticleId == Article.Id).Count();

// bereken het gemiddelde hiervan

decimal score = 1.0M \* total / count;

// haal het artikel object op (het record van het betrokken artikel)

Article = \_context.Article.FirstOrDefault(m => m.Id == Article.Id);

// pas de score aan van dit artikel volgens het gemiddelde dat we zonet berekend hebben

Article.Score = score;

// bewaar het record

\_context.Attach(Article).State = EntityState.Modified;

\_context.SaveChanges();

}

* We kunnen ook in de **OnPost** terechtkomen omdat de gebruiker ofwel een commentaar heeft toegevoegd, gewijzigd of wenst te verwijderen.  
  In de methode komt een argument toe met de naam **savecontent** (hoe die hier komt zien we straks). We zullen het zo regelen dat indien dit argument de waarde -1 heeft dit betekent dat de gebruiker zijn commentaar wenst te wissen, heeft dit argument de waarde 1 dan betekent dit dat we de commentaar die ingevoerd werd zullen gebruiken om zijn record aan te passen (m.a.w. de inhoud van het veld comment te vullen). Ter herinnering, een bezoeker kan pas commentaar plaatsen wanneer hij eerder al een score (sterretjes) heeft ingevoerd, dus we zijn zeker dat er reeds een record in de tabel Scores aanwezig is met het Id van de bezoeker en het Id van het artikel : meer nog, de prop Score is reeds gevuld met de inhoud van dat record.

private void UpdateComment(string comment)

{

Score.Comment = comment;

\_context.Attach(Score).State = EntityState.Modified;

\_context.SaveChanges();

}

private void RemoveComment()

{

Score.Comment = "";

\_context.Attach(Score).State = EntityState.Modified;

\_context.SaveChanges();

}

1. De Details.cshtml file

Eerst en vooral zullen we in de Details.cshtml file een aantal nieuwe variabelen nodig hebben.  
De meeste van deze variabelen zullen we gebruiken om zaken zichtbaar/onzichtbaar te maken, te accentueren …

We sommen eerst alle variabelen op die we nodig zullen hebben (gemarkeerd in groen) en geven daarna een woordje uitleg :

@page "{id:int?}"

@model SchoolShop.Pages.Articles.DetailsModel

@{

ViewData["Title"] = "Details artikel";

string imagePath = "~/images/" + (Model.Article.ImagePath ?? "noimage.jpg");

string userScore = "";

if (Model.Score != null)

userScore = Model.Score.Stars.ToString();

string scoreBlockStyle = "visibility: hidden; ";

string defaultScoreButtonStyle = "background:transparent;";

string userScoreButtonStyle = "background:oldlace;";

string selectedButtonStyle = "";

string myReviewStyle = "visibility: hidden; ";

if (Model.Score != null)

myReviewStyle = "visibility: visible; ";

ViewData["Email"] = "";

ViewData["Loginstyle"] = "visibility: visible; ";

ViewData["Logoutstyle"] = "visibility: hidden; ";

ViewData["Configstyle"] = "visibility: hidden; ";

if (!string.IsNullOrEmpty(Model.Availability.Email))

{

scoreBlockStyle = "visibility: visible; ";

ViewData["Email"] = Model.Availability.Email;

ViewData["BasketCount"] = Model.Availability.BasketCount;

ViewData["Loginstyle"] = "visibility: hidden; ";

ViewData["Logoutstyle"] = "visibility: visible; ";

if (Model.Availability.IsAdmin)

{

ViewData["Configstyle"] = "visibility: visible; ";

}

}

}

* Helemaal bovenaan zorgen we voor een route : “{id:int?}”

Deze route zal er voor zogen dat wanneer we in de OnGet en OnPost een argument voorzien met de naam “int? id” hier automatisch het ID van het betrokken artikel zal in terecht komen : de property Article is immers als eerste gebonden aan deze pagina (BindProperty)

* In de variabele userScore zullen we de (eventuele) score bewaren die de gebruiker eerder ingevoerd heeft voor dit artikel. We kunnen weten als de gebruiker al eerder een score invoerde indien Model.Score verschillend is van null.
* We hebben afgesproken dat een gebruiker pas een score zal kunnen geven wanneer hij zich aangemeld heeft. In de variabele scoreBlokcStyle bewaren we een stijlkenmerk (standaard verborgen) die we straks zullen gebruiken om de knoppen waarmee een score kan ingevoerd worden al dan niet te tonen. Wanneer we merken dat de gebruiker ingelogd is gaan we de stijl wijzigen in zichtbaar.
* We zullen straks 5 knoppen aanbieden (met sterretjes) waarmee de gebruiker zijn score zal te kennen kunnen geven. Wanneer een gebruiker al eerder voor het betrokken artikel een score uitbracht willen we dat visueel aantonen (dus de score die hij eerder meegaf).  
  In de variabele defaultScoreButtonStyle bewaren we een stijl die de knoppen onopvallend zal maken.  
  In de variabele userScoreButtonStyle bewaren we een stijl die een knop een meer opvallende achtergrond zal geven (deze stijl gaan we uiteraard gebruiken om duidelijk te maken wel score er eerder al werd ingegeven).

In de variabele selectedButtonStyle gaan we straks ofwel de inhoud van defaultScoreButtonStyle overnemen ofwel de inhoud van userScoreButtonStyle

* We stelden eerder al dat een gebruiker pas commentaar zal kunnen schrijven wanneer hij eerder al een score had ingegeven. M.b.v. de variabele myReviewStyle zullen we er voor zorgen dat de invoer van commentaar al dan niet mogelijk wordt gemaakt.

Onder de DIV waarin je de beschrijving van het artikel afbeeldt (maar boven de DIV waarin de Edit button zit, +/- regel 80) neem je onderstaande (uitgebreide) code over.  
De code is veel te lang om hier een volledige uitleg rond te schrijven, maar in de les wordt de volledige code samen doorlopen.

<div class="form-group row">

<span class="col-sm-2 col-form-label font-weight-bold">Score</span>

<div class="col-sm-10">

@for (int i = 0; i < (int)Math.Round(Model.Article.Score); i++)

{

<i class="fas fa-star" style="color:gold;"></i>

}

@for (int i = (int)Math.Round(Model.Article.Score); i < 5; i++)

{

<i class="far fa-star" style="color:gold;"></i>

}

<h6>@Html.DisplayFor(model => model.ScoreCount) beoordelingen.</h6>

</div>

</div>

<div style="@scoreBlockStyle">

<input **type**="hidden" **asp-for**="Article.Id" />

<form method="post">

<hr />

<h3>Oordeel zelf : </h3>

<p>

@if (userScore == "1")

selectedButtonStyle = userScoreButtonStyle;

else

selectedButtonStyle = defaultScoreButtonStyle;

<button type="submit" **asp-route-star**="1"

class="btn btn-sm btn-light"

title="1 ster : barslecht"

style="@selectedButtonStyle">

<i class="fas fa-star" style="color:gold;"></i>

<i class="far fa-star" style="color:lightgrey;"></i>

<i class="far fa-star" style="color:lightgrey;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

</button>

</p>

<p>

@if (userScore == "2")

selectedButtonStyle = userScoreButtonStyle;

else

selectedButtonStyle = defaultScoreButtonStyle;

<button type="submit" **asp-route-star**="2"

class="btn btn-sm btn-light"

title="2 sterren : niet zo goed"

style="@selectedButtonStyle">

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

</button>

</p>

<p>

@if (userScore == "3")

selectedButtonStyle = userScoreButtonStyle;

else

selectedButtonStyle = defaultScoreButtonStyle;

<button type="submit" **asp-route-star**="3"

class="btn btn-sm btn-light"

title="3 sterren : goed"

style="@selectedButtonStyle">

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

</button>

</p>

<p>

@if (userScore == "4")

selectedButtonStyle = userScoreButtonStyle;

else

selectedButtonStyle = defaultScoreButtonStyle;

<button type="submit" **asp-route-star**="4"

class="btn btn-sm btn-light"

title="4 sterren : zeer goed"

style="@selectedButtonStyle">

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="far fa-star" style="color: lightgrey;"></i>

</button>

</p>

<p>

@if (userScore == "5")

selectedButtonStyle = userScoreButtonStyle;

else

selectedButtonStyle = defaultScoreButtonStyle;

<button type="submit" **asp-route-star**="5"

class="btn btn-sm btn-light"

title="5 sterren : uitstekend"

style="@selectedButtonStyle">

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

<i class="fas fa-star" style="color:gold;"></i>

</button>

</p>

<hr />

<div style="@myReviewStyle">

<h3>Mijn review : </h3>

<div class="row">

<div class="col-sm-10">

<textarea **asp-for**="Score.Comment" class="form-control" rows="5"></textarea>

</div>

<div class="col-sm-2">

<button type="submit" class="btn btn-lg btn-info"

**asp-route-savecomment**="1">

<i class="far fa-save" style="font-size:18pt;"></i>

</button>

<button type="submit" class="btn btn-lg btn-danger"

**asp-route-savecomment**="-1">

<i class="fas fa-trash-alt" style="font-size:18pt;"></i>

</button>

</div>

</div>

<hr />

</div>

<h3>Wat anderen vonden : </h3>

<div class="row">

<table class="table">

<tbody>

@foreach (var item in Model.Reviews)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.User.FirstName)

&nbsp;

@Html.DisplayFor(modelItem => item.User.Name)

</td>

<td>

stars

@for (int i = 0; i < item.Stars; i++)

{

<i class="fas fa-star" style="color:gold;"></i>

}

@for (int i = item.Stars; i < 5; i++)

{

<i class="far fa-star" style="color:gold;"></i>

}

</td>

<td>

@Html.DisplayFor(modelItem => item.Comment)

</td>

</tr>

}

</tbody>

</table>

</div>

</form>

</div>

1. De Index.cshtml file

We wensen in onze index pagina bij elk artikel ook de score af te beelden in de vorm van sterretjes. We passen dus ook nog onze index pagina aan. We hoeven in de onderliggende code (index.cshtml.cs) geen aanpassingen te doen, gezien we in onze pagina over elke artikel object beschikken en de score daarin vervat zit. Opgepast, dit is een gebroken getal die we wel nog eerst zullen afronden naar een getal zonder cijfers na de komma zodat we netjes onze sterretjes zullen kunnen afbeelden.

In de foreach-lus, net onder de toekenning van de waarde aan imagePath maak je een nieuwe variabele aan met de naam roundScore waarin we dus de afgeronde score zullen bewaren (+/- regel 84) :

@foreach (var item in Model.Article)

{

string imagePath = "~/images/" + (item.ImagePath ?? "noimage.jpg");

string roundScore = item.Score.ToString("0");

<div class="col-6 col-lg-4 p-3">

…

Vervolgens neem je onder de <H5> tag waarmee je prijs afbeeldt onderstaande code over (+/- regel 100) :

<a **asp-page**="./Details"

**asp-route-id**="@item.Id"

**asp-route-categoryid**="@Model.SelectedCategoryId"

**asp-route-brandid**="@Model.SelectedBrandId"

class="btn">

<div class="card-body text-left" style="min-height:400px;">

<h5>Merk : @Html.DisplayFor(modelItem => item.Brand.BrandName)</h5>

<h5>Prijs : @Html.DisplayFor(modelItem => item.Price)</h5>

<h5>

Score :

@for (int i = 0; i < Int32.Parse(roundScore); i++)

{

<i class="fas fa-star" style="color:gold;"></i>

}

@for (int i = Int32.Parse(roundScore); i < 5; i++)

{

<i class="far fa-star" style="color:gold;"></i>

}

</h5>

<hr />

<div class="text-center">

<img class="imageThumbnail" **src**="@imagePath" **asp-append-version**="true" />

</div>

</div>

</a>