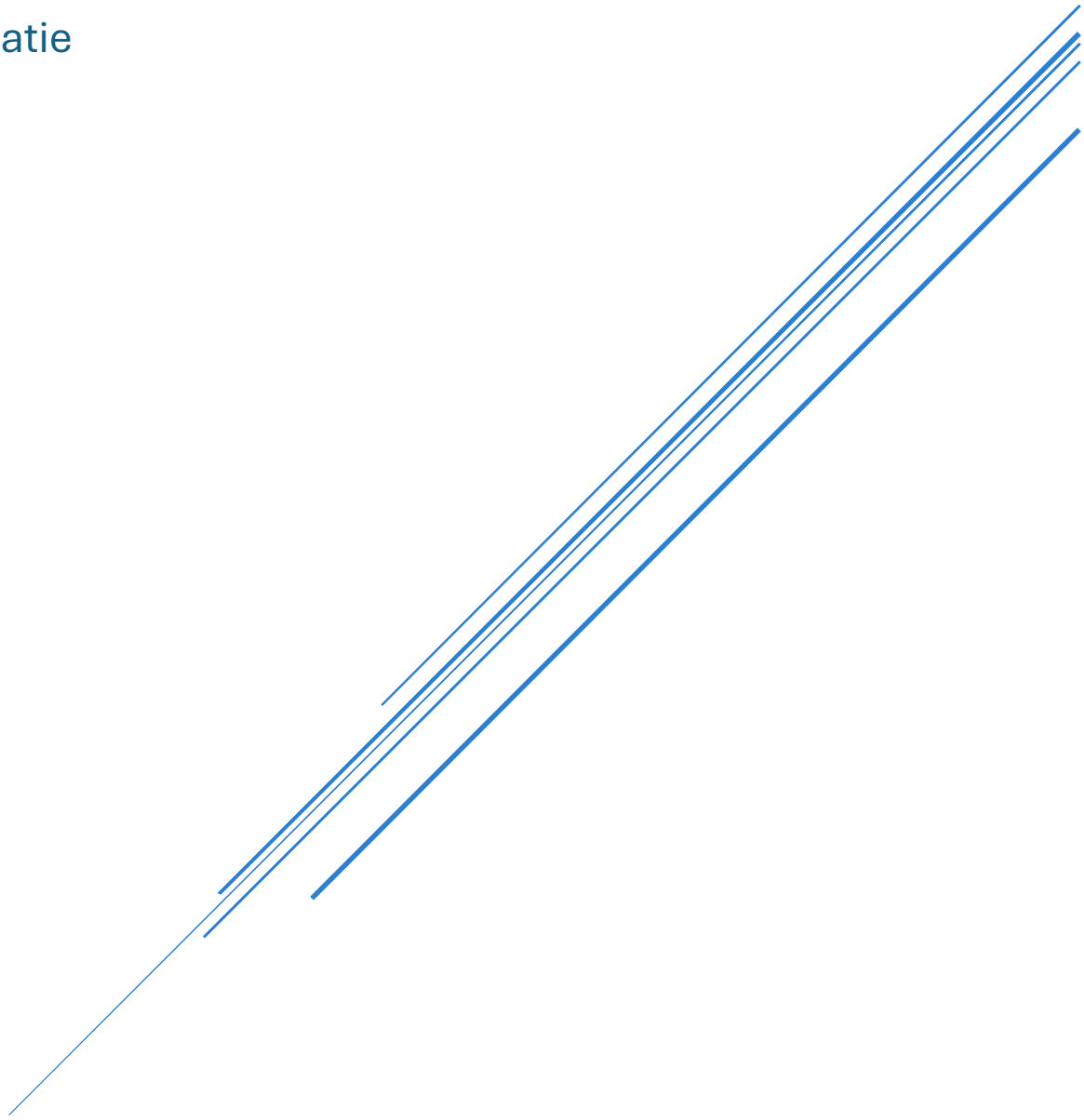
GAMEINFOAPI

Documentatie



Inhoudsopgave

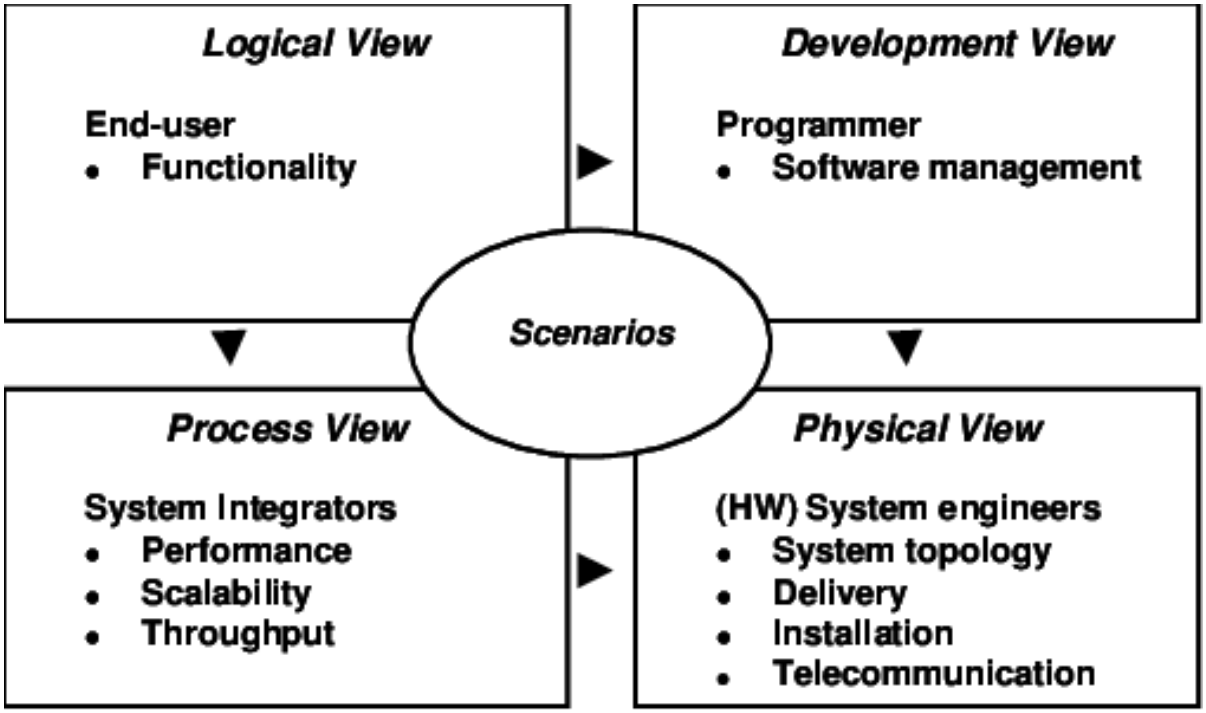
1. Software development cycle

1.1. Analyse

Gedurende de analyse fase ging het voornamelijk om verdieping in verschillende best practices binnen de backend van een web-API. De verschillende gevonden best practices zijn meegenomen in het software ontwerp. Er wordt gebruik gemaakt van entiteiten en DTO's waarin DTO's gebruikt worden om vanuit de frontend de database aan te passen. Hierbij kunnen entiteiten bijvoorbeeld wachtwoorden of andere gevoelige data bevatten en worden deze in de DTO's niet meegenomen. Op dit moment bevatten de DTO's alle data uit de entiteiten omdat de database geen gevoelige data bevat maar mocht dit later wel het geval zijn is deze structuur dus al toegepast.
Verder wordt er gebruik gemaakt van controllers om de verschillende CRUD-operaties uit te kunnen voeren deze controllers maken op hun beurt weer gebruik van repositories voor logica die helpt de database uit te lezen en op basis hiervan return waardes te bepalen. Voor de communicatie met de database vanuit de repositories wordt een aparte databasecontext klassen gebruikt
De repositories maken gebruik van verschillende interfaces om ervoor te zorgen dat de applicatie overzichtelijk en schaalbaar blijft.

1.2. Ontwerp

Voor de onderbouwing van het ontwerp is gebruik gemaakt van het 4 + 1 model. Binnen dit hoofdstuk worden de verschillende views (Logical, Development, Process en Physical) omschreven en onderbouwt gericht op de GameInfoAPI project opbouw.



1.2.1. Logical view

Dit perspectief beschrijft de functionaliteit die het systeem biedt aan de eindgebruikers. Hierin worden de klassen en hun relaties weergegeven. Waaronder de Game klassen die bestaat uit een int Id, string Title, string Description, DateTime ReleaseDate, int AuthorId, author Author, int BestPlayerId, player Bestplayer verder een Author klassen die bestaat uit een int Id, string Name en als laatst een Player klassen met een int Id en string Name.

Primary Keys:

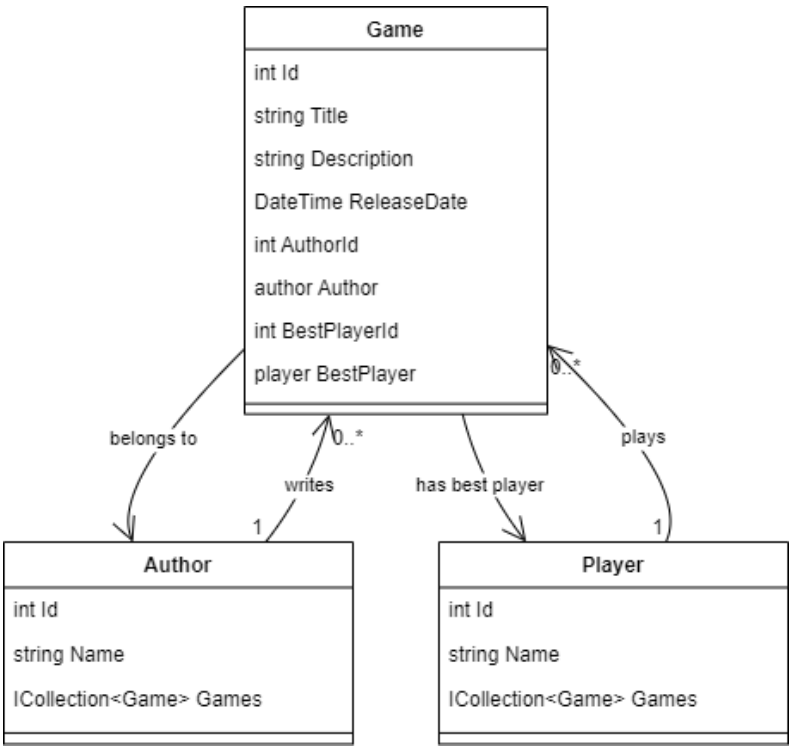
Elke klasse (Game, Author, Player) heeft een primaire sleutel (Id) die ervoor zorgt dat elk record in de respectievelijke tabel uniek is.

Foreign Keys:

AuthorId in de Game klasse is een vreemde sleutel die de relatie legt met de Author klasse, wat betekent dat elke game een specifieke auteur heeft.

BestPlayerId in de Game klasse is een vreemde sleutel die de relatie legt met de Player klasse, wat betekent dat elke game een beste speler heeft.

Deze structuur zorgt ervoor dat de gegevens goed georganiseerd zijn en dat de relaties tussen games, auteurs, en spelers duidelijk en effectief worden beheerd. Het gebruik van primaire en vreemde sleutels zorgt voor data-integriteit



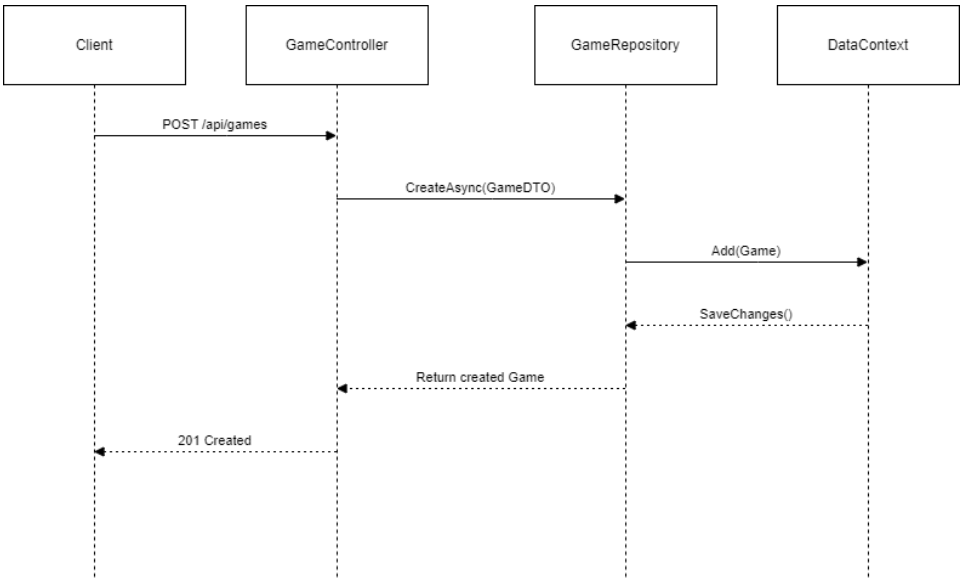
1.2.2. Development view

Dit perspectief beschrijft de organisatie van de software en het project. Hierin wordt de indeling in namespaces en bestanden weergegeven.

```
GameInfoAPI/
├── Controllers/
│   ├── AuthorController.cs
│   ├── GameController.cs
│   └── PlayerController.cs
├── Data/
│   └── DataContext.cs
├── DTOs/
│   ├── AuthorDTO.cs
│   ├── GameDTO.cs
│   └── PlayerDTO.cs
├── Entities/
│   ├── Author.cs
│   ├── Game.cs
│   └── Player.cs
├── Repositories/
│   ├── AuthorRepository.cs
│   ├── GameRepository.cs
│   ├── IAuthorRepository.cs
│   ├── IGameRepository.cs
│   ├── IPlayerRepository.cs
│   └── PlayerRepository.cs
├── Program.cs
└── appsettings.json
```

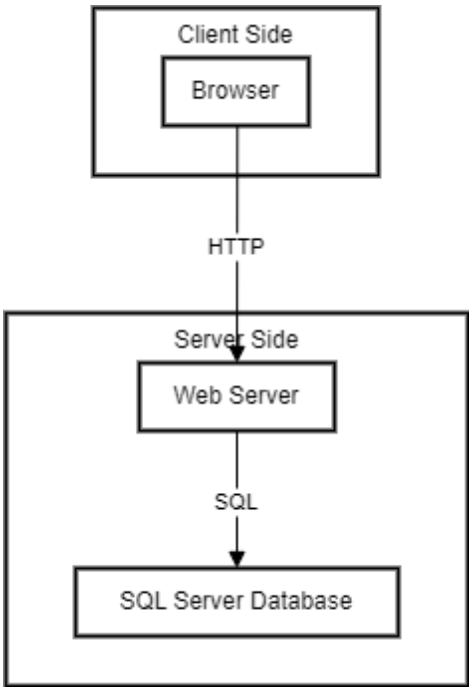
1.2.3. Process view

Dit perspectief beschrijft de communicatie tussen de verschillende delen van het systeem tijdens de uitvoering. Met een sequence diagram wordt hier beschreven hoe een game zal worden aangemaakt.



1.2.4. Physical view

De physical view bevat een hardware gerelateerd overzicht van hoe het systeem tussen de verschillende componenten communiceert. Zo wordt er gebruik gemaakt van een browser om met de frontend te kunnen verbinden. Verder communiceert deze frontend met de applicatie (server) die dan vervolgens aanpassingen doorvoert binnen de database.



1.3. Realisatie

Gedurende de realisatie is er gefocust op het ontwerp zoals beschreven uit te werken, wel zijn er een aantal onderdelen waarvoor in het ontwerp nog geen concreet besluit was genomen aan toegevoegd.
Er is besloten dat alle entiteiten de mogelijkheid krijgen om gebruik te maken van de CRUD-opties Get, GetById, Post, Put en Delete, wat deze operaties inhouden en uitvoeren komt later in deze documentatie aanbod.
Verder zijn de best practices die gedurende de analyse naar voren zijn gekomen nageleefd wat gezorgd heeft voor een overzichtelijk en schaalbaar project waarin correct gebruik word gemaakt van nodige design patterns en frameworks.

1.4. Testen

Binnen de test fase zijn verschillende manieren van testen naar voren gekomen. Zo is als eerst de requirement lijst van brightspace nagelopen om er zeker van te zijn dat het project aan alle eisen voldoet en zijn er specifieke requirements die van tevoren waren opgesteld. Verder zijn unit testen en integratie testen binnen een apart test project toegevoegd om de code te kunnen dekken en testen mocht deze uitgebreid worden. Als laatste zijn een aantal acceptatie testen uitgevoerd om aan te tonen dat de applicatie aan de afgesproken werking voldoet.

1.4.1. Requirements

Functionele Requirements:

CRUD-operaties voor Games:

De applicatie biedt eindgebruikers de mogelijkheid om games aan te maken, te lezen, bij te werken en te verwijderen.

De eindgebruiker moet een beste speler en auteur kunnen koppelen tijdens het aanmaken of bewerken van een game.

CRUD-operaties voor Authors:

De applicatie ondersteunt het lezen en maken van auteurs.

CRUD-operaties voor Players:

De applicatie ondersteunt het lezen en maken van spelers.

Relaties tussen Entiteiten:

De applicatie beheert relaties tussen entiteiten, zoals de relatie tussen een Game en zijn Author, en de relatie tussen een Game en de BestPlayer.

Niet-functionele Requirements:

Veiligheid:

De applicatie maakt gebruik van DTO’s om gevoelige data binnen de database te beschermen.

Prestaties en Schaalbaarheid:

De applicatie is ontworpen om makkelijk aanpasbaar te zijn, de applicatie is overzichtelijk en volgens een logische indeling opgesteld.

Database-interactie:

De applicatie maakt gebruik van Entity Framework (ORM) voor efficiënte database-interactie.

Robuustheid en Foutafhandeling:

De applicatie kan omgaan met foutscenario's en geeft betekenisvolle foutmeldingen terug aan de eindgebruikers.

Testbaarheid:

De applicatie bevat een testproject waarmee functionaliteiten kunnen worden getest om de betrouwbaarheid van de applicatie te blijven garanderen.

1.4.2. Unit test

Voor het unit testen van de applicatie is gekozen om deze te focussen op de repositories. Binnen de repositories wordt op basis van data vanuit de database een return waarde gecreëerd. Op basis van deze return waardes kan de controller dan een beslissing maken over de input die vanuit de frontend binnenkomt. Dit betekent dus dat het grootste gedeelte van de logica en behandelen van data binnen deze repositories gebeurt dus door deze in te dekken met unit testen de core van de applicatie volledig gedekt is.

1.4.3. Integratie test

Voor de Integratie testen is gekozen om de front-end en controllers na te bootsen door de DTO's met test data te vullen deze vervolgens langs de front-end naar de database te communiceren en aan de hand van de front-end response te kijken of deze connectie correct werkt en de response overeenkomt met de verwachting.

1.4.4. Acceptatie test

Test Scenario 1: Speler aanmaken

Testgeval: Een gebruiker wil een nieuwe speler aanmaken.

Stappen:

Verstuur een POST-verzoek naar /api/players met de gegevens van de speler.

Controleer de statuscode van de respons (201 Created).

Controleer of de respons de juiste gegevens van de speler bevat.

Verstuur een GET-verzoek naar /api/players/{id} om te controleren of de speler is toegevoegd.

Verwacht Resultaat:

Statuscode is 201 Created.

Respons bevat de juiste gegevens van de speler.

De GET-verzoek geeft de toegevoegde speler terug.

Test Scenario 2: Spel aanmaken en beste speler instellen

Testgeval: Een gebruiker wil een nieuw spel aanmaken en een beste speler instellen.

Stappen:

Verstuur een POST-verzoek naar /api/games met de gegevens van het spel en de ingevulde playerid van de beste speler.

Controleer de statuscode van de respons (201 Created).

Controleer de statuscode van de respons (200 OK).

Verstuur een GET-verzoek naar /api/games/{id} om te controleren of de beste speler is ingesteld.

Verwacht Resultaat:

Statuscode van het POST-verzoek is 201 Created.

Statuscode van het PUT-verzoek is 200 OK.

De GET-verzoek geeft het spel terug met de ingestelde best player-id & name.

Test Scenario 3: Auteur aanmaken en spel toewijzen

Testgeval: Een gebruiker wil een nieuwe auteur aanmaken en een spel aan deze auteur toewijzen.

Stappen:

Verstuur een POST-verzoek naar /api/authors met de gegevens van de auteur.

Controleer de statuscode van de respons (201 Created).

Verstuur een PUT-verzoek naar /api/games/{gameId} om de auteur aan een spel toe te wijzen doormiddel van de author-id.

Controleer de statuscode van de respons (200 OK).

Verstuur een GET-verzoek naar /api/games/{gameId} om te controleren of de auteur aan het spel is toegevoegd.

Verwacht Resultaat:

Statuscode van het POST-verzoek is 201 Created.

Statuscode van het PUT-verzoek is 200 OK.

De GET-verzoek geeft het spel terug met het toegewezen author-id & name.

2. Frameworks & Design patterns

Binnen het project worden het Repository Pattern en het Data Transfer Object Pattern gebruikt om de applicatie gestructureerd, schaalbaar en onderhoudbaar te houden.

Repository Pattern

Het Repository Pattern wordt toegepast om de toegang tot de onderliggende database gescheiden en geordend te houden. In plaats van directe interactie met de database, werkt de applicatie met repositories die de logica voor gegevensopslag en -opvraging beheren. Dit zorgt ervoor dat de business logica gescheiden blijft van de datalaag, waardoor de code beter testbaar en onderhoudbaar wordt. Voor elk van de entiteiten, zoals Game, Author en Player, is er een repository. Deze repositories bieden methoden om gegevens op te halen, zoals GetById, GetAll, en methoden voor het toevoegen, bijwerken en verwijderen van data.

DTO Pattern

Het Data Transfer Object (DTO) Pattern wordt in het project gebruikt om de data die tussen de applicatie en database worden uitgewisseld, te structureren en te optimaliseren. DTO's zijn eenvoudige objecten die geen logica bevatten, maar alleen data. Ze worden gebruikt om alleen de benodigde gegevens over te dragen tussen de API en de repositories. Dit zorgt voor extra veiligheid omdat op deze manier gegevens zoals bijvoorbeeld wachtwoorden niet zomaar worden overgestuurd.

Door deze patterns te implementeren, blijft de code schoon en gestructureerd, en wordt het eenvoudiger om wijzigingen door te voeren zonder grote delen van de applicatie te hoeven herschrijven.

3. API Crud Operations

GET - Alle resources ophalen

De GET-operatie wordt gebruikt om een lijst van alle data voor een specifieke entiteit op te halen. Binnen de GameInfoAPI zou bijvoorbeeld met GET een lijst van alle games in het systeem opgehaald kunnen worden. Dit is een "read"-operatie in het CRUD-principe en maakt het mogelijk voor clients om alle beschikbare gegevens te bekijken.

GET by ID – Specifieke data ophalen

De GET by ID-operatie biedt de mogelijkheid om specifieke data op te halen aan de hand van een uniek ID. Bijvoorbeeld, een GET-aanroep naar /games/{id} retourneert de details van een specifieke game met dat unieke ID. Dit maakt het mogelijk om specifieke informatie te verkrijgen zonder een volledige lijst van alle data binnen deze entiteit op te vragen.

POST – Nieuwe data creëren

De POST-operatie wordt gebruikt om een nieuwe data in het systeem te creëren. Wanneer een client een POST-aanroep doet naar bijvoorbeeld /games met de nodige informatie (zoals de titel, beschrijving en releasedate van de game), zal de API een nieuwe game aanmaken en deze opslaan in de database. Dit valt onder de "create" in CRUD en zorgt ervoor dat nieuwe gegevens via een ap call in de database belanden.

PUT - Bestaande data bijwerken

De PUT-operatie wordt gebruikt om bestaande data bij te werken. Als een client een PUT-aanroep doet naar /games/{id}, met bijgewerkte gegevens voor een specifieke game, zal de API deze gegevens gebruiken om de bestaande data in de database te vervangen. Dit is de "update" functionaliteit binnen CRUD, waarmee je gegevens kunt aanpassen zonder nieuwe objecten te hoeven creëren.

DELETE - Data verwijderen

De DELETE-operatie wordt gebruikt om data uit het systeem te verwijderen. Bijvoorbeeld, een DELETE-aanroep naar /games/{id} zal de game met het opgegeven ID verwijderen uit de database. Dit is de "delete" functionaliteit van CRUD en zorgt ervoor dat ongewenste data in de database permanent kan worden verwijderd.

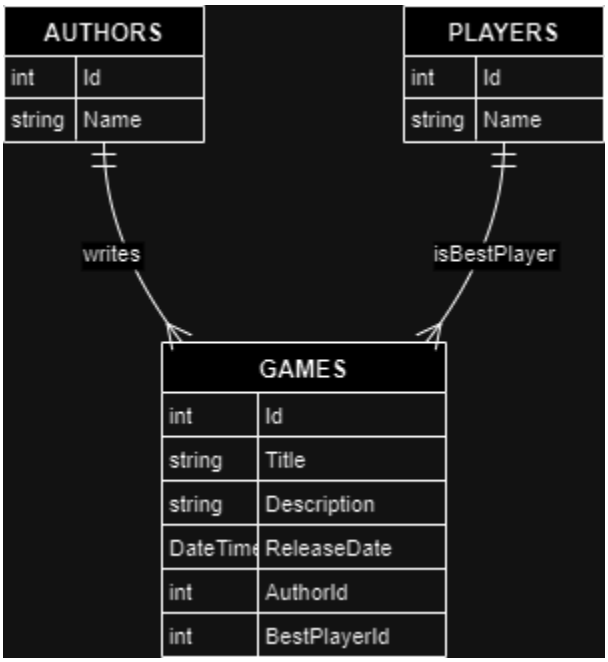
4. Database & ERD

De database van de GameInfoAPI kan informatie over games, auteurs, en spelers beheren. De structuur is relationeel, waarbij de gegevens in verschillende tabellen worden opgeslagen die onderling verbonden zijn door primaire en vreemde sleutels.

Games: De Games-tabel slaat informatie op over verschillende games, zoals de titel, beschrijving, releasedate, en gerelateerde auteurs en beste spelers. De primaire sleutel van deze tabel is Id.

Authors: De Authors-tabel bevat gegevens over de auteurs van games, met Id als primaire sleutel. Deze tabel is gerelateerd aan de Games tabel via een vreemde sleutel namelijk AuthorId.

Players: De Players-tabel houdt informatie bij over spelers, met Id als primaire sleutel. Deze tabel is ook gerelateerd aan de Games tabel via een vreemde sleutel namelijk BestPlayerId.



5. Github repository

Het volledige project is terug te vinden op GitHub in een public repository. Deze repository bestaat uit het GameInfoAPI project, het GameInfoAPI.Test project, een docketfile waarmee het project in een container gerunt kan worden, een read me file waarin de repository en het project worden uitgelegd en een main.yml file waarin gebruik wordt gemaakt van GitHubActions.

5.1. Repository

De repository ziet er als volgt uit en kan gevonden worden met de bijbehorende link onder deze afbeelding.

https://github.com/JornNeijssen/GameInfoApi

5.2. Read me

Binnen de repository is een read me file opgesteld die zowel een uitleg van het project als een uitleg van het clone proces beschrijft.

5.3. Docker

De repository bevat ook een docker file door middel van deze docker file kan een container image van de GameInfoAPI worden gerunt. Deze docker file wordt ook gebruikt binnen de ci cd pipeline (GitHubActions) om het project naar een repository op dockerhub te pushen, dit gebeurt alleen wanneer het project geen errors bevat en correct gerunt kan worden.

5.4. Ci cd pipeline

Er wordt gebruik gemaakt van een ci cd pipline door middel van GitHubActions. Dit is terug te vinden in de workflow file Main.yml. Wanneer er een nieuwe push naar de master branch plaats vindt wordt deze file gerunt. Binnen deze file staat de logica om in te kunnen loggen met secrets op een dockerhub repository waar het project ook is terug te vinden. Deze secrets zijn opgezet in github zelf en zorgen ervoor dat de token die gebruikt wordt om het project op de dockerhub repository te kunnen update niet als plain tekst in de main.yml file staat en dus niet kan lekken.