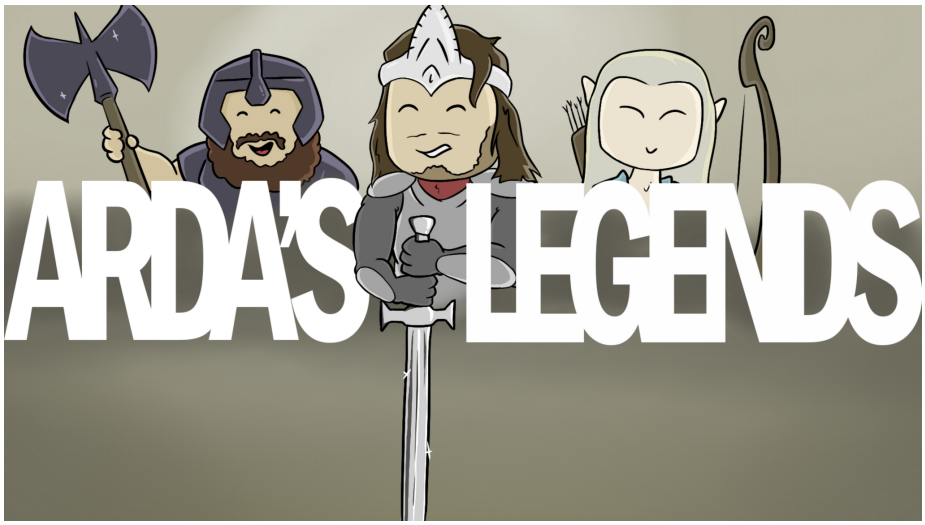
Dokumentation: Semesterbegleitleistung Datenbanken 2

Ardas Legends Minecraft Server - Roleplaying System

Luis Brinkmoeller und Moritz Rohleder



Fachhochschule Dortmund
Deutschland
23.06.2023

Inhaltsverzeichnis

1 Aufgabe 1c: Anwendungsszenario

Unser Anwendungsszenario ist eine Datenbank für einen Minecraft Server mit Rollenspiel Aspekten. Auf dem Server ist es möglich einen Rollenspiel Charakter (RP-Char) zu steuern, einem Volk beizutreten, Regionen zu erobern und Kriege zu führen.[1]

Zu einem Spieler werden die DiscordID sowie der In-Game-Name (IGN) und die Unique User ID (UUID) von Minecraft gespeichert.
Die Spieler können einem Volk beitreten, zu welchem eine ID als Primärschlüssel, ein Name und einen Völker spezifischer Buff gespeichert werden. Außerdem werden ein Hexa-dezimaler Farbwert und die In-Game benutzte Gesinnung gespeichert. Die Wahl des Volkes kann jederzeit geändert werden, wobei man maximal einem Volk angehören kann.
Die Völker können sich verbünden.

Der Spieler kann einen RP-Char erstellen und steuern. Zu diesem Charakter wird eine ID, ein Name, ein Titel und die Option, ob er PvP[2] oder PvE[3] Spielstil möchte, gespeichert.
Desweiteren gehört auch immer Ausrüstung dazu. Zu einem Ausrüstungsgegenstand gehören die ID, der Namen und die Haltbarkeit. Er kann entweder aus einer Waffe, mit Waffentyp und Schaden, oder einem Rüstungsteil, mit Rüstungstyp und Schutz, bestehen.
Ein RP-Char kann durch die Welt reisen, dementsprechend wird immer die aktuelle Position des RP-Chars in Form der Region gespeichert.
Ein Spieler, der einen RP-Char hat kann der Anführer eines Volks werden.
Ein Volk kann nur einen Anführer gleichzeitig haben.

Spieler eines Volks können in den Regionen von Mittelerde Standorte, wie Dörfer, Städte, Burgen und Festungen bauen.
Eine Region ist ein Teil von Mittelerde und wird in der Datenbank mit einer ID, einem Namen, einem Typ (Wald, Ebene, Berge, etc.) und allen benachbarten Regionen gespeichert.
Jedes Volk startet in einer festen Region, wo ein kleines Dorf steht.
Bauen einer oder mehrere Spieler einen Standort im Spiel, können Sie bei den Verwaltern des Servers beantragen, dass dieser die Region für ihr Volk kontrolliert.

Ein Standort hat eine ID und bekommt von den Erbauern einen Namen. Je nachdem, wie dieser ist, wird der Typ bestimmt (Hamlet, Village, Town, Capital, Keep, Castle, Stronghold) und in der Datenbank gespeichert. Ebenso werden dessen Koordinaten aus der Welt von Mittelerde gespeichert.

[1] Mehr Informationen zu dem System, dass auf dem Server benutzt wird, können Sie der PowerPoint entnehmen. [1]
[2] Player vs Player
[3] Player vs Entity

An einem Standort können bestimmte Gebäude, wie ein Lazarett, ein Hafen oder eine Botschaft stehen. Ist das der Fall, werden Funktionen, wie die Schiffsreise, die Heilung von Truppen und Allianzen, möglich. Diese Gebäude werden in einer eigenen Entität dargestellt und die Funktion des Gebäudes in dem Claimbuild wird mit der Beziehung gespeichert.

Neben den bestimmten Gebäude kann ein Standort auch Produktionsstätten haben, welche diesen mit Ressourcen versorgen. Zu einer Produktionsstätte werden einmal eine ID und der Typ (Farm, Schlachthaus, Fischerhütte, Holzfäller, Lager, Steinbruch, Mine, etc.) gespeichert. Ist eine Produktionsstätte vorhanden, kann man sich je nach dem Typ aussuchen, welche Ressourcen man haben möchte. Abhängig von der Ressource wird die Anzahl festgelegt und mit dieser an der Beziehung gespeichert.

Aussicht:

Jedes Volk kann mehrere Truppen haben, diese können militärischer Funktion sein (Army) oder zivile Funktionen abdecken (Trader Company). Zu einer Truppe gibt es eine ID und einen Namen.
Jeder Trupp (unabhängig der Funktion) hat einen Ursprungs-Standort. Bei den Zivilisten, kann dies eine Stadt (Town) oder eine Hauptstadt (Capital) sein. Bei den militärischen kann das eine Burg (Castle), eine Festung (Stronghold), eine Stadt (Town) oder eine Hauptstadt (Capital) sein. Jeder Standort kann nur eine Truppe von jedem Typ hervorbringen. Ausgenommen ist die Hauptstadt, welchen zwei militärische Truppen hervorbringen kann.
Eine Truppe kann an einem Standort stationiert oder in der Welt unterwegs sein. Dementsprechend wird die aktuelle Position (Region) gespeichert und wenn die Truppe stationiert ist, auch der Standort.
Um außerhalb der Ländereien des Volkes zu reisen, muss ein Trupp von einem RP-Char kontrolliert werden.

Eine militärische Truppe hat zusätzliche Informationen, die gespeichert werden müssen. Zum einen besteht ein militärischer Trupp aus Soldaten. Zu jedem Soldatentypen, den es im Spiel gibt, wird ein Bezeichner, die Ausrüstung und Token-Kosten gespeichert. Zum anderen hat eine Armee nur eine begrenzte Zahl an Token zur Verfügung.

[3]Folgende Entitäten sind nicht implementiert gehören aber eigentlich auch dazu, der Vollständigkeit halber aber Dokumentiert hier.

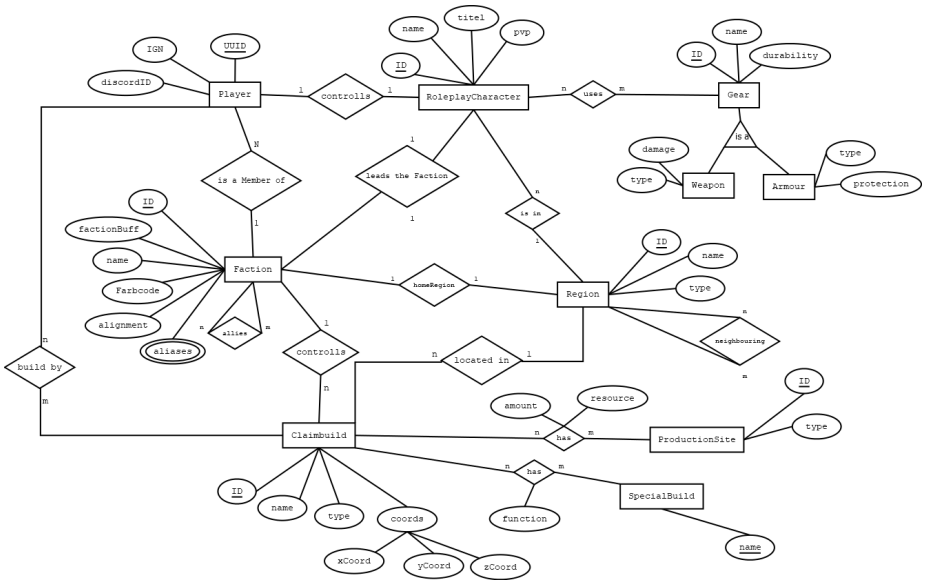


Figure 1: EER-Diagramm

2 Aufgabe 2a: Dokumentation des Hibernate Projekts

2.1 Textuelle Erläuterung des OR-Mappings

Für das OR-Mapping benutzt man mit Hibernate und JPA die *@Entität* Annotation an Klassen, welche auf die Datenbank gemapped werden sollen.
Eine Datenbank-Tabelle braucht auch einen Primärschlüssel, der wird mit der *@Id* Annotation markiert wird. Soll der Primärschlüssel automatisch generiert werden, kann die Annotation *@GeneratedValue* genutzen werden.
Es ist im Bereich des möglichen, Enums in der Datenbank zu speichern, dazu wird die *@Enumerated* Annotation benutzt um die Art fest zu legen, wie dies zu geschehen hat.
Beziehungen werden mit den Annotationen *@OneToOne*, *@OneToMany* und *@ManyToMany* markiert.
Im Folgenden geben wir Beispiele aus unserem Projekt [2]

```
/*
 * Hier stehen die Imports
 */
@Entity
public class ProdSite {
    @Id
    @GeneratedValue
```

```
    private int id;
    @Enumerated(EnumType.STRING)
    private ProductionSiteType type;
    @OneToMany(mappedBy = "productionSite")
    private Set<UsedProductionSite> usingClaimbuilds;
    /*
     * Hier stehen Konstruktoren, Getter und Setter
     */
}
```

2.1.1 Strukturiertes Attribut

Für das Strukturierte Attribut haben wir die *@Embaddable* Annotation an der Klasse CoordinatesEmbed benutzt.
Mit der Embaddable sagen wir, dass diese Klasse in einer anderen Entität eingebettet wird.

```
@Embeddable
public class CoordinatesEmbed {
    private double x;
    private double y;
    private double z;

    //Hier folgen Konstruktoren, Getter und Setter
}
```

In der Klasse Claimbuild nutzen wir dann die *@Embedded* Annotation um die Klasse CoordinatesEmbed einzubetten.
Die Annotation *@AttributeOverrides* überschreibt die Namen der Spalten in der Entität, wo sie eingebettet werden.

```
@Entity
public class Claimbuild {
    /*
     * Hier sind die "primitiven" Attribute der Klasse
     * Claimbuild
     */
    @Embedded
    /*@AttributeOverrides(
            @AttributeOverride(name = "x", column = @Column(name = "cbXCoord")),
            @AttributeOverride(name = "y", column = @Column(name = "cbYCoord")),
            @AttributeOverride(name = "z", column = @Column(name = "cbZCoord"))
    )*/
    private CoordinatesEmbed coords;
```

```
    /*
     * Hier folgt der Rest der Klasse Claimbuild
     */
}
```

2.1.2 Mehrwertiges Attribut

Unser mehrwertiges Attribut sind die Alias eines Volks (Faction).
Für die Modellierung dieser haben wir die Annotation *@ElementCollection()* an einem Set aus Strings verwendet.
Durch diese Annotation wird im Hintergrund eine extra Tabelle von Hibernate erzeugt.
Das bringt zum einen den Vorteil, dass die Anzahl der Alias theoretisch unbegrenzt ist, als auch deren Länge deutlich länger ausfallen kann, als bei der Alternative.
Die Alternative wäre mit einem Converter zu arbeiten, der alle Einträge der Liste in eine einzige Tabellen-Spalte schreibt (*@Convert* Annotation). Bei dieser Alternative, wären Anzahl und Länge der Aliasse durch die maximal Länge der Spalte begrenzt sind.

```
/*
 * Hier stehen die Imports
 */
public class Faction {
    @Id
    @GeneratedValue
    private int id;
    @Column(name = "faction", nullable = false)
    private String name;
    @Column(name = "buff", nullable = false)
    private String buff;
    @Column(name = "color", nullable = false)
    private String colorCode;
    @Column(name = "alignment", nullable = false)
    @Enumerated(EnumType.STRING)
    private Alignment alignment;
    @ElementCollection()
    private Set<String> aliases = new HashSet<>();
    @OneToOne
    @JoinColumn(name = "regionumber", referencedColumnName = "regionnumber")
    private Region homeRegion;

    @OneToMany(mappedBy = "faction")
    private Set<Player> players;
```

```
    @OneToMany(mappedBy = "ally")
    private Set<Faction> allies;

    @OneToMany(mappedBy = "controllingFaction")
    private Set<Claimbuild> claimbuilds;

    @OneToOne
    @JoinColumn(name = "leaderChar_id", referencedColumnName = "id")
    private RPChar leader;

    public Faction(){

    }

    public Faction(String name, String buff, String colorCode, Alignment alignment){
        this.name = name;
        this.buff = buff;
        this.colorCode = colorCode;
        this.alignment = alignment;
    }
    /*
     * Hier folgen Getter und Setter
     */
}
```

2.1.3 Vererbungsbeziehung

Die Vererbungsbeziehung zwischen den Klassen Weapon, Armour und Gear haben wir mit der Annotation *@MappedSuperclass* dargestellt, da die Klasse Gear abstract ist.

```
/*
 * Hier stehen die imports
 */
@MappedSuperclass
public abstract class Gear {
    @Id
    @GeneratedValue
    private int id;
    private String name;
    private int durability;

    public Gear() {
    }
```

```
    public Gear(String name, int durability) {
        this.name = name;
        this.durability = durability;
    }
    /*
     * Hier folgen Getter und Setter, etc.
     */
}
```

Mit der Annotation taucht Gear gar nicht in der Datenbank auf, sondern nur Armour und Weapon mit den Attributen von Gear. Da von Gear auch keine Objekte erstellt werden sollen, sondern Gear immer Armour oder Weapon ist, passt das genau in unser Anwendungsszenario.

```
/*
 * Hier stehen die imports
 */
@Entity(name = "Weapon")
public class Weapon extends Gear{
    private String type;
    private double dmg;

    @ManyToMany(mappedBy = "weapons")
    private List<RPChar> rpchars = new ArrayList<>();

    public Weapon() {
        super();
    }

    public Weapon(String name, int durability, String type, double dmg) {
        super(name, durability);
        this.type = type;
        this.dmg = dmg;
    }
    /*
     * Hier folgen Getter und Setter, etc.
     */
}
```

In den Klassen Weapon und Gear benutzen wir die *@Entity* Annotation und erben Gear. Das ist alles was wir für des Mappen der Vererbungsbeziehung benötigen.

```
/*
 * Hier stehen die imports
```

```java
 */
@Entity(name = "Armour")
public class Armour extends Gear{
    private String type;
    private double protection;

    @ManyToMany(mappedBy = "armours")
    private List<RPChar> rpchar = new ArrayList<>();
    public Armour() {
        super();
    }

    public Armour(String name, int durability, String type, double protection) {
        super(name, durability);
        this.type = type;
        this.protection = protection;
    }
    /*
     * Hier folgen Getter und Setter, etc.
     */
}
```

3 Aufgabe 3: OR Datenbankmodell

```
/*
 * Coordinates_Typ
 * speichert x-, y- unt z-Koordinaten
 */
CREATE TYPE Coordinates_Type AS OBJECT (
    xCoord  FLOAT,
    yCoord  FLOAT,
    zCoord FLOAT
);
/
/*
 * Claimbuild_Typ
 * speichert ein Claimbuild mit dessen ID, Namen, Typ und Koordinaten
 */
CREATE TYPE Claimbuild_Type AS OBJECT (
    cbID INT,
    cbName VARCHAR(50),
    cbType varchar(50),
    coordinates Coordinates_Type
);
/
/*
 * Aliases_Typ
 * speichert eine ID und den AliasNamen, sprich das Alias,in einem eigenen Typ.
 * wir nutzen einen eignene Typ,
 * da wir keine Obergrenze haben und somit eine Nested-Table nutzen.
 */
 CREATE TYPE Aliases_Type AS OBJECT (
    aliasID INT,
    aliasName VARCHAR(50)
);
/
CREATE TYPE Aliases_NT AS TABLE OF REF Aliases_Type;
/*
 * Faction_Type
 * speichert eine Faction mit der ID, dem Namen, dem Buff,
 * der Farbe, das In-Game-Alignment und den Alias-Namen.
 */
CREATE TYPE Faction_Type AS OBJECT (
    FacID INT,
    FacName VARCHAR(50),
    FacBuff VARCHAR(200),
    FacColor VARCHAR(7),
    Alignment VARCHAR(50),
```

```
    Aliases Aliases_NT
);
/

CREATE TYPE Gear_Type AS OBJECT
(
    gearID INT,
    gearName VARCHAR(50),
    durability FLOAT
)NOT FINAL;
/
CREATE TYPE Weapon_Type UNDER Gear_Type
(
    weaponType VARCHAR(50),
    damage FLOAT
);
/
CREATE TYPE Armour_Type UNDER Gear_Type
(
    armourType VARCHAR(50),
    armourProt FLOAT
);
/
/*
 * ============================
 * Tabellen deklaration ab hier
 * ============================
 */

CREATE TABLE Factions OF Faction_Type(
    FacID PRIMARY KEY
)
    NESTED TABLE Aliases STORE AS Aliases_NT_TAB;
/
 CREATE TABLE Claimbuilds OF Claimbuild_Type;
/
CREATE TABLE Armour OF Armour_Type;
/
CREATE TABLE Weapon OF Weapon_Type;
```

References

[1] Moritz Rohleder und Luis Brinkmöller. *LotR MC-Server System*. URL: https://docs.google.com/presentation/d/15pT1os93NsZp5_2S2r89S5uVOg2PpnIu/edit?usp=drive_link&ouid=101991561915549422326&rtpof=true&sd=true.

[2] Luis Brinkmöller und Moritz Rohleder. *AL System DB2*. URL: https://github.com/MoritzRohleder/AL_Systems_DB2.