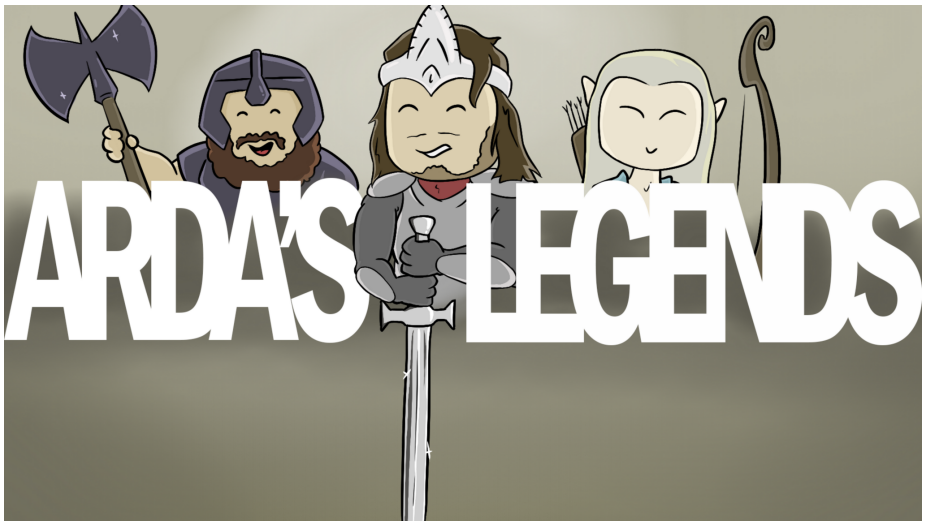
Dokumentation: Semesterbegleitleistung Datenbanken 2

Ardas Legends Minecraft Server - Roleplaying System

Luis Brinkmoeller und Moritz Rohleder



Fachhochschule Dortmund
Deutschland
23.06.2023

Contents

1 Aufgabe 1c: Anwendungsszenario

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

Zu einem Spieler werden DiscordID sowie In-Game-Name (IGN) und Unique User ID (UUID) von Minecraft gespeichert.
Spieler können einem Volk beitreten, zu dieser werden eine ID als Primärschlüssel, ein Name, einen Völker spezifischer Buff, ein Hexa-dezimaler Farbwert und die In-Game benutzte Gesinnung. Die Wahl des Volkes kann jederzeit geändert werden, aber man kann immer nur einem oder keinem Volk angehören.
Ein Volk kann sich mit anderen Völkern verbünden.

Ein Spieler kann einen RP-Char erstellen und steuern. Zu einem RP-Char wird eine ID, ein Name, ein Titel und die Option, ob er PvP oder PvE Spielstil möchte gespeichert.
Zu einem RP-Char gehört immer auch Ausrüstung. Ein Ausrüstungsgegenstand hat einmal eine ID, einen Namen, Haltbarkeit und kann entweder eine Waffe, mit Waffentyp und Schaden, oder ein Rüstungsteil, mit Rüstungstyp und Schutz, sein.
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 Region, wo ein kleines Dörfchen schon steht, die sind fest.
Bauen einer oder mehrere Spieler einen Standort im Spiel, können Sie bei den Verwaltern des Servers beantragen, dass dieser Standort die Region für ihr Volk kontrolliert.

Ein Standort hat eine ID und bekommt von den Erbauern einen Namen. Je nachdem, wie der Standort gebaut ist, wird der Typ des Standorts bestimmt (Hamlet, Village, Town, Capital, Keep, Castle, Stronghold) und in der Datenbank gespeichert, genauso wie die Koordinaten, wo der Standort im Spiel gebaut ist.
An einem Standort können bestimmte Gebäude, wie ein Lazarett, ein Hafen

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

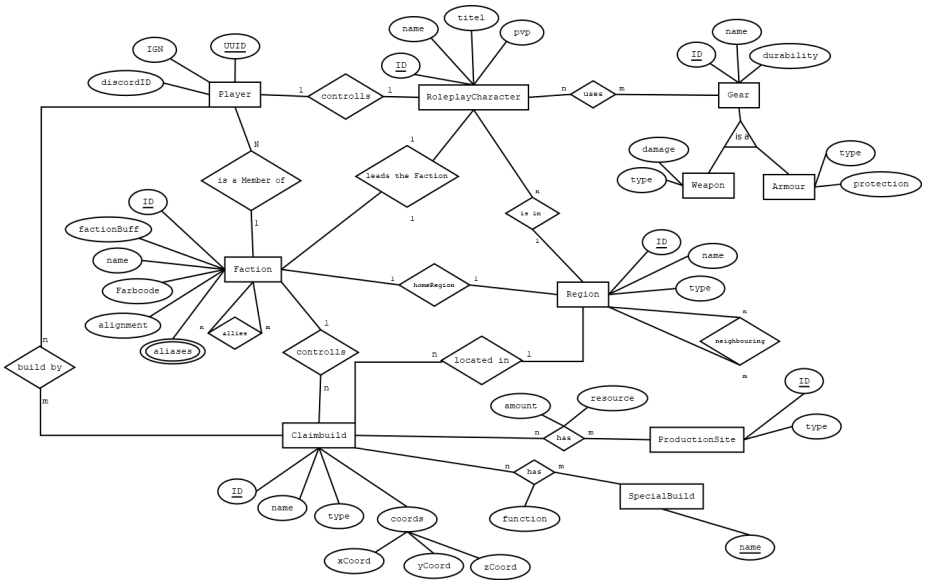


Figure 1: EER-Diagramm

oder eine Botschaft sein. Steht ein solches Gebäude an dem Standort werden Funktion, wie Schiffsreise, 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. Hat man eine Produktionsstätte kann man sich je nach dem Typ aussuchen, welche Resourcen man haben möchte und abhängig von den gewählten Resourcen wird dann die Anzahl festgelegt. Das wird 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

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

Stadt (Town) oder eine Hauptstadt (Capital) sein. Jeder Standort kann nur eine Truppe von jedem Typ hervorbringen, ausgenommen die Hauptstadt, die kann zwei militärische Truppen hervorbringen.
Eine Truppe kann an einem Standort stationiert sein, 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. Eine Armee hat nur eine begrenzte Anzahl an Token zur Verfügung.

2 Aufgabe 2a: Doku 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. Möchte man, dass der Primärschlüssel automatisch generiert werden kann man die Annotation *@GeneratedValue* nutzen.
Möchte man Enums in der Datenbank speichern, kann man mit der *@Enumerated* Annotation eine Art festlegen, wie das geschehen sollte. Wir haben uns dafür entschieden, die Enums als Strings in der Datenbank zu hinterlegen.
Beziehungen werden mit den Annotationen *@OneToOne*, *@OneToMany* und *@ManyToMany* markiert.

```
/*
 * 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 mehrwerties Attribut sind die Aliases 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 Aliases theoretisch unbegrenzt ist, als auch deren Länge deutlich länger ausfallen kann, als bei der aliternative.
Die Alternative wäre mit einem Converter zu arbeiten, der alle Einträge der Liste in eine einzige Tabellen-Spalte spreibt (*@Convert* Annotation). Bei dieser Alternative, wären Anzahl und Länge der der Aliases 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 wir von Gear auch keine Objekte erstellt werden sollen, sondern Gear immer Armour oder Weapon ist, ist das auch genau das was wir wollen.

```
/*
 * 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
 */
@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

Test Test

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.