

RUHR-UNIVERSITÄT BOCHUM

# WEB-ENGINEERING

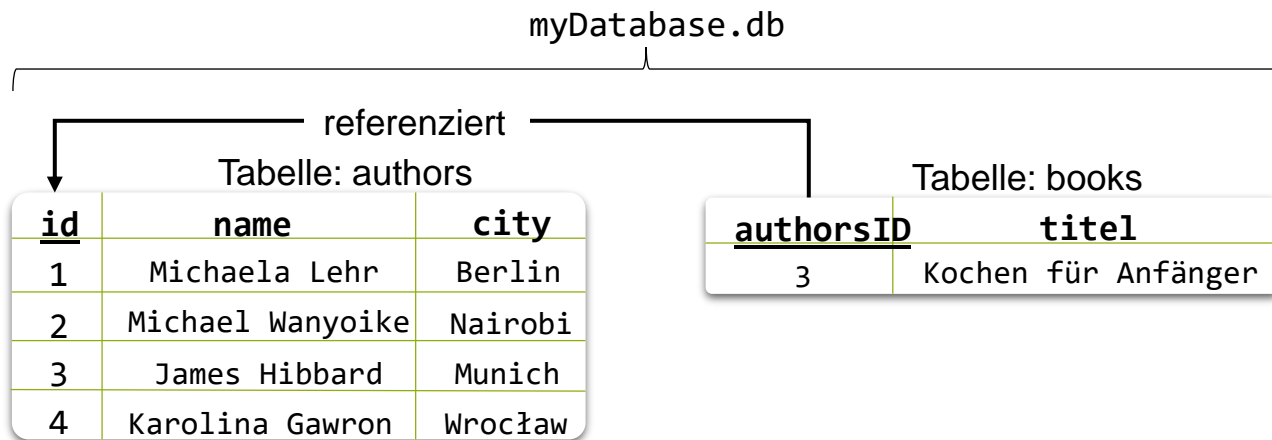
Sommersemester 2023



Informatik  
im Bauwesen

# Wiederholung zu Datenbanken

- DBs dienen als `persistenter Datenspeicher`
- Daten in DBs werden in `Relation` zueinander gesetzt  
(Relationale Datenbank)
- Ein Datenbanksystem **oder** DBs **gewährleistet die** Vollständigkeit, Richtigkeit **und** Verfügbarkeit **von Daten.**



# Wiederholung zu Datenbanken

- SQL ist eine typisierte Abfragesprache für Datenbanken
- Treiber ermöglichen die Kommunikation mit spezifischen Datenbanken
- Mittels SQL Sprachen werden `queries` verschickt mit denen Datensätze erweitert, gelöscht, abgefragt **oder** manipuliert werden können
- Zusammenfassung kennengelernter Befehle:

`CREATE TABLE, PRIMARY KEY, INSERT INTO, SELECT, DELETE`

```
const mysql = require('mysql');
const con = mysql.createConnection({ ... });

con.query('SELECT * FROM mydb.authors', (err, rows) => {
    //DO SOMETHING HERE

});

con.end();
```

# Herausforderung im Umgang mit Datenbanken

## Interaktion mit SQL über mit Datenbank-Treiber schwierig

- SQL Queries werden als String kommuniziert, was fehleranfällig ist und präzise Syntax-Kenntnisse erfordert

## Ausgeführte SQL Queries verändern die Datenbank nachhaltig

- Änderungen werden nicht nachverfolgt
- Keine Versionierung und Einsicht darüber welcher Nutzer wie die Datenbank verändert hat

# Impedance Mismatch

## Objektifizierung von Datenbanken

- Relationale Datenbanken basieren auf mathematische Grundlagen der relationalen Algebra
- Ein Objekt kapselt Zustände und Verhalten, wodurch diese eine eigenständige Identität erhalten
- Dieser Widerspruch wurde als `Impedance Mismatch` bekannt

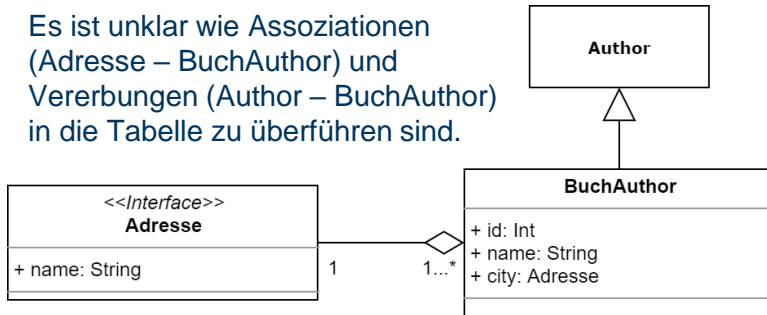
Im wesentlichen können folgende Konfliktpunkte herausgestellt werden:

- Struktur → Eine Klasse in objektorientierter Programmierung (OOP) kann Teil einer Klassenhierarchie sein, was in einem relationalen Modell nicht existiert und daher schwer zu überführen ist.

# Impedance Mismatch

## Objekt-Modell

Es ist unklar wie Assoziationen (Adresse – BuchAuthor) und Vererbungen (Author – BuchAuthor) in die Tabelle zu überführen sind.



## relationales Modell

<u>id</u>	name	city
1	Michaela Lehr	Berlin
2	Michael Wanyoike	Nairobi
3	James Hibbard	Berlin
4	Karolina Gawron	Wrocław

# Impedance Mismatch

Im wesentlichen können folgende Konfliktpunkte herausgestellt werden:

- Identität → In einem relationalen Modell wird die Identität durch den Datensatz (insb. Primärschlüssel) ausgezeichnet. In OOP verfügt jedes Objekt durch Instanziierung über eine Eigene Identität (Objekt-Referenz). Ein Vergleich von Daten funktioniert daher grundlegend unterschiedlich.

## Objekt-Modell

objA  
(idRef=1)



objB  
(idRef=3)



Identität: `objA==objB` (false)

Übereinstimmung: `objA.equals(objB)` (true)

## relationales Modell

<u>id</u>	name	city
1	Michaela Lehr	Berlin
2	Michael Wanyoike	Nairobi
3	James Hibbard	Berlin
4	Karolina Gawron	Wrocław

# Impedance Mismatch

Im wesentlichen können folgende Konfliktpunkte herausgestellt werden:

- Datenkapselung → In OOP wird die Veränderung von Inhalt eines Objekts über Methoden begrenzt. Bei einem relationalen Modell existiert solch ein Schutzmechanismus standardmäßig nicht. Die Einträge einer Datenbank werden meist direkt aktualisiert.

## Objekt-Modell

Geldbeutel
-geld:float -inhaber:String
+einzahlen(betrag:float):boolean +auszahlen(betrag:float):boolean +getInhaber():String

```
michaelaObj.einzahlen(106.00);  
//Summe 230.50 Euro
```

## relationales Modell

Tabelle: Geldbeutel

<u>id</u>	inhaber	geld
1	Michaela Lehr	124.50
2	Michael Wanyoike	2350.25

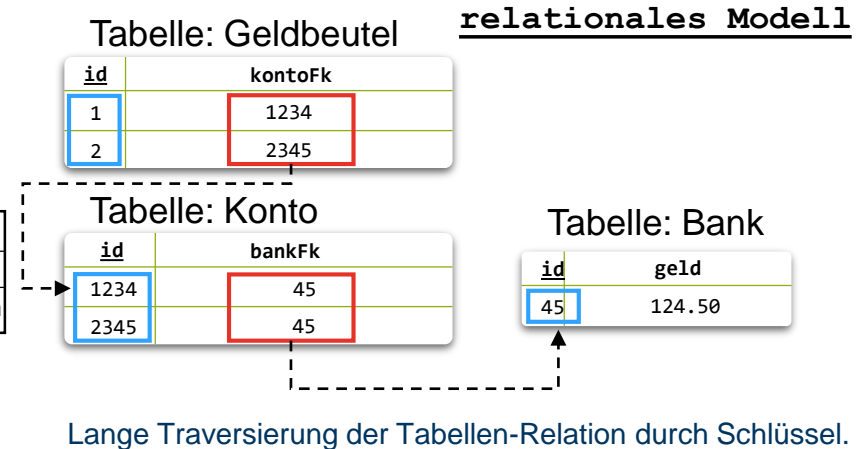
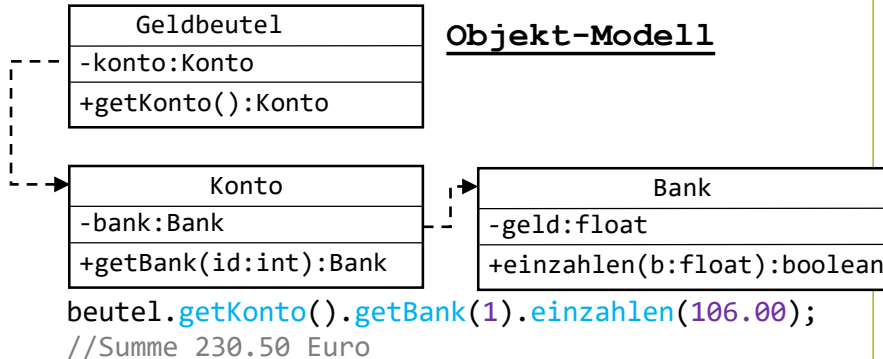
```
UPDATE Geldbeutel SET geld=230.50 WHERE  
id = 1;
```



# Impedance Mismatch

Im wesentlichen können folgende Konfliktpunkte herausgestellt werden:

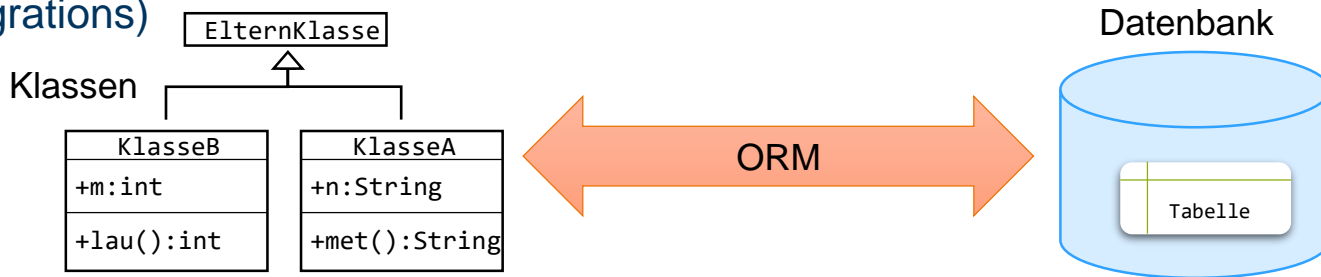
- Arbeitsweise → Um ein Wert in OOP zu aktualisieren wird eine Verkettung von Objekten bis zur Quelle traversiert. In einer relationalen Datenbank wird dies über den Abgleich von Schlüssel lokalisiert, was eine grundlegend unterschiedliche Arbeitsweise ist.



# Object-Relational Mapper

Ein Object-Relational Mapper (ORM) ist eine Bibliotheken um Objekte in eine Datenbank zu überführen.

- Ist darauf spezialisiert Probleme des `Impedance Mismatch` aufzulösen
- Bildet eine Schicht zwischen Anwendung und Datenbank
- Verfügt über eine modell-basierte Syntax aufbauend auf SQL, durch welche die Interaktion zwischen Programmcode und Datenbank ineinander greift (Sequelize)
- Durch ORM können versionierte und skalierbare Datenbanken erstellt werden (Migrations)



# Web-Engineering

## **Sequelize**

# Sequelize

**Sequelize ist ein Promise-basiertes ORM für Postgres, MySQL, SQLite und Microsoft SQL Server.**

- Sequelize erfordert die Angabe eines Dialekts
- Ein Dialekt ist der Datenbank-Treiber mit der Sequelize sich verbinden und arbeiten soll
- Der Sequelize Code ist unabhängig vom Dialekt und kann beim wechseln von einem zum anderen Datenbanksystem übernommen werden



Kommandozeile

```
C:\Users\...> npm install sequelize  
C:\Users\...> npm install mysql2
```

Weitere Dialekte: `sqlite3`, `mariadb`,  
`tedious` (Microsoft SQL Server), `pg`  
`pg-hstore` (Postgres)

```
const Sequelize = require('sequelize');
```

# Sequelize

## Verbindung zu einer Datenbank aufbauen

- Die Instanziierung eines Sequelize-Objekts stellt die Verbindung zu einer Datenbank her
- Option 1: Übergabe von separaten Parametern

```
const sequelize = new Sequelize('mydb', 'username', 'password', {  
  host: 'localhost',  
    dialect: 'mysql', /* 'mariadb' | 'postgres' | 'mssql' */  
    dialectOptions: { ... } //Für mysql optional  
});
```

- Option 2: Übergabe einer Verbindungs-URI

```
const adresse = 'mysql://user:pass@localhost:3306/mydb';  
const sequelize = new Sequelize(adresse);
```

# Sequelize

## Testen und schließen der erstellten Verbindung

- Die Datenbankverbindung kann mit der `authenticate`-Methode getestet werden:

```
sequelize.authenticate().then(() => {  
  console.log('Connection has been established successfully.');
```

```
}).catch(err => {
```

```
  console.error('Unable to connect to the database:', err);
```

```
});
```

- Sequelize hält standardmäßig die Datenbankverbindung offen und wickelt alle Anfragen (queries) über diese ab
- Die Verbindung kann durch die `close`-Methode manuell getrennt werden. Geschieht dies nicht, wird die Verbindung für folgende Queries aufrecht erhalten

```
sequelize.close()
```

# Sequelize

## Modelle sind die Essenz von Sequelize.

- Ein Modell ist eine Abstraktion einer Tabelle in einer Datenbank
  - Beinhaltet Informationen über Tabellennamen, Spalten und den dazugehörigen Datentyp
  - Es sind ES6 Klassen, welche von `Sequelize.Model` erben
- Modelle in Sequelize tragen Namen
- Diese Namen müssen nicht mit dem Namen der repräsentierten Tabelle übereinstimmen
- Modellnamen sind in der Regel im Singular geschrieben während Tabellennamen im plural geschrieben werden  
→ User (Klasse) || Users (Tabelle)

# Sequelize

## Erstellen eines Datenbank Modells

```
class Mitarbeiter extends Sequelize.Model {}
```

Ein Sequelize Modell muss  
von der Klasse `Model` erben.

```
Mitarbeiter.init({  
  userName: {  
    type: DataTypes.STRING,  
    allowNull: false  
  },  
  birthday: {  
    type: DataTypes.DATE  
  }  
}, {  
  sequelize,  
  modelName: 'user'  
});
```

Übergebener Inhalt ist vergleichbar mit  
der SQL Syntax für `Create Table`.

Die Modell-Bezeichnung ist standardmäßig im  
plural zu schreiben. Aus dem Modellnamen  
`user` wird automatisch `users`.



# Sequelize

**Innerhalb von Sequelize muss jede definierte Spalte typisiert sein.**

Die Datentypen können nach dem Import des `DataTypes` Moduls genutzt werden.

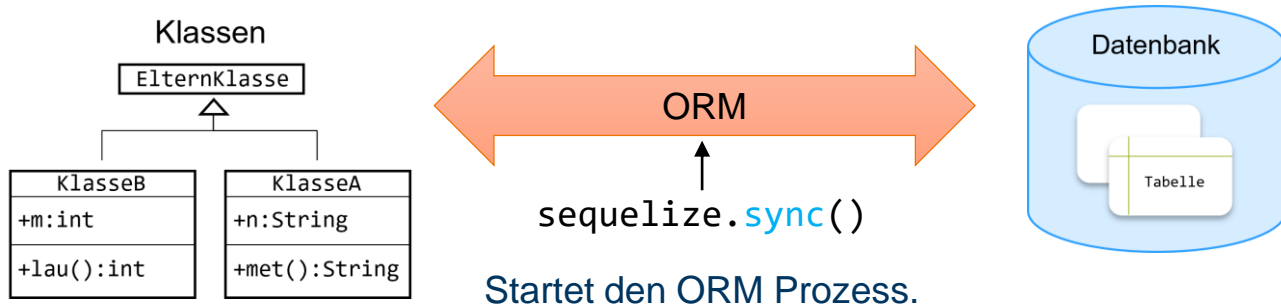
```
const { DataTypes } = require("sequelize");
```

	Sequelize	SQL
Strings {	<code>DataTypes.STRING</code>	<code>VARCHAR(255)</code>
	<code>DataTypes.STRING(1234)</code>	<code>VARCHAR(1234)</code>
	<code>DataTypes.TEXT</code>	<code>TEXT</code>
Ganze Zahlen {	<code>DataTypes.BOOLEAN</code>	<code>TINYINT(1)</code>
	<code>DataTypes.INTEGER</code>	<code>INTEGER</code>
	<code>DataTypes.BIGINT</code>	<code>BIGINT</code>
Fließkomma-Zahlen {	<code>DataTypes.FLOAT</code>	<code>FLOAT</code>
	<code>DataTypes.DOUBLE</code>	<code>DOUBLE</code>

# Sequelize

Bei der Definition eines Modells werden einige Informationen an Sequelize übergeben.

- Die Sequelize `sync`-Methode kann genutzt werden um die Datenbank mit dem Modell zu synchronisieren (gibt ein Promise zurück)
  - Hierbei wird lediglich die Datenbanktabelle und nicht das interne JavaScript-Modell angepasst
  - Sequelize übersetzt Objekte über ORM zu Datenbank-Tabellen



# Sequelize

## Modelle gezielt synchronisieren.

- Einzelne Modelle können auch gezielt mit der DB synchronisiert werden

```
class Mitarbeiter extends Sequelize.Model {}
```

```
Mitarbeiter.init({ ... });
```

```
Mitarbeiter.sync();
```

Erstellt eine Neue Tabelle in der Datenbank, falls es diese noch nicht gibt.

```
Mitarbeiter.sync(
```

```
  { force : true };
```

Falls die Tabelle schon existiert, wird diese erst gelöscht und dann neu erstellt.

```
Mitarbeiter.sync(
```

```
  { alter : true };
```

Vergleicht Unterschiede und führt nur benötigte Änderungen durch.

```
Mitarbeiter.drop();
```

Löscht die Tabelle zu einem Modell.  
`sequelize.drop()` löscht alle Tabellen!

# Sequelize

## Instanzen

- Eine Instanz einer Klasse repräsentiert ein Objekt aus dessen Modell
  - Die Instanzen sind einer Zeile der Tabelle in der Datenbank zugeordnet
  - Durch Instanzen werden die Informationen von Tabellenzeilen bearbeitet
  - Modellinstanzen sind Data Access Objects (DAO) und mit der Datenbank verknüpft
  - Die `create`-Methode kombiniert `build` und `save` in einem Aufruf

```
const jane = Mitarbeiter.create({  
  userName: 'janedoe',  
  birthday: new Date(1980, 6, 20)  
});  
  
jane.name = 'Marie';  
  
jane.save();
```

Über die  
Model-Methode `create()` wird  
eine Instanz des Mitarbeiter Modells  
erstellt

Die `save`-Methode aktualisiert die  
DB bei Änderungen.

# Sequelize

## Insert Into:

```
const jane = Mitarbeiter.create({  
  userName: 'janedoe',  
  birthday: new Date(1980, 6, 20)  
});
```

```
const mike = Mitarbeiter.create({  
  userName: 'mikeDan',  
  birthday: new Date(1991, 4, 20)  
});
```

Die DB-Tabellen-  
einträge werden über die  
create-Methode erstellt.

**Tabelle:** Mitarbeiter

id	name	birthday	createdAt	updatedAt
1	janedoe	1980-07-19 22:00:00	2020-06-0...	2020-06-0...
2	mikeDan	1991-05-19 22:00:00	2020-06-0...	2020-06-0...
NULL	NULL	NULL	NULL	NULL

# Sequelize

## Querying Select:

```
Mitarbeiter.findAll({  
  where: {  
    userName: 'janedoe'  
  }  
});
```

Abfrage aller Daten mit  
dem `username = janedoe`

**Tabelle:** Mitarbeiter

id	name	birthday	createdAt	updatedAt
1	janedoe	1980-07-19 22:00:00	2020-06-0...	2020-06-0...
2	mikeDan	1991-05-19 22:00:00	2020-06-0...	2020-06-0...
NULL	NULL	NULL	NULL	NULL

# Sequelize

## Querying Delete:

```
Mitarbeiter.destroy({  
  where: {  
    userName: 'mikeDan'  
  }  
});
```

Löschen aller Daten mit  
dem `username = mikeDan`

**Tabelle:** Mitarbeiter

id	name	birthday	createdAt	updatedAt
1	janedoe	1980-07-19 22:00:00	2020-06-0...	2020-06-0...
2	mikeDan	1991-05-19 22:00:00	2020-06-0...	2020-06-0...
NULL	NULL	NULL	NULL	NULL

# Sequelize

## Getter & Setter

Sequelize bietet die Möglichkeit eigene **Getter** und **Setter** für **Modell-Attribute** zu definieren

- Mit Sequelize können auch die virtuellen Attribute angegeben werden
- Die Attribute des Sequelize-Modells sind in der SQL-Tabelle nicht vorhanden
- Die Attribute werden automatisch von Sequelize übertragen

```
class Mitarbeiter extends Sequelize.Model {  
  get name() {  
    return this.name();  
  }  
  set name(value) {  
    this.setDataValue('name', value);  
  }  
}
```

Der bisher leere  
Anweisungsblock des Modells  
Mitarbeiter wird um die  
getter und setter-Methoden  
erweitert.



# Sequelize

## Erstellen und testen der Getter und Setter-Methoden:

```
Mitarbeiter.init({  
  ...  
});  
  
sequelize.sync({force:true}).then(() => {  
  const mike = Mitarbeiter.create({  
    userName : 'mikeDan',  
    birthday: new Date(1991, 4, 20)  
  });  
  
  console.log(mike.name);  
  mike.name = "Gustav"  
  console.log(mike.name);  
});
```

Durch das `force`-Attribut werden erst leere Tabellen erzeugt und anschließend mit Daten gefüllt (wenn das Promise erfüllt wurde).

Erstellung eines Mitarbeiter Objekts

mikeDan

Gustav

Aufruf der Getter und Setter Methoden

# Sequelize

## Sequelize unterstützt die Standardzuordnungen bei Relationen

- Standardzuordnungen:  
`Eins-zu-Eins`, `Eins-zu-Viele` und `Viele-zu-Viele`
- Relationen in Konventionellen SQL werden mit einer Kombination aus `join` und `select` Anweisungen erzeugt
- In Sequelize gibt es Hilfsmethoden für Zuordnungen:  
`BelongsTo`, `HasMany`, `BelongsToMany`, `HasOne`
- Relationen werden kombiniert, um Zuordnungen anzugeben

# Sequelize

## Assoziationen

- **BelongsTo Relation**  $\rightarrow$  **A.belongsTo(B)**
  - Zwischen A und B besteht eine Eins-zu-Eins-Beziehung, wobei der Fremdschlüssel im Quellmodell (A) definiert ist.
- **HasMany Relation**  $\rightarrow$  **A.hasMany(B)**
  - Zwischen A und B besteht eine Eins-zu-Viele-Beziehung, wobei der Fremdschlüssel im Zielmodell (B) definiert ist.

# Sequelize

## Assoziationen

- **BelongsToMany Relation**  $\rightarrow$  **`A.belongsToMany(B, { through: 'C' })`**
  - Zwischen A und B besteht eine Viele-zu-Viele-Beziehung  
C ist eine Kombinationstabelle
- **HasOne Relation**  $\rightarrow$  **`A.hasOne(B)`**
  - Eine Eins-zu-Eins-Beziehung zwischen A und B, wobei der Fremdschlüssel im Zielmodell (B) definiert ist.

# Sequelize

## Beispiel einer Eins-zu-Eins Relation:

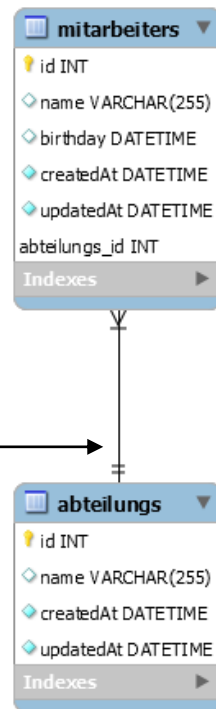
Eins-zu-Eins-Beziehung → `hasOne` und `belongsTo`.

`Mitarbeiter.hasOne(Abteilung);` oder

`Abteilung.belongsTo(Mitarbeiter);`

`sequelize.sync();`

Ein Mitarbeiter ist genau  
einer Abteilung zugeordnet



# Sequelize

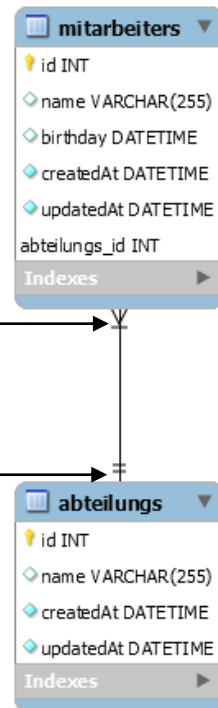
## Beispiel einer Eins-zu-Viele Relation:

Eins-zu-Viele-Beziehung → `hasMany` und `belongsTo`

```
Abteilung.hasMany(Mitarbeiter);  
Mitarbeiter.belongsTo(Abteilung);  
sequelize.sync();
```

Eine Abteilung hat 1 ...  
N Mitarbeiter

Ein Mitarbeiter ist genau  
einer Abteilung  
zugeordnet



# Sequelize

## Beispiel einer Viele-zu-Viele Relation:

**Viele-zu-Viele-Beziehung** → zwei `belongsToMany` Aufrufe zusammen

```
let conTab = sequelize.define(
  'kurs_has_student', {}
);

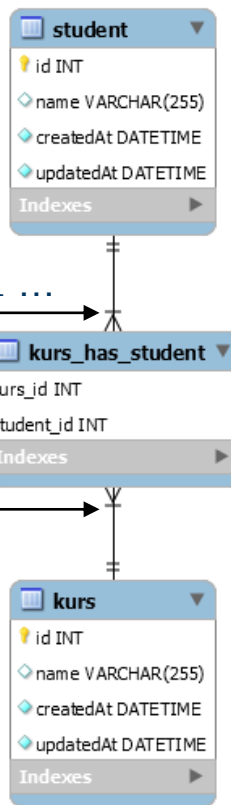
Kurs.belongsToMany(
  Studenten,
  { through: conTab }
);

Studenten.belongsToMany(
  Kurs,
  { through: conTab }
);

sequelize.sync();
```

Ein Student besucht 1 ...  
N Kurse

Ein Kurs hat 1 ... N  
Studenten



# Sequelize

## Den Wert einer Relation zuweisen/setzen:

- Eine Relationen erzeugt Hilfsmethoden zum Setzen von Fremdschlüssel über Objekte

```
Mitarbeiter.hasOne(Abteilung);
Abteilung.belongsTo(Mitarbeiter);
...
let transferData = async () => {
  const abteilung = await Abteilung.create({
    name: 'Verwaltung'
  });
  const mike = await Mitarbeiter.create({
    name: 'mikeDan',
    birthday: new Date(1991, 4, 20)
  }).then(mitarbeiter => {
    mitarbeiter.setAbteilung(abteilung);
  });
};
transferData();
```

Async/await ist erforderlich  
um auf den Datentransfer der  
create-Methoden zu warten,  
bevor die Bearbeitung  
fortgesetzt wird!

Die Methode setAbteilung  
ist nur durch die Relation  
verfügbar! Setzt den  
Fremdschlüssel.

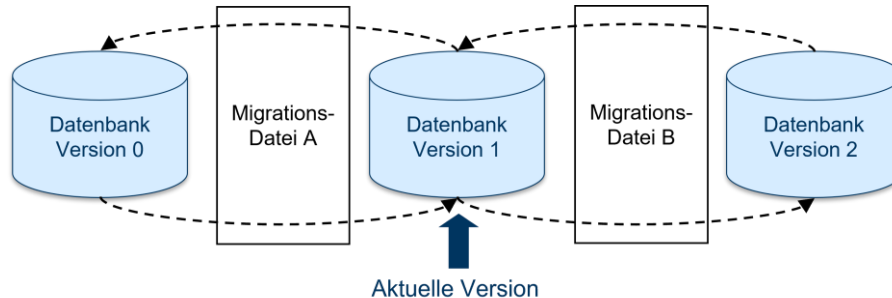


# Web-Engineering **Migrations**

# Migrations

## Was sind Migrations und wofür werden sie benötigt?

- Eine Migration kapselt Zustände und Anweisungen von Datenbanken
  - Gekapselte Anweisungen können wahlweise in die Datenbank überführt werden
  - Ermöglicht eine Versionskontrolle und Änderungs-Nachverfolgung
  - Änderungen können wieder rückgängig gemacht werden



# Migrations

Migrations können durch das **sequelize-cli** Modul genutzt werden.

Kommandozeile

```
C:\Users\...> npm install sequelize-cli  
C:\Users\...> npx sequelize-cli init
```

npm → Verwaltet Pakete.

npx → Führt Node-Pakete aus.

Um ein leeres Projekt zu erstellen muss der `init` Befehl ausgeführt werden.

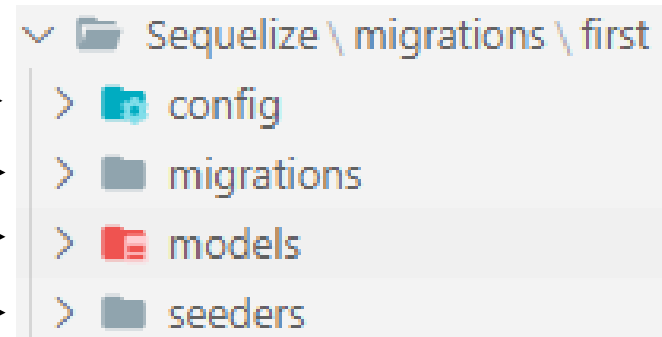
Enthält Dateien zur Kommunikation für die CLI  
/ Datenbankverbindung.

Enthält alle Migrations-Dateien.

Enthält alle Modelle.

Enthält alle Seeder-Dateien.  
(Diese können Tabellen mit Beispieldaten füllen)

Die erzeugte Ordnerstruktur:



# Migrations

## Die sequelize-cli config-Datei

- Die Attribute in `development`, `test` und `production` stellen unterschiedliche Umgebungen für die Entwicklung bereit
- Die Werte für diese Attribute werden automatisch erzeugt und sollten angepasst werden
- So können unterschiedliche Datenbanksystem mit unterschiedlichen Dialekt übergreifend verwendet werden

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  },
  "test": {
    ... ,
    "database": "database_test"
  },
  "production": {
    ... ,
    "database": "database_production",
  }
}
```

# Migrations

## Erstellung der ersten Migration in der Kommandozeile

- Es wird ein Zustand für eine Datenbank über `model:generate` erzeugt

- In etwa vergleichbar mit einem Git-Commit

- Der `model:generate` Befehl erstellt ...

- ... eine Migrations-Datei im `migrations`-Ordner:

XXXXXXXXXXXXXXXXX-create-user.js

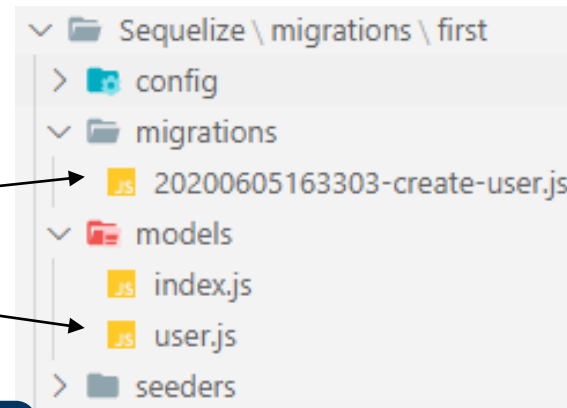
- ... eine Modell-Datei im `models`-Ordner: `user.js`

Der `model:generate` Befehl hat zwei Optionen:

Kommandozeile

```
C:\Users\...>npx sequelize-cli model:generate  
--name User --attributes firstName:string,lastName:string
```

--name: der Name des Modells  
--attributes: die Modellattribute



# Migrations

## Anpassen des Inhalts in der erzeugten xxx-create-user.js Datei.

- Das `QueryInterface` wird genutzt um Änderungen auf DBs zu kommunizieren

```
module.exports = {  
  up: (queryInterface, Sequelize) => {  
    return queryInterface.createTable('Users', {  
      id: {  
        allowNull: false, primaryKey: true,  
        autoIncrement: true, type: Sequelize.STRING  
      },  
      titel: { type: Sequelize.STRING }  
    });  
  },  
  down: (queryInterface, Sequelize) => {  
    return queryInterface.dropTable('Users');  
  }  
};
```

} Logik für das Aufspielen  
einer neuen Tabelle.

} Logik für das Entfernen einer  
existierenden Tabelle.

# Migrations

## Weitere Methoden des queryInterface.

- Das `QueryInterface` wird genutzt um Änderungen auf DBs zu kommunizieren

```
queryInterface.addColumn('TableName', 'ColumnName', {  
  type: Sequelize.INTEGER,  
  references: {  
    model: 'TableName',  
    key: 'id'  
  },  
  onUpdate: 'CASCADE',  
  onDelete: 'SET NULL'  
});
```

Manuelle definition einer  
Foreign Key Spalte.

# Migrations

```
queryInterface.removeColumn('TableName', 'ColName',  
{ /* query options */ });
```

```
queryInterface.changeColumn('TableName', 'ColName', { ... });
```

```
queryInterface.bulkInsert('Users', [{ TABLEDATA }], {});
```

```
queryInterface.bulkDelete('Users', null, {});
```

Die bulk-  
Methoden dienen  
dabei als Befehl  
Daten in die  
Tabelle  
einzupflegen oder  
zu entfernen. Wird  
im wesentlichen  
vom Seeder  
verwendet!

**API-Referenz:** <https://sequelize.org/master/class/lib/dialects/abstract/query-interface.js~QueryInterface.html>



# Migrations

## Migrations aufspielen.

Die `db:migrate` Anweisung ermöglicht Anwenden und Entfernen einer Migration.

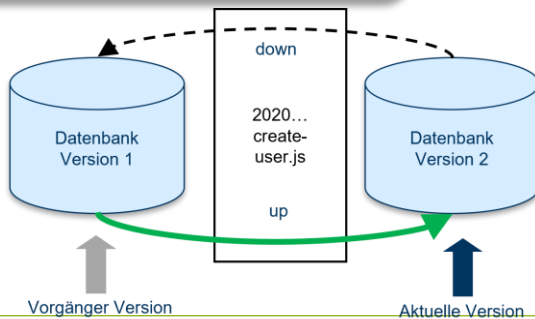
- Vorgang wird über eine spezielle Tabelle namens `SequelizeMeta` in DB koordiniert

### Kommandozeile

```
C:\Users\...> npx sequelize-cli db:migrate
C:\Users\...> npx sequelize-cli db:migrate
--to filename.js
```

Es werden die bisher nicht ausgeführten Migrationsdateien ausgeführt.

Eine spezifische Migration aufspielen.



# Migrations

## Migrations rückgängig machen.

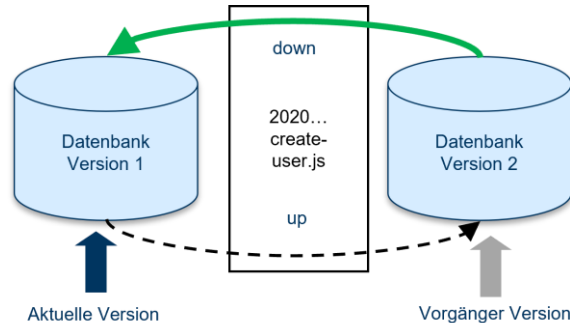
Kommandozeile

```
C:\Users\...> npx sequelize-cli db:migrate:undo  
C:\Users\...> npx sequelize-cli db:migrate:undo:all  
C:\Users\...> npx sequelize-cli db:migrate:undo:all  
--to XXXXXXXXXX-create-posts.js
```

Die letzte Migration  
rückgängig machen.

Alle Migration`s  
rückgängig machen.

Spezifische Migration  
rückgängig machen.



# Migrations

## Was sind Seedings und wofür werden sie benötigt?

- Mit Seed-Dateien können mehrere Datenbank-Tabellen mit Beispiel oder Testdaten aufgefüllt werden
- Seedings haben dieselbe Schreibweise wie gewöhnliche Migration-Dateien
- Sie entkoppeln Änderungen am Schema von dem Datensatz!

Kommandozeile

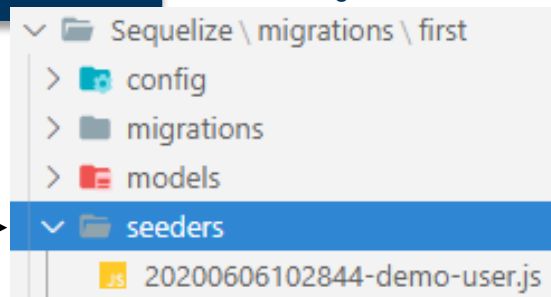
```
C:\Users\...> npx sequelize-cli seed:generate --name demo-user
```

--name: der Name des Seeds

Die erzeugte Ordnerstruktur:

Der `seed:generate` Befehl erstellt  
eine Datei im `seeders`-Ordner:

XXXXXXXXXXXXXXXXX-demo-user.js



# Migrations

## Seedings ausführen

- Nach ausführen des `db:seed:all` Befehls sind die seeds an die Datenbank angebunden

Kommandozeile

```
C:\Users\...> npx sequelize-cli db:seed:all
```

Kommandozeile

```
C:\Users\...> npx sequelize-cli db:seed:undo  
C:\Users\...> npx sequelize-cli db:seed:undo:all  
C:\Users\...> npx sequelize-cli db:seed:undo:undo  
-seed XXXXXXXXXX-seed.js
```

Den letzten Seed rückgängig machen.

Alle Seed's rückgängig machen.

Spezifische Seed's rückgängig machen.