Datenbankprogrammierung mit gorm

- Das erste Beispiel stammt aus dem gorm Tutorial
  - GoDoc zu gorm
- Wir bilden eine struct auf die Datenbank ab
  - Wir nutzen die struct Model
  - Sie enthält vordefinierte Felder:
    Id → Primärschlüssel
    created, updated, deleted → Zeitangaben über diese Datenbankoperationen

```
type Product struct {
        gorm.Model
        Code string
        Price uint
}
```

```
type Model struct {
    ID        uint `gorm:"primary_key"`
    CreatedAt time.Time
    UpdatedAt time.Time
    DeletedAt *time.Time `sql:"index"`
}
```

Datenbankprogrammierung mit gorm

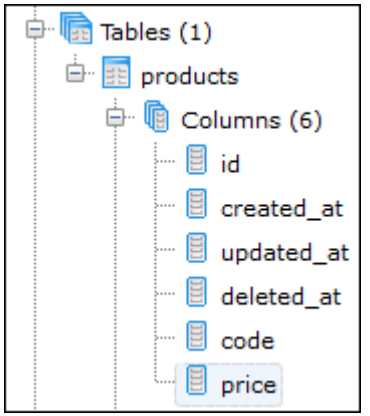
- Zuerst muss eine Verbindung zur Datenbank aufgebaut werden
- Wenn Verbindung nicht möglich ist, Abbruch
- Am Ende des Programms muss die Verbindung wieder geschlossen werden → defer db.Close()

```
// Demonstriert Benutzung der gorm Bibliothek,
// basiert auf dem Quickstart von Jinzhu
func main() {
        db, err := gorm.Open("postgres",
                "user=oosy dbname=gorm1 password=oosy2016 sslmode=disable")
        if err != nil {
                panic("failed to connect database")
        }
        defer db.Close()
```

Datenbankprogrammierung mit gorm

- AutoMigrate() legt das Datenbankschema für die struct an oder aktualisiert es
  - Achtung, es werden nur neue Spalten hinzugefügt, keine Spalten gelöscht oder im Typ geändert
  - Es werden nur öffentliche Felder der struct berücksichtigt
- In pgAdmin kann man sich die DDL (Data Definition Language) ansehen

```
// Migrate the schema
db.AutoMigrate(&Product{})
```




```
CREATE TABLE public.products
(
    id integer NOT NULL DEFAULT nextval('products_id_seq'::regclass),
    created_at timestamp with time zone,
    updated_at timestamp with time zone,
    deleted_at timestamp with time zone,
    code text COLLATE "default".pg_catalog,
    price integer,
    CONSTRAINT products_pkey PRIMARY KEY (id)
)
```

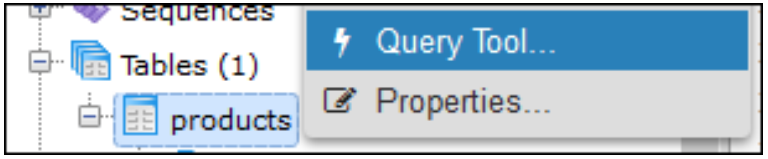
Datenbankprogrammierung mit gorm

- Jetzt kann mit den Daten gearbeitet werden
  - Ein neues Objekt in die Datenbank speichern
  - Ein Objekt aus der Datenbank in eine Variable lesen
  - Ein Datenfeld in der Datenbank aktualisieren
    - Es können auch nur einzelne Felder aktualisiert werden → Tutorial
  - Ein Objekt aus der Datenbank löschen

```
// Create
db.Create(&Product{Code: "L1212", Price: 1000})
// Read
var product Product
// find product with smallest id
db.First(&product)
// Update product's price to 2345
product.Price = 2345
db.Save(&product)
// Delete - delete product
db.Delete(&product)
```

Datenbankprogrammierung mit gorm

- Mit dem Query-Tool von pgAdmin kann leicht überprüft werden, was in die Datenbank geschrieben wurde



gorm1 on postgres@PostgreSQL 9.6

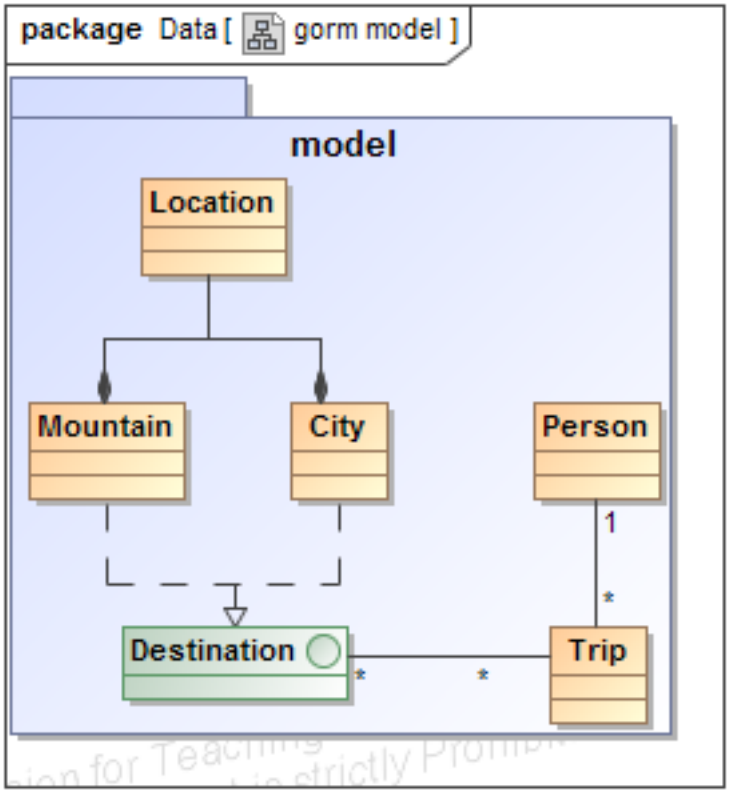
```
1  select * from products;
```

Data Output | Explain | Messages | History

	id integer	created_... timesta...	updated... timesta...	deleted_... timesta...	code text	price integer
☐	5	2016-10-...	2016-10-...		L1212	1000
☐	4	2016-10-...	2016-10-...		L1212	2345

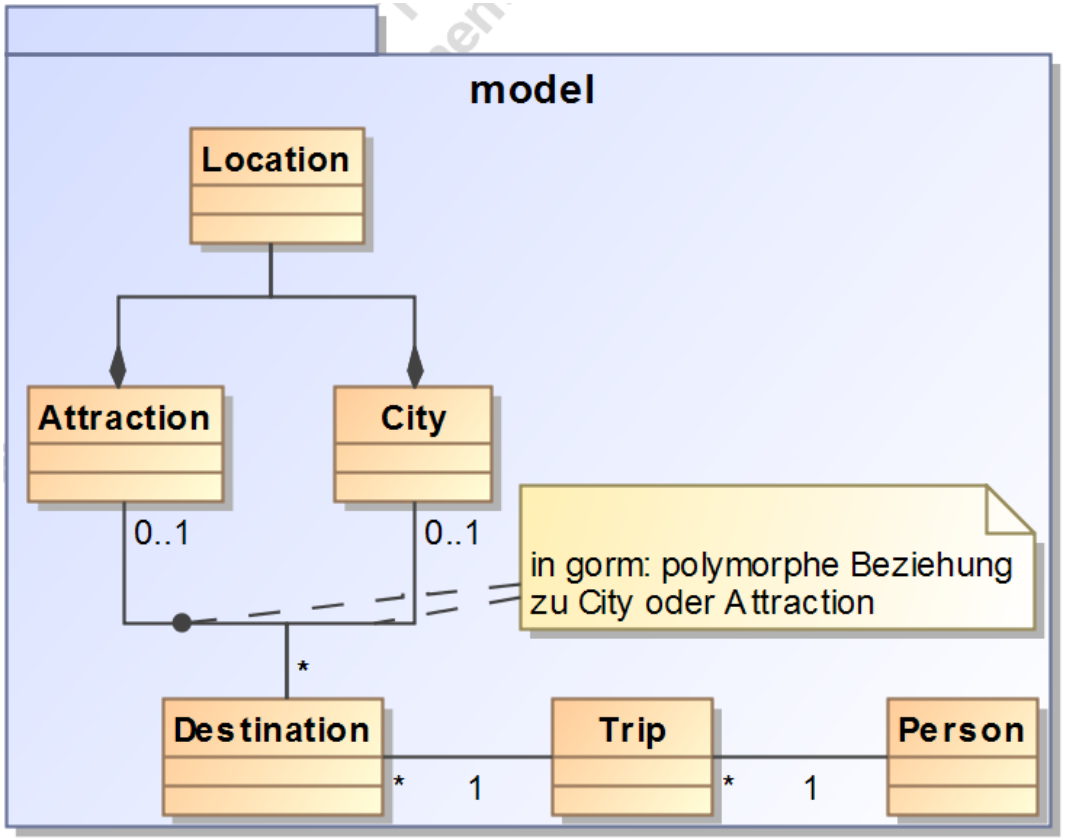
Ein komplexeres Beispiel für gorm

- Von einer Person können auf einem Trip mehrere Destinationen (Berge, Städte) besucht werden
- Ein Trip gehört immer zu einer Person, aber sie kann mehrere Trips planen
- Die gleichen Destinationen können in vielen Trips besucht werden
- Es gibt viele Personen



Beispiel verbessert

- Gorm braucht zum Datenbank-zugriff ein Objekt einer struct, kein Interface
- Vererbung gibt es nicht in Go
- M:N Beziehungen sind bei genauerer Betrachtung oft 1:N–N:1 Beziehungen, weil noch zusätzliche Information zur Beziehung hinzugefügt werden soll
  - Aus dem Interface Destination wird eine Klasse (UML-Sichtweise) bzw. eine struct (Go-Sichtweise)



Beispiel: Transaktionen und Locking

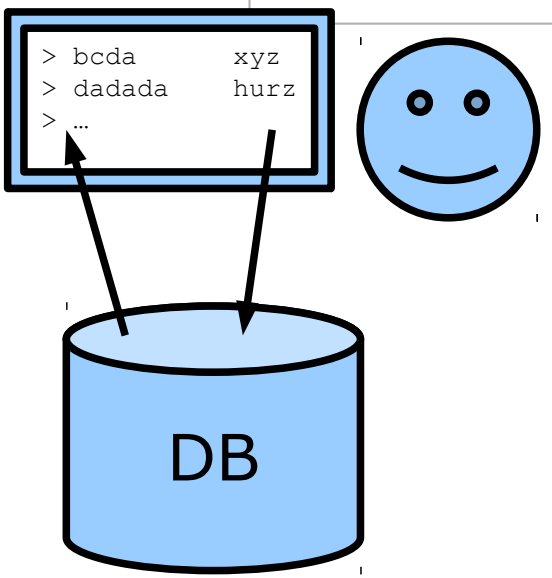
- Transaktion: Mehrere Datenbankoperationen müssen gemeinsam ausgeführt werden
  - Man lässt diese Operationen innerhalb einer Transaktion ablaufen
  - Commit: Alle Operationen werden gemeinsam ausgeführt
  - Rollback: Alle Operationen werden rückgängig gemacht
- Lock: Während einer Transaktion kann die Datenbank nicht von anderen Prozessen verändert werden

```
// beginne Transaktion: Alle
// Datenbankoperationen nur mit tx!
tx := db.Begin()
// lies Objekt aus DB in p1
tx.First(&p1)
// ändere p1
p1.Price = 1111
p1.Code = "P4711"
// sichere p1 zurück in die DB
db.Save(&p1)
// beende Transaktion
tx.Commit()
```

Beispiel: Optimistic Locking, lange Transaktionen

- Benutzer bearbeitet Daten
  - Daten lesen und anzeigen
  - längere Bearbeitung
- Daten in DB unverändert ?
  - Ja: Daten speichern
  - Nein: Nicht speichern
    - Veränderungen anzeigen
    - Benutzereingaben und geänderte Daten zusammenführen
    - Automatisch oder manuell

```
// Zeitstempel auf fremden update überprüfen
if db.Where("updated_at = ?", p2.UpdatedAt).
// nur speichern, wenn keine Änderung
        Save(&p2).
// Anzahl der geänderten Zeilen in DB testen
        RowsAffected == 0 {
        // geänderte Daten integrieren ...
        fmt.Printf("Not saved to DB\n")
} else {
        // alles OK, kein Update-Konflikt
        fmt.Printf("Saved to DB\n")
}
```



Beispiel: Speichern von assoziierten Objekten (1:1)

- Destination hat eine Assoziation zu City
  - Speichen einer Destination speichert auch die assoziierte City
  - Umgekehrt geht es nicht!

```
type Destination struct {
      Model
      Dest City
}
```

Assoziation

```
city := model.New(aCity)
dest := model.Destination{Dest: city}
// create dest sichert auch city in
// die DB -> kaskadieren
db.Create(&dest)
```

```
type City struct {
      Model
      Location
      Name          string
      Inhabitants   int
      DestinationID uint
}
```

Fremdschlüssel

Abhängige Objekte laden: Preloading

- gorm kann mit Preload("Attributname") die über eine Assoziation verbundenen Objekte beim Laden eines Objekts mitladen
- Das geht sowohl beim Laden eines einzelnen Objekts mit First(…) als auch beim Laden einer Objektmenge mit Find(…)
- Es geht nur in einer Richtung der Assoziation
- Preload kann auch über mehrere Assoziationen gehen (nested preloading)

```
var destinations []model.Destination
// alle Destinations aus der Datenbank
// in ein Array einlesen.
// Das Feld Dest bleibt leer.
db.Find(&destinations)
```

```
// Preload holt die Objekte der
// Assoziation "Dest" in das Objekt
db.Preload("Dest").Find(&destinations)
```

```
// Preload Person -> Trips -> Cities
db.Preload("Trips").
      Preload("Trips.Cities").
      First(&kirki, kirk.ID)
```

Assoziierte Objekte laden: Related bei 1:N und 1:1

- Mit Related(…) können die über eine Assoziation verbundenen Objekte in eine Variable geladen werden
- Variable kann sein:
  - Feld des assoziierten Objekts
  - beliebige andere Variable
- Wenn Feldname != Feldtyp muss der Feldname angegeben werden, der im Parameter von Model(…) verwendet wird
- Es geht in beide Richtung der Assoziation

```
// (1:N): von Person -> Trip
db.Model(&rara).Related(&rara.Trips)
// (N:1): von Trip -> Person
db.Model(&lara.Trips[0]).
       Related(&fromTrip)
```

```
// Lade Destination, die auf "Köln"
// zeigt, mit Related in Variable
var cgn model.City
var dcgn model.Destination
// First mit Query-Parameter aufrufen
db.First(&cgn, "name = ?", "Köln")
// (1:1) City -> Destination
// Fremdschlüsselname ohne _ID
db.Model(&cgn).
      Related(&dcgn, "Destination")
```

Assoziierte Objekte laden: Related bei M:N

- M:N Assoziationen können bidirektional implementiert werden
  - []slice bei beiden assoziierten structs anlegen mit dem Typ der Gegenseite
  - In Trip: `Cities []City` `` `gorm:"many2many:trip_city;"` ``
  - In City: `Trips []Trip` `` `gorm:"many2many:trip_city;"` ``
  - Gleiche Annotation für die Korrelationstabelle
- Namen der Assoziationsvariablen müssen anscheinend angegeben werden
  - Beispiel in persist4.go

```
var muc model.City
var toMuc []model.Trip
db.First(&muc, "name = 'München'")
// (M:N) City -> Trip
db.Model(&muc).
      Related(&toMuc, "Trips")
```

```
var tripCities []model.City
// (M:N) Trip -> City
db.Model(&kirki.Trips[0]).
      Related(&tripCities, "Cities")
```

Über die Assoziation hinweg suchen

- Oft soll ein Objekt gesucht werden, dessen assoziiertes Objekt eine Bedingung erfüllt
  - Zwei Zugriffe auf die DB oder ein Join

```
// Finde Destination, die auf "München"
// zeigt, mit zwei Datenbankzugriffen
var muc model.City
var dmuc model.Destination
db.First(&muc, "name = ?", "München")
// First mit Query-Parameter aufrufen
db.Preload("Dest").First(&dmuc,
                muc.DestinationID)
```

```
// Finde Destination, die auf "Berlin" zeigt, mit einem Datenbankzugriff
// in JOIN muss die "fremde" Tabelle, zu der der Join läuft, vorn stehen
var dmuc1 model.Destination
db.Joins("JOIN cities On cities.destination_id = destinations.id"+
      " AND cities.name = ?", "Berlin").Preload("Dest").First(&dmuc1)
```

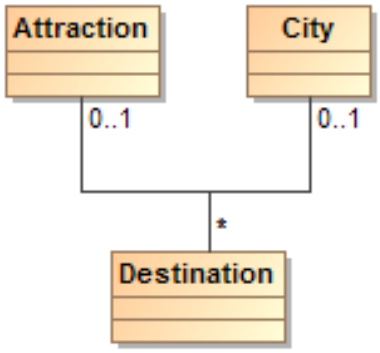
Definition einer polymorphen Assoziation in gorm

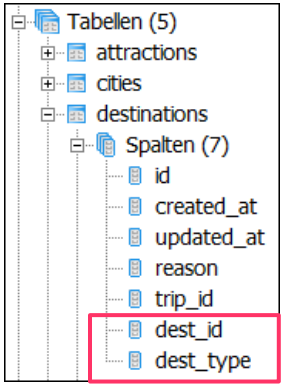
- Polymorph heißt hier, dass eine Assoziation zu mehreren unterschiedlichen structs (d.h. Tabellen in der DB) führen kann
- Gorm unterstützt Polymorphie nur bei 1:1 und 1:N Assoziationen
- Ein Fremdschlüssel (dest_id) verweist auf Einträge in zwei unterschiedlichen Tabellen
- Das Typ-Attribut (dest_type) gibt die referenzierte Tabelle an
- Zeile nicht trennen:
  `` `gorm:"polymorphic:Dest;"` ``

```
type Attraction struct {
    …
    Destination []Destination `gorm:
                "polymorphic:Dest;"`
}
```

```
type City struct {
    …
    Destination []Destination `gorm:
                "polymorphic:Dest;"`
}
```

```
// Reiseziel
type Destination struct {
    …
    DestID   uint
    DestType string
}
```





Einfügen in eine polymorphe Assoziation in gorm

- Objekte werden in einer polymorphen Assoziation wie in anderen Assoziationen auf der Seite hinzugefügt, wo sich das Feld oder Slice befindet (nicht der Fremdschlüssel)
  - Im Beispiel: City, Attraction
- Save bewirkt auch hier, dass auch das assozierte Objekt in die Datenbank geschrieben wird
  - Der richtige Fremdschlüssel-Wert wird gespeichert
  - Das Feld mit dem Tabellennamen verweist automatisch auf die richtige Tabelle
  -

```
var dests []model.Destination
var cities []model.City
db.Find(&cities,
    "name in ('Köln', 'München'," +
        " 'Düsseldorf')")
for _, c := range cities {
    dest := model.Destination{
        Reason: "Karneval"}
    c.Destination = append(
        c.Destination, dest)
    db.Save(&c)
}
```

```
dest := model.Destination{
        Reason: "Schloßbesichtigung"}
var att model.Attraction
db.First(&att, "name like 'Neuschw%'")
att.Destination = append(
        att.Destination, dest)
db.Save(&att)
```

Zugriff über eine polymorphe Assoziation in gorm

Zwei Varianten

1) die originalen structs aus der Datenbank füllen
   - Testen auf Tabellen-name
   - Richtigen Typ lesen
2) nur bestimmte Informatio-nen aus der Datenbank ho-len
   - lokal eine struct mit den benötigten Feldern anlegen
   - das XxxType Feld zur Auswahl der Daten-bank-Tabelle nutzen

```
db.Find(&dests)
for _, dest := range dests {
    var city model.City
    var attr model.Attraction
    fmt.Printf("Reiseziel %s: ", dest.Reason)
    // ausführliche Variante 1:
    // Polymorphes Objekt vollständig lesen
    if "cities" == dest.DestType {
        db.First(&city, dest.DestID)
        fmt.Printf("City %s\n", city.Name)
    } else {
        db.First(&attr, dest.DestID)
        fmt.Printf("Attraction %s\n", attr.Name)
    }
    // kompakte Variante 2: nur die Werte
    // lesen, die gebraucht werden
    var any struct{
        Name string
    }
    db.Table(dest.DestType).
        Where("ID = ?", dest.DestID).Scan(&any)
    fmt.Printf("\t%s\n", any.Name)
}
```