COMP8005 – Applied Web Development

Introduction to Eloquent ORM

Topics

- More on Object-Relational Mapping
- The impedance mismatch
- The ActiveRecord pattern / anti-pattern
- Eloquent ORM

Object-Relational Mapping

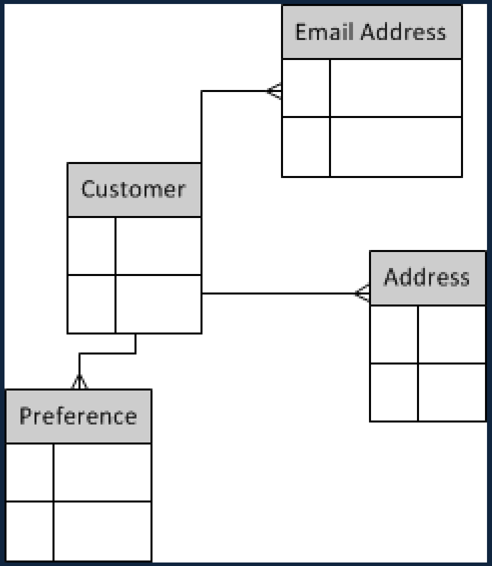
- Two models in an application that can cause headaches when trying to move from one to the other:
  - The domain model (e.g. modelled using class diagrams)
  - The data model (e.g. relational data model modelled using ER diagrams)
- Persisted data lives in the database and the data model
- In-memory application data resides in the domain model

Impedance Mismatch

- The **impedance mismatch** refers to the mismatch between the tables in our database and the classes in our application
- In the relational data model, normalisation reduces duplication at the expense of multiple joined-up tables
- Let’s look at the example of a customer in terms of the relational and domain models

Normalised ER Model

- Customer has many (repeating) email addresses, billing or delivery addresses and preferences



Class Model

```
class Customer {
  int id;
  String name;
  Address[] addresses;
  Email[] emails;
  Preference[] prefs;
}
```

Mapping

- Must get data from 4 tables into 1 object
- Object-Relational Mappers help you do this
- You configure the mappings so that when you retrieve a customer, not only is the customer object created, but its internal arrays or collections are populated
- Can also handle more complex mappings, e.g. where a table is split among several objects

Hibernate

- Hibernate started out as a Java ORM, but has been ported to other languages
- Handles complex mapping and you can configure objects to have their inner collections or arrays populated automatically or not
- Can recognise when objects are "dirty", i.e. have changed from what is currently stored in the database
- You can "save" an object and all its inner objects will be persisted to the database… if they are "dirty", otherwise will ignore the save request
- We will look at some brief Hibernate examples later

ActiveRecord

- ActiveRecord is a very simple design pattern
- You don’t cater for complex mappings like the previous example
- Instead you have a one-to-one mapping between table and object
- This is in contrast to a domain-driven approach which is focused on objects, such as real world objects

ActiveRecord

- In some respects it is an anti-pattern as it can result in an "anaemic domain model" – one that goes against object-oriented design where you start with an object model that could differ from the ER model
- It can violate the "single responsibility principle" of OO design – i.e. the ActiveRecord object takes on more responsibility than just regular data access – leading to a lack of cohesion that can have knock-on affects on things like testing

ActiveRecord

- Remember the layering from the last set of slides?
- The Domain object was passed from layer to layer
- With ActiveRecord, the domain object is also the persistence layer, so it violates the traditional multilayered architecture

ActiveRecord

- With all that said, for rapid application development (RAD) it can offer some benefits, particular simplicity and speed to develop, as well as an ability to quickly develop prototypes to aid agile development
- It may be more suited to small to medium-scale applications, of which there are many
- Not every organisation is a massive enterprise and this will usually satisfy their needs
- You need to decide if on the one hand the convention over configuration advantages are outweighed by the other factors outlined

ActiveRecord

- Main point is: choose ActiveRecord in some cases and go for a domain-driven approach (e.g. without an ORM or with a more complex DataMapper ORM like Hibernate) in others – comes down to being that well-rounded software developer capable of making those kind of informed decisions
- In Laravel we will begin with looking at its ActiveRecord implementation: Eloquent ORM
- We will also look at more domain-driven and layered approaches that start with class diagrams without reference to an ER model – we can achieve this in the same way as Spring does, with IoC and dependency injection

Eloquent ORM

- The accompanying tutorials show how to create migrations, which are scripts that create tables in the database
- They also show how to start with simple Eloquent model classes
- In the last set of slides, we looked at the most simple of Eloquent model classes…

Model Class in Laravel

```
<?php
  class Book extends Eloquent { }
?>
```

- No content?
- No requirement to write your own methods / functions
- You are inheriting some basic functions from the Eloquent ORM class

Model Class in Laravel

- e.g. *`all()`* retrieves all rows from a table
- We make a static call to `Book::all()` in our code to retrieve all book rows
- E.g.

```
$books = Book::all();
foreach($books as $book) {
   // do something with the $book,
   echo '<p>' . $book->title . '</p>';
}
```

Other Eloquent Functions

- Query for a single row by primary key:
  - $book = Book::find(1)

    // if using a simple integer id
  - $book = Book::find(“123456789-123”);

    // if your primary key was a varchar like an
    // ISBN

Other Eloquent Functions

- Saving a record

```
$book = new Book;
$book->title = ‘The Shining’;
$book->author_id = 1;
// etc.
$book->save();
```

- Updating an existing record

```
$book = Book::find(1);
$book->isbn = ‘234234222’;
$book->save();
```

Other Eloquent Functions

- Deleting a record

```
$book = Book::find(1);
$book->delete();
```

- Or more directly…

```
Book::destroy(1);
```

One-To-Many

Can represent a one-to-many relationship

E.g.

```
class Book extends Eloquent {
  public function author() {
    return $this->belongsTo('Author');
  }
}
```

Can then use for example

```
$author = Book::find($id)->author;
```

Other Eloquent Functions

- Let’s take a spin through the official Eloquent web documentation for a comprehensive list of functions / options:
  - http://laravel.com/docs/eloquent

Hibernate Example

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;

   public Employee() {}
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }

etc......
```

Hibernate Example

- Other “annotations”
  - @GeneratedValue – use with @Id to indicate the key is auto-generated
  - @ManyToOne and @OneToMany and @JoinColumn, e.g. in an Author entity class:

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = “book_id", nullable = false)
public Book getBook() { return this.book; }
```