“Classes” in your projects

React
Components

Other classes, e.g.
game “engine”

Models
Model data

Transform data, validations, etc.

“Knows”, associations
Movie model (M in MVC)

Movie “resource” starts as a simple object (POJO), later transitions to ORM model

- Validate user rating is 0-5
- Provide “virtual” attributes that transform data
- Express associations between models

```javascript
class Movie extends Model {
    static get tableName() {
        return 'Movie';
    }

    ... static get relationMappings() {
        ...
    }
}
```

Movie table in database

ORM is a design pattern for mapping database schema to object
These ORM models exist only on the server

Communicate with client via JSON (POJO)

3-tier Architecture

Presentation Tier
- Web Server (e.g. Apache, NGinx)

Logic Tier
- App. Server (e.g. NodeJS)

Persistence Tier
- Database (e.g. Mongo, PostgreSQL)

Routing & Controllers (e.g. Express)

Models (e.g. knex, objection)

MVC
**Recall: Film Explorer CRC cards**

<table>
<thead>
<tr>
<th>User</th>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
<td><strong>Collaborator</strong></td>
</tr>
<tr>
<td>Knows user’s name</td>
<td>Rating</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>Knows movies I rated</td>
<td>User</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Genre</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
<td><strong>Collaborator</strong></td>
</tr>
<tr>
<td>Knows rating</td>
<td></td>
</tr>
<tr>
<td>Knows its owner</td>
<td>User</td>
</tr>
<tr>
<td>Knows its movie</td>
<td>Movie</td>
</tr>
</tbody>
</table>

*“many to many”*  *“one to many”*

*Kent Beck & Ward Cunningham, OOPSLA 1989*
Thinking in relations/associations

• “HasOne” / “BelongsToOne”
  One-to-one relationship, e.g. Supplier and Account
• “HasMany” / “BelongsToOne”
  One-to-many relationship, e.g. Movie and Genre
• “ManyToMany”
  Many-to-many relationship (often called “has many through”), e.g. User and Movie through Rating
You are developing an application for a veterinarian’s office. How would you model the relation between Customer and Animal?

A. One-to-one, e.g. “HasOne”
B. One-to-many, e.g. “HasMany”
C. Many-to-many, e.g. “HasManyThrough”
RESTful URLs for associations

RESTful routes for “has many” association, e.g. Movie “has many” Genres

<table>
<thead>
<tr>
<th>Route</th>
<th>Controller Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/movies/:movie_id/genres</td>
</tr>
<tr>
<td>GET</td>
<td>/movies/:movie_id/genres/:id</td>
</tr>
<tr>
<td>PUT</td>
<td>/movies/:movie_id/genres/:id</td>
</tr>
<tr>
<td>DELETE</td>
<td>/movies/:movie_id/genres/:id</td>
</tr>
<tr>
<td>GET</td>
<td>/movies/:movie_id/genres</td>
</tr>
</tbody>
</table>
Routes aren’t just for the server

Routes are a description of our data model

• A “traditional” application often has views associated with association routes
  
  GET /movies/:movie_id/genres/new is a form to add genre to :movie_id, which then creates a request to
  POST /movies/:movie_id/genres

• You can use a similar approach to organize your React views

  <Switch>
      <Route exact path="/" component={Home} />
      <Route path="/movies/:id" component={Movie} />
      <Route path="/about" component={About} />
  </Switch>

**React Router**
Client-Server

HTTP & URI

HTML, JSON, ...

3-tier Architecture

MVC

Client (e.g. browser) → Internet → Site

Web Server (e.g. Apache, NGinx) → App. Server (e.g. NodeJS) → Database (e.g. Mongo, PostgreSQL)

Presentation Tier

Logic Tier

Routing & Controllers (e.g. Express) → Models (e.g. knex, objection)

Persistence Tier
Why a database?

What are the limitations of the memory-backed server (or what missing features do we want)?

```javascript
app.get('/api/movies/:id', (request, response) => {
    response.send(movies[request.params.id]);
});
```

- Efficient random access when total dataset is too large to fit in memory
- Fast and complex queries (not fast or complex)
- Model relationships within the data
- Transactions and other forms of fault tolerance
- Security (and management tools)
Database client and server

App. Server (e.g. NodeJS) 

Database (e.g. Mongo, PostgreSQL)

Persistence Tier

Message-based protocol (over TCP/IP, etc.)

Often separate server or process

Interface is typically SQL or custom DSL
# SQL vs. NoSQL

Really: Relational vs. Non-Relational

<table>
<thead>
<tr>
<th></th>
<th>Relational (RDBMS)</th>
<th>Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>Table-oriented</td>
<td>Document-oriented, key-value, graph-based, column-oriented, ...</td>
</tr>
<tr>
<td><strong>Schema</strong></td>
<td>Fixed schema</td>
<td>Dynamic schema</td>
</tr>
<tr>
<td><strong>Joins</strong></td>
<td>Used extensively</td>
<td>Used infrequently</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>SQL</td>
<td>Custom query language</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>ACID</td>
<td>CAP</td>
</tr>
</tbody>
</table>

```sql
SELECT * FROM people
WHERE age > 25;
```

```javascript
db.people.find(
    { age: { $gt: 25 } }
)
```
RDBMS mental model

Noun/Model, e.g. “Movie” ⇔ Table

Model attributes ⇔ Columns

### Genre

**Schema (name and type)**

**Movie table**

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>overview</th>
<th>release_date</th>
<th>poster_path</th>
<th>vote_average</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Star...</td>
<td>Princes...</td>
<td>1977-05-25</td>
<td>/tvSLB...</td>
<td>7.7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2001: A...</td>
<td>Huma...</td>
<td>1968-04-05</td>
<td>/90T7...</td>
<td>7.5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Primary key:** Unique identifier for record (can be 1+ columns)
RDBMS vocabulary

DB instance (e.g. PostreSQL)

Has 0+

Databases

Has 0+

Tables

Contains 0+

Rows

With 1+

Attributes/Columns

Index
Optimized lookup tables (e.g. tree) for specific columns

Cursor
Iterator into the result set that can obtain a few documents at a time

Each table has a schema with types, optional primary key, optional constraints
Example SQL statements

“Raw” queries

```
SELECT columns FROM table WHERE conditions;
INSERT INTO table(columns) VALUES (values);
UPDATE table SET column=value, ... WHERE conditions;
CREATE TABLE table (column Type, ...);
```

Our typical usage

SQL Query Builder (knex.js)

ORM (objection,js)

PostgreSQL
MySQL
sqlite

Schemas
Queries
Marshaling to JSON
Associations
Validation
Managing Schema: Migrations

Customer data is critical! How do you evolve your application without destroying any data?

• Maintain multiple databases (e.g. test, development, production, ...)
• Change schema/data with scripted *migrations*

**Migrations** create/delete tables, add/remove/modify columns, modify data, etc.

Advantage of migrations:
+ Track all changes made to DB
+ Manage with VCS
+ Repeatable
Validation (recall aspects & AOP)

Mechanisms for validating model data?
• Schema itself (unique, not null, etc.)
• Requirements specified in Model

```javascript
Movie.query().patchAndFetchById(..., { rating: 10 })
```
```
Movie.fromJson properties: {
  ...  
  rating: {
    type: ['integer', 'null'],
    minimum: 0,
    maximum: 5
  },
}
```
“When invalid movie data is sent in a request, then the requester should receive a '400 Bad Request' response”. Which of the below, if any, is **NOT** required to implement this scenario in a DRY manner?

A. Model validations  
B. Express middleware  
C. Code in one or more Express route handlers  
D. All of the above elements are actually required
**CRC cards to DB schema**

<table>
<thead>
<tr>
<th>User</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Knows user’s name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knows movies I rated</td>
<td>Rating</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Movie</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Knows its title</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knows its plot overview</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Know which genres it is</td>
<td>Genre</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Knows rating</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knows its owner</td>
<td>User</td>
</tr>
<tr>
<td></td>
<td>Knows its movie</td>
<td>Movie</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Genre</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Knows its descriptor</td>
<td></td>
</tr>
</tbody>
</table>
Two approaches to Movie ⇔ Genre

De-normalized Approach

<table>
<thead>
<tr>
<th>Movie table</th>
<th>Normalized Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Serialize multiple genres into attribute, e.g. “[63, 14]”

- Fewer tables and joins
- Variable sized records
- Trickier to search

Normalized Approach

<table>
<thead>
<tr>
<th>Movie table</th>
<th>Genre table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Foreign Key referencing Movie.id
Associations: “One-to-many” example

**Genre table**

<table>
<thead>
<tr>
<th>movielId</th>
<th>genreId</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>12</td>
<td>63</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

*Foreign Key referencing Movie.id*

GET /api/movies/11

Movie.query().findById(id).eager('[genres]')

SELECT `Movie`.* FROM `Movie` WHERE `Movie`.`id` = 11;
SELECT `Genre`.* FROM `Genre` WHERE `Genre`.movieId IN (11);

SELECT `Movie`.`id` AS `id`, ..., FROM `Movie` LEFT JOIN `Genre` AS `genres` ON `genres`.movieId = `Movie`.`id` WHERE `Movie`.`id` = 11

Fetch the movie *and* all of its genres
### Joins as filtered cartesian product

**Movie × Genre cartesian product**

<table>
<thead>
<tr>
<th>Movie.id</th>
<th>...</th>
<th>Genre.movielId</th>
<th>Genre.genreId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

Movie.id = Genre.movielId
Ratings: A “many-to-many” association

Foreign Keys and Primary Key

<table>
<thead>
<tr>
<th>moviId</th>
<th>userId</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>53</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Get a movie with it’s ratings?

GET /api/movies/12
Movie.query().findById(id).eager('[ratings]')

Create a new rating for a movie?

POST /api/ratings
Rating.query().insert({...})

Or from a movie
POST /api/movies/12/ratings
movie.$relatedQuery('ratings').insert({...})

Insert rating without either related model object (User or Movie)

Insert rating from Movie object
Where do the foreign keys go?

• “HasOne” / “BelongsToOne”
  Foreign key typically in the “BelongsToOne” side (although could be reversed)

• “HasMany” / “BelongsToOne”
  Foreign key in “BelongsToOne” side (the “many” model)

• “ManyToMany”
  Foreign keys in join model, e.g. Rating in “User and Movie through Rating”
Which of the following is the best migration (schema) for the Genre table in the Film Explorer? Note that `onDelete('CASCADE')` specifies that rows are deleted from the table if that corresponding row is deleted from the parent table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Schema</th>
</tr>
</thead>
</table>
| A.     | table.increments('id');
         | table.integer('movieId');
         | table.integer('genreId'); |
| B.     | table.integer('movieId');
         | table.integer('genreId');
         | table.primary([ 'movieId', 'genreId' ]); |
| C.     | table.integer('movieId')
         | .references('id')
         | .inTable('Movie');
         | table.integer('genreId');
         | table.primary([ 'movieId', 'genreId' ]); |
| D.     | table.integer('movieId')
         | .references('id')
         | .inTable('Movie')
         | .onDelete('CASCADE');
         | table.integer('genreId');
         | table.primary([ 'movieId', 'genreId' ]); |