

We previously introduced the role of the server: to provide persistence, a means for communicating between users and a secure environment (controlled by you) for operations that could/should not be performed by the untrusted client. Today we will focus of the first of those roles – persistence, that is saving data for future use. Although in doing so we will also touch on data integrity and other issues.

Why a database?

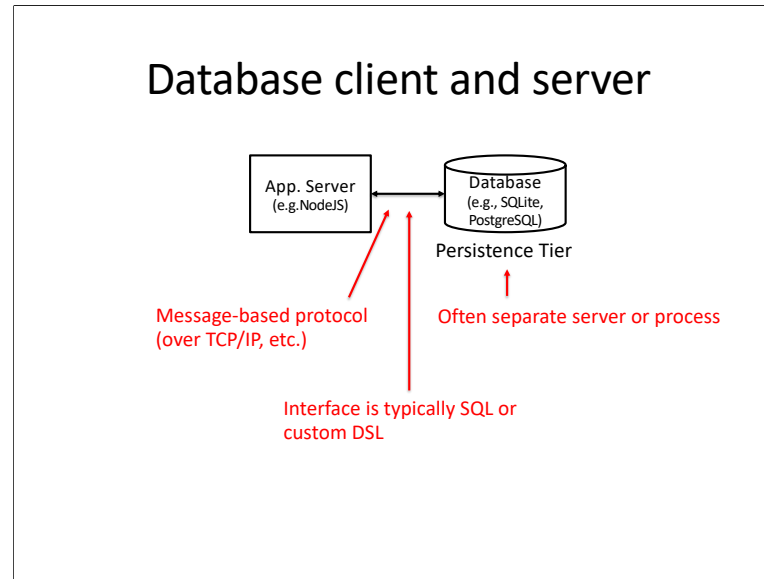
What are the limitations of the memory-backed server (or what missing features do we want)?

```
// pages/api/articles/[id].js
nc.get((request, response) => {
  response.status(200).json((articles.get(req.query.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)

Our memory backed server worked (and was nice and simple) but had a major limitation, all modifications were lost when we reloaded the server. That is obviously not something we can tolerate in a real application. But the in-memory approach also has many other limitations that make it a poor choice for any real application.

What are those limitations, or perhaps from a more optimistic perspective, what are benefits of using a real database system as the persistence tier for our application (other than that when restart out server/the database the data is still there!)




In the typical approach the database is its own server, and our application server (i.e., running on Node) communicates with the database server via TCP/IP or some other message-based protocol. That interaction occurs via SQL (a standardized language for querying relation databases) or a custom domain-specific language.

Although I should note that one of the databases we will use extensively, SQLite, is not a server. It is a library that runs inside the “client” process, accessing a database stored entirely within a single file. As an aside, SQLite is one of the most widely-used pieces of SW ever. It is embedded in many tools, e.g., web browsers, iOS, etc.,.


SQL vs. NoSQL

Really: Relational vs. Non-Relational

| | Relational (RDBMS) | Non-Relational |
|---------------------|--------------------|---|
| Data | Table-oriented | Document-oriented, key-value, graph-based, column-oriented, ... |
| Schema | Fixed schema | Dynamic schema |
| Joins | Used extensively | Used infrequently |
| Interface | SQL | Custom query language |
| Transactions | ACID | CAP |



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



```
db.people.find(
  { age: { $gt: 25 } }
)
```

I just mentioned the term “relational database”. That is a term for a particular class of database management tools, it also sometimes referred to via “SQL”, which is really the name of the query language used by relational DBs. The alternative is often called “NoSQL”, or more appropriately “non-relational” databases. An example of the latter would be Google’s Firebase.

One of the decisions we will need to make in our project is what kind of database to use. Like many decisions we encounter in class there is no right answer – although the entire Internet will have an opinion – just tradeoffs. From my perspective, NoSQL is more flexible, and perhaps easier to get started, but the flexibility begets challenges in managing your data. In contrast, relational databases have a slightly steeper learning curve but force us to organize our data in helpful ways. A relevant analogy might be a statically-typed languages like Java (relational) vs. dynamic languages like Python (non-relational). The latter is quick to get going but is susceptible to type errors that are not possible in Java...

Before we can pick a database, however, we need to figure out how the data in our application is structured (i.e., determine the data model as discussed previously) and only then can we pick a database that make sense for our application. From my perspective, a well-designed data model is more important than the choice of database system.

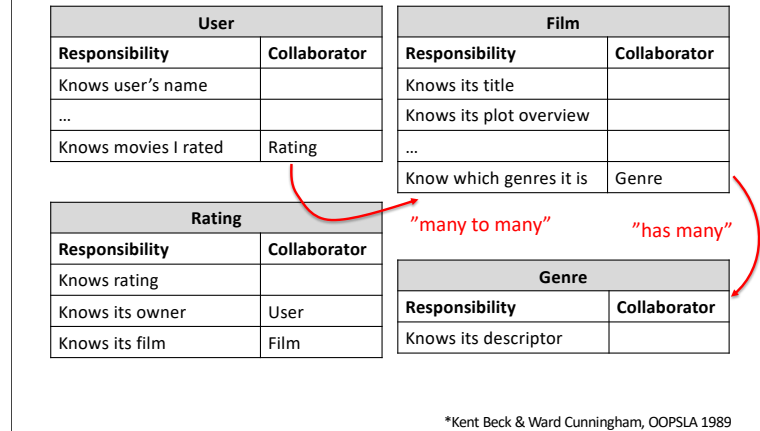
Glossary:

SQL: Structured Query Language

ACID: ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.

CAP: Two of consistency (most recent data), availability, and partition tolerance.

Recall: Film Explorer CRC cards



Recall CRC cards are like user stories, but for classes. Each index card contains:

- On top of the card, the class name
- On the left, the responsibilities of the class, i.e., what this class "knows" and "does". For example, a "car" class may know how many seats and doors it has and could "do" things like stop and go.
- On the right, the collaborators (other classes) with which this class interacts to fulfill its responsibilities

Like User Stories, using an index card limits complexity and helps designers focus on the essentials of the system. The CRC cards guide the design of our models and database schema.

The "knows" are going to become the fields that we store in our database for each model and the knows/collaborators define the relationships or associations between those models. Recall that:

- A film has a one-to-many relationship with genres (i.e., film "has many" genres)
- There is a many-to-many relationship between Users and Films via the ratings. Often called a "has many-through" association.

These terms are semi-formal (note different tools use slightly different names, but the concepts are the same), that is they map directly to the design of the database

schema. We are effectively designing database tables as we work out these relations. That said, I encourage you to approach the data modeling from this “direction”, that is start by modeling the nouns in your application (and their relationships) then choose and design your database instead of starting with the database design (and not try to go the other way, that is start with the database, then develop the data model).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

Thinking in relations/associations

- “HasOne” / “BelongsToOne”
One-to-one relationship, e.g., Supplier and Account
- “HasMany” / “BelongsToOne”
One-to-many relationship, e.g., Film and Genre
- “ManyToMany”
Many-to-many relationship (often called “has many through”), e.g., User and Film through Rating

The relations/associations you will typically encounter are listed here (again a quick reminder that different tools will use slightly different terminology, but the concepts are the same). We can think of these associations as design patterns that will enable us to utilize libraries/frameworks for the “parts that are the same every time”, i.e., automatic validations, optimized queries and more.

A note: the “through” modifier can be applied to “HasMany” as well, and typically implies that you might want to work with the “through” noun independently of the two sides of the relationship. We still would want to identify the relationship as one-to-many to take advantage of built-in validations.

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"

Answer: B

A customer can have many animals (pets), but each animal is presumably owned by a single customer. Although we could imagine situations though where C might be needed... What would such an example be? Multiple customers were owners/responsible parties for a pet.

We could imagine there is data associated with the specific Customer-Animal relation (e.g., insurance), however that each association may be its own entity doesn't itself change that it is a one-to-many relation.

True or False? Two models can only have one relation.

- A. True
- B. False

Answer: False

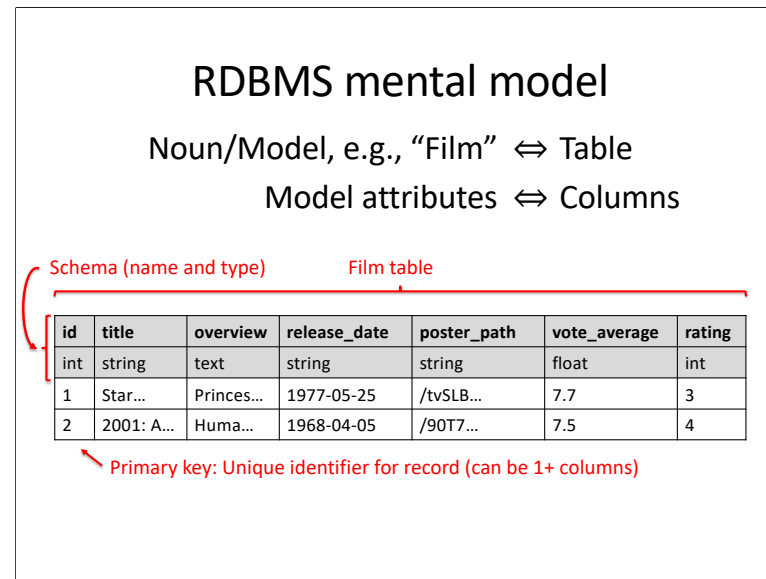
Consider an application where a user can make and like comments. A user has many comments via posting (a has-many relation), and user also has many comments via liking (a many-to-many or has-many-through relation).

Interlude: RESTful URLs for associations

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

| Route | Controller Action |
|---------------------------------------|---|
| POST /api/films/:film_id/genres | Create new genre associated with film with id == :film_id |
| GET /api/films/:film_id/genres/:id | Read genre with id == :id associated with film with id == :film_id |
| PUT /api/films/:film_id/genres/:id | Update genre with id == :id associated with film with id == :film_id from request |
| DELETE /api/films/:film_id/genres/:id | Delete genre with id == :id associated with film with id == :film_id |
| GET /api/films/:film_id/genres | Read genres of film with id == :film_id |

The associations can be directly translated to URLs. Here the nesting implies genres associated with a specific film. In theory we can go infinitely deep, in practice we shouldn't go more than one or two levels, otherwise it gets unwieldy.



With our CRC cards we focused on modeling our data independent of how it is stored. We will now implement those models using a relational database. Our mental model is a table (e.g., a spreadsheet table). The attributes/columns are typically the “knows” in your CRC cards, that is the schema is nearly direct translation of the CRC card.

Primary key is a unique identifier for a record (that should be not be reused). Often it is an arbitrary (auto-incrementing) integer, e.g., the “id” in Simplepedia, but does not need to be (and it can even be a composite of multiple columns) as long as it is unique. Schema includes type (storage size) and can further include indexes (think hash tables or trees) to speed up queries and other constraints, like not null.

RDBMS vocabulary

DB instance (e.g., PostgreSQL)

Has 0+

Databases

Has 0+

Tables

Contains 0+

Rows

With 1+

Attributes/Columns

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

Index

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

Cursor

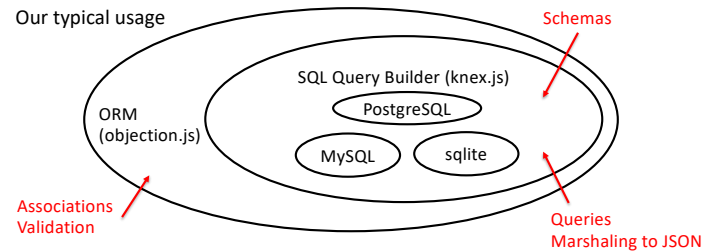
Iterator into the result set
that can obtain a few
documents at a time

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



We generally won't write "raw" queries, instead we will use the knex.js query builder to abstract DB-specific differences, handle "safe" parameters substitution, etc.. We will further wrap knex with an ORM library (Object Relational Mapping) that provides an object-oriented interface to our database (stay tuned).

Managing Schema: Migrations

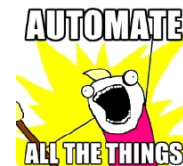
Customer data is critical! How do you evolve your application without destroying any data?

- 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



Migrations are the answer to how we smartly evolve our database schema at all stages in our application lifecycle, from creating the initial database schema to safely evolving the database to add features to our production application (which presumably has customer data in it). While we could modify our database manually. We won't. Instead, instead we define a series of migrations scripts that evolve the schema from an empty database to the desired state.

Each migration has two parts, an “up” function that makes the desired changes, e.g., creating a table, adding column, etc. and the a “down” function that reverts those changes. Performing the up function and then the down function should return the database to its prior state. Each migration is incremental, that is makes the “next” set of changes to the prior database/schema data. For example, if you add a feature that needs a new column in an existing table, we create a migration that adds that column (and sets an appropriate value for existing entries).

Migrations are a key part of our “overall” DevOps approach.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

Frequent error: Migrations are tracked by date-time

migrations/20190424165216_users_and_articles.js

20190424165216
Date and time for this migration

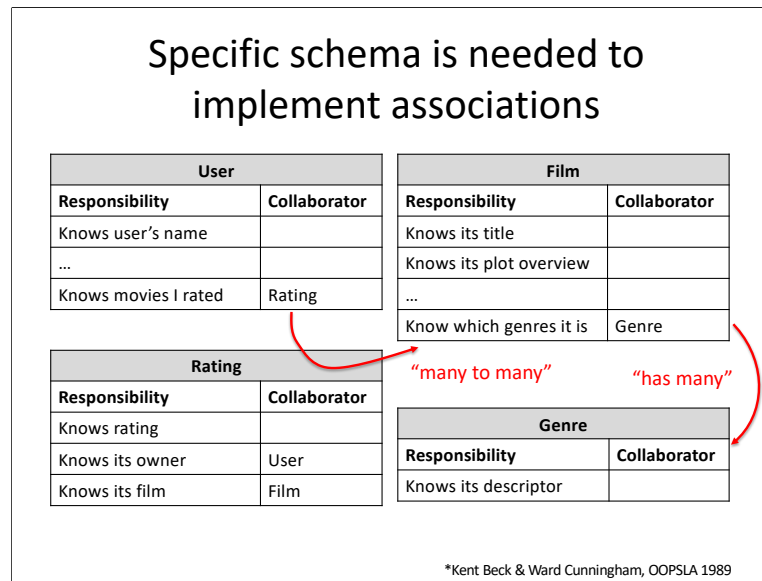
Knex et al. apply migrations in *date-time order* and *track the last migration applied*

- Applying migrations multiple times won't have any effect
- Modifying and re-applying a migration won't have any effect

If you modify a migration, rollback then reapply

Why does it work this way? The goal for migrations is to enable to you to evolve a database that is in use, that is has customer data that you don't want to lose. Thus, we don't want to "double apply" the changes in a migration. And there isn't the expectation that we would go back and modify already implied migrations because that would invalidate the data.

Modifying a migration is common during development. If you do so, either delete the database and rebuild from scratch (easy with SQLite – just delete the file - less so with other RDBMSs) or more robustly, rollback the relevant migrations (invoking the "down" operations) then reapply after the making the edit.



These associations have specific schema associated with them. That is the association will determine what columns we need in our database. Specifically ...

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

Two approaches to Film ⇔ Genre

De-normalized Approach

| Film table | | |
|------------|-----|--------|
| id | ... | genres |
| 1 | | |
| 2 | | |
| 3 | | |

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

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

Normalized Approach

| Film table | | Genre table | |
|------------|-----|-------------|---------|
| id | ... | filmid | genreid |
| 1 | | 1 | 63 |
| 2 | | 1 | 14 |
| 3 | | 2 | 14 |

Foreign Key referencing
Film.id links tables

The first approach is what we would implement in a memory backed server (and most NoSQL DBs). All the data for film, including its one or more genres are packed together in a single, albeit variable sized, object. No additional data is required. In contrast a normalized approach breaks the film into two fixed size parts, the film itself and separate genre entries, that are linked together via foreign keys, that is we need to include additional columns/attributes - the filmid column - to create the connection between the two tables.

Normalization: Eliminating repeated information in a table by decomposing repeating entries into separate linked tables. The data is reconstructed via join operations (to come...)

Specifically, "first normal form" enforces these criteria: 1) Eliminate repeating groups in individual tables, 2) Create a separate table for each set of related data, 3) Identify each set of related data with a primary key

The second, normalized, approach is typically used with RDBMS (these are the relations in the name).

Foreign keys enforce the connection between two tables. A foreign key is a constraint not a type. To insert an entry into Genre, there must be a corresponding entry in the Film (it will fail if there is no Film). Ensuring that all foreign key references are valid is

termed maintaining “referential integrity”, and is one of the benefits of RDBMs.

Associations: “One-to-many” example

GET /api/films/11

↓ Desired response

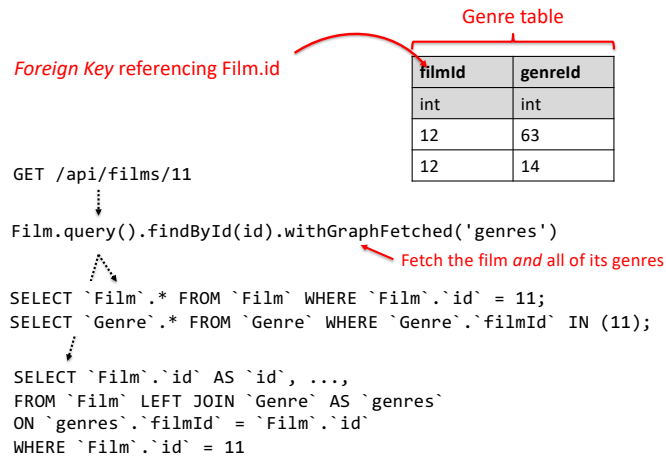
```
{
  "id": 11,
  "title": "Star Wars: Episode IV - A New Hope",
  ...
  "genres": [{ "filmId": 11, "genreId": 12 } ...]
}
```

From Film table

From Genre table

When we implement this route to on the film explorer server we want to obtain all the data for this film, including its genres. But now that data is separate. How do we combine the data from the two tables?

Associations: “One-to-many” example



Here we see two approaches to construct the entire object returned by this route:

1. The multiple queries approach first gets the film, then the associated genres.
2. The join approach conceptually builds a table with a row for each combination of movie and genre (with both sets of columns and duplicated movie entries) and then filters that table according to the join conditions and where clause, etc.

The former requires more queries (latency) but it is simpler to parse into tree of Objects. The latter is one (complex) query but parsing results into objects will be a little more involved.

In practice we will not implement these queries directly. Instead, we will use the (Object.js) ORM to create the queries for us. Here we are telling Object.js to eagerly, as opposed to lazily, fetch and populate the genres. The ORM uses the associations defined in the model class to generate the appropriate query and construct the final object.

Joins as filtered cartesian product

Film × Genre cartesian product

| Film.id | ... | Genre.filmId | Genre.genreId |
|---------|-----|--------------|---------------|
| 1 | | 1 | 63 |
| 2 | | 1 | 63 |
| 3 | | 1 | 63 |
| 1 | | 1 | 14 |
| 2 | | 1 | 14 |
| 3 | | 1 | 14 |
| 1 | | 2 | 14 |
| 2 | | 2 | 14 |
| 3 | | 2 | 14 |

Film.id == Genre.filmId


Joins are such a key feature of an RDBMS I want to briefly expand on what is going on behind the scenes. Our mental model for joins is a filtered cartesian product. That is the database system is creating all combinations of entries from the Film table and the Genres table and then only keeping those where the join criteria, in this case that Film.id == Genre.filmId, is true (the actual implementation is more efficient than that though...).

Interlude: Refining server responses

```
GET /api/films/11
  ↓ Desired response
{
  "id":11,
  "title":"Star Wars: Episode IV - A New Hope",
  ...
  "genres":[{"filmId": 11, "genreId": 12 } ...]
}

const { genres, ...film } = ...;
response.send({
  ...film,
  genre_ids: genres.map(g => g.genreId)
});

{
  "id":11,
  "title":"Star ...",
  ...
  "genre_ids":[12,28,878]
}
```



Our previous response was the direct output produced by Objection.js as it joined the Film and its genres. In many cases that is exactly what we want. But we notice it contains lots of extraneous data (i.e., repeats of the filmId). We could imagine want to strip that out, or otherwise modify the response. One place to do is in the route handler, API handler. Keep in mind that is just JavaScript code and so we can execute other code, like transforming the genres. If we wanted to do this every time with your model, we are best off integrating that transformation into the model itself..

Ratings: A “many-to-many” association

Foreign Keys and Primary Key

| filmId | userId | rating |
|--------|--------|--------|
| int | int | int |
| 12 | 4 | 2 |
| 53 | 4 | 3 |

Rating
"Join Table"

Get a movie with its ratings?

```
GET /api/films/12
Film.query().findById(id).withGraphFetched('ratings')
```

Create a new rating for a movie?

```
POST /api/ratings
Rating.query().insert({...}) ← Insert rating without either related
Or from a movie                model object (User or Film)
POST /api/films/12/ratings
movie.$relatedQuery('ratings').insert({...})
                                ← Insert rating from Film object
```

The last assumes we have obtained the movie and the user already in the handler, e.g. via `fetchById`.

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 Film through Rating”

These are established designs for these relations, that is once you define the relation we know where the keys need to go. We don’t need to figure that out every time.

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.

| | |
|---|--|
| <p>A.</p> <pre>table.increments('id'); table.integer('filmId'); table.integer('genreId');</pre> | <p>B.</p> <pre>table.integer('filmId'); table.integer('genreId'); table.primary(['filmId', 'genreId']);</pre> |
| <p>C.</p> <pre>table.integer('filmId') .references('id') .inTable('Film'); table.integer('genreId'); table.primary(['filmId', 'genreId']);</pre> | <p>D.</p> <pre>table.integer('filmId') .references('id') .inTable('Film') .onDelete('CASCADE'); table.integer('genreId'); table.primary(['filmId', 'genreId']);</pre> |

Answer: D

Answers A & B are missing foreign key constraints and thus will not enforce that a genre entry must be associated with a valid film. When a Film is a deleted, we want to delete all of its genre entries, onDelete('CASCADE') does that. Answer C/3 could work but would leave orphan entries in the Genre table if a film is deleted.

Film model (M in MVC)

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

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

```
class Film extends Model {
  static get tableName() {
    return 'Film';
  }
  ...
  static get relationMappings() {
    ...
  }
}
```



Film table in database

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |

ORM is a design pattern for mapping database schema to object

In the Film Explorer application, the model is a Film. In our initial practicals, there is no explicit model class, just a plain old JavaScript object (POJO) representing the records in the table. Depending on the application we might not need much more. But as we saw already, Film Explorer could and does benefit from established design patterns and built-in functionality offered by an ORM library (Object Relational Mapping). ORMs are a design pattern for mapping database schema to an object whose methods/properties correspond to attributes in DB, DB queries, etc.

We already saw use of the ORM model to express and implement associations between models (the eager withGraphFetched), the other features are:

Validations: Exactly what they sound like, example of Aspect-oriented programming (i.e., these validations are relevant everywhere the model is created/used. Instead re-implementing that code, we do it once).

Virtual attributes: Convenience “attributes” derived from actual attributes/columns in DB.

Just how different of a database can a given ORM support? Some... Across different RDMSs, e.g., sqlite, MySQL and PostgreSQL. Yes. Relational vs. Non-relational? No.

Validation (recall aspects & AOP)

Mechanisms for validating model data?

- Schema itself (unique, not null, etc.)
- Requirements specified in Model

```
Film.query().patchAndFetchById(..., { rating: 10 })
    ↓
Film.fromJson ← ..... properties: {           Model Schema
    ↓
throw ValidationError                          rating: {
    ↓                                           type: ['integer', 'null'],
                                           minimum: 0,
                                           maximum: 5
                                           },
400 Bad Request                                }
```

Schema alone, i.e., type, is insufficient to enforce the range. Note that any constraint could be implemented in this way (not just range).

“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 *AoP*
- B. NextJS/next-connect middleware
- C. Code in one or more NextJS API handlers
- D. All the above elements are required

Answer: C

No specific code should be required in the route handlers. A validation error should result in a rejected Promise, that rejected promise could be detected by the NextJS invoke the error handling middleware.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.