Databases

- Association ≙ connection between two values in a db
- Cartesian product ≙ every combination of everything
  - Join to specify a key to combine on
- PK ≙ unique col in a table that identifies the rows
- Foreign Key (FK) ≙ key in one table that refers to the primary key of another
- Join ≙ queries that combine records from 2 or more tables using PKs and FKs

ActiveRecord Associations

- Allows manipulating DB-managed associations more easily
- After setting things up correctly, you don't have to worry (much) about keys and joins
- eg:

```
class Movie < ActiveRecord::Base
    has_many :reviews
end
class Review < ActiveRecord::Base
    belongs_to :movie
end
```

- `belongs_to` ≙ owning relationship; a review owns the foreign key to the movie
- eg migration:

```
class AddReviews < ActiveRecord::Migration
    def self.up
        create_table :reviews do |t|
            t.integer 'potatoes'
            t.references 'movie'
            t.references 'movigoer'
        end
    end
end
```

- Rails infers the model from the column name
- `references` automatically turns `movie` into `movie_id` in the DB
- You can automatically create reviews from a diff controller eg: `@movie.reviews.build(potatoes: 5)` or `@movie.reviews.create(potatoes: 5)`
  - Can also do `@movie.reviews << @new_review` to append it

Using Associations

- Models must have attribute for FK of owning object
  - ActiveRecord manages this field in both db and in-memory AR object
- Adding a one-to-many association
  1. Add `has_many` to owning model and `belongs_to` to owned model
  2. Create migration to add foreign key to owned side that references owning side
  3. Apply migration

Through-Associations

- For Many-to-many Associations
- Cannot use `has_many` and `belongs_to`
- Solution: Create a new to model the multiple associations
- eg: add a `movigoer_id` to a `review` so that it has foreign keys in `moviegoer` and `movies` separately
  - Still have `belongs_to` relations

- eg for moviegoer: `has_many :movies, through: :reviews`
- eg for movie: `has_many :moviegoers, through: :reviews`
- Enables `@user.movies` and `@movie.users`

RESTful Routes for Associations

- Nested routes:

```
resources :movies do
    resources :reviews
end
```

- Says that there will be a movie id and review id for the routes

Helper method	RESTful Route and action	
movie_reviews_path(m)	GET /movies/:movie_id/reviews	index
movie_review_path(m)	POST /movies/:movie_id/reviews	create
new_movie_review_path(m)	GET /movies/:movie_id/reviews/new	new
edit_movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id/edit	edit
movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id	show
movie_review_path(m,r)	PUT /movies/:movie_id/reviews/:id	update
movie_review_path(m,r)	DELETE /movies/:movie_id/reviews/:id	destroy

Figure 1: Screenshot_2023-10-02_at_4.22.02_PM.png

-
- Note: in the controller, you must use `movie_id` instead of just `id`
  - Generally want to make sure the movie exists before making a new review
- The `form_with` param `url` can take in a list of objects now such as `[movie, review]`
  - Same thing for `redirect_to`
- The `form_for` takes in the url such as `movie_review_path(@movie, @review)`
  - The inner nested resource defines the model to direct to

Referential Integrity

- If you delete a movie with reviews, `movie_id` field of those reviews then refers to nonexistent PK
- You can automatically delete them with `has_many :reviews, dependent: :destroy`
- You can automatically 'orphan' (no owner) with `has_many :reviews, dependent: :nullify`
  - Warning: `@review.movie` may no longer exist
- You can also do lifecycle callbacks to do other things (eg: merging)
- Make sure to test for this

Other Topics (Not on Exam)

- Single-Table Inheritance (STI) $\triangleq$ table joined with self
- Polymorphic Associations $\triangleq$ table owned by multiple models
- Self-referential eg: `has_many :through`
- Many declarative options on manipulating associations (like validations)

MVC Review

- Minimize code in reviews
- Minimize fat controller methods (move to model if possible)
  - Factor out code when possible (keep main classes slim)
    - Presenter (view object)
    - Value object
    - Adapter/Decorator class, Helper modules
  - DRY out code
    - Concerns (set of simple modules that live in `app/concerns`)
    - Automations (eg: create fake staging data)
      - Usually in `lib/tasks`
  - Encapsulate coupling among models
    - Service object (form object, query object)
    - Policy object: who's allowed to do what

DRYing Out Queries with Reusable Scopes

- Scopes can be stacked $\triangleq$ `Movie.for_kids.with_good_reviews(3)`
  - Evaluated lazily
- eg:

```
class Movie < ApplicationRecord
    has_many :reviews
    scope :for_kids, -> { where(rating: ['G', 'PG']) }
```