DRYing your Rails app

Code reuse the Rails way

Based on Ch. 5 of the "Engineering Software as a Service" book by Fox & Patterson

April 26, 2016. Seminar given during the course MAC5714 — Object Oriented Programming (IME-USP).
Débora Setton, Hans Bejar, Thiago Petrone.

First things first

- **This presentation is available at:**

  http://github.com/deborasetton/rottenpotatoes-rails/blob/master/docs/drying-rails.pdf

  (OR: `https://goo.gl/0CrGO9`)

- We'll use the **Rotten Potatoes** application to give examples (a domain everyone is familiar with from the homeworks).

- **All Rotten Potatoes examples are available on GitHub:**

  http://github.com/deborasetton/rottenpotatoes-rails/commits/master

- Examples are identified throughout the slides with GitHub code ▶

- Interruptions are welcome at any time

Now that the details are taken care of...

Let's talk about code reuse!

Code reuse mechanisms for each layer

		View	**1.** Partials
View	+	**M**odel	**2.** Validations
		Controller	**3.** Filters
Controller	+	**M**odel	**4.** Third-party authentication
		Model	**5.** Associations
		Controller	**6.** RESTful routes for associations
		Model	**7.** Scopes

We'll cover the **WHAT**, **WHEN** and **HOW** for each

1. Partials

View

1. Partials

View

WHAT: fragments of view code (Haml, ERB) that live in their own files and are included in views using the `render` method.

WHEN: the same code is being copy-pasted into two or more views with minor or no differences.

Rendered from: `new.html.haml`

Create new movie

Title:

Country:

Director:

Release date:

Rating: *Select...*

Save

Rendered from: `edit.html.haml`

Editing: "Pulp Fiction"

Title: *Pulp Fiction*

Country: *US*

Director: *Quentin Tarantino*

Release date: *1994*

Rating: *R*

Update

1. Partials

View

WHAT: fragments of view code (Haml, ERB) that live in their own files and are included in views using the `render` method.

WHEN: the same code is being copy-pasted into two or more views with minor or no differences.

Rendered from: `new.html.haml`

Create new movie

Title:

Country:

Director:

Release date:

Rating: *Select...*

Rendered from: `edit.html.haml`

Editing: "Pulp Fiction"

Title: *Pulp Fiction*

Country: *US*

Director: *Quentin Tarantino*

Release date: *1994*

Rating: *R*

Update

DUPLICATION

1. Partials

GitHub code ▶ View

HOW:

1. Create a file under `views` for the partial. The file name must start with an underscore, e.g., `_form.html.haml`.
2. Extract the duplicated code into this partial and make the necessary adjustments (if any).
3. In the original views, replace the old code with a call to the `render` method.

new.html.haml

```
%h1 Create new movie
= render partial: 'form'
= submit_tag 'Save'
```

edit.html.haml

```
%h1 Editing: #{@movie.title}
= render partial: 'form'
= submit_tag 'Update'
```

_form.html.haml

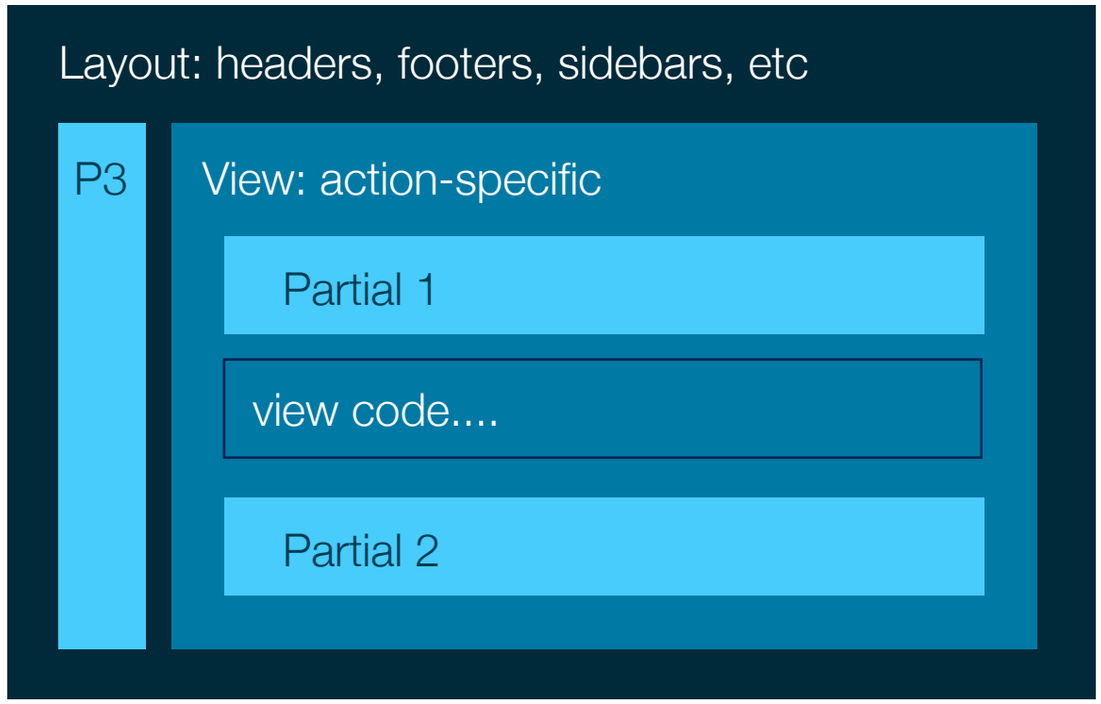
```
form fields
  go here
```

Views

Partial

MORE ABOUT PARTIALS:

- Any chunk of view code can be turned into a partial.
- Views can include multiple partials.
- `render` is **not** equivalent to an `#include` in C. Views and partials are processed in different scopes.
- Partials != Layouts.



2. Validations

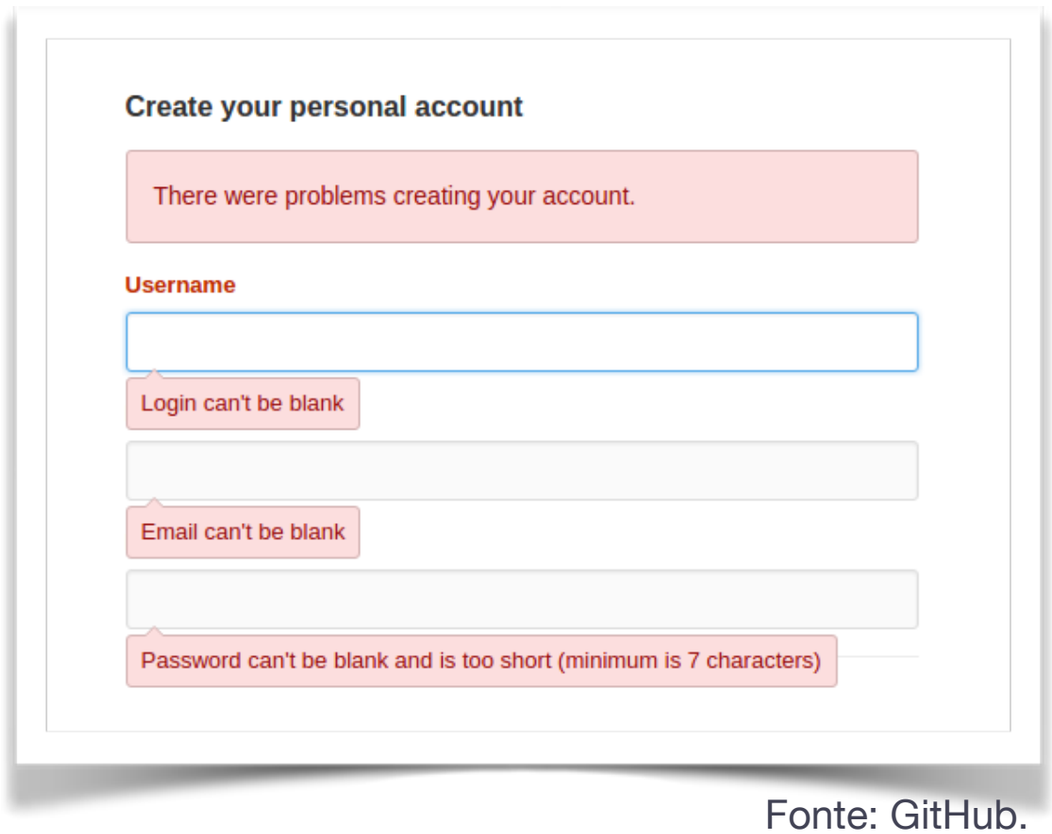
Model + **V**iew

2. Validations

Model

WHEN:

- You want to make sure **persisted data will always be valid** according to your application's rules.
  - Usernames must contain only alphanumeric characters
  - Age must be a number >= 18
- You want to display user-friendly error messages

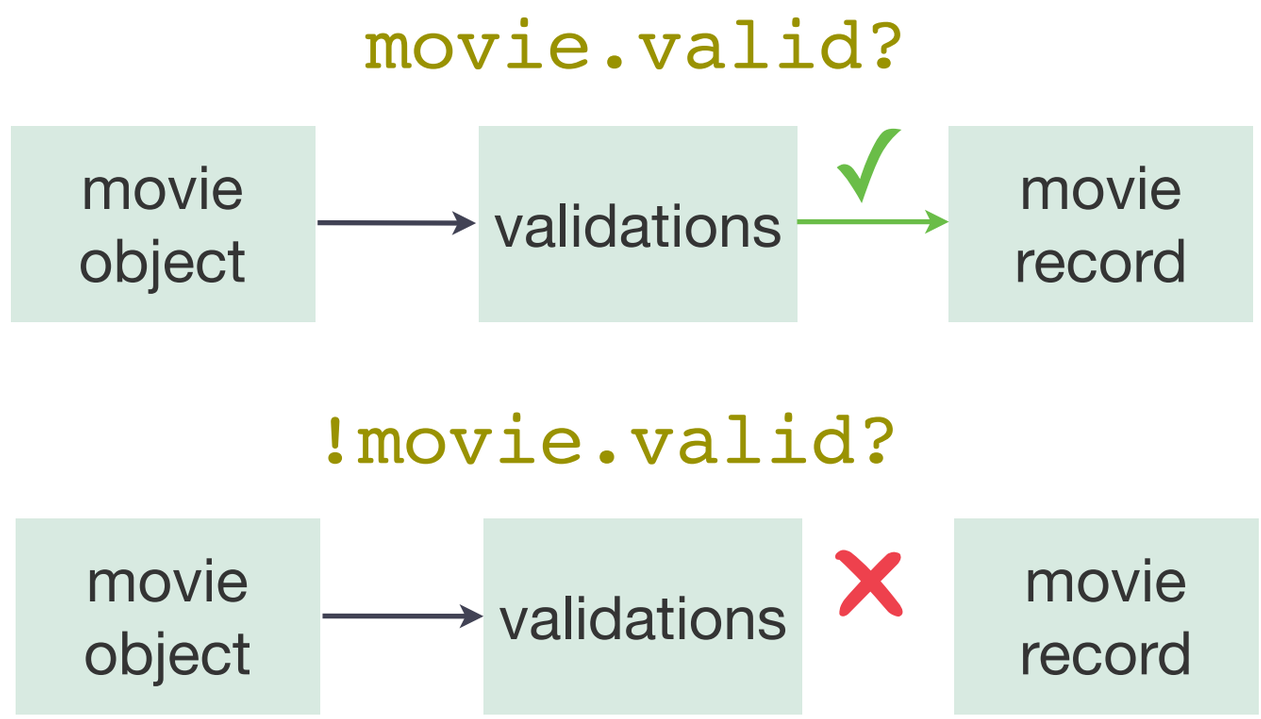


Fonte: GitHub.

2. Validations

WHAT:

- A collection of pre- or custom-defined methods added to model classes. These methods will be **automagically called by the framework** before data is persisted in the database.
- A technique borrowed from aspect-oriented programming to execute code in specific points of the application without explicitly invoking it.



2. Validations

GitHub code ▶ **M**odel

HOW:

- In your models: use built-in validation helpers or define your own.

```ruby
class User < ActiveRecord::Base
  # Built-in validations helpers.
  validates :name, presence: true
  validates :age, numericality: { greater_than: 18 }

  # Custom validation method.
  validate  :username_is_cool

  def username_is_cool
    errors.add(:username, "is not cool") unless username.cool?
  end
end
```

- In your views, use the `errors` object.

```ruby
@movie.errors.full_messages
@movie.errors[:title]
```

2. Validations

When are validations called?

Need to know **the lifecycle of Active Record objects.**

```
@movie = Movie.where(
  title: 'Kill Bill').first
```

↓

```
@movie.title = nil
```

↓

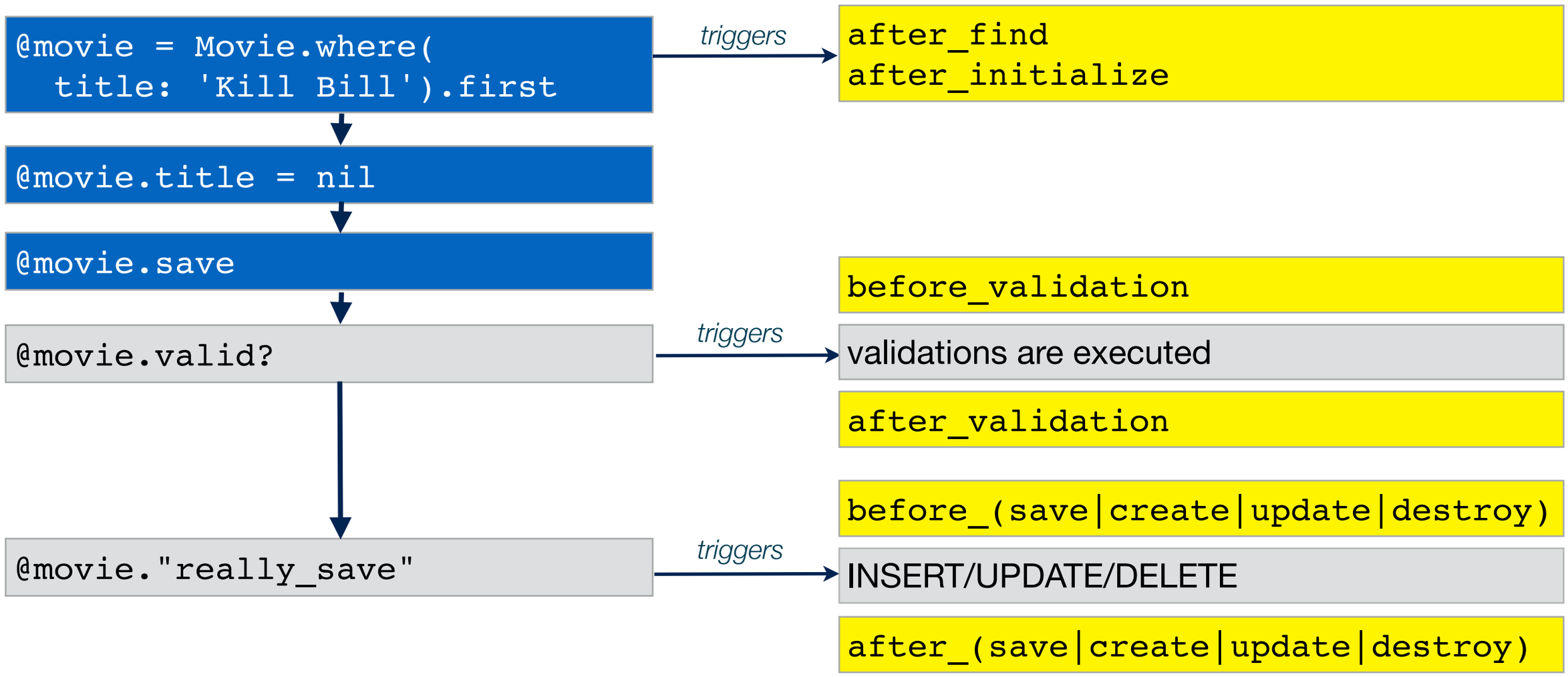
```
@movie.save!
```

```
ActiveRecord::RecordInvalid:
Validation failed: Title can't
be blank
```

2. Validations

When are validations called?

Need to know **the lifecycle of Active Record objects.**



3. Filters

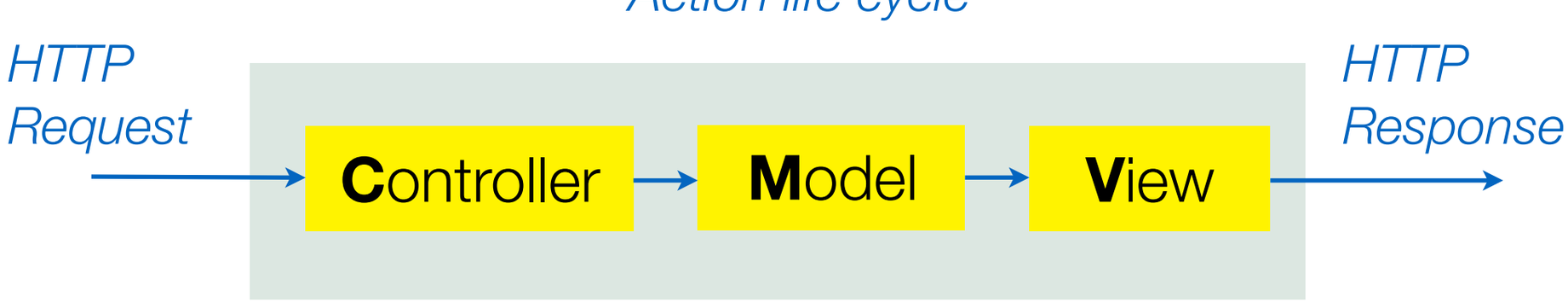
Controller

3. Filters

WHAT:

- Analogous to ActiveRecord **callbacks**, but for controllers: code that will be executed without you calling it directly in specific points of the **request life cycle**.

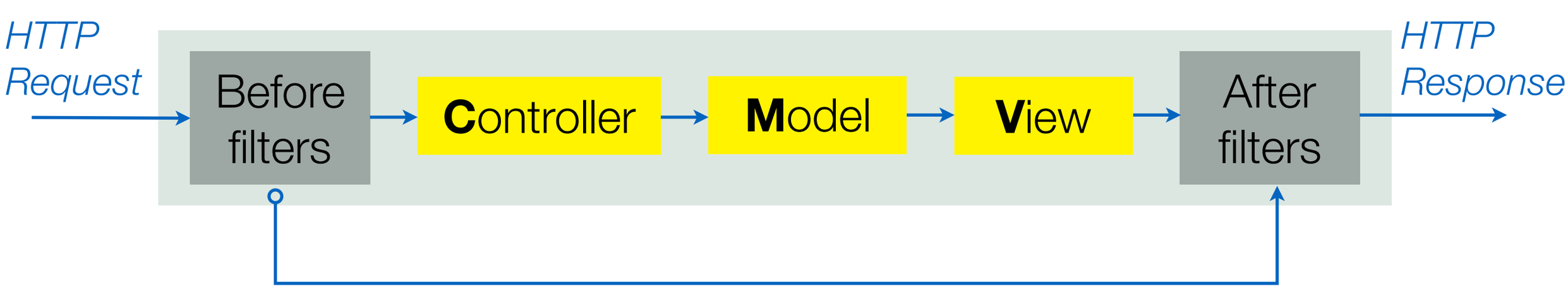
Action life cycle



WHAT *(continued)*:

- Analogous to ActiveRecord **callbacks**, but for controllers: code that will be executed without you calling it directly in specific points of the **request life cycle**.

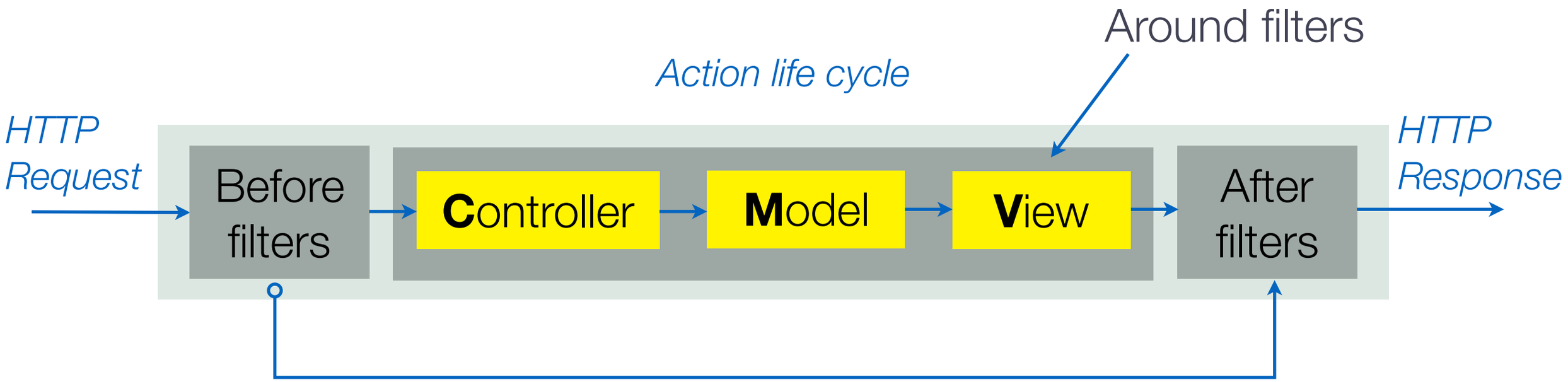
Action life cycle



You can prevent the entire action from happening!

WHAT *(continued)*:

- Analogous to ActiveRecord **callbacks**, but for controllers: code that will be executed without you calling it directly in specific points of the **request life cycle**.



You can prevent the entire action from happening!

Around filters: calling the action is your responsibility.

WHEN:

- You need to **check whether some conditions are true** before allowing an action to happen—e.g., user must be logged in.
- You need to **setup variables** for the action and views—e.g., setup the `@user` variable from session information.
- You need to **log information** about an action—e.g., "action 'show' took 400ms to run".

HOW:

```ruby
class ApplicationController
  before_action :authenticate
  around_action :log_request_duration

  protected

  def log_request_duration
    start = Time.now
    yield # Do the action
    Rails.logger.info("Duration: #{Time.now - start}")
  end

  def authenticate
    @user = User.where(id: session[:user_id]).first

    if @user.nil?
      redirect_to :login_path
    end
  end
end
```

- Both filters will always be called, for **every action** in the application, unless a `skip_*` directive is used.
- Multiple filters can be called for a single action.

4. Third-party authentication

Model + **C**ontroller

4. Third-party authentication

Model | Controller

WHAT: when credentials established and verified by an unrelated party, called an **identity provider**, are used to identify and sign in users to your application.

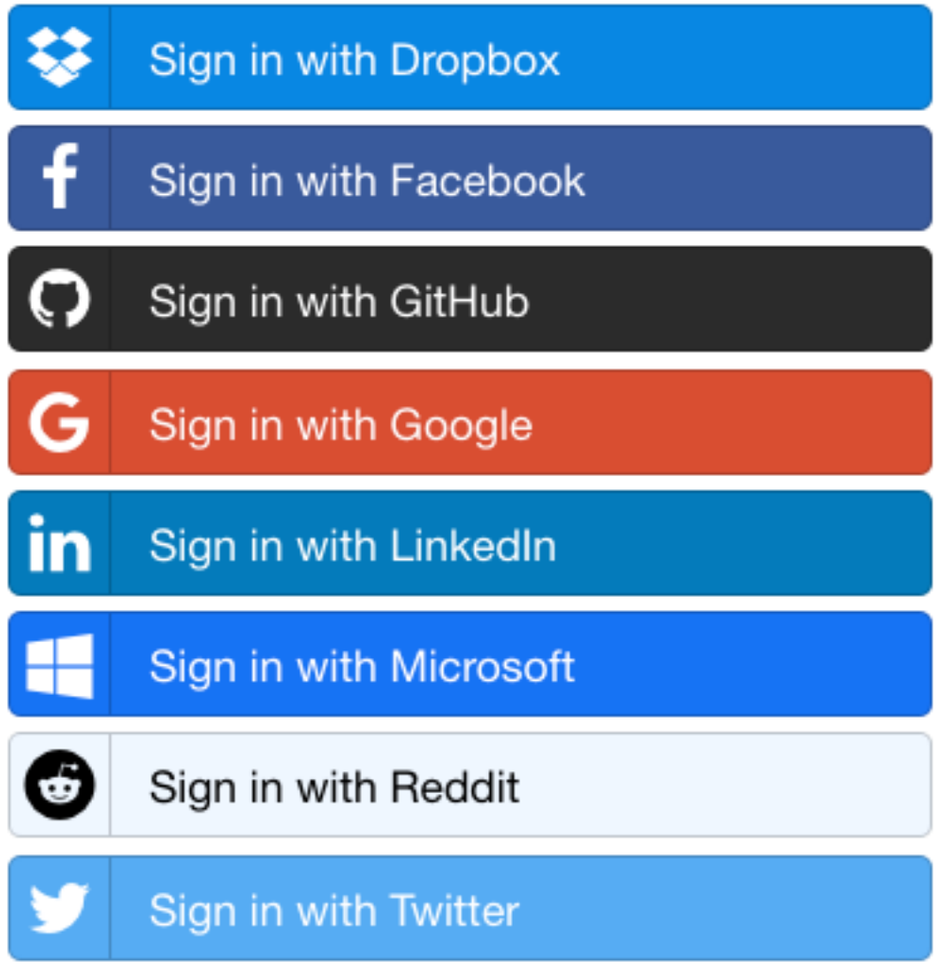


Image source: https://lipis.github.io/bootstrap-social/

4. Third-party authentication

Model | Controller

WHAT: when credentials established and verified by an unrelated party, called an **identity provider**, are used to identify and sign in users to your application.

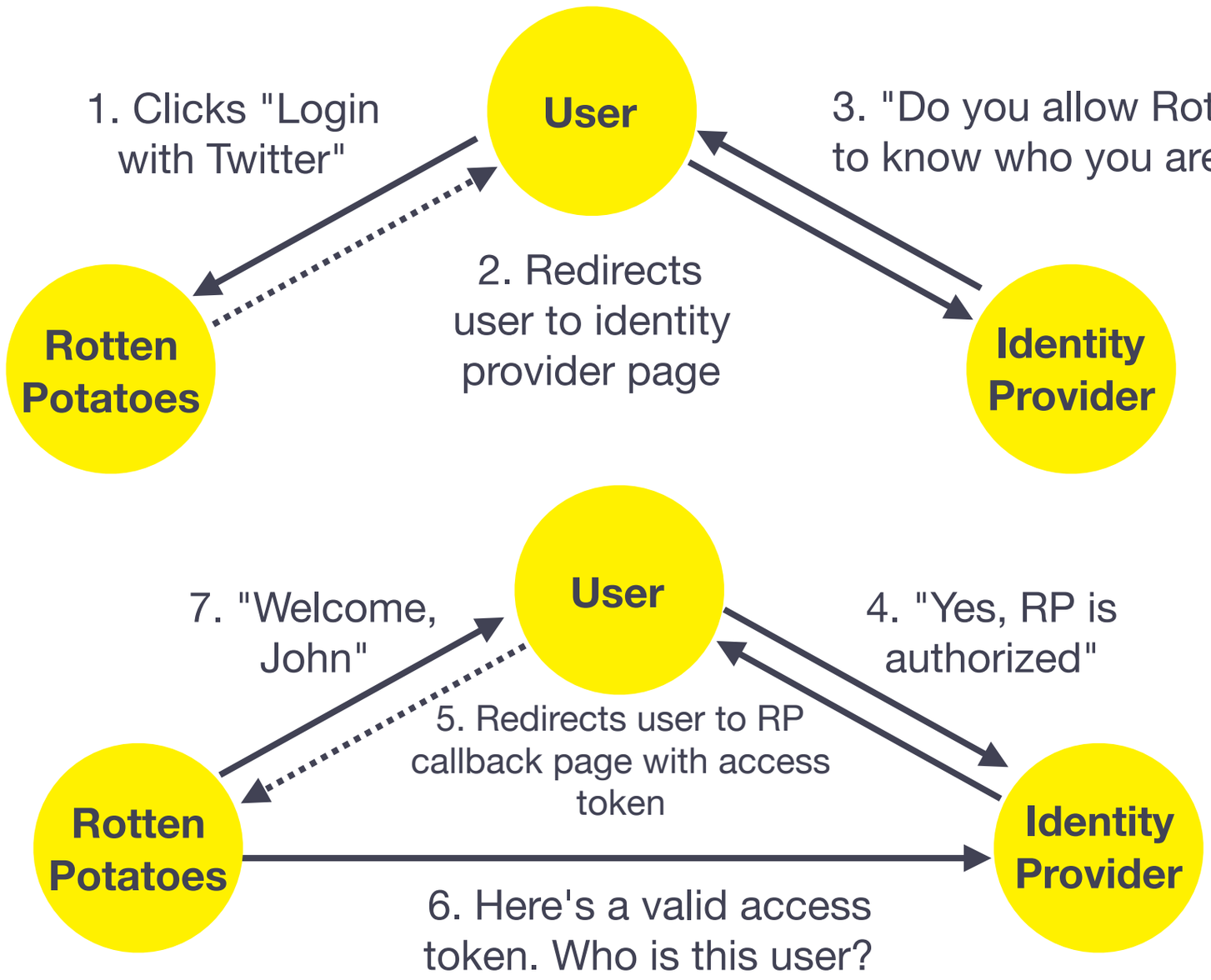


Diagram adapted from the SaaS book.

4. Third-party authentication

Model **C**ontroller

WHEN:

- You don't want to roll your own authentication solution, i.e., you want to **reuse verified identities** and keep your app DRY.
- Your application requires user authentication; and/or
- You want to perform actions on behalf of your users—e.g., post something on their Facebook wall.

4. Third-party authentication

GitHub code ▶ | **M**odel | **C**ontroller

HOW:

- Use the OmniAuth gem to take care of the details, together with specific OmniAuth gems for the providers you want to support:

```
# In your Gemfile
gem 'omniauth'
gem 'omniauth-twitter'
```

- Register your application with each identity provider to obtain **API keys** and **API secret keys**.
- Tell OmniAuth about the keys (don't add this to version control!)

```
# config/initializers/omniauth.rb
Rails.application.config.middleware.use OmniAuth::Builder do
  provider :twitter, "APP_KEY", "APP_SECRET"
end
```

HOW *(continued)*:

- Create a model to store user information:

```
# In a terminal
rails generate model Moviegoer name:string provider:string uid:string
rake db:migrate
```

```
# app/models/moviegoer.rb
class Moviegoer < ActiveRecord::Base
  def self.create_with_omniauth!(auth)
    Moviegoer.create!(
      provider: auth['provider'],
      uid:      auth['uid'],
      name:     auth['info']['name']
    )
  end
end
```

HOW *(continued)*:

- Update your views to add login and logout links:

```
- # app/views/layouts/application.html.haml
#login
- if @current_user
  %p.welcome Welcome, #{@current_user.name}!
  = link_to 'Logout', logout_path
- else
  %p.login= link_to 'Log in with Twitter', OmniAuth.login_path(:twitter)
```

- Add routes for login and logout actions:

```
# config/routes.rb
get 'auth/:provider/callback' => 'sessions#create'
get 'logout'                  => 'sessions#destroy'
get 'auth/failure'            => 'sessions#failure'
```

HOW *(continued)*:

- Update your controllers:

```ruby
# ApplicationController
before_action :set_current_user, :authenticate!

def set_current_user
  @current_user = Moviegoer.where(id: session[:user_id]).first
end

def authenticate!
  unless @current_user
    redirect_to OmniAuth.login_path(:twitter)
  end
end
```

```ruby
# MoviesController
skip_before_filter :authenticate!, only: [ :show, :index ]
```

HOW *(continued)*:

- Update your controllers:

```
# SessionsController
skip_before_action :authenticate!

def create
  auth = request.env['omniauth.auth']
  user = Moviegoer.where(provider: auth['provider'], uid: auth['uid'] ).first
  unless user
    user = Moviegoer.create_with_omniauth!(auth)
  end
  session[:user_id] = user.id
  redirect_to movies_path
end

def failure
  flash[:notice] = 'Could not login'
  redirect_to root_path
end

def destroy
  session.delete(:user_id)
  flash[:notice] = 'Logged out successfully'
  redirect_to movies_path
end
```

5. Associations

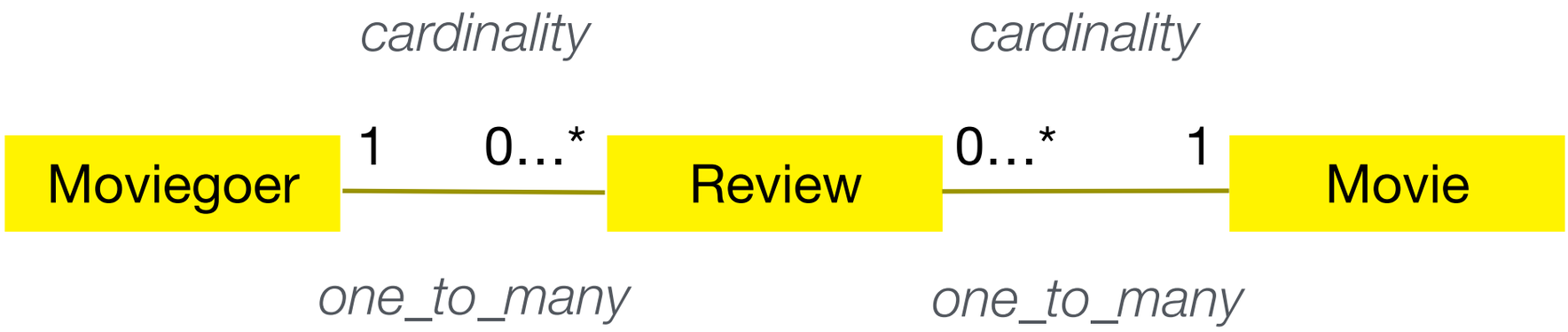
Model

WHAT:

- An association is a **logical relationship** between two entities.
- Associations are implemented as a set of class methods that tie objects together through foreign keys.

WHEN:

- Entities in your application are related and you want to access one entity through another—e.g., you have a moviegoer object and want all of their reviews.



Foreign keys is the mechanism that supports associations in relational databases, such as MySQL, PostgreSQL, SQLite, etc.

movies

id	title	rating
13	Pulp Fiction	PG-13
41	Kill Bill	PG
43	Reservoir Dogs	R

reviews

id	movie_id	moviegoer_id	potatoes
21	41	1	5
22	13	2	3
23	13	1	4

moviegoers

id	username
1	Alice
2	Bob
3	Carol

In SQL, the **join** operation from relation algebra is used to find associated records:

```
SELECT reviews.*
FROM movies JOIN reviews ON movies.id=reviews.movie_id
WHERE movies.id = 41
```

OO: object of a class has a direct reference to its associated objects.

How to **use** associations in Rails:

```
reservoir_dogs = Movie.find_by_title('Reservoir Dogs')
alice, bob     = Moviegoer.find(alice_id, bob_id)

# Alice likes the movie, Bob hates it
alice_review = Review.new(potatoes: 5)
bob_review   = Review.new(potatoes: 2)

# Add these reviews to the movie object's `reviews` association and
# update the database.
reservoir_dogs.reviews = [alice_review, bob_review]

# A moviegoer has many reviews.
alice.reviews << alice_review
bob.reviews   << bob_review

# How can we find out who wrote each review?
reservoir_dogs.reviews.map { |r| r.moviegoer.name } # => ['alice', 'bob']
```

How to **setup** associations in Rails:

- Create Reviews table to store review information:

```
# In a terminal
rails generate migration CreateReviews

# db/migrate/*_create_reviews.rb
class CreateReviews < ActiveRecord::Migration
  def change
    create_table :reviews do |t|
      t.references :moviegoer
      t.references :movie
      t.integer    :potatoes
      t.text       :comments
    end
  end
end
```

HOW to **setup** associations in Rails *(continued)*:

- Update the models involved in the relationship:
- **`belongs_to`** for Review:

```ruby
class Review < ActiveRecord::Base
  belongs_to :movie
  belongs_to :moviegoer
end
```

- **`has_many`** for Movie and Moviegoer:

```ruby
class Movie
  has_many :reviews
  # other model code...
end
```

HOW to use associations in Rails:

- Some of the most useful methods

```
# Fetch all reviews that belong to the movie.
m.reviews

# Replace the set of owned reviews with the new set [r1, r2].
m.reviews = [r1, r2]

# Add the review r1 to the set of m's reviews. This change is saved in
# the database immediately.
m.reviews << r1

# Get the movie associated with review r.
m = r.movie

# Associates review r with movie m.
r.movie = m
```

5. Associations

- `Build` vs. `create` for associations

new	save
`m.reviews.build(potatoes: 5)`	`m.reviews.create(potatoes: 5)`

- **`ActiveRecord::Associations`**

- `has_many` implies a collection of the owned object (Reviews), we can use all the collection idioms on it.
- `belongs_to` gives **Review** objects a **movie** instance, the review belongs to at most one movie.

Other types of associations

```
class Project < ActiveRecord::Base
  belongs_to              :portfolio
  has_one                 :project_manager
  has_many                :milestones
  has_and_belongs_to_many :categories
end
```

- One-to-one

```
class CEO < ActiveRecord::Base
  has_one :office
end
class Office < ActiveRecord::Base
  belongs_to :ceo    # foreign key - employee_id
end
```

- Many-to-many

```
class Programmer < ActiveRecord::Base
  has_and_belongs_to_many :projects       # foreign keys in the join table
end
class Project < ActiveRecord::Base
  has_and_belongs_to_many :programmers    # foreign keys in the join table
end
```

Foreign keys are just one way to implement associations.

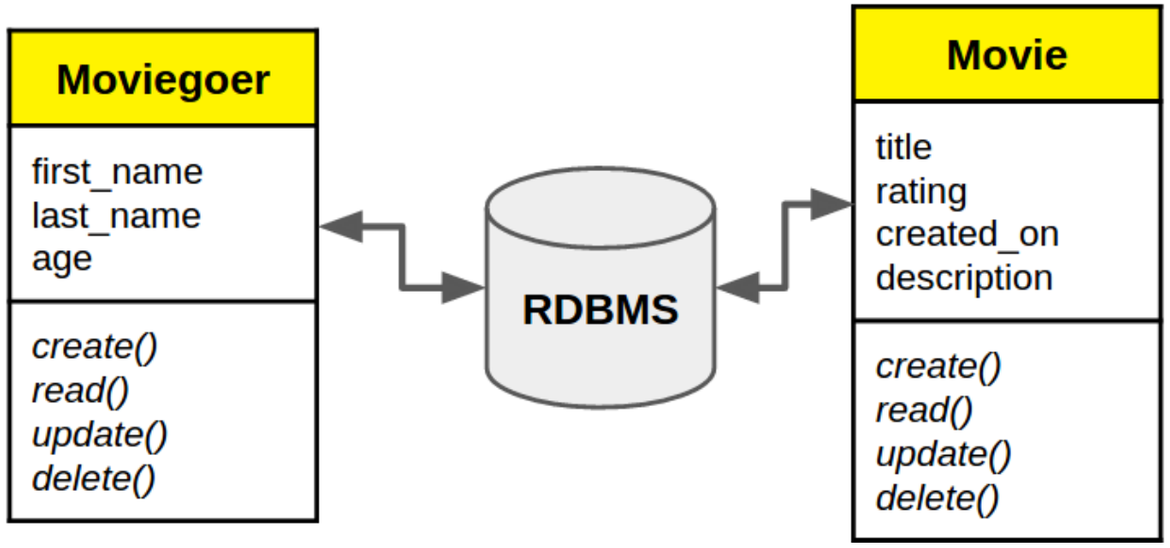
ActiveRecord:

- Foreign-key-based associations can become complex, that can limit the scalability and it is the first bottleneck in 3-tiers architecture.
- Relational pattern. Uses RDBMS.

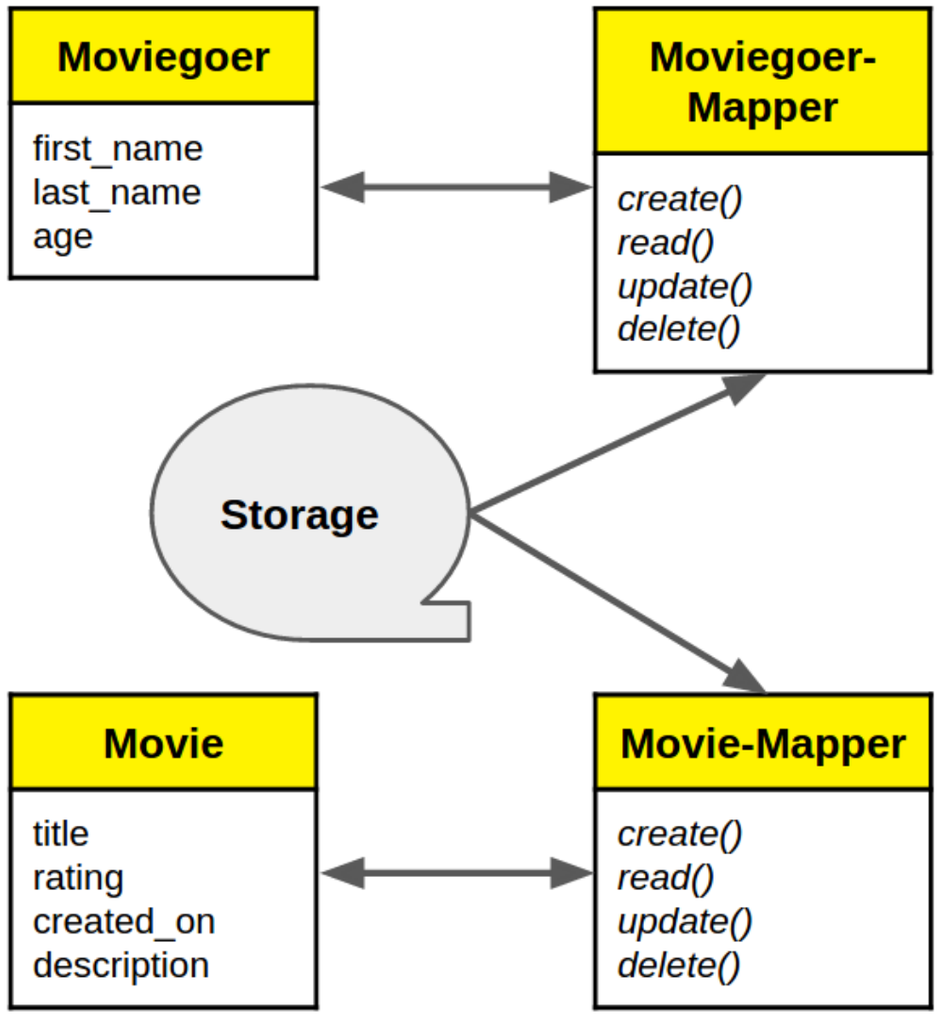
Data Mapper:

- It is an architectural pattern which defines how each model and its associations are represented.
- Doesn't rely on foreign key support.
- NoSQL

ActiveRecord



DataMapper



Google AppEngine, PHP, Sinatra

5.1. Through-associations

Model

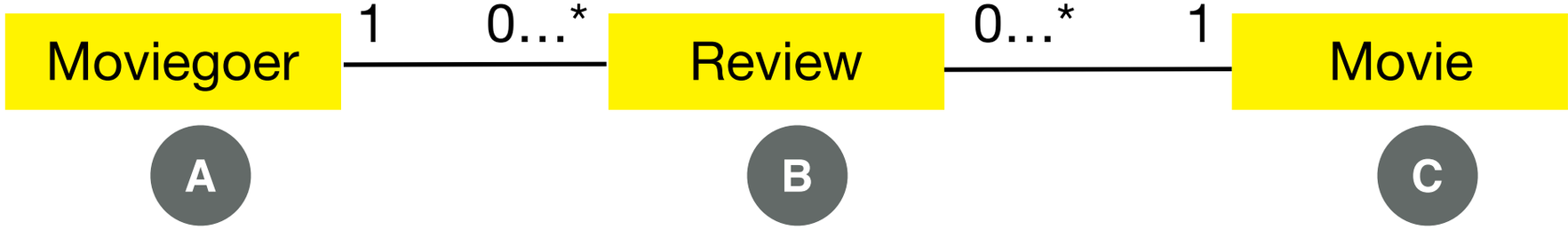
5.1. Through-associations

GitHub code ▶ | **M**odel

WHAT: an indirect association between two entities.

WHEN: A and **C** have `has_one` or `has_many` to a common **B**. This means that **A** and **C** are related via a `many_to_many` association.

A **moviegoer** has many **movies** through their **reviews**.



HOW: what are the movies Alice has reviewed?

```ruby
# app/models/moviegoer.rb:
class Moviegoer
  has_many :reviews
  has_many :movies, through: :reviews
  # ...other moviegoer model code
end
```

```ruby
# Client code:
alice =
  Moviegoer.find_by_name('Alice')

alice_movies = alice.movies
```

MORE ABOUT ASSOCIATIONS:

- They can be validated—e.g., make sure a review has an associated movie before saving it.

```
class Review < ActiveRecord::Base
  # A review is valid only if it's associated with a movie:
  validates :movie_id, presence: true

  # We can also require that the referenced movie itself be valid
  # in order for the review to be valid:
  validates_associated :movie
end
```

- They accept options that define what happens when related objects are deleted

```
# All reviews that belong to a movie will be deleted from the database if
# the movie is destroyed.
has_many :reviews, dependent: :destroy
```

6. RESTful routes for associations

Controller

6. RESTful routes for associations

Controller

WHAT:

- Routes that allow the modification, in a RESTful way, of nested resources (resources that **belong to** a parent resource).
- They are an elegant way of specifying a parameter that must always be present, for all CRUD actions (the `id` of the owner of the relationship).

WHEN:

- You want to perform CRUD operations on a resource that belongs to another (e.g., **reviews** belong to **movies**).
- You want to do this is a RESTful way, without using hidden form parameters or `session` variables.

6. RESTful routes for associations

GitHub code ▶ | Controller

HOW:

- Use Ruby blocks to nest resources in `routes.rb`:

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

- That will define the following **named routes** (run `rake routes`):

```
        Prefix Verb   URI Pattern                                       Ctrl#Action
 movie_reviews GET    /movies/:movie_id/reviews(.:format)               reviews#index
new_movie_review GET  /movies/:movie_id/reviews/new(.:format)           reviews#new
edit_movie_review GET /movies/:movie_id/reviews/:id/edit(.:format)      reviews#edit
     movie_review GET /movies/:movie_id/reviews/:id(.:format)           reviews#show
```

- Example:

```
PATCH /movies/10/reviews/2
```

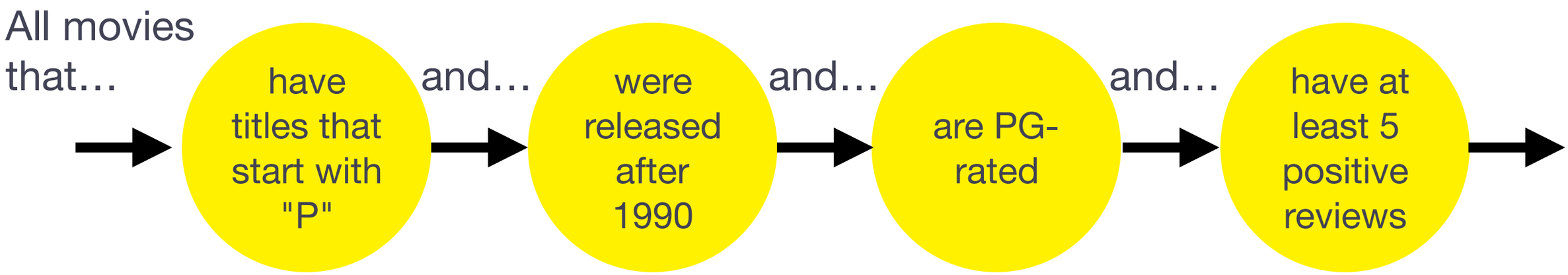
```
params[:movie_id] == 10; params[:id] == 2
```

7. Scopes

Model

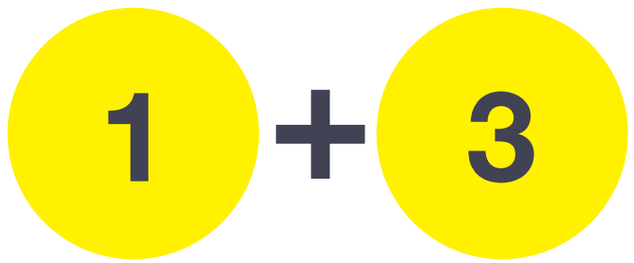
WHAT:

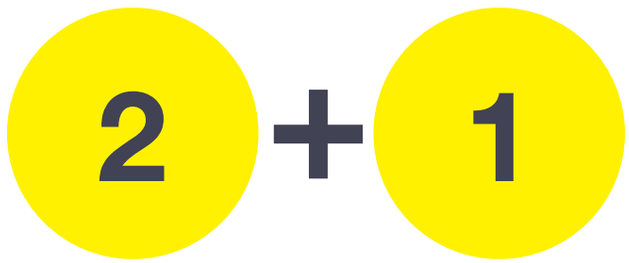
- Named fragments of ActiveRecord queries that are composable, i.e., that may be chained to form a more complex query.
- They rely on the **lazy evaluation** feature of `ActiveRelation` to only go to the database when an actual object from the result set is required.



WHEN:

- You want to reuse ActiveRecord filters in a **mix and match** fashion (as many as you want, in any order you want).





7. Scopes

GitHub code ▶ **M**odel

HOW:

Model

```ruby
class Movie < ActiveRecord::Base
  scope :for_kids, -> { where(rating: ['P', 'PG']) }

  scope :released_after, ->(year) {
    where("release_date >= ?", "Jan 1 #{year}".to_date) }

  def self.title_start_with(letter)
    where("title LIKE '?%'", letter)
  end
end
```

Controller

```ruby
class MoviesController < ApplicationController
  def index
    @movies = Movie.all

    if params[:released_after]
      @movies = @movies.released_after(params[:released_after])
    end

    if params[:for_kids]
      @movies = @movies.for_kids
    end
  end
end
```

The End.

April 26, 2016. Seminar given during the course MAC5714 — Object Oriented Programming (IME-USP).

Débora Setton, Hans Bejar, Thiago Petrone.