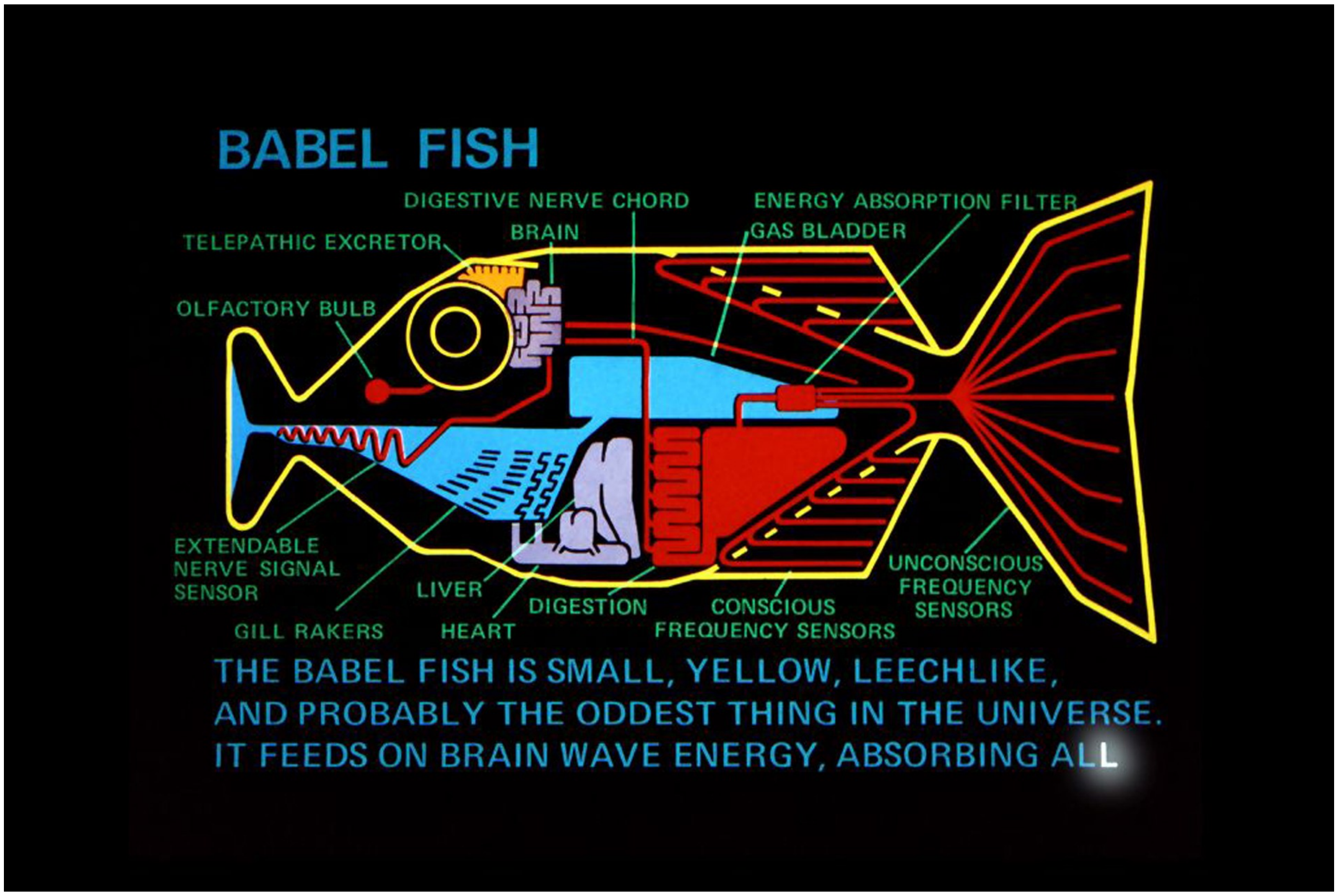
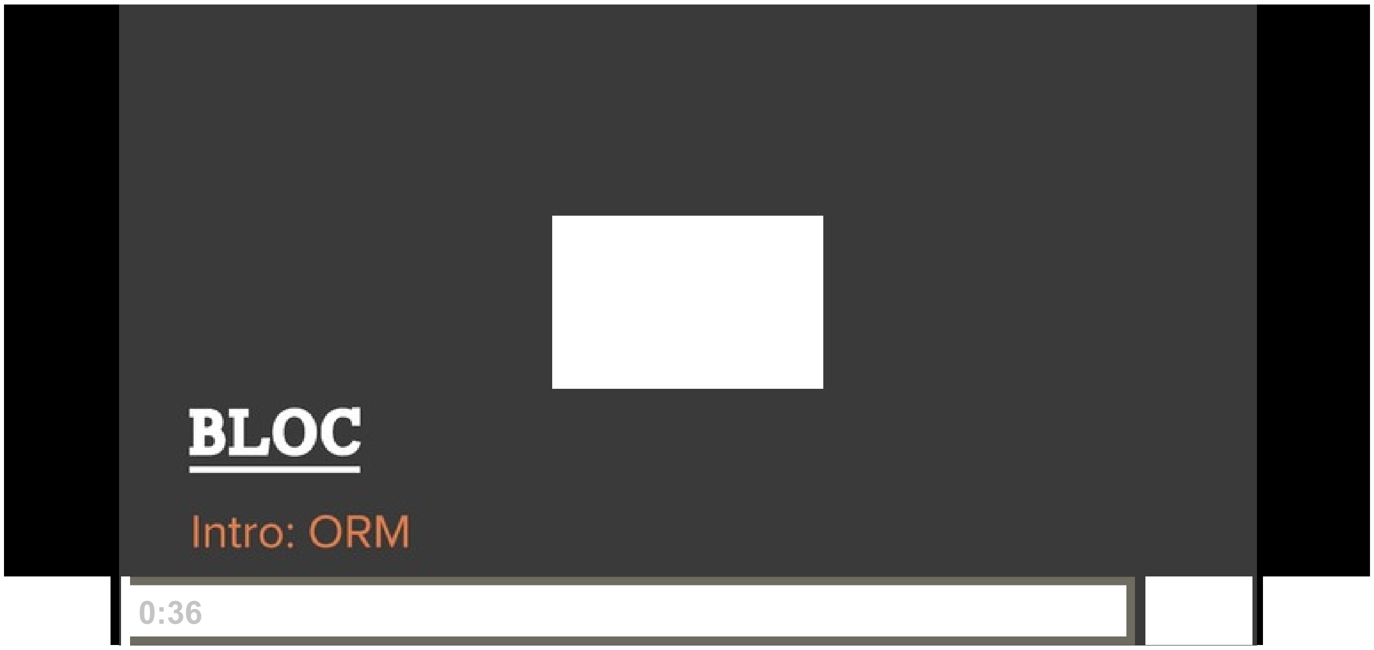
29 Rails: Object Relational Mapping


BABEL FISH
DIGESTIVE NERVE CHORD
ENERGY ABSORPTION FILTER
TELEPATHIC EXCRETOR
BRAIN
GAS BLADDER
OLFACTORY BULB
EXTENDABLE NERVE SIGNAL SENSOR
LIVER
DIGESTION
CONSCIOUS FREQUENCY SENSORS
UNCONSCIOUS FREQUENCY SENSORS
GILL RAKERS
HEART
THE BABEL FISH IS SMALL, YELLOW, LEECHLIKE, AND PROBABLY THE ODDEST THING IN THE UNIVERSE. IT FEEDS ON BRAIN WAVE ENERGY, ABSORBING ALL

> "Meanwhile, the poor **Babel fish**, by effectively removing all barriers to communication between different races and cultures, has caused more and bloodier wars than anything else in the history of creation."
>
> — **Douglas Adams**

Object Relational Mapping



As we learned in the last checkpoint, a model is a Ruby class that must also be represented as a database table. This implies that Rails must communicate with a database - and it does - but not without some complexity.

Communication between two systems which "speak different languages" is inherently complex because a translation service is required. As a Rails developer, you are essentially using two systems - Rails, which speaks Ruby, and a database, which speaks SQL.

Object Relational Mapping, or ORM is similar to a translation service, in that it provides a way for Rails developers to manipulate a database using Ruby, rather than writing SQL. Rails employs an ORM library named `ActiveRecord` to provide this translation service. To explore how Rails leverages ORM, we'll experiment with the Rails console. Let's watch a video introducing SQL and the `ActiveRecord` ORM:

Rails Console

The Rails console loads our application in a shell, and provides access to Rails methods, app-specific methods, persisted data, and Ruby. To launch the console from the command line, enter:

Terminal

```
$ rails c
```

And you should see the following message and prompt, or something very similar:

Console

```
Loading development environment (Rails 4.2.5)
2.2.1 :001 >
```

Because the console provides access to our application code, we can create posts and comments within the console, from the command line. Let's create a new post instance:

Console

```
> Post.create(title: "First Post", body: "This is the first post in our system")
```

> Creating a post would not be possible in IRB because posts and comments are specific to our application. Ruby (which is the only language that IRB understands) would not know what a "Post" is, and would throw an error.

Here's what we did:

- Called the `create` method on `Post`. This created a new row in the `posts` table. The `create` method is not Ruby - it's part of the `ActiveRecord` class that `Post` inherits from. The first line in our `post.rb` file - `class Post < ActiveRecord::Base` - declares this inheritance and gives `Post` access to the `create` method.
- Passed a hash to the `create` method. The hash was comprised of two keys: `title` and `body`, and two values.

Earlier we stated that Rails and the database don't speak the same language - but we just created a new database row via a Rails method, in the Rails console. This is ORM at work. The **`create` method** is part of the `ActiveRecord`, which is Rails' ORM library. `create` translates this: `Post.create(title: "First Post", body: "This is the first post in our system")` into SQL. We'll evaluate the resulting SQL in the next section.

SQL

Structure Query Language, or SQL, is the common language for all databases. Though some database technologies employ their own flavor of SQL, all SQL flavors are similar in syntax and usage. Active Record is a robust ORM library, and translates Rails code into a specific flavor of SQL, which it detects automatically from the database. This means that we don't need to worry about the nuanced SQL flavors. Let's review the SQL that was executed when we called `create` - it was printed in our Rails console:

Console

```
   (0.1ms)  begin transaction
# #1
  SQL (0.8ms)  INSERT INTO "posts" ("title", "body", "created_at", "updated_at") VAL
# #2
   (0.6ms)  commit transaction
=> #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

At #1, we add a row to the posts table using the `INSERT INTO` SQL statement. `"title", "body", "created_at", "updated_at"`, are the column names (i.e. attributes) on the `posts` table. The list of values after `VALUES (?, ?, ?, ?)` in brackets (`[["title", "First Post"] ... ]`) are values that correspond to the column names. The `created_at` and `updated_at` columns are default columns that Rails adds automatically, which is why we didn't need to specify them in the `create` call.

At #2, we commit the transaction which executes `INSERT INTO`. Commit statements end a SQL transaction and make all changes permanent. A transaction is one or more SQL statements that a database treats as a single unit.

We now have one row in the posts table.

Retrieving Information

It is important to remember that a row in a table corresponds to an instance of a class. Like a class instance, a row in a database table is unique. ORM allows us to retrieve information stored in a row and map it to a class instance that we create in our application. Let's retrieve a row from the posts table and map it to an instance of the `Post` class:

Console

```
> post = Post.first

  # #3
  Post Load (0.2ms)  SELECT  "posts".* FROM "posts"  ORDER BY "posts"."id" ASC LIMIT
 => #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

At #3 `Post.first` executes a `SELECT` SQL statement and fetches the first row from the posts table. `SELECT` is used to fetch a set of records from one or more tables.

After the first row is fetched, `ActiveRecord` converts the row's data into an instance of `Post`, or a post object. This post object is then assigned to the `post` variable. `ActiveRecord` makes this conversion from a database record to Ruby object possible.

Now that our instance is assigned, print it to view its value:

Console

```
> post
 => #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

`post` is populated by the first row of data (currently the *only* row of data) in our posts database table.

Let's add a comment to the post we retrieved:

Console

```
> post.comments.create(body: "First comment!")
   (0.1ms)  begin transaction
  SQL (0.4ms)  INSERT INTO "comments" ("body", "post_id", "created_at", "updated_at"
   (0.7ms)  commit transaction
 => #<Comment id: 1, body: "First comment!", post_id: 1, created_at: "2015-06-10 19:
```

Because we chained the method calls - `post.comments.create` - ActiveRecord interpreted this as "create a new comment for the first post". If we didn't specify the post to create a comment for, ActiveRecord would not have been able to update the `post_id`, which is critical because it defines the relationship between posts and comments. Inspect `post.comments`:

Console

```
> post.comments
Comment Load (2.6ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id"
=> #<ActiveRecord::Associations::CollectionProxy [#<Comment id: 1, body: "First comm
```

`post.comments` returns an `ActiveRecord::Association` because a comment *depends* on a given post. We'll explore associations in the next section.

`ActiveRecord` Associations

We defined the relationships between posts and comments in their respective classes, with `has_many` and `belongs_to`. These relationships are known as **associations**.

The `belongs_to :post` declaration in `Comment` generates a `post` method for each comment, giving us the ability to call `.post` on an instance of `Comment` and retrieve the associated post. The database stores this relationship, by keeping a `post_id` (foreign key) for each comment.

Retrieve the first comment in the comments table, and assign it to a `comment` variable:

Console

```
> comment = Comment.first
```

Fetch the post that is associated with `comment`:

Console

```
> comment.post
  Post Load (0.4ms)  SELECT  "posts".* FROM "posts" WHERE "posts"."id" = ? LIMIT 1
 => #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

Let's create another comment on `post`:

Console

```
> post.comments.create(body: "Second comment!")
```

The `has_many :comments` declaration in `Post` is the counterpart of `belongs_to :post`. The posts table makes no reference to comments. There's no comment_id column or array of comment_ids in the posts table. Instead, this relationship is stored in the comments table exclusively. A post retrieves its associated comments by fetching all the comments with a post_id that matches the id of the post. Storing the relationship in the comments table is a database strategy to allow data to be intersected or joined in an efficient manner.

Now that we have two comments associated with a single post, let's iterate over them using Ruby:

Console

```
# #4
>  post.comments.each { |comment| p comment.body }
# #5
  Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_i
"First comment!"
"Second comment!"
 => [#<Comment id: 1, body: "First comment!", post_id: 1, created_at: "2015-06-10 19
```

At #4, the `|comment|` block argument represents an instance of `Comment` with each iteration. We call `body` on each comment instance to retrieve the comment's body attribute from the database.

At #5, the `SELECT` statement fetches all the comments with the given `post_id`.

Recap

Concept	Description
Rails Console	Provides command line access to a Rails application and Ruby.
SQL	SQL is a language for communicating with a relational database.
Object Relational Mapping	Object-Relational Mapping (ORM) is a design pattern that connects the objects of an application to tables in a database. Using ORM, the properties and relationships of objects in an application can be connected to a database without the need to write SQL statements.
`ActiveRecord`	`ActiveRecord` is Rails' ORM library.

29. Rails: Object Relational Mapping

Assignment	**Discussion**	**Submission**

Create a new Git feature branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

By default, the Rails console uses the standard IRB shell, but it can also use alternative shells. One alternative is **Pry**, which adds many extra features that aren't available using the IRB shell.

1. Add `pry-rails` to the `:development` group in `Gemfile` and run `bundle` to install it.
2. Start the Rails console. With `pry-rails` installed, the console will use `Pry` by default.
3. Type `help` to see a list of the commands Pry provides. Some of the most useful are **listed below**.
4. Assign the first post to a variable and then print it to view its value.
5. Use `self.methods.sort` to inspect all the methods on the first post. You can `cd` into an object as though it's a directory, e.g. `cd post`.

   > To exit the list of methods, use `exit`.

6. Print the `nesting` information for the first post.

7. Use `self` and `save!` to change the `title` and `body` of the first post.
8. Issue a standalone `self` to print the post and verify the new `title` and `body`.

Copy the console output of items 4-8 to a **Gist** named `bloccit-rails-console-assignment` and submit the Gist link to your mentor.

Pry Command	Description
`help`	Show a list of commands or information about a specific command.
`cd`	Move into a new context (object or scope).
`ls`	Show the list of vars and methods in the current scope.
`nesting`	Show nesting information.
`reset`	Reset the REPL to a clean state.
`whereami`	Show code surrounding the current context.
`wtf?`	Show the backtrace of the most recent exception.

`exit`	Pop the previous binding.
`exit-program`	Exit Pry unconditionally.
`recognize-path`	See which route matches a URL.
`show-models`	Show all models.
`show-routes`	Show all routes in match order.

Commit your assignment in Git. See **Git Checkpoint Workflow: After Each Assignment** for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the assignment. If you struggle to complete the assignment, submit your best effort to your mentor *before watching a solution video*.

ORM Solution

assignment completed

?