University of Sheffield

COM1001 SPRING SEMESTER

Professor Phil McMinn

p.mcminn@sheffield.ac.uk

Object-Relational Mapping

How Do We Access a Database in Ruby?

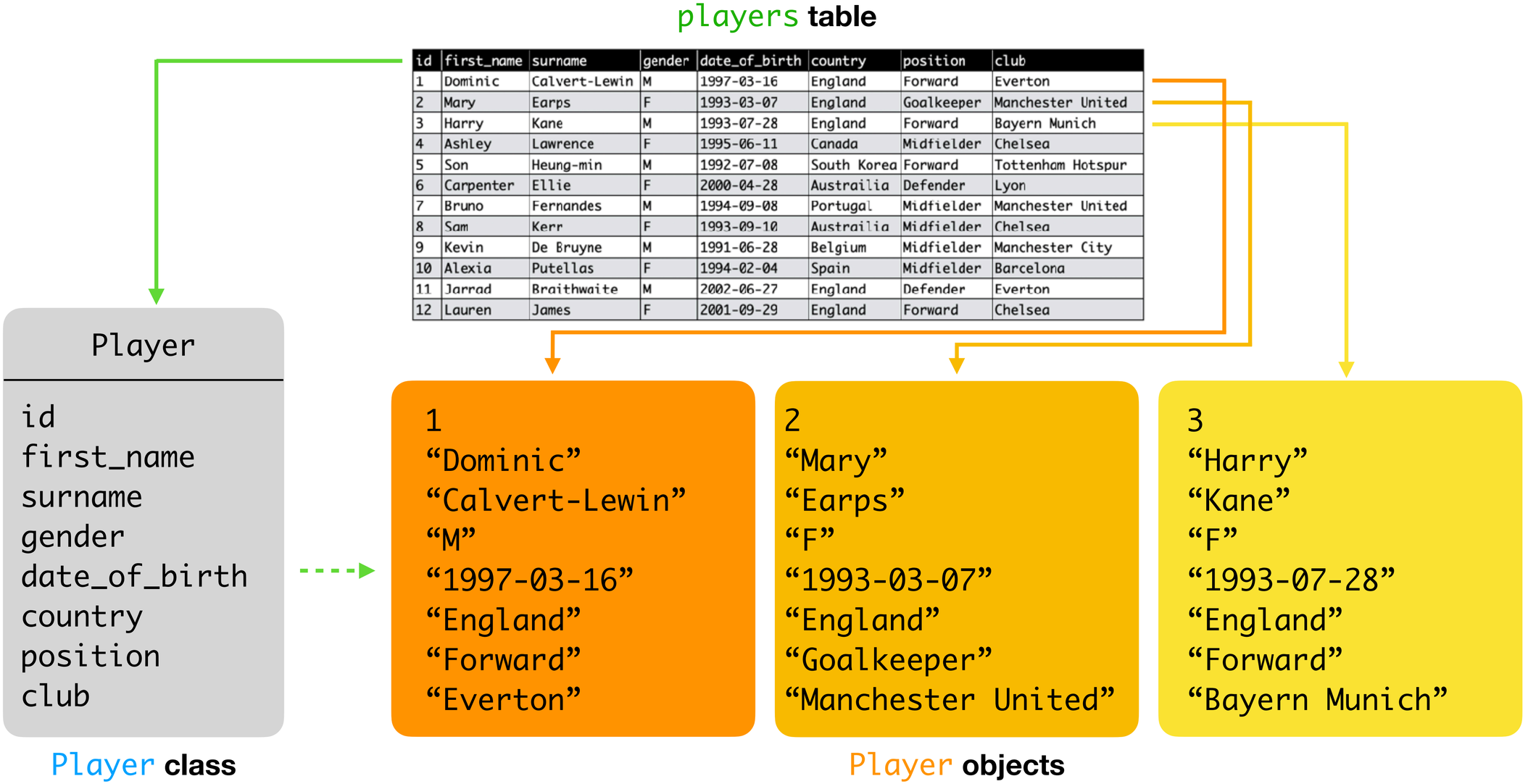
By far the easiest way to interact with a database in an Object-Oriented language such as Ruby is by using a third-party library or framework that gives automates the principles of **Object-Relational Mapping (ORM).**

Object-Relational Mapping is where:

- **A class is created for each table.** The class definition provides instance variables for each column, and corresponding getters and setters to set them. **The class definition encompasses the table definition.**

- **Objects of the class are instantiated for each row of data in the table.** The instance variables for each column are set to the values for that column in the row.

Object-Relational Mapping



Object-Relational Mapping Frameworks

An object-relational mapping (ORM) framework:

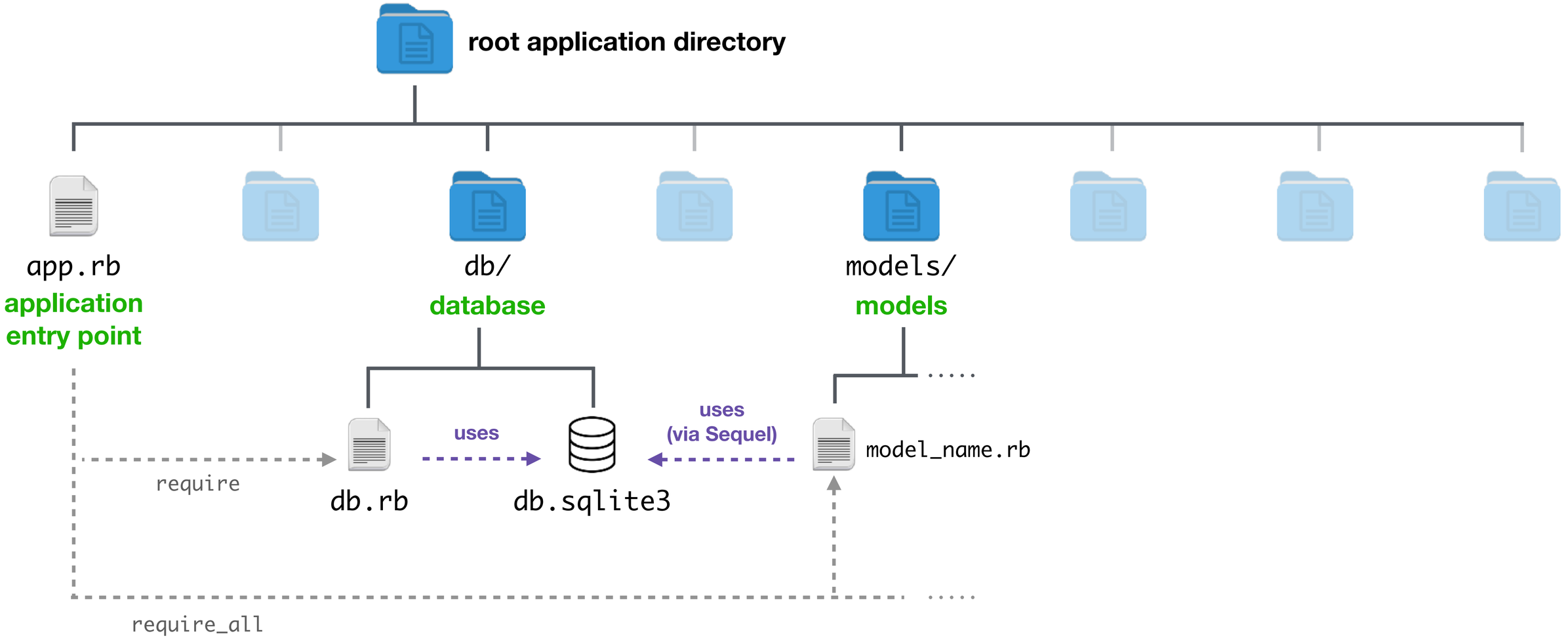
Automatically creates classes from table definitions (i.e., the Player class for the players table). These are also referred to as **Models**.

Automatically instantiates objects for each row of the table **as they are needed** from the database

> (It does not attempt to recreate the whole database as objects in memory, which could quickly exhaust memory resources if the database is very large, and thereby defeat the point of using a database in the first place)

We are going to use the **Sequel** ORM framework for Ruby in our Sinatra Applications.

Models


root application directory
app.rb
application
entry point
db/
database
models/
models
db.rb
db.sqlite3
model_name.rb
uses
uses
(via Sequel)
require
require_all

Coding a Model at the Most Basic Level

Model classes, at a minimum, must:

(1) extend `Sequel::Model`

```
class Player < Sequel::Model
end
```

This is the beginnings of our **Model** class for the `players` table

"<" means "extends" in Ruby.

(2) have a name that maps to a database table

Sequel uses a convention to map model classes to database tables – it takes the class name, converts it from camel case to snake case and pluralises it.

In other words, Sequel will match *classes* called `ClassName` with *tables* called `class_names`.

So Sequel figures the `Player` class represents the `players` table.

What This Gives Us

```ruby
class Player < Sequel::Model
end
```

By extending `Sequel::Model`, Sequel figures this is class is intended to be a model, and because it is called `Player`, it maps to the `players` table.

By interrogating the database schema, and through Ruby magic behind the scenes, Sequel is able to provide the model class a getter and setter for each column of the table.

Player
id
first_name
surname
gender
date_of_birth
country
position
club

```
1
“Dominic”
“Calvert-Lewin”
“M”
“1997-03-16”
“England”
“Forward”
“Everton”
```

The member variables of objects instantiated from the class then take on values of specific rows in the database, as demonstrated next…

What This Gives Us

In a controller class, we can then write Ruby “queries” on the tables represented by the model:

```ruby
players = Player.all
players.each do |player|
  puts "#{player.firstname} #{player.surname}"
end
```

“`all`” is a class-level method (equivalent to a “`static`” method in Java). It gets all the records from the table and returns them as an array of model instances (i.e., `Player` objects)

This loop iterates over the `players` array. In the loop body, the code we can get hold of the field values for each column by calling accessor methods which have the same names as the fields we are interested in.

Logging – db/db.log

```
I, [2023-11-02T15:34:02.270494 #62735]  INFO -- : (0.000095s) PRAGMA foreign_keys = 1
I, [2023-11-02T15:34:02.270529 #62735]  INFO -- : (0.000007s) PRAGMA case_sensitive_like = 1
I, [2023-11-02T15:34:02.270726 #62735]  INFO -- : (0.000030s) SELECT sqlite_version()
I, [2023-11-02T15:34:02.270831 #62735]  INFO -- : (0.000056s) PRAGMA table_xinfo(‘players')
I, [2023-11-02T15:34:02.271114 #62735]  INFO -- : (0.000068s) SELECT * FROM `players`
```

If we open up the `db/db.log` file, we can see the `SELECT` statement that Sequel generated by virtue of our code calling the `all` method to get hold of all records in the table.

As you would expect, the `SELECT` statement used is `SELECT * FROM players`.

This shows how we can write Sequel API calls to generate SQL statements.

If Sequel does not seem to be behaving or returning the records that we were anticipating, we can consult the log file to see what SQL statements it is generating and see if they match our expectations.

More Examples – using `where` and `count`

```ruby
supplied_club = "Everton"

players = Player.where(club: supplied_club)
num_players = players.count

if num_players.zero?
  puts "Sorry there are no players for that club."
else
  players.each do |player|
    puts "#{player.first_name()} #{player.surname()}"
  end
end
```

This variable doesn't have to be fixed of course, it could be supplied by a user, e.g. via a form.

We then use the (static) `where` method on the `Player` class to get hold of all players with this club.

The "where" clause here is provided as a key-value pair, in a special form of Ruby syntax. The key (the column name in the table) is written as shown, with a colon following it, followed itself by a value or a variable.

The `where` method returns a special type of Sequel object called a `Dataset`. This object has a number of useful methods including the `count` method (which returns the number of records/objects in the `Dataset`.

It can also be iterated over, like the array in the previous example where we used the `all` method on the `Player` class, as opposed to `where` as used here.

Getting One Specific Record

```ruby
supplied_id = 1

player = Player.first(id: supplied_id)
if player.nil?
  puts "No player exists with that ID"
else
  puts "#{player.first_name} #{player.surname}"
end
```

The `first` method is called in the same way as the `where` method in the previous example, except of course it returns the first record the database retrieves rather than all of them.

This is useful when there **should only be one record**, for example when we're **looking up a record by its primary key**, as we are doing here.

Create, Update, Delete

```ruby
# Create a new player instance
player = Player.new
player.first_name = "Marcus"
player.surname = "Rashford"
player.club = "Manchester United"

# Save to the database
player.save_changes

# Update his club and save again
player.club = "Manchester City"
player.save_changes

# Now delete
player.delete
```

Creates a new player instance in memory *only* (i.e., *not* in the database yet)

This triggers Sequel to generate an SQL `INSERT` statement and send it to the database

Since the record already exists in the database, Sequel now generates an SQL `UPDATE` statement to update the corresponding record in the database

This triggers Sequel to generate an SQL `DELETE` statement to remove the corresponding record.

Create, Update, Delete

```ruby
# Create a new player instance
player = Player.new
player.first_name = "Marcus"
player.surname = "Rashford"
player.club = "Manchester United"

# Save to the database
player.save_changes

# Update his club and save again
player.club = "Manchester City"
player.save_changes

# Now delete
player.delete
```

We can see the effect of these SQL statements in the log (`db/db.log`):

```
…
INFO -- : (0.000218s) INSERT INTO `players` (`first_name`, `surname`,
`club`) VALUES ('Marcus', 'Rashford', 'Manchester United')
…
INFO -- : (0.000093s) UPDATE `players` SET `club` = 'Manchester City'
WHERE (`id` = 13)
…
INFO -- : (0.000216s) DELETE FROM `players` WHERE `id` = 1
```

("…" indicate parts of the log removed for brevity)

Adding Further Methods To Models

```ruby
class Player < Sequel::Model
  # Get a string of the player's name in one method
  def name
    "#{first_name} #{surname}"
  end

  # Get the player's age, based on their date_of_birth
  def age(at_date = Date.today)
    dob = Date.strptime(date_of_birth, "%Y-%m-%d")
    TimeDifference.between(dob, at_date).in_years.floor
  end
end
```

Our model classes don’t have to remain empty.

This is the place where business logic on data can be implemented, or common routines for processing it.

SQL Injection

Another advantage of using ORM/Sequel to write the SQL statements for us is that it automatically sanitises user inputs by escaping them.

Suppose we got some input club from the user and attempted to insert it into a string to construct an SQL `SELECT` query:

```
query = “SELECT * FROM players WHERE club=‘#{club}’”
```

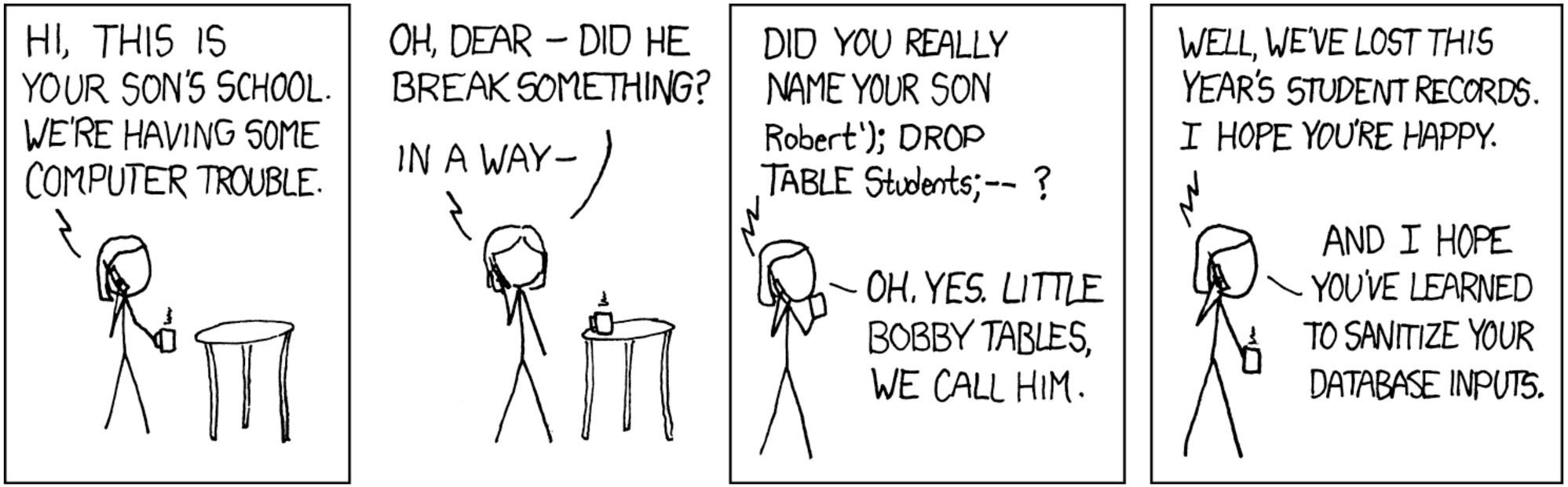
The string the user entered `“Manchester United”` would be:

```
query = “SELECT * FROM players WHERE club=‘Manchester United’”
```

Suppose the user entered `“Manchester United’; DELETE * FROM players; --”`

What would happen?

The `DROP TABLE` command in SQL deletes all records *and* the table from the database's schema!



However, Sequel takes care of the SQL injection problem for us, so there is no need to worry about it.

Documentation

As with all technologies it's not possible to teach *everything* you might need.

But, given this lecture as a starting point, you can look for what you need in the documentation that Sequel provides (this is a further useful skill to develop).

See `https://sequel.jeremyevans.net/documentation.html`

The README.md of the GitHub page also contains some useful examples: `https://github.com/jeremyevans/sequel`

If you cannot find what you want, you can also ask for help in the laboratory sessions.

Live Demonstration:

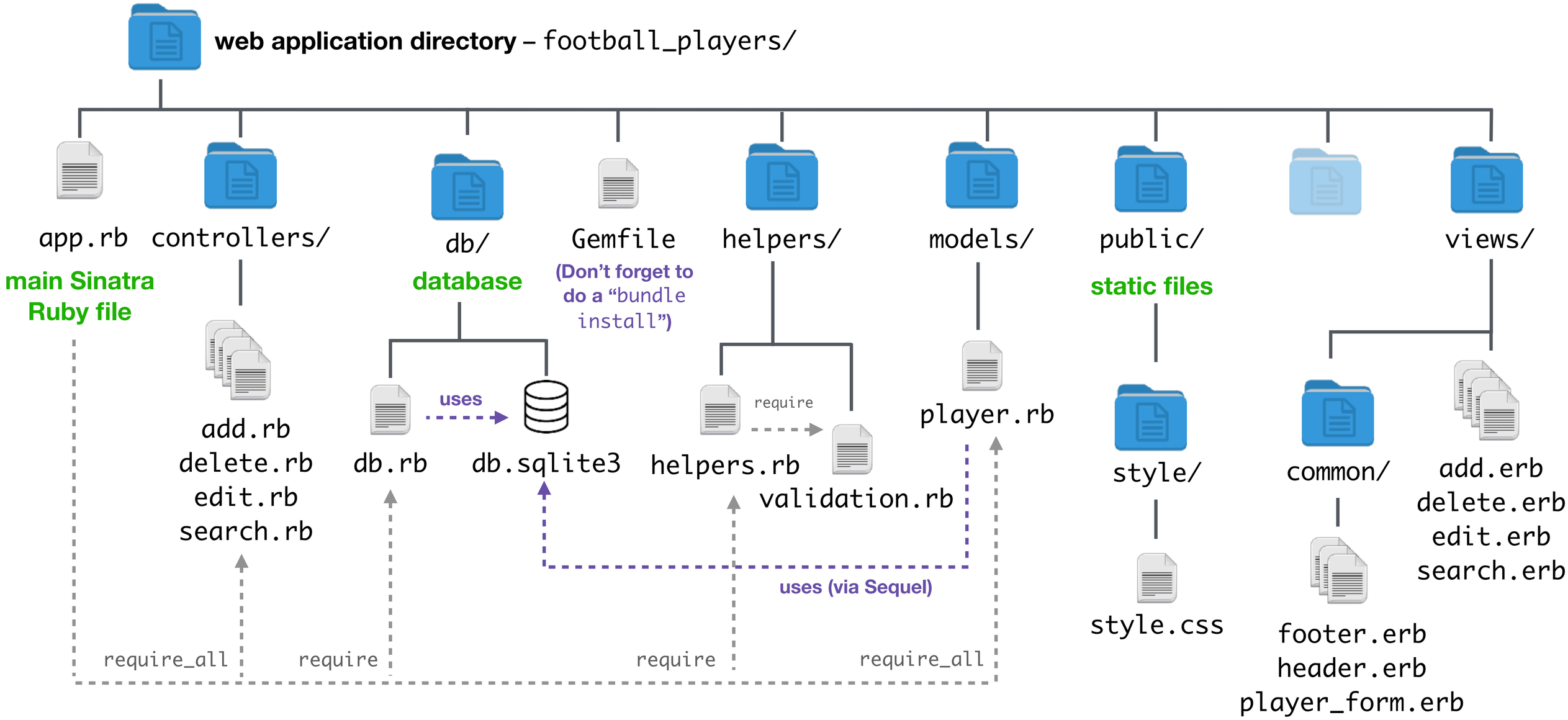
`week3/football_players_example`

(from the COM1001 GitHub repository)

Featuring:

- How everything is wired together in the app
  - From search to edit
- Validation

Understanding the File Structure



Validation with Models

```ruby
require "time_difference"

class Player < Sequel::Model
  include Validation
  extend Validation

  def self.id_exists?(id)
    return false if id.nil? # check the id is not nil
    return false unless str_digits?(id) # check the id is an integer
    return false if Player[id].nil? # check the database has a record with this id

    true # all checks are ok - the id exists
  end

  # ...

  def validate
    super
    errors.add("first_name", "cannot be empty") if !first_name || first_name.empty?
    errors.add("surname", "cannot be empty") if !surname || surname.empty?
    errors.add("gender", "cannot be empty") if !gender || gender.empty?
    errors.add("club", "cannot be empty") if !club || club.empty?
    errors.add("country", "cannot be empty") if !country || country.empty?
    errors.add("position", "cannot be empty") if !position || position.empty?
    errors.add("date_of_birth", "cannot be empty") if !date_of_birth || date_of_birth.empty?
    return unless date_of_birth && !str_yyyy_mm_dd_date?(date_of_birth)

    errors.add("date_of_birth", "is invalid")
  end
end
```

football_players/models/player.rb

Models move the heavy lifting of validation out of the controller and into the model, for better re-use across controllers.

If this looks odd to you, then I'm inclined to agree! It's the way Ruby makes the methods of helper models like `helpers/validation.rb` available to both class and instance methods (`extend` is for class methods and `include` is for instance methods).

`self.method_name` is the syntax Ruby uses to denote a method is a class method (i.e., the equivalent of static methods in Java). That is, this method would be called by using the class name as follows: `Player.id_exists(id)`

Validation is performed with models by overriding a method called `validate`, and by first calling its parent using `super`. (The same principles as Java apply here.)
Then we proceed to validate each field. If there is an error, we add it to an object called `errors` using its `add` method, supplying the field name and the error message. Note we can add more than one error message for a field.

Validation – Control Flow in the Controller

```ruby
get "/edit" do
  id = params["id"]
  @player = Player[id] if Player.id_exists?(id)
  erb :edit
end

post "/edit" do
  id = params["id"]

  if Player.id_exists?(id)
    @player = Player[id]
    @player.load(params)

    if @player.valid?
      @player.save_changes
      redirect "/"
    end
  end

  erb :edit
end
```

football_players/controllers/edit.rb

The `edit` route is passed an id of a player (this is encoded as a query string in the URL for editing the player on the search page. We have to check this id is valid. (The URL could have been edited by the user and this could be a source of security problems or crashes for some applications.)

Editing is more suitable for post, since a specific edit is a “one-time” action. So the form in the get route posts the data to this post route.

The code for this route again checks the id exists. If so it loads and sanitises the new, edited data into the player object using its `load` method (see the code in `models/player.rb`).

Although we defined the validation code in a method called `validate`, the method we need to call in the controller is a method inherited from `Sequel::Model` called `valid?`

If the data is valid, the controller saves it back to the database and redirects the user back to the search page (which is the “`/`” route).

ORM – Summary

Object-Relational Mapping (ORM) frameworks provide way for object-oriented languages to interact with relational databases.

- ORM maps **tables to classes**, and **rows to objects**. The resultant classes are referred to as **models**.

Sequel is an example of an ORM for Ruby.

- To create a model class that maps to a table in our database, we need to **follow the convention** that the class name is the non-plural, camel-cased version of the table name, so that **Sequel can match the class to a table.** That is the model class `ClassName` will be mapped to the table `class_names`.

By virtue of the object-oriented principle of inheritance our model class imports functionality from `Sequel::Model`.

- This includes the ability to obtain individual row values, while also querying, inserting, updating and deleting records in the underlying database table.

Sanitisation and validation can be handled by models, to save having to duplicate the same code in different routes/controllers