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?

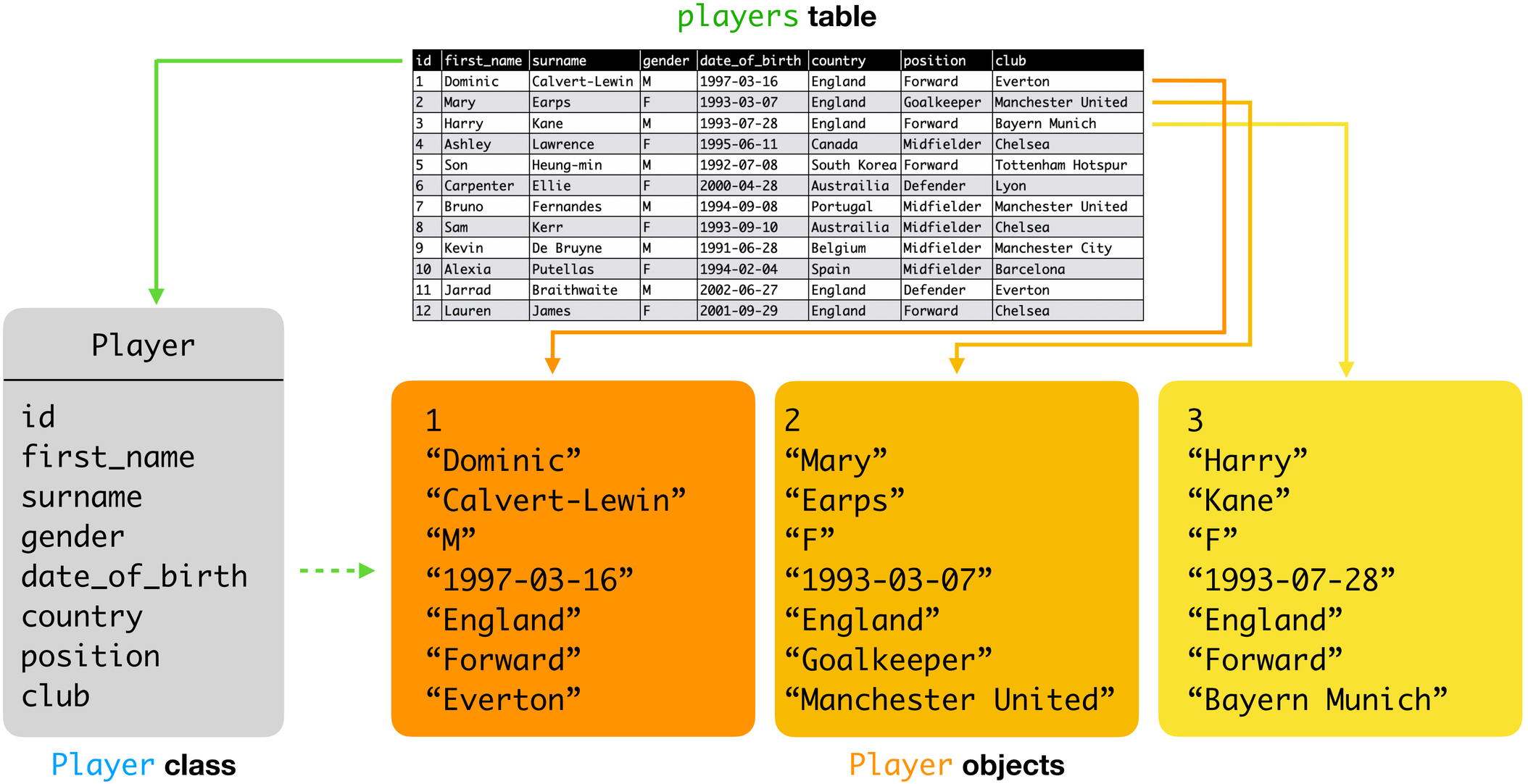
By far the easiest way to interact with a database in an Object-Oriented language such as Ruby is by using a third-party library or framework that gives automates the principles of **Object-Relational Mapping (ORM).**

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.

Sequel: Required Gems

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

class Player < Sequel::Model
end

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

We need the `logger`, `sequel` and `sqlite3` gems.

The `sqlite3` gem is often accidentally forgotten, leading to errors, so don't forget to include it in your `Gemfile`!

The `logger` gem is only needed if we plan to get Sequel to write the SQL statements it generates to interact with the database to a log. This is useful for debugging, so we need this gem too.

Sequel: The Global DB object

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

class Player < Sequel::Model
end

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

Sequel works by having a global constant, which by convention is called `DB`. This object manages the connection to the database we're using.

It is setup in the style shown, with a call to the static method `sqlite` of the `Sequel` class.

We supply it with two parameters – the first is the path to the `sqlite3` database file. The second is a `Logger` object, which is created using the path to a text file ("`db.log`") to which logging statements are output to. This is useful for debugging.

Sequel: Defining Models

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

class Player < Sequel::Model
end

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

This is how we create a class for the `players` table, referred to as a **Model**.

The `<` symbol means “extends” in Ruby. So the `Player` class extends the `Sequel::Model` class of the Sequel framework.

Sequel realises the `Player` class is intended to map to the `players` table. **This is by convention – class the class name is depluralised and camel-cased.**

That is, Sequel will match *classes* called `ClassName` with *tables* called `class_names`.

Then, it automatically creates accessor methods in the class for each column. Although we cannot see these written into the class definition for `Player`, they exist, as can be seen here, later in the code.

Sequel: Using Models

```ruby
require "logger"
require "sequel"

DB = Sequel.sqlite("football_players.sqlite3",
                   logger: Logger.new("db.log"))

class Player < Sequel::Model
end

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

In Sequel, we perform the equivalent of `SELECT` queries by invoking static methods of the model.

These are inherited from the `Sequel::Model` class, so we do not see them as part of our class (the `Player` class) itself.

Here, we are using the `all` method to get hold of all records in the table. The `all` method returns an array of `Player` objects, which we can iterate over.

In the loop body, we 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.

Sequel: Logging

```
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 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`.

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`

```
puts "Please enter a player's club:"
supplied_club = gets.chomp

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
```

In this example we get some data (a football club name) from the user at the command line

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
puts "Please enter a player's ID:"
supplied_id = gets.chomp

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:

```
…
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}’”
```

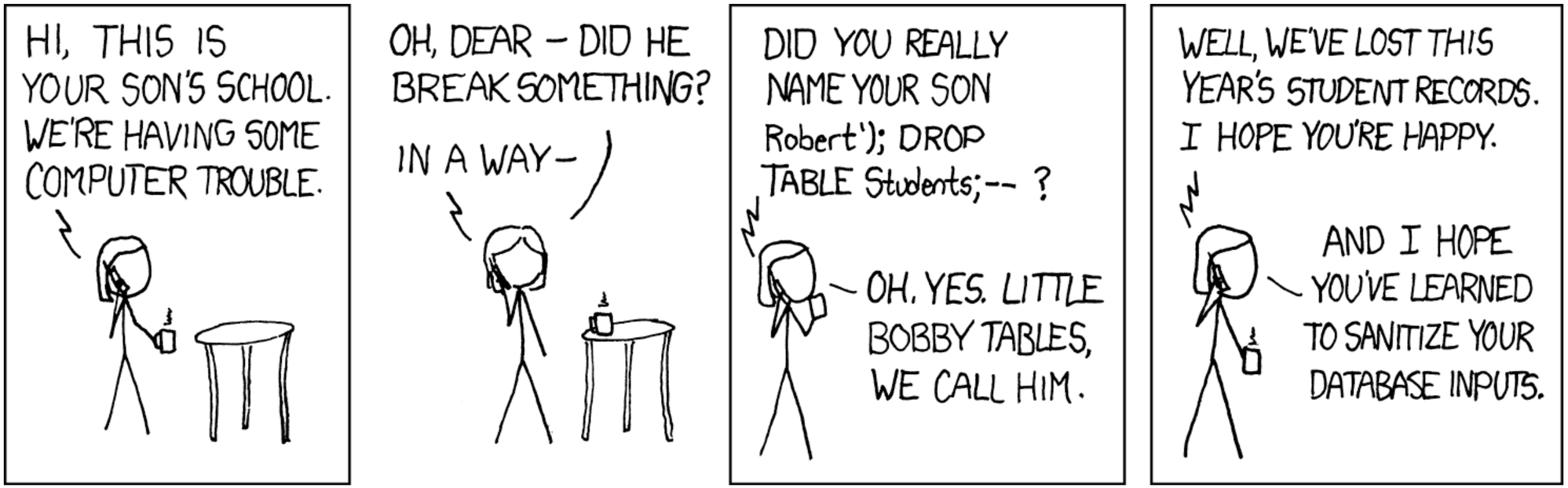
The string the user entered `“Manchester United”` would be:

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

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

What would happen?

xkcd – Little Bobby Tables



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:

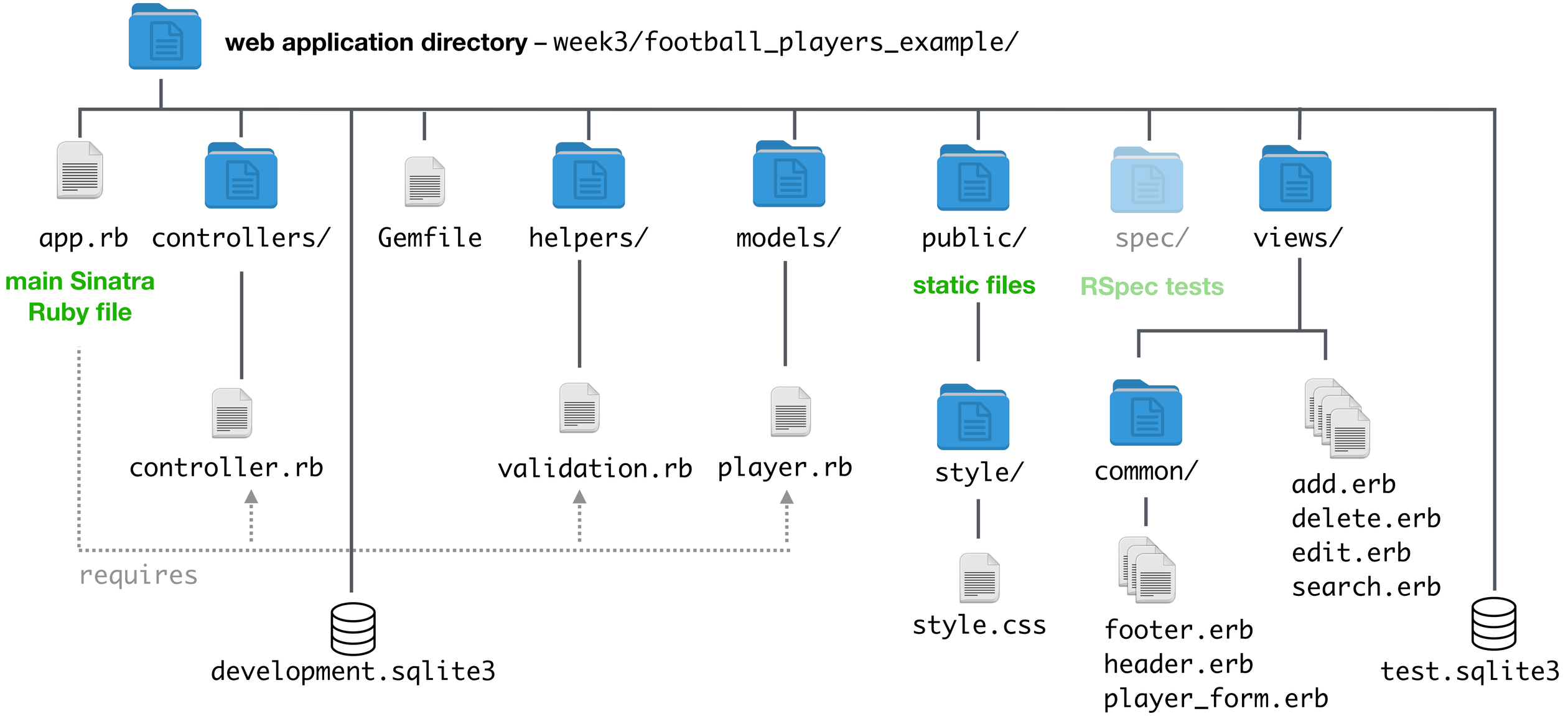
`week3/football_players_example`

(from the COM1001 GitHub repository)

Featuring:

- Different databases – development and production
- How everything is wired together in the app

Understanding the File Structure



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.