Advanced Web Programming

Yuan Wang

2019 Spring

Lecture 11

Ruby - Sinatra - RECAP

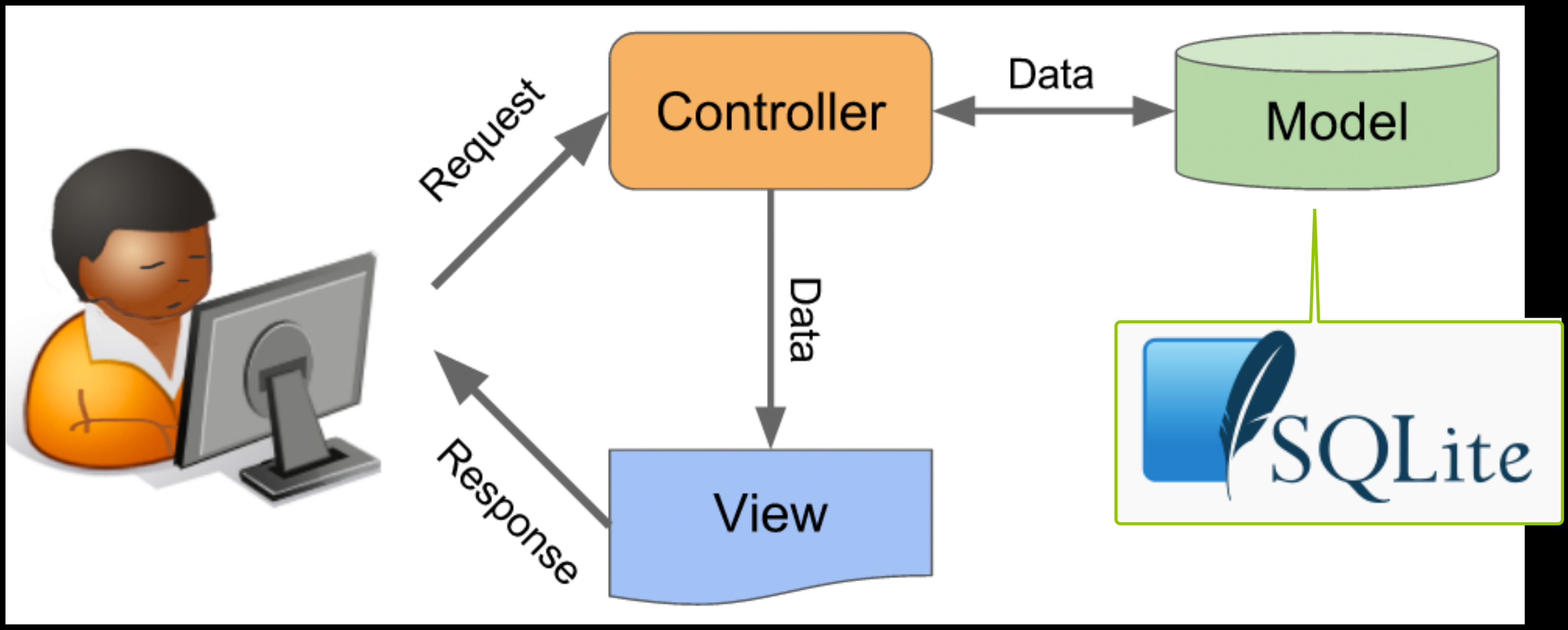
Songs by Sinatra

Home About Contact

this is my home page

Ruby - Sinatra

MVC - model - sqlite

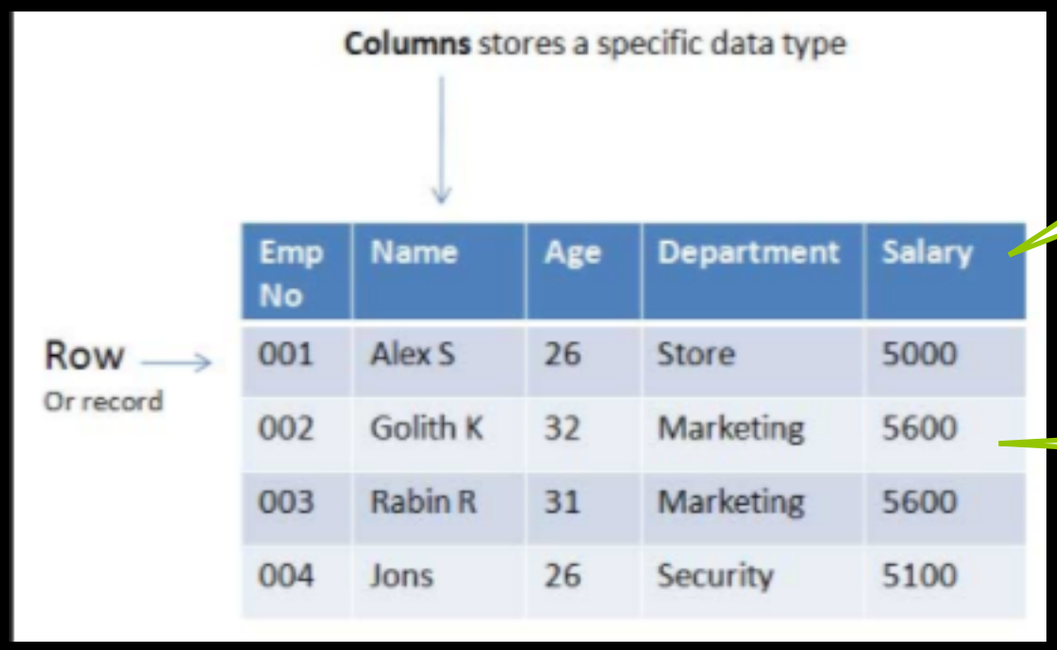


Ruby - Sinatra - Database

How to use database in Sinatra:

Database have tables,
table have rows,
rows have columns.

A table is very much like a class in Ruby, each rows is an instance (object) of this class, each column is a property.

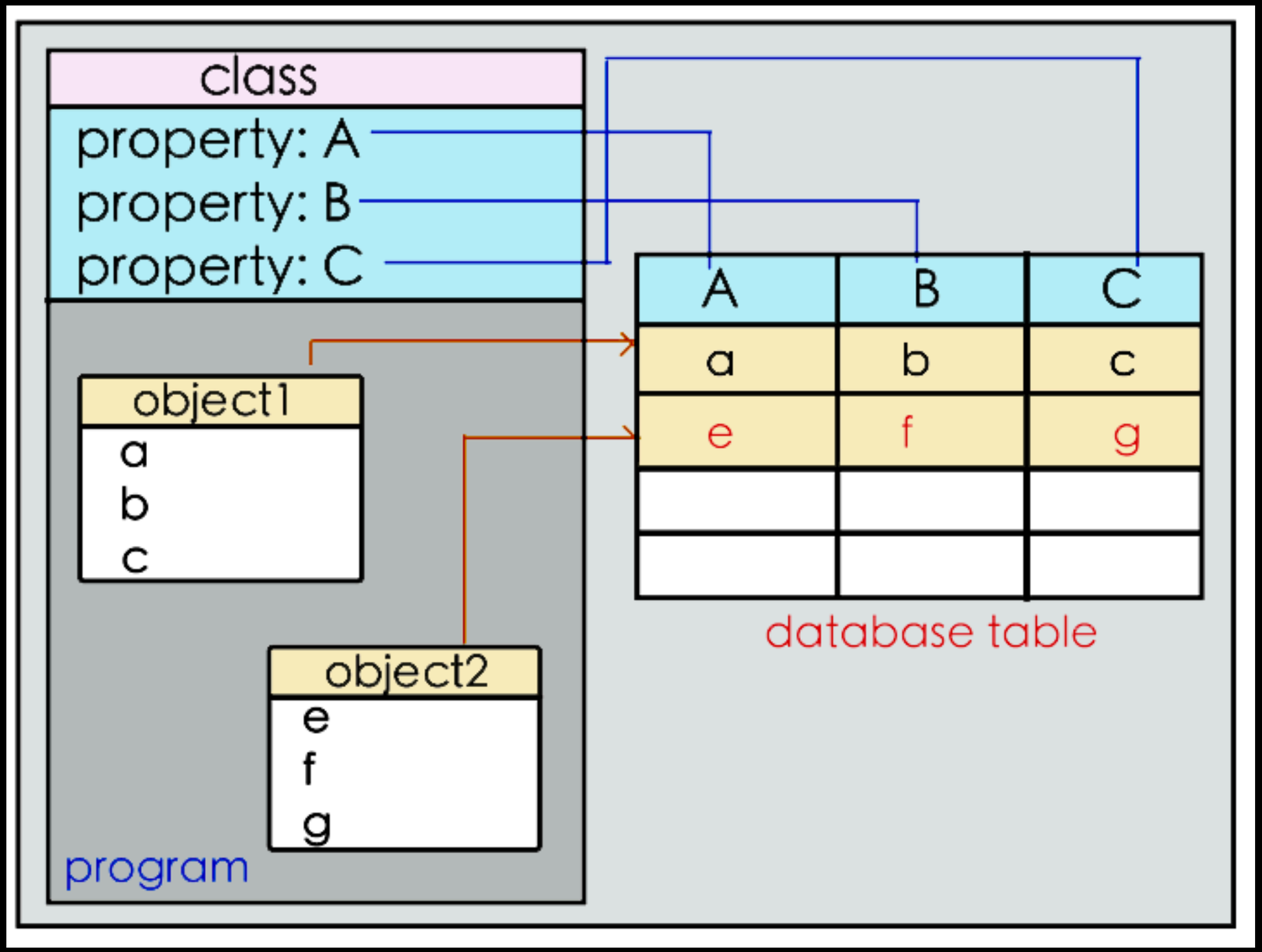


class “Employee”

one row is one instance of Employee

Ruby - Sinatra - Database

Like this:



Ruby - Sinatra - Database

If we can have some abstraction, so that operations on class and object can be mapped to database table and rows,

then database operation inside Sinatra application will be much easier than writing SQL statements,

The answer to that is:

ORM - Object Relational Mapper

Ruby - Sinatra - ORM

ORMs

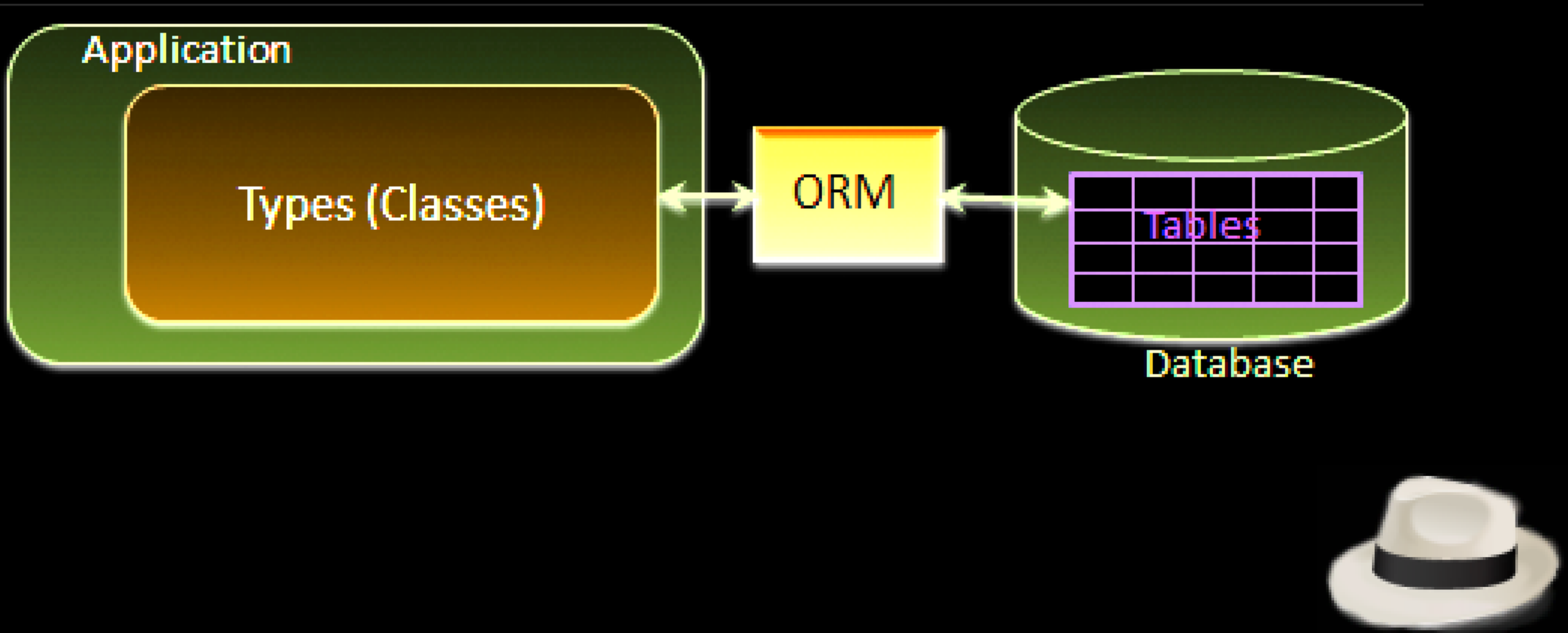
detailed interaction with database (SQL query, etc) are wrapped in API in programming language (Ruby)

we can then interact with database using API of this language.

this way, you naturally make use of the feature of the language (Ruby)

Ruby - Sinatra - Database

So an ORM is mapping tables to your Ruby classes.



Ruby - Sinatra - ORM

Different ORMs options for Sinatra

- Active Record (used in Ruby on Rails)
  http://rubygems.org/gems/activerecord

- DataMapper
  http://datamapper.org

we are using this for Sinatra!

- Sequel
  http://sequel.rubyforge.org

Ruby - Sinatra - ORM

Using DataMapper

Ruby - Sinatra - DataMapper

get the library

```
> gem install data_mapper
```

main gem

extra functionalities

data_mapper library including these gems:

- dm-core
- dm-migrations
- dm-aggregates
- dm-constraints
- dm-transactions
- dm-serializer
- dm-timestamps
- dm-validations
- dm-types

```
> gem install dm-sqlite-adapter
```

this allows the DataMapper to communicate with the SQLite database, we do NOT need to explicitly require this gem in our program

Ruby - Sinatra - DataMapper

Using DataMapper

require ‘dm-core’
require ‘dm-migrations’

dm-core: the main DataMapper gem

dm-migrations: create tables for you

or

require ‘data_mapper’

will access all the gems.

Ruby - Sinatra - DataMapper

First:

Connecting to database:

```
DataMapper.setup( name_of_connection,
                  connection_info)
```

class method of main class "DataMapper"

Ruby - Sinatra - DataMapper

```
DataMapper.setup(conn_name, url)
```

conn_name (database repository):
:default

url:
'protocol://username:password@localhost:port/path/to/repository'

database information: types, access, database file

Ruby - Sinatra - DataMapper

example - for sqlite database

```
DataMapper.setup(:default,
“sqlite3://#{Dir.pwd}/song.db”)
```

this will create a file called “song.db” if it doesn't exist, under the current directory. (using class method Dir.pwd)

username and password not supported

Ruby - Sinatra - DataMapper

example (for MySQL database)

```
DataMapper.setup(:default,
“mysql://user:password@hostname/database”)
```

example (for postgres database)

```
DataMapper.setup(:default,
“postgres://root:secretpass@127.0.0.1/
my_test_db”)
```

Ruby - Sinatra - DataMapper

Another format:

using hash for connection info

```
DataMapper.setup(conn_name, hash)
```

example:

```
DataMapper.setup(:default, {
  :adapter  => 'adapter_name_here',
  :database => 'path/to/repo',
  :username => 'username',
  :password => 'password',
  :host     => 'hostname'
})
```

Ruby - Sinatra - DataMapper

To setup other conn_name, you need
to have :default first, then set up another connection:

```
DataMapper.setup(:another, url)
```

then you have two connections (:default
and :another)

To use specific connection:

```
DataMapper.repository(:another, url) do
    Students.first      # retrieve first object.
end
```

Ruby - Sinatra - DataMapper

Using DataMapper

define a class to represent table (a Song for example)

```
class Song
  include DataMapper::Resource   # mixin
  property :id, Serial
  property :title, String
  property :lyrics, Text
  property :length, Integer
  property :released, Date
end
```

auto increment primary key

a table class is called:

"model"

property method to define column

column name by symbols

column type

Ruby - Sinatra - DataMapper

then call:

DataMapper.finalize

It is for validity/integrity checking, initialize all properties. Some frameworks like Rails will do this for you.

finalize method should be called after loading all models, and before the application starting accessing table.

Ruby - Sinatra - DataMapper

then call:

DataMapper.auto_migrate!
or
DataMapper.auto_upgrade!

destructively drops and recreates tables to match your model definitions

upgrading your datastore to match your model definitions, without actually destroying any already existing data.

Save all of the above in a ruby file song.rb

migration is the process of changing the schema (table definition) of the database

Ruby - Sinatra - DataMapper

Examples:

DataMapper.auto_migrate!
DataMapper.auto_upgrade!

```
require 'dm-core'
require 'dm-migrations'

DataMapper::Logger.new($stdout, :debug)
DataMapper.setup(:default, "sqlite3:///#{Dir.pwd}test.db")

class Person
 include DataMapper::Resource
  property :id,   Serial
  property :name, String, :required => true
end
DataMapper.auto_migrate!

class Person
  property :hobby, String
end
DataMapper.auto_upgrade!
```

turn on error logging to STDOUT

```
# ~ (0.015754) SET sql_auto_is_null = 0
# ~ (0.000335) SET SESSION sql_mode =
'ANSI,NO_BACKSLASH_ESCAPES,NO_DIR_IN_CREATE,NO_ENGINE_SUBSTITUTION,NO_UNSIGNED_SUBTRACTION,TRADITIONAL'
# ~ (0.283290) DROP TABLE IF EXISTS `people`
# ~ (0.029274) SHOW TABLES LIKE 'people'
# ~ (0.000103) SET sql_auto_is_null = 0
# ~ (0.080191) CREATE TABLE `people` (`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`name` VARCHAR(50) NOT NULL, PRIMARY KEY(`id`)) ENGINE = InnoDB CHARACTER SET utf8
COLLATE utf8_general_ci
# => #<DataMapper::DescendantSet:0x101379a68 @descendants=[Person]>
# ~ (0.000111) SET SESSION sql_mode =
'ANSI,NO_BACKSLASH_ESCAPES,NO_DIR_IN_CREATE,NO_ENGINE_SUBSTITUTION,NO_UNSIGNED_SUBTRACTION,TRADITIONAL'
# ~ (0.000932) SHOW VARIABLES LIKE 'character_set_connection'
# ~ (0.000393) SHOW VARIABLES LIKE 'collation_connection'
```

create table

```
# ~ (0.000612) SHOW TABLES LIKE 'people'
# ~ (0.000079) SET sql_auto_is_null = 0
# ~ (0.000081) SET SESSION sql_mode =
'ANSI,NO_BACKSLASH_ESCAPES,NO_DIR_IN_CREATE,NO_ENGINE_SUBSTITUTION,NO_UNSIGNED_SUBTRACTION,TRADITIONAL'
# ~ (1.794475) SHOW COLUMNS FROM `people` LIKE 'id'
# ~ (0.001412) SHOW COLUMNS FROM `people` LIKE 'name'
# ~ (0.001121) SHOW COLUMNS FROM `people` LIKE 'hobby'
# ~ (0.153989) ALTER TABLE `people` ADD COLUMN `hobby`VARCHAR(50)
# => #<DataMapper::DescendantSet:0x101379a68 @descendants=[Person]>
```

add column and upgrade table

Ruby - Sinatra - DataMapper

We can check this out in IRB.

do this so that Song class is available

```
$ irb
>require ‘./song’
>Song.auto_migrate!    # that’s what
                       # dm_migrations is for
```

first time to run #auto_migrate!, will create a new file song.db (in the example above), create a table called “songs” in the database with specified columns for each property

Ruby - Sinatra - DataMapper

CRUD (Create, Read, Update, Delete)

Now we can start working on the table using object

Create: song = Song.new

an object (in memory) is created. that is, a new row in the tables.

to save to the database(from memory):

song.save

Ruby - Sinatra - DataMapper

To add data to the table row:

```
song.title = “My Way”
song.lyrics = “this is my way”
song.length = 323
song.released = Date.new(1969)
song.save
```

Ruby - Sinatra - DataMapper

Or

```
Song.create(title: “New York New York”,
            lyrics: “new york, new york”,
            length: 210,
            released: Date.new(1960))
```

this method will “create + save” the song to database.

```
Song.count      # this will be 2
```

Ruby - Sinatra - DataMapper

Read from table:

```
Song.all          # return all the songs created
                  # in an array
Song.all.reverse
Song.all(:title.like => ‘way%')   # query
Song.get(id)    # get the object with primary id
Song.get(1)     # primary id is 1
Song.first      # first object
Song.last       # last record
song = Song.first(title: “My Way”)    # query for first
                                      #matching record
```

Ruby - Sinatra - DataMapper

Read from table: more query examples

pass a hash

```
# 'gt' means greater-than. We can also do 'lt'.
Person.all(:age.gt => 30)

# 'gte' means greater-than-or-equal-to. We can also do 'lte'.
Person.all(:age.gte => 30)

# 'not' allows you to match all people without the name "bob"
Person.all(:name.not => 'bob')

# If the value of a pair is an Array, we do an SQL IN-clause for you.
Person.all(:name.like => 'S%', :id => [ 1, 2, 3, 4, 5 ])

# Does a NOT IN () clause for you.
Person.all(:name.not => [ 'bob', 'rick', 'steve' ])
```

#gt, #gte, #not, #like, #not, #eql are additions to Symbol class

Ruby - Sinatra - DataMapper

Update

```
song.update(title: “Your Way”)
song.title     # =>Your Way
```

Ruby - Sinatra - DataMapper

Delete

```
Song.create(title: “When I fall in love”)
Song.last.destroy
```

chain method call to delete the last created song

Ruby - Sinatra - DataMapper

Checking database using IRB and sqlite3 (model is defined in song.rb)

```
$irb
irb> require “./song”
irb> Song.auto_migrate!
irb> song = Song.new
irb> song.save
irb> song.title = “song 1”
irb> song.length = 240
irb> song.release = Date.new(2016)
=> #<Date: 2016-01-01 ((2457389j,0s,0n),+0s,
2299161j)>
irb> song.save
```

```
$sqlite3 song.db
sqlite> .tables
songs
sqlite>sqlite> .schema songs
CREATE TABLE "songs" ("id" INTEGER NOT NULL
PRIMARY KEY AUTOINCREMENT, "title" TEXT,
"lyrics" TEXT, "length" INTEGER, "release"
DATE);

sqlite> select * from songs
1|song 1|this is a new song|100|2016-01-01
```

Ruby - Sinatra - DataMapper

Add model/view/route to our Frank Sinatra web application

Code explanation……

Ruby - Sinatra - DataMapper

End