Integrallis

Ruby on Rails

Full Steam Ahead

with Chris Irish

Overview

Part 2 - Intro to Rails

In **Part 2** we'll build our first Rails application

Objectives

- To gain an understanding of the **structure** of a Rails application
- To learn how Rails implements the **MVC** pattern
- To get used to the flow of creating and enhancing a Rails application
- To familiarize yourself with the console and the rails & rake commands

Rails

Some Background

Ruby on Rails
What is Ruby on Rails?

Powerful framework for building web applications

100% open-source under the MIT license

Actively maintained and developed by a top-notch core team

http://en.wikipedia.org/wiki/Ruby_on_Rails#History

Ruby on Rails
Rails 3

In large part a merge with another framework called **Merb**

Became one the first major frameworks to fully embrace **REST**

ActiveRecord revamped to use **Arel**, for programatic query building

Easy unobtrusive JavaScript helpers

Explicit dependency management with Bundler

jQuery became the default JavaScript library shipped with Rails

CoffeeScript and SCSS support on by default

Introduced the Asset Pipeline to make JS and CSS first class citizens

Streaming response API

Convention over configuration / smart defaults

Don't repeat yourself (DRY)

Optimize for developer happiness

Full Steam RoR

What makes Ruby on Rails great?

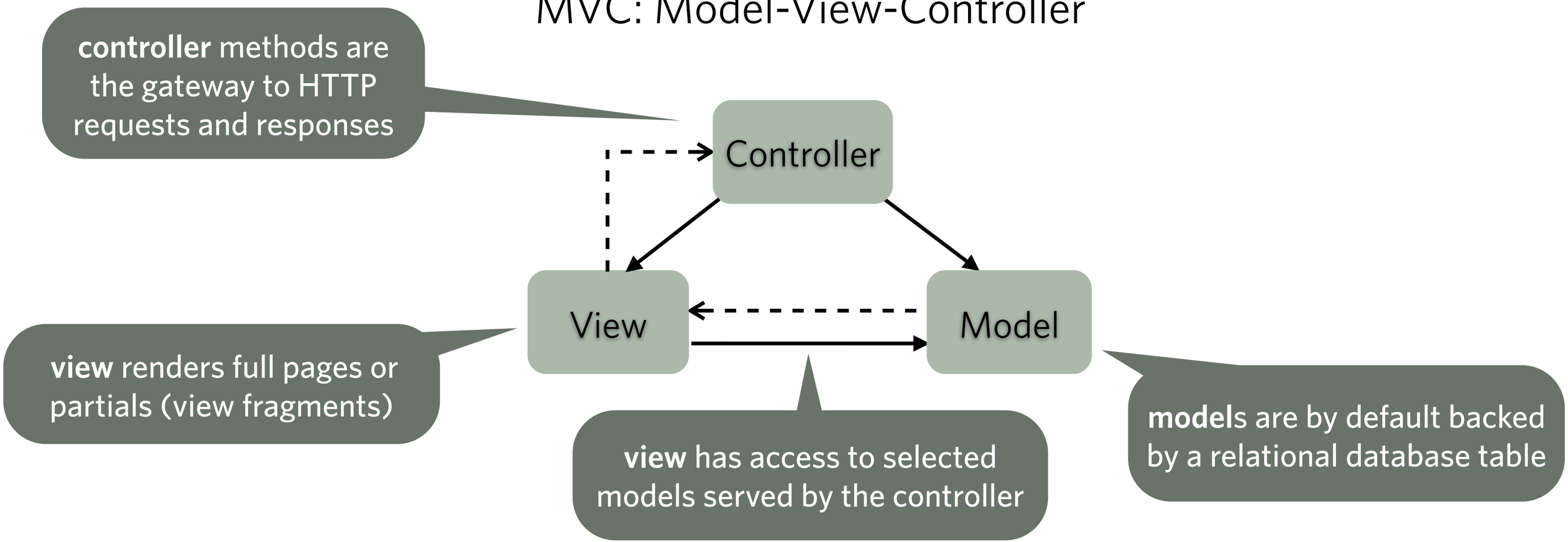
Ruby

Community

Migrations

Active Support

MVC: Model-View-Controller



Rails divides an application codebase following the MVC pattern

Request Handling
The Request-Response Pipeline

Router - "Connecting URLs to Code"

Controller - Process the request and produce an appropriate response

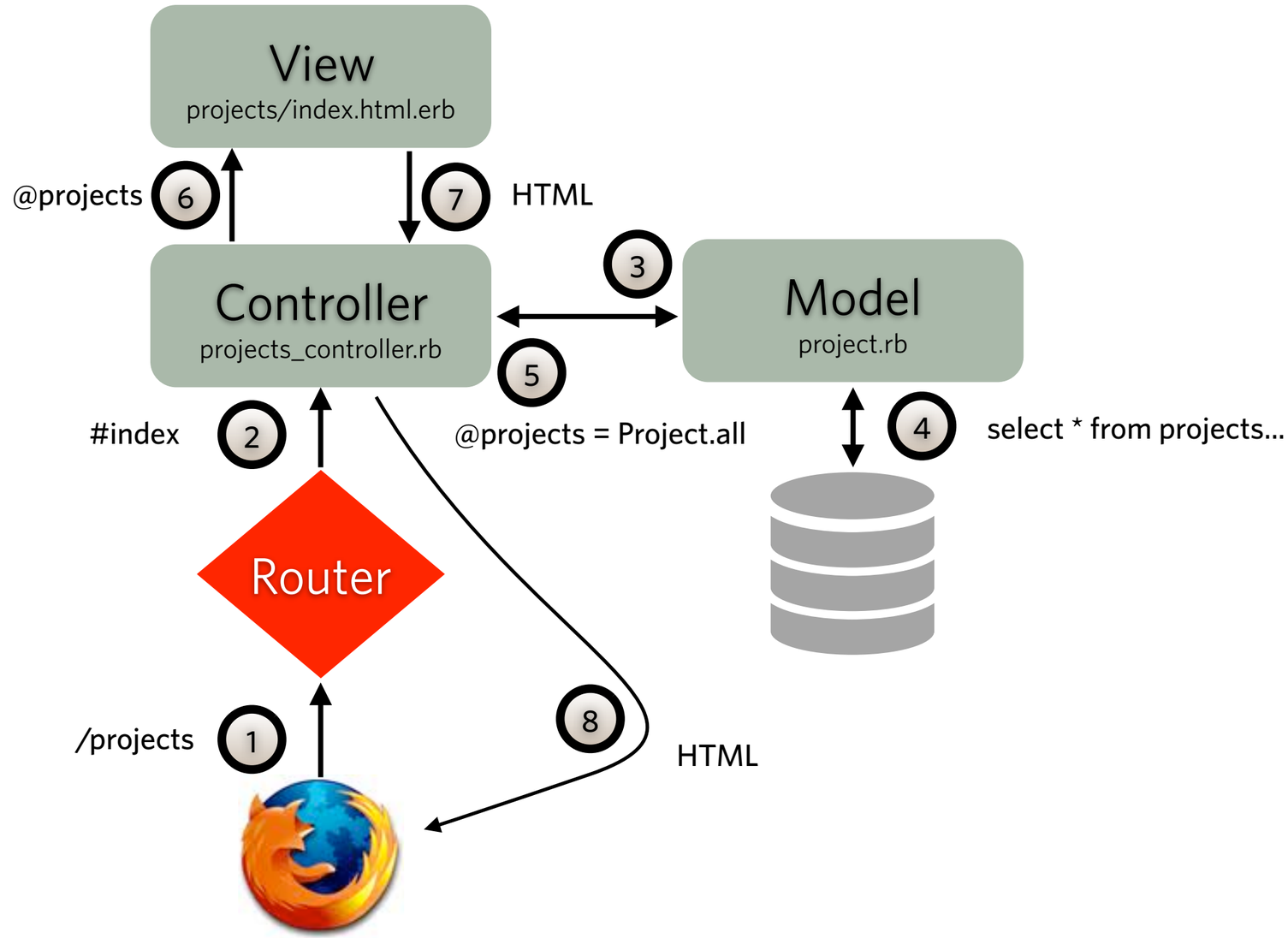
Model - Retrieve objects from the database + Manage persistence logic (validations, serialization, etc.)

View - HTML (or other) templates rendered and returned as the response body

Request Handling

The Request-Response Pipeline

1. User requests /projects
2. Rails router forwards the request to projects_controller#index action
3. The index action creates the instance variable @projects by using the Project model all method
4. The all method is mapped by ActiveRecord to a select statement for your DB
5. @projects returns back with a collection of all Project objects
6. The index action renders the index.html.erb view
7. An HTML table of Projects is rendered using ERB (embedded Ruby) which has access to the @projects variable
8. The HTML response is returned to the User



Rails

Laying the Tracks

Ruby on Rails

Using rails new

- Let's create a Rails application using the **rails new** command:

```
/>rails new friends
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      ...
/>cd friends/
```

- Rails will generate the friends application and run the bundle command

Ruby on Rails
A First Look

- Let's take a quick tour of the top level directories:

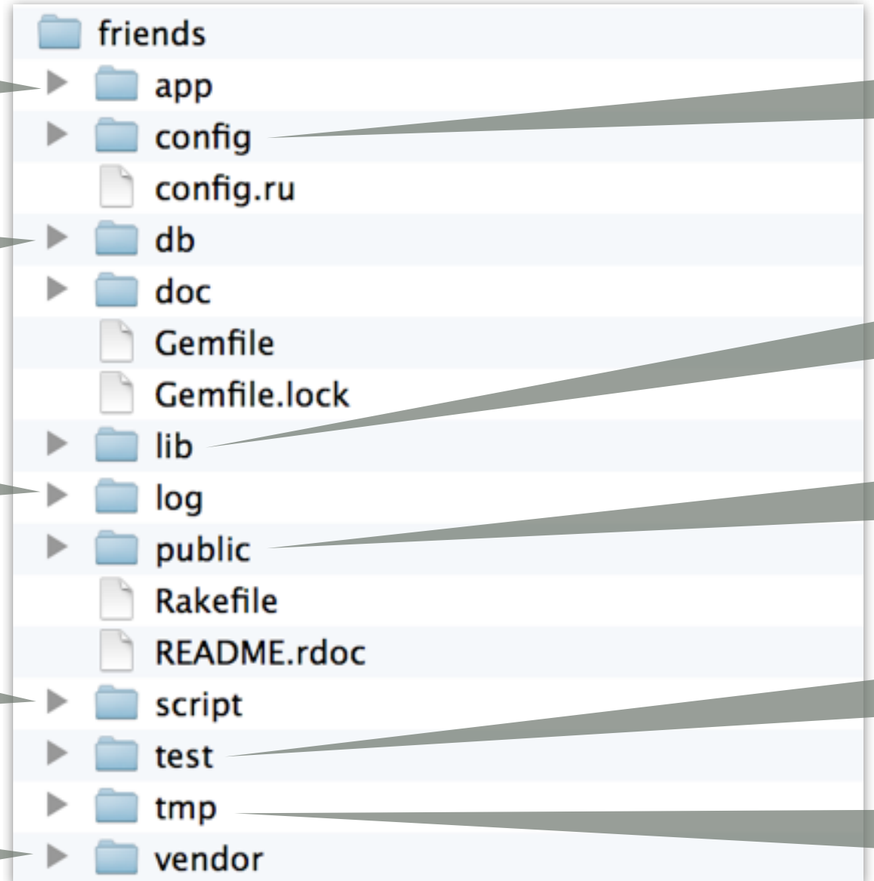
Folder for models, views and controllers along with (view) helpers, assets, and mailers. Most of your work will happen here!

All DB migrations, the generated schema.rb and potentially DB file-based artifacts when using SQLite

Here you'll find a log file per environment (development.log, test.log, production.log)

Contains a single script; rails. The rails command contained in the Gem will invoke this script when invoked from the root of your app

Third party code, assets (old style plugins - deprecated in Rails 4)



Folders for configuring the 3 default environments, database.yml and routes.rb (routing mapping). Also contains initializer files, localization files

Rails designates lib to store assets (your own non-app specific libraries) and (rake) tasks

Static HTML files that get picked up by the router based on name and other static assets that can be publicly reached

Default test directory for Test::Unit based tests

Temporary runtime artifacts like sessions, sockers, cache, pids

Controllers
app/controllers

- Controllers live in app/controllers
- By default we are given a base controller ApplicationController, your controllers will inherit from it
- ActionController::Base provides us with the facilities to deal with requests, parameters, sessions, responses, redirects and rendering

```
class ApplicationController < ActionController::Base
  protect_from_forgery
end
```

app/controllers/application_controller.rb

Models
app/models

- Models live in app/models
- Models are Ruby objects associated with one or more database entities
- A freshly created Rails app has no models
- We'll generate our first model soon

Views
app/views

- Views live in app/views
- Views by default are ERB (Embedded RuBy) templates
- A freshly create application provides a basic layout ERB template

```
<!DOCTYPE html>
<html>
<head>
  <title>Friends</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

app/views/layouts/application.html.erb

Helpers
app/helpers

- View Helpers live in app/helpers
- Helpers are Ruby modules whose methods are mixed-in your views
- All helpers are mixed into the view automatically

```
module ApplicationHelper
end
```

app/helpers/application_helper.rb

Config
Configuration Directory

- Your Application class lives in a module named after your application
- In application.rb a config object is available which can be use to configure many areas of the application
- Rails will call this Application class (making Rails a framework and not a library)

```
module Friends
  class Application < Rails::Application
    config.encoding = "utf-8"
    config.filter_parameters += [:password]
```

config/application.rb

Per Environment Overrides

config/environments

Contains 3 files which override **application.rb** configuration depending on the "environment" the app is running in

development.rb

production.rb

test.rb

Initializers
config/initializers

Initializer files are loaded after the framework and gem dependencies are loaded

Lots of gems have configurations options which need to be set

Rails includes a few of it's own initializers

Routes
config/routes.rb

- Routes map URLs to controller actions
- The routes file is where all the urls your app responds to are declared

```
Friends::Application.routes.draw do
  # The priority is based upon order of creation:
  # first created -> highest priority.

  # Sample of regular route:
  #   match 'products/:id' => 'catalog#view'
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  #   match 'products/:id/purchase' => 'catalog#purchase', :as => :purchase
  # This route can be invoked with purchase_url(:id => product.id)
```

config/routes.rb

DB Directory

db/

Database migrations will be kept here

Database seeds are usually handled in `db/seeds.rb`

Something called `schema.rb` will also be held here

Lib Directory

/lib

The `lib` directory is where custom code that is _not_ application specific goes

If you might extract something into a gem, it should go into `lib/`

Public Directory
/public

Rails used to keep JS, CSS, and images in public

It often became a junk drawer

Script Directory

/script

Contains a single script; rails.

The rails command contained in the Gem will invoke this script when invoked from the root of your app

Test Directory

/test

By default Rails generates a test directory

It's designed to be used with `Test::Unit`, a testing library that ships with Ruby

We'll be using `RSpec` which, by convention, keeps tests in `spec/`

Vendor Directory

/vendor

Sometimes you'll use external libraries which aren't packaged as gems.

Put those in `vendor/`

Active Record

The Pattern

Architectural Pattern named by Martin Fowler in 2003

Database table expressed/map by a class

An instance of the class maps to one row in the table

Both the class and its instances are enhanced at runtime with persistence logic

Active Record

The M in MVC

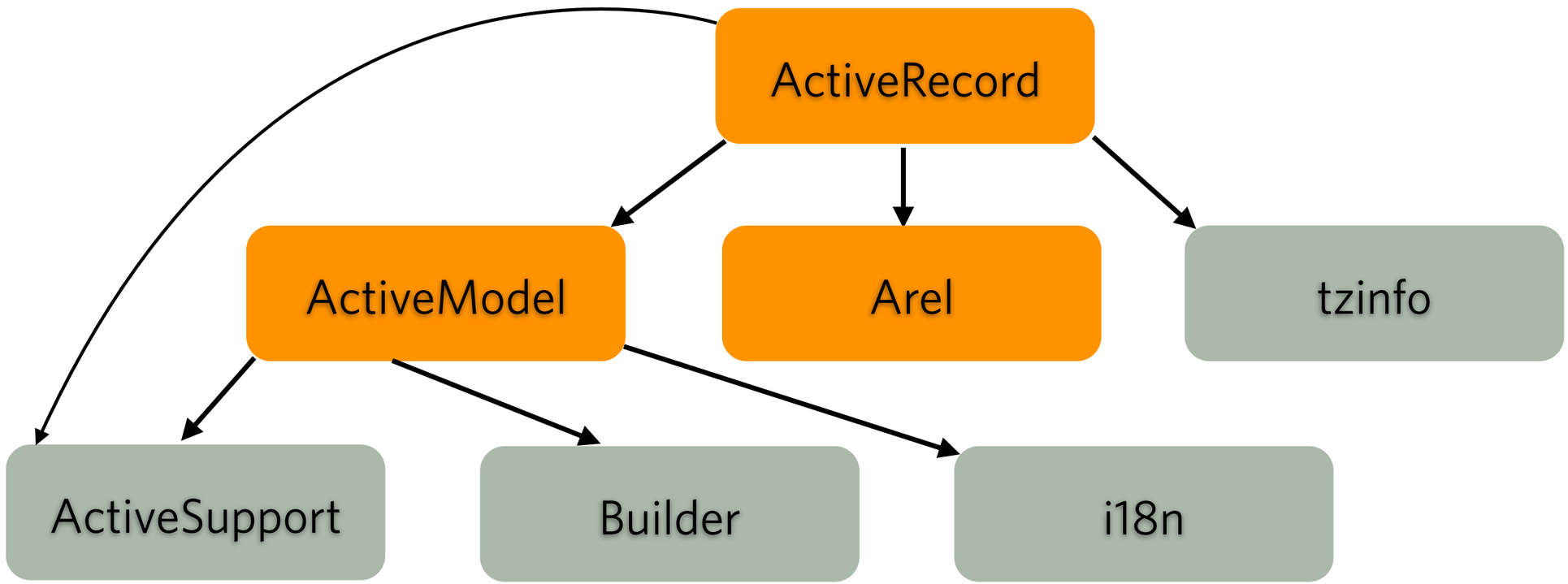
- **ActiveRecord** is a Ruby library that provides an implementation of the Active Record Pattern and a powerful object-relational mapper (ORM), it allow you to:
  - build a domain model by mapping database tables to Ruby classes
  - express relational associations with simple class methods
  - use a simple API for CRUD operations and a fluid API for queries

Active Record

The M in MVC

- **ActiveRecord 3.x** depends on ActiveModel and Arel (a.k.a ActiveRelation)
  - **ActiveModel:** Provides a collection of model related utilities that you can mixin into POROs (Plain Old Ruby Objects) and that ActiveRecord includes by default such as attributes methods, callbacks, conversions, dirty attributes, naming, lifecycle observers, serialization, translation and validations
  - **Arel:** A relational algebra library that replaces the old ad-hoc query generation, provides lazy evaluation and better chaining of query fragments by returning ActiveRecord::Relation which is only evaluated (query executed) when accessing it as a collection

ActiveRecord and Friends



ActiveRecord
SQL Generation

- ActiveRecord objects facilitate CRUD operations against a relational database
- For example, the save method will generate an insert statement as shown below

```
user = User.new
user.name = "Erin"
user.email = "erin@example.com"
user.save
```

```
INSERT INTO "users" ("email", "name") VALUES ("erin@example.com", "Erin")
```

ActiveRecord
SQL Generation

- Persistence operations that are not tied to a specific database row such as finding records are bound to the class object
- For example, the find method is available on a ActiveRecord class object:

```
User.find(42)
```

```
SELECT "users".* FROM "users" WHERE "users"."id" = 42 LIMIT 1
```

ActiveRecord
SQL Generation

- Models are initialize with a hash of attributes
- Getters and setters can also be used

```
user = User.new(:name => "Erin")
user.email = "erin@example.com"

user.persisted? #=> false
user.save
user.persisted? #=> true
```

```
users = User.where(:name => "Phyllis")
```

```
SELECT "users".* FROM "users" WHERE "users"."name" = 'Phyllis'
```

SQLite
Self-contain Server-less DB

The easiest way to get started is SQLite

SQLite is a "self-contained, serverless, zero-configuration, transactional SQL database engine"

```
gem 'sqlite3'
```

Migrations
Managing an evolving Schema

Rails helps us create and manage a database using `ActiveRecord::Migration` and the provided generators

```
class CreateUsers < ActiveRecord::Migration
  # ...
end
```

Generating a Model

Mapping a class to a table

- Rails provides a number of generators for development
- Rails even provides a generator framework for libraries to provide their own generators (more on that later)
- Generators can be accessed through the `rails` executable

```
/>rails generate model user name:string email:string
      invoke  active_record
      create    db/migrate/20121207034947_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/unit/user_test.rb
      create      test/fixtures/users.yml
```

Generating a Model

Mapping a class to a table

- Models inherit from **ActiveRecord::Base**
- Notice we don't declare what columns the DB table has... why not?
- **attr_accessible** is a white list of attributes that are allowed to be mass-assigned

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
end
```

Running Migrations

Managing an evolving Schema

Until now we've used the `rails` executable to run commands on the project

Now we'll need another tool: Rake

Rake
Ruby's Make

Rake is a pure Ruby DSL, no XML files to edit

The Rails development process uses Rake extensively

```
# Rakefile
task :test do
  system("rspec spec/")
end
```

```
/>rake test
```

Running Migrations

Managing an evolving Schema

- Database related **rake** tasks are namespaced under **db**
- The output confirms the table was created
- Let's see what else changed:
  - db/schema.rb
  - db/development.sqlite

```
/> rake db:migrate
==  CreateUsers: migrating ==========================================
-- create_table(:users)
   -> 0.0017s
==  CreateUsers: migrated (0.0018s) =================================
```

Generated Schema

db/schema.rb

- **schema.rb**: is a Ruby representation of the structure of the database
- It should never be edited directly
- The database can be recreated from from it

```
ActiveRecord::Schema.define(:version => 20121207034947) do

  create_table "users", :force => true do |t|
    t.string   "name"
    t.string   "email"
    t.datetime "created_at", :null => false
    t.datetime "updated_at", :null => false
  end

end
```

Rails Console

Rails REPL

- Notice that the User model already knows what fields exist based on the database table columns
- It also adds an **id** field which represents the row id for a particular instance.
- If a record has not been saved yet, the **id** will be nil

```
/> rails console
Loading development environment (Rails 3.2.11)
>> User
=> User(id: integer, name: string, email: string, created_at: datetime, updated_at: datetime)
```

ActiveRecord
Creating a Record

- Here we'll create and retrieve our first user
- **rails console** gives us details about the executed queries
- The create method is equivalent to **#new** and **#save** in sequence

```
>> User.create(:name => "Leonard", :email => "leonard@example.com")
(0.1ms)  begin transaction
  SQL (7.5ms)  INSERT INTO "users" ("created_at", "email", "name", "updated_at")
VALUES (?, ?, ?, ?)  [["created_at", Sun, 03 Feb 2013 17:12:01 UTC +00:00],
["email", "leonard@example.com"], ["name", "Leonard"], ["updated_at", Sun, 03 Feb
2013 17:12:01 UTC +00:00]]
   (0.8ms)  commit transaction
```

ActiveRecord

Finding a Record by Id

- A saved object can be retrieved by its id using the find method
- The find method executes a select statement immediately

```
>> User.find(1)
  User Load (0.3ms)  SELECT "users".* FROM "users" WHERE
"users"."id" = ? LIMIT 1  [["id", 1]]
 => #<User id: 1, name: "Leonard", email: "leonard@example.com",
created_at: "2013-02-03 17:12:01", updated_at: "2013-02-03
17:12:01">
```

ActiveRecord
Many Ways to Find

- ActiveRecord also provides alternative ways to retrieve objects

```
>> User.first
  User Load (0.2ms)  SELECT "users".* FROM "users" LIMIT 1
 => #<User id: 1, name: "Leonard", email: "leonard@example.com",
created_at: "2013-02-03 17:12:01", updated_at: "2013-02-03
17:12:01">
>> User.last
```

ActiveRecord
Many Ways to Find

- Most common operations are encapsulated in simple methods, for example selection all records (select * from table) is accomplished with the **#all** method

```
>> User.all
  User Load (0.2ms)  SELECT "users".* FROM "users"
 => [#<User id: 1, name: "Leonard", email: "leonard@example.com",
created_at: "2013-02-03 17:12:01", updated_at: "2013-02-03
17:12:01">]
```

ActiveRecord
Conditions

- Find “all” with conditions:

```
>> User.find(:all, :conditions => {:name => 'Leonard'})
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE
"users"."name" = 'Leonard'
 => [#<User id: 1, name: "Leonard", email: "leonard@example.com",
created_at: "2013-02-03 17:12:01", updated_at: "2013-02-03
17:12:01">]
```

- As we learned before, the find method executes the selected statement immediately, taking away any chances of refining it

ActiveRecord
ActiveRecord::Relation

- That's where ARel comes in... The where method now returns an ActiveRecord::Relation which can be chained

```
>> users = User.where('email like ?', '%@example.com')
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE (email like
'%@example.com')
 => [#<User id: 1, name: "Leonard", email: "leonard@example.com", created_at:
"2013-02-03 17:12:01", updated_at: "2013-02-03 17:12:01">, #<User id: 2, name:
"William", email: "bill@example.com", created_at: "2013-02-03 18:07:17",
updated_at: "2013-02-03 18:07:17">]
1.9.3-p374 :019 > users.class
 => ActiveRecord::Relation
```

ActiveRecord
ActiveRecord::Relation

- We can, for example, chain an order clause to our where statement:

```
>> users = User.where('email like ?', '%@example.com').order("name DESC")
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE (email like
'%@example.com') ORDER BY name DESC
 => [#<User id: 2, name: "William", email: "bill@example.com", created_at:
"2013-02-03 18:07:17", updated_at: "2013-02-03 18:07:17">, #<User id: 1, name:
"Leonard", email: "leonard@example.com", created_at: "2013-02-03 17:12:01",
updated_at: "2013-02-03 17:12:01">]
```

ActiveRecord

ARel

- In ActiveRecord 3.x we have a lot of new chain-able methods:
  - where
  - having
  - select
  - group
  - order
  - limit
  - offset
  - joins
  - includes
  - lock
  - readonly
  - from

ActiveRecord
Model Associations

- ActiveRecord provides several association types:
  - belongs_to
  - has_many
  - has_one
  - has_and_belong_to_many
  - has_many :through

ActiveRecord
Model Associations

- ActiveRecord covers the different types of relational associations:
  - **One to One:**
    has_one, belongs_to
  - **Many to One:**
    has_many, belongs_to
  - **Many to Many:**
    has_and_belong_to_many, has_many :through

ActiveRecord
One to One

- For a **One to One** association use **has_one** in the owner model and **belongs_to** in the owned model

```
class User < ActiveRecord::Base
  has_one :office
end

class Office < ActiveRecord::Base
  belongs_to :user
end
```

The offices table will contain a foreign key (user_id)

- In a One to One relationship use:
  - **has_one:** when the foreign key is in the other table in the association
  - **belongs_to:** when the foreign key is the current table

ActiveRecord
One to Many

- For a **One to Many** association use **has_many** in the owner model and **belongs_to** in the owned model

```
class User < ActiveRecord::Base
  has_many :comments
end

class Comment < ActiveRecord::Base
  belongs_to :user
end
```

The comments table will contain a foreign key (user_id)

ActiveRecord
Many to Many - HABTM

- The simple join table **Many to Many** strategy is supported by **has_and_belongs_to_many**

```
class Reviewer < ActiveRecord::Base
  has_and_belongs_to_many :articles
end

class Article < ActiveRecord::Base
  has_and_belongs_to_many :reviewers
end
```

The simple join table would be articles_reviewers containing reviewer_id and article_id columns

- In a **has_and_belongs_to_many** association operations work just like they would in a **has_many** association with the difference being that both members of the association can perform the same operations

ActiveRecord
Many to Many - HMT

- The **has_many :through** scheme provides a full blown "connecting" model to connect the two models in the association

```
class Review < ActiveRecord::Base
  belongs_to :reviewer
  belongs_to :article
end

class Reviewer < ActiveRecord::Base
  has_many :articles, :through => :reviews
end

class Article < ActiveRecord::Base
  has_many :reviewers, :through => :reviews
end
```

Scaffold Generator

Quick & Dirty

- Scaffold a good way to get to know Rails
- Don't use scaffold generators in production

```
/>rails g scaffold friend name phone email twitter
      invoke  active_record
      create    db/migrate/20121207161021_create_friends.rb
      create    app/models/friend.rb
      invoke    test_unit
      ...
```

Rails Server

Running our App

- In development Rails uses WEBrick as a server
- WEBrick is not recommended for production use.
- Open a browser a go to `localhost:3000` or `0.0.0.0:3000`

```
/>rails server
=> Booting WEBrick
=> Rails 3.2.11 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-12-07 09:32:40] INFO  WEBrick 1.3.1
[2012-12-07 09:32:40] INFO  ruby 1.9.3 (2012-10-12) [x86_64-darwin11.4.0]
[2012-12-07 09:32:40] INFO  WEBrick::HTTPServer#start: pid=53868 port=3000
```

Generated Index

No Thanks

- New Rails app have an basic home page with info about your environment
- Let's remove it!
- Now we get `No route matches [GET] "/"`
- No problem it's expected since we haven't declared a root route yet

```
/>rm public/index.html
```

Root Route

Declaring a Home

- Visiting localhost:3000 will now try to route to friends#index
- But we get an error because we haven't run the latest migration
- Let's try that now...

```
/> rake db:migrate
==  CreateFriends: migrating
=============================================
-- create_table(:friends)
   -> 0.0115s
==  CreateFriends: migrated (0.0116s)
=========================================
```

```
Friends::Application.routes.draw do
  resources :friends
  root :to => 'friends#index'
end
```

Resource Routes

Rails-flavored REST

- The resources method allows us to create a resource oriented set of routes that map to the traditional CRUD actions

```
Friends::Application.routes.draw do
  resources :friends
end
```

```
/> rake routes
    friends GET    /friends(.:format)          friends#index
            POST   /friends(.:format)          friends#create
 new_friend GET    /friends/new(.:format)      friends#new
edit_friend GET    /friends/:id/edit(.:format) friends#edit
     friend GET    /friends/:id(.:format)      friends#show
            PUT    /friends/:id(.:format)      friends#update
            DELETE /friends/:id(.:format)      friends#destroy
       root        /                           friends#index
```

Thanks

http://integrallis.com