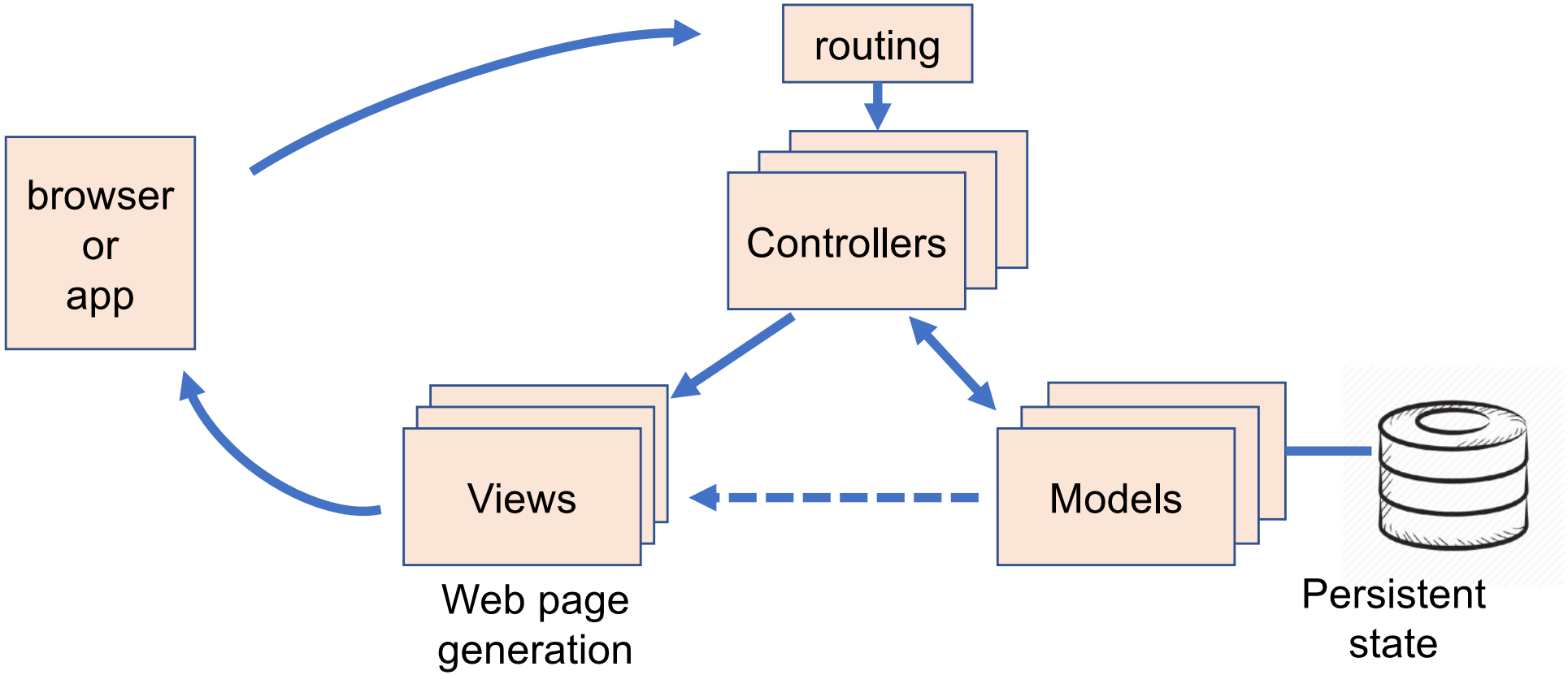
Software Engineering
ECE444

Rails Overview Continued…

Michael Stumm
ECE
University of Toronto

Recall: MVC Architecture:

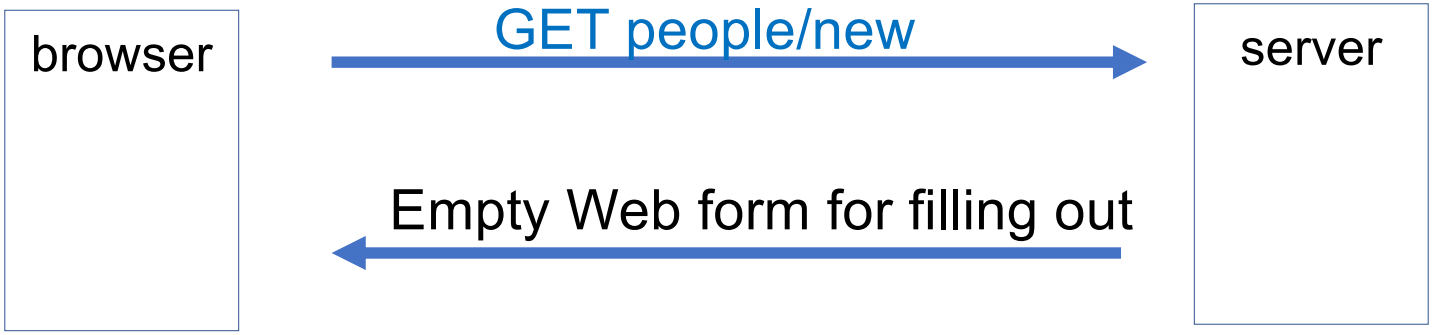
- Architectural pattern: Model-View-Controller (MVC) structure for organizing your code



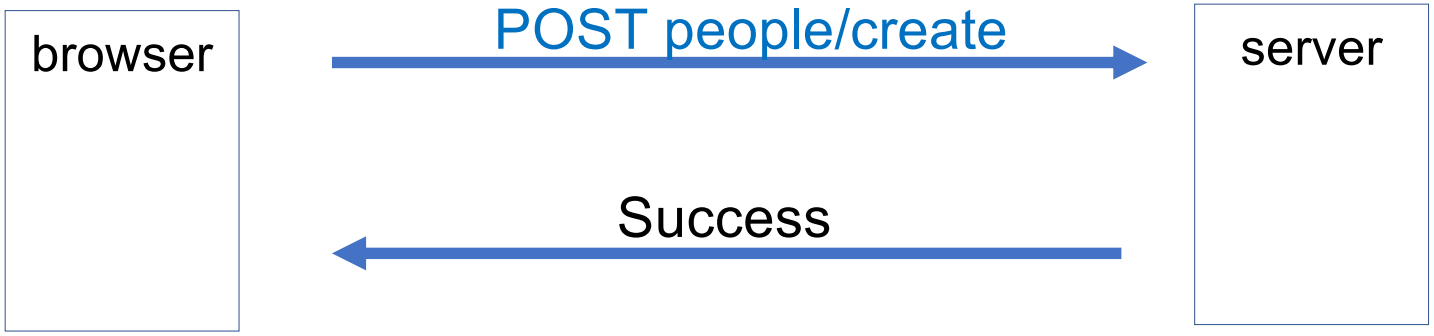
- Developers write code for all three independently
  Rails takes care of the independent plumbing

Recall: RESTful interfaces; e.g., Create

- CRUD methods on resources; e.g.,
  1. request form for new user from `new` action of `People` controller



  2. request person be created by `create` action of `People` controller



Recall: Routing

- Routing table created automagically for you if you use CRUD interface on resources; e.g., students

Prefix	Verb	URI Pattern	Controller#Action
students	GET	/students	students#index
	POST	/students	students#create
new_student	GET	/students/new	students#new
edit_student	GET	/students/:id/edit	students#edit
student	GET	/students/:id	students#show
	PATCH	/students/:id	students#update
	PUT	/students/:id	students#update
	DELETE	/students/:id	students#destroy

Note: Prefix of helper methods to create paths and URI's

- `student_path(@student)` → `/students/#{@student.id}`
- `edit_student_path(@student)`
  → /students/#{@student.id}/edit

Customize helper method:

- get '/students/region-1/top-20', to: 'students#tops', as: "r1_top20"
  ➔ `r1_top20_path` → /students/region-1/top-20

Databases

- Typically relational database:
  e.g., SQLite3, MySQL, PostgreSQL, Oracle, DB2
  - data stored in tables; e.g.,

attribute

lname	fname	salut	DoB	street_addr	. . .

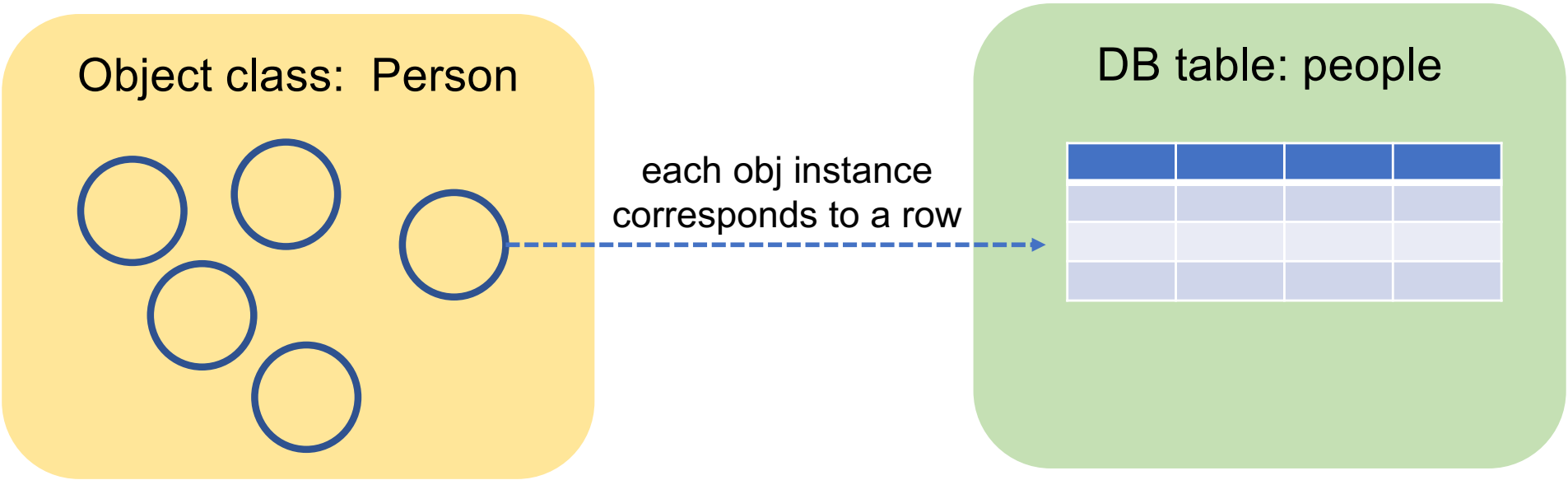
row

- More recently: key-value stores; e.g., RocksDB

- on each DB table, once defined: 4 key operations
  - Create (new row)
  - Read (a row)
  - Update (a row)
  - Delete (a row)
- usually programs issue SQL stmts to DB for each operation

Object Relational Mapping (ORM)

- Alternative approach to using SQL
- Idea: programmer only operates on objects and classes – never on DB directly. Magic maps objs to DB



- Non-trivial to do, but there are such ORM libraries e.g., ODB (C++), Active JDBC & Enterprise Javabeans
- Active Record is ORM layer supplied with Rails
  ➔ the M in MVC

We will describe, but first . . .

Convention over Configuration I

- A common theme in Rails
- Rails makes appropriate assumptions and does everything for you automagically
- Based on idea: if everyone configures app in the same way most of the time then this should be the default way.
- E.g., naming convention
  - DB table names: plural lowercase_snake
    e.g., `star_ratings`
  - Model class name: singular CamelBack
    e.g., `StarRating`
  - File name for model: singular lowercase_snake
    e.g., `app/models/star_rating.rb`

Convention over Configuration II

- Each automagically table gets id column (autoincremented)
- Also `created_on, updated_on` columns
- links to id of another table:
  `xxx_id` where table is plural of `xxx`
- Class methods:
  `StarRating.find( 3 )`
  `StarRating.new`
  `StarRating.where( "…SQL-like where clause…")`
  ....
  many more
-

DB table	Model class	ref to table element
people	Person	person_id
registrations	Registration	registration_id
items	Item	item_id

Active Record Model I

- Create: in `app/models/person.rb`

```
class Person < ApplicationRecord
end
```

- This automagically creates model mapped to people DB table. Automatically finds attributes.
- You can then (e.g., in controller):

```
p = Person.new
p.lname = "Smith"
…
p.save
```

or:

```
p = Person.create( lname:"Smith", fname:… )

Person.create!( lname:"Smith", fname:… )
```

Active Record Model II

- You can then do any one of:

```
people = Person.all  # collection of all ppl
p = Person.first
p = Person.find_by( lname: "Smith" )
p = Person.find( 45 )
p = Person.where( " … SQL query … " )
```

R

- You can then do:

```
p = Person.find_by( lname: "Smith" )
p.lname = "Smithy"
p.save
```

or

```
p.update( lname: "Smithy" )
```

U

- You can then do:

```
p = Person.find_by( lname: "Smith" )
p.destroy
```

D

Active Record Model III

- Active Record uses Convention over Configuration
  1. name of class ('`Person`') auto-identifies DB table ('`people`')
  2. DB auto-queried to identify columns
  3. gives every '`Person`' object attribute getter and setting fcts

     These are in part smart:

```
p = Person.new
p.dob = 'Oct 21, 1999'
p.dob = 15-Nov-89'
```

Two things to note:

1. Modifying an object doesn't modify the DB entry unless you `save/create`
   - `save/create` returns `nil` if unsuccessful
   - `save!/create!` don't, but may throw an exception
2. Deleting an object deletes the DB entry, but doesn't remove the obj, although you can no longer modify it.

Migrations I

- To create and modify DB tables

```
> rails generate migration create_people
```

➔generates a file under db/migrate:

```
xxxxxxxxxxx_create_people.rb

class CreatePeople  ActiveRecord::Migration
   def change
       create_table 'people'  do | t |
          t.string  'lname'
          t.string  'fname'
          t.date  'dob'
          …
          t.timestamps
       end
     end
  end
```

Migrations II

- Then

```
> rake db:migrate
```

➔applies all migrations not yet performed

- This works on all sorts of DBs

In fact: you often work on 3 at the same time

1. production (e.g., Oracle) named library_production
2. dev (e.g., MySQL) named library_development
3. test (e.g., SQLite3) named library_test
   -- gets recreated on every test

which ones?: configured in `config/database.yml`

Migration Advantages

- Can identify each migration, and know which one(s) applied and when
  - Many migrations can be created to be *reversible*
- Can manage with version control
- *Automated == reliably repeatable*

Automating things to make them repeatable is key to managing complexity

Theme: *don't do it—automate it*

- *specify* what to do, create tools to automate

- Applying migration also records in DB itself which migrations have been applied

Migrations: more to come. . .

- E.g., modifying exiting DB:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
   end
end
```

to add columns and create indices

- Can also reverse most migrations…

Rails Cookery

- Augmenting app functionality == adding models, views, controller actions

To *add a new model* to a Rails app:

- (or change/add attributes of an existing model)

1. Create a migration describing the changes:
   `rails generate migration` (gives you boilerplate)
2. Apply the migration: `rake db:migrate`
3. If new model, create model file `app/models/`*model*`.rb`
4. Update test DB schema: `rake db:test:prepare`
5. Eventually deploy: `heroku rake db:migrate`

Seeding the DB

- when developing RoR code: often useful to fill DB
- in `db/seeds.rb`:

```
# seed the DB
many_people = [
  {lname:'Smith', fname:'Susan', … },
  {lname:"Singh", fname: 'Yatish', …},
  . . .
]

many_people.each do |p|Person.create!(p) end
```

```
> rake db:seed
```

Where to place code for logic: Recall

- In theory: arbitrarily spread over models, controllers and views --- different styles

- My recommendation:
  - Start by defining model/DB data and how it is organized ➔ will dominate design, and if designed well (and naturally) the rest will become evident.

Interactions between M, V, and C

GET /students/:56 → students#show

```
class StudentsController <ApplicationController
 def show
   @s = Student.find( #{params[:id]} )
   # if error: handle it;
 end
```

Note:
Instance var @s, set in controller available in view. (Local vars are not!)

show.html.erb:

```
   <html>
   . . .
   <%= Hello #{@s.fname} #{@s.lname} %>
   . . .
```

Interactions between M, V, and C II

GET /students/ → students#index

```
class StudentsController <ApplicationController
 def index
   @studs = Student.all
end
```

index.html.erb:

```
<h1>Students</h1>
<table>
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
```

Interactions between M, V, and C II

GET /students/ → students#index

```
class StudentsController <ApplicationController
 def index
   @studs = Student.all
end
```

index.html.erb:

```
<h1>Students</h1>
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Facts</th>
      <th colspan="3"></th>
    </tr>
  </thead>
```

Index view (cont.)

```
<tbody>
  <% @studs.each do |s| %>
    <tr>
      <td><%= s.fname %></td>
      <td><%= s.lname %></td>
      <td><%= link_to 'Show', s %></td>
      <td><%= link_to 'Edit', edit_student_path(s) %></td>
      <td><%= link_to 'Destroy', s, method: :delete,
                data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
</tbody>
</table>
<br>
<%= link_to 'New Student', new_student_path %>
```

"<%=": if it returns value
"<%": if not

Simple Form Processing I

- View Form Helper designed to work with models
- Core method is form_with
- E.g. simple, artificial form with no fields:

```
<%= form_with do %>
  Form contents
<% end %>
```

generates HTML code:

```
<form accept-charset="UTF-8" action="/"
          data-remote="true" method="post">
  <input name="authenticity_token"
         type="hidden"
         value="J7CBxfH…wJ108nDHX/8Cts=" />
  Form contents
</form>
```

Simple Form Processing II (with fields)

- E.g. new.html.erb:

```
<h1>Add a new person</h>
<%= form_with model: @person, local: true,
              url: people_path
              class: "fancy_form" do |f| %>
  <%= f.text_field :fname %>
  <%= f.text_field :lname %>
  <%= f.submit "Add" %>
<% end %>
```

Don't use Ajax (the default)

Give it the right URI
(Otherwise it goes back to new.)

Simple Form Processing III (with fields)

- will emit new.html:

```
<h1>Add a new person</h>
<form class="fancy_form" action="/people"
    accept-charset="UTF-8" data-remote="true"
    method="post">
  <input type="hidden"
    name="authenticity_token"
    value="NRkFyRWxdYNf…8wL781/xlrzj63TA==" />
  <input type="text" name="person[fname]"
    id="article_fname" />
  <input type="text" name="person[lname]"
    id="article_fname" />
  <input type="submit" name="Add"
    value="Create" data-disable-with="Create" />
</form>
```

Simple Form Processing IV (with fields)

- in `create` controller: access submitted entries through `params` hash:

```
@stud = Student.new
@stud.fname= params[:student][:fname]
@stud.lname= params[:student][:lname]
@stud.save!
redirect_to new_student_path
```

or more conveniently:

```
@stud = Student.new( params[:student])
@stud.save!
```

but is potentially dangerous; better:

```
@stud = Student.new( params.require(student).
                     permit(:fname, :lname) )
@stud.save!
```

helpers for many things in for; e.g.,

check box	color field	date field
datetime field	datatime_local	email field
label	month field	number field
passwd field	phone field	radio button
range field	search field	telephone field
text area	time field	URL field
week field	etc.	etc.

Time to get very busy…

Homework:

1. Read Getting Started with Rails:
   https://guides.rubyonrails.org/getting_started.html

2. Implement the Blog Application described there

3. Do everything up to Section 5.11

You must demonstrate working app and demonstrate working knowledge of app to TAs in Labs next week.