ICOM 6034
Website engineering

Dr. Roy Ho
Department of Computer Science, HKU

Session 6: The server side - Part II

Objectives

- Introduction to Ruby on Rails
  - Ruby – the programming language for Rails
  - Rails – a server-side MVC framework
    - MVC with Rails
    - "Convention over configuration"
    - Implementing the "model" class with object-relational mapping (ORM)
    - Migration and scaffolding
- The "socialization" aspects of Web 2.0
  - Content management systems (CMS)
  - Examples: Drupal and Wiki
- Summary of Part 2:
  - A "framework-centric approach" to rapid website/webapp development
- The Group Project

Introduction to Ruby

Ruby

- The programming language for Rails (“Ruby on Rails”)
  - i.e., “Ruby <=> Rails” is like “PHP <=> Laravel”
- Goal: write more-maintainable code in less lines
- Open-source and purely object-oriented
- Core principle (also adopted by Rails):
  - “Principle of least effort”

Principle of least effort

- Developers don't like to waste time writing tedious code
- Less code also means less bugs
  - But how? By the use of conventions
- Examples:
  - With Ruby, you can have multiple assignment: `x, y = y, x` to swap the values of `x` and `y`
  - Convenient operations on arrays:

    diff = ary1 – ary2

    union = ary1 + ary2
  - Many other examples as we will see

Example of syntax: Euclid's algorithm

- For finding the greatest common divisor (GCD) of two numbers
-
```
def euclid x, y
  while x != y
    if x > y then x -= y end
    if y > x then y -= x end
  end
  x
end
```

This is the return value – the syntax is so concise!

-
```
puts 'Enter two numbers:'
STDOUT.flush
x = Integer(gets)
y = Integer(gets)
puts "The GCD of #{x} and #{y} is #{euclid x, y}"
```

Strings can be quoted by single or double quotes.

Doubly quoted strings can include values for evaluation, e.g., #{expression}

-
```
Enter two numbers:
24
108
The GCD of 24 and 108 is 12
```

General properties (1/2)

- Case sensitive
- All values are objects (there is no “primitive data type”)
- Variables are typeless and need not be declared
- Other conventions:
  - Multiword variables use underscores (e.g., student_name), not camelCase
  - Indentation is two spaces, not tab
  - Ruby uses :: to mean the same as . in Java
- Commonly-used string operations
  - + is used to concatenate strings
  - the to_s method – convert other things to strings
  - downcase, upcase – modify capitalization
  - strip – remove leading and trailing spaces

General properties (2/2)

- **Arrays** are untyped, for example `[1, 2, "hi"]`
- Defining an array by using array literal:
  - `a = [1, 1, 2, 3, 5, 8, 13, "hi"]`
- Array indexes are zero based: `a[2] == 2`
- Arrays can be expanded
  - `a = a + [21, 34]`
    `puts a`
    - `[1, 1, 2, 3, 5, 8, 13, "hi", 21, 34]`
- Ruby has **ranges**, such as `1..10`
  - `..` is an inclusive range, `...` is an exclusive range
    - So `1..4` and `1...5` both mean 1, 2, 3, 4
- Ruby supports **hashing**: `{ :banana => 'yellow', :cherry => 'red' }`

If statements

- Multi-line version:
  - if ***condition*** then
    code
    elsif ***condition*** then
    code
    else
    code
    end
  - The "then" is optional at the end of a line

```
if a < b
  a = 1
end
```

- Single-line versions:
  - if ***condition*** then ***code*** elsif ***condition*** then ***code*** else ***code*** end
    - "then" is required to separate condition from code

```
if a < b then a = 1 end
```

  - ***statement*** if ***condition***

```
a = 1 if a < b
```

Loops in Ruby

- A code block is a set of statements enclosed by begin..end, by do..end, or by {..}

- Ruby has several loops
  - while ***condition*** do
    statements
    end
  - begin
    statements
    end while ***condition***
  - until ***condition***
    statements
    end
  - begin
    statements
    end until ***condition***
  - for ***variable*** in ***range*** do
    statements
    end
  - loop do
    statements
    end
  - ***statement*** while ***condition***
  - ***statement*** until ***condition***
  - loop { ***statements*** }

- Loops are not used as often in Ruby as in other languages
- Instead, Ruby developers tend to use the more powerful iterators

Iterators

- In Ruby, loops are considered “low-level”, which are used only when there is no appropriate iterator

- General syntax of iterator:
  object*.*iterator { |***value***| ***statement*** }
  or
  object*.*iterator do |***value***|
  statements
  end
- … where object is typically a collection, e.g., an array, a range, or a hash, e.g.,

- a = [1, 1, 2, 3, 5, 8, 13]
- a.each { |i| print " #{i}" }
  - Output: 1 1 2 3 5 8 13

each is a predefined *iterator method,* which returns elements of the collection object (“a”) one at a time.

- a.each do |i|
  print " #{i}"
  end
  - Output: 1 1 2 3 5 8 13

- By convention, curly brackets are used for single-line blocks, do…end for multi-line blocks

Other examples of iterators

- ***n***.times – do a block for ***n*** times
  - E.g., “5.times” would do a block 5 times – “5” is also an object
- n.downto(***limit***) – step from ***n*** down to and including ***limit***
- n.upto(***limit***) – step from ***n*** up to and including ***limit***
- string.each_line – get each line from a string

Object orientation

- class Employee < Person # '<' means “extends” – concise!
  @@number = 0 # class variable
  attr_accessor :name # creates a getter & setter for
  # for an instance variable @name
  # attr_reader for getter
  # attr_writer for setter

  def initialize(name) # this is the constructor
  @name = name # instance variable
  @@number += 1 # class variable
  end
  end

- teacher = Employee.new(“Steven")

Rails

What is Rails?

- Rails is a server-side framework for building web apps, based on Ruby
- Powerful support for building MVC, database-centric web applications
- Invented during the development of a real webapp (Basecamp)
- "Convention over configuration"
  - In the past, most frameworks for .NET or Java required a lot of configuration code (e.g., in XML) to specify where everything was, and how the parts were related/connected
  - Rails tends to avoid configuration code; instead, it assumes **standard** configurations (i.e., the "conventions")
  - Extensive conventions can significantly simplify coding
    - E.g., assuming name of model class = name of database table; the 'id' attribute = name of primary key column in the DB, etc.
    - By following common conventions, some developers report that they can reduce the total configuration code by a factor of five over similar Java frameworks.
    - As we will see, many of these conventions are also assumed in Laravel, which was largely inspired by Rails

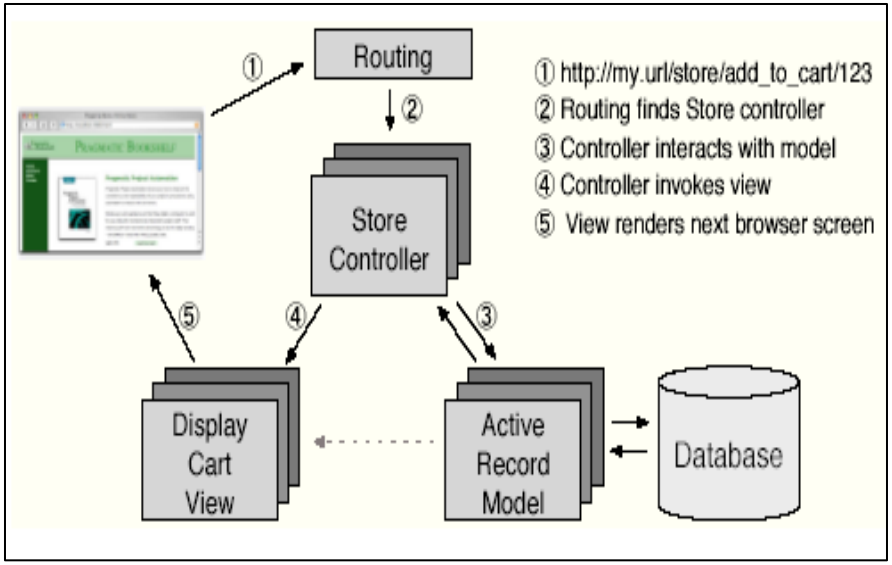
Rails

- Building blocks:
  - The Ruby language
  - A database, e.g., SQlite or MySQL
  - A Ruby-capable web server, e.g., WEBrick, Lighttpd, Mongrel, etc.
  - The Rails framework files and tools

- Installation – two options:
  - Use the GEM packaging tool
    - Ruby and GEM can be installed through an installer. After that, Rails packages can be installed through GEM
  - Use the RailsInstaller for Windows and MacOS

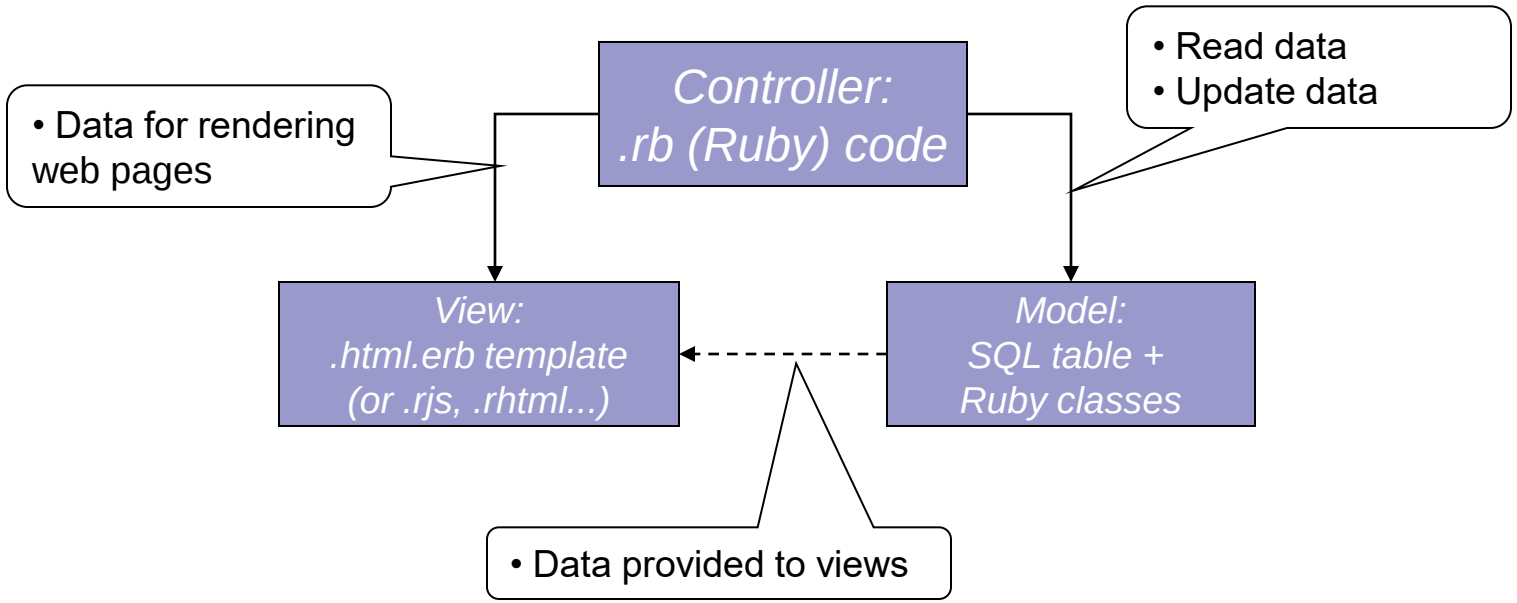
Rails and MVC

- Conceptually similar to Laravel (and other MVC frameworks)
  - An incoming request is first sent to a router.
    - Router sends the request to the appropriate method (action) in the controller code
  - The action may
    - look at the submitted data in the request (if any),
    - interact with the model, and
    - prepare information for the view, which renders an HTML page to the client.



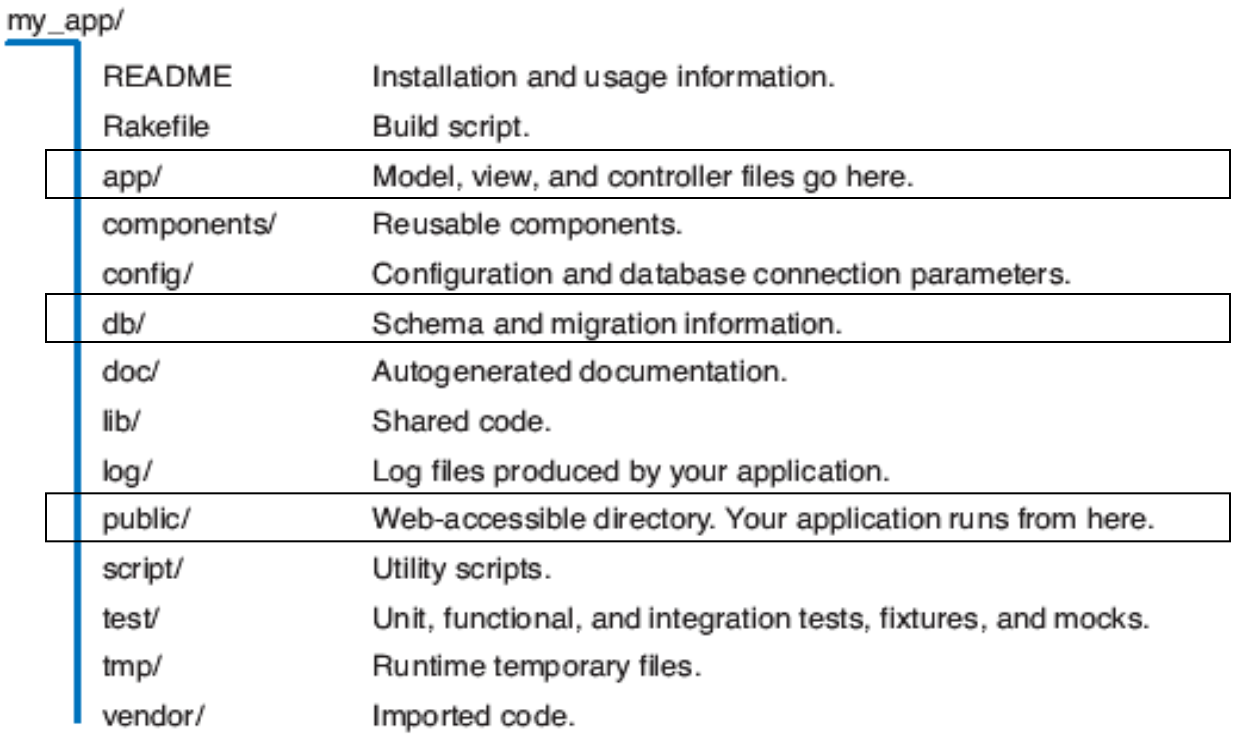
Controller and view

- Controller - logical center of an application
  - Defined in a .rb file - .rb = “Ruby script”
  - Contains methods (“actions”), which are connected to views
- Controller class variables are available to views
  - => Controller sets the variables, which can then be used by the views
- View
  - Programs that contain embedded Ruby code (like .blade.php) for creating web pages and dynamic content
  - File extension: .html.erb (or .rjs or .rhtml)

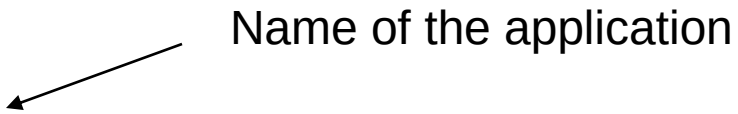


Creating a Rails Project

- rails my_app



Example: Hello World

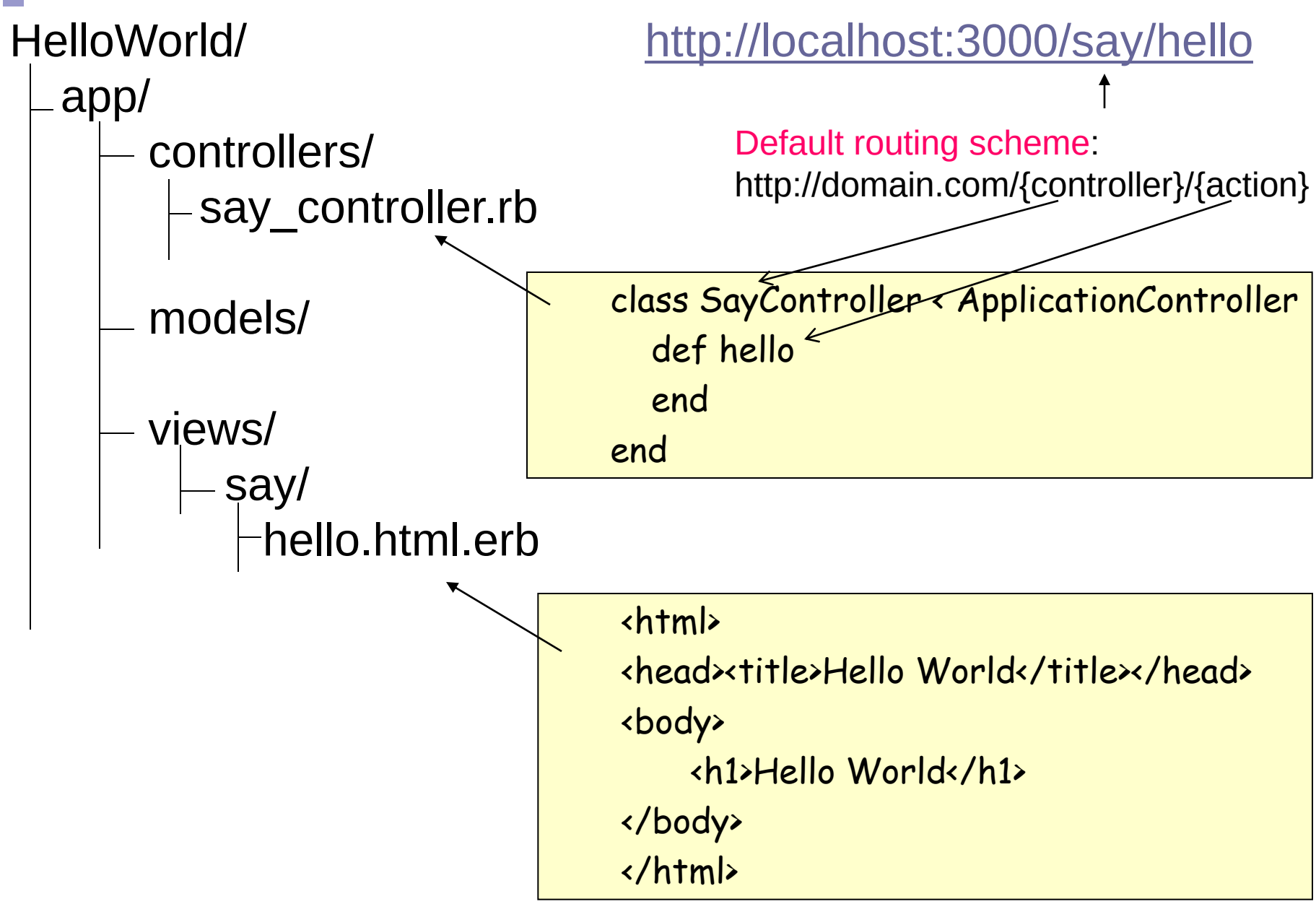


- `rails HelloWorld`
- `ruby script/generate controller Say`

- The created controller:
  - app/controllers/say_controller.rb

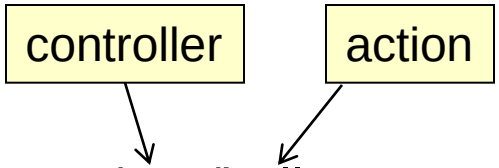
```
class SayController < ApplicationController
end
```

  - ^-- only the skeleton is created
- Let's add an action and the corresponding view…


HelloWorld/
app/
controllers/
say_controller.rb
models/
views/
say/
hello.html.erb
http://localhost:3000/say/hello
Default routing scheme:
http://domain.com/{controller}/{action}
class SayController < ApplicationController
def hello
end
end
<html>
<head><title>Hello World</title></head>
<body>
<h1>Hello World</h1>
</body>
</html>

Rails naming conventions

- Controller naming
  - Request — http://example.com/say/hello
  - Class — SayController
  - Action/method — hello
  - File — app/controllers/say_controller.rb
  - Layout — app/views/layouts/say.html.erb
- View naming
  - File — app/views/say/hello.html.erb
  - Helper — app/helpers/say_helper.rb



say.html.erb is the default layout for the entire Say controller.

"layout" is similar to Laravel's Blade templating engine

Used for simplifying "view"; more on this later

Rails dynamic content

- Dynamic content can be embedded in .html.erb files
  - “embedded ruby code”
- Examples of embedded code:
  - <%= Time.now %>
  - <%= 1.hour.from_now %>
  - Or a loop:
    - <% 3.times do |count| -%>
    - <%= count %>: Hello
    - <% end -%>

“=”: print out the evaluated value of the expression

Note the concise syntax of Ruby

putting the - sign(-%>) at the end eliminates the newlines in the generated html

Rails dynamic content

- Controller's instance variables can be used in View

```
class SayController < ApplicationController
  def hello
    @time = Time.now
  end
end
```

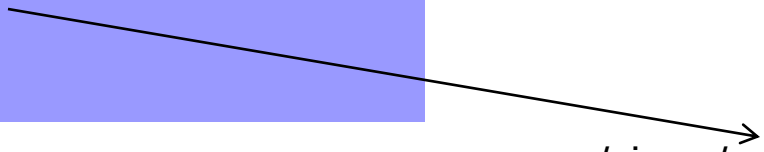
```
<html>
<head><title>Hello World</title></head>
<body>
    <h1>Hello World</h1>
    It is now <%= @time %>
</body>
</html>
```

Layouts

- Specify a unified look across pages
- You can have an layout per view, per controller and/or per application
  - view_name.html.erb
  - controller_name.html.erb
  - application.html.erb
- Once a layout is defined, you can remove those portions from the views

app/views/layouts/application.html.erb

```
<html>
<head><title>Hello World</title></head>
<body>
        <%= yield %>
</body>
</html>
```



app/views/say/hello.html.erb

```
<h1>Hello World</h1>
```

Helpers

- A module (class) containing methods to help a view
  - Each controller can have its own helper
- Used to make the view code shorter and cleaner

app/helpers/say_helper.rb:

```
module SayHelper
  def file_list
    list = '<ol>'
    for file in Dir.glob('*') do
      list += '<li>' + file + "\n"
    end
    list += '</ol>'
  end
end
```

app/views/say/index.html.erb:

```
<html>
<head><title>Index</title></head>
<body>
        <h1>File listing</h1>
        <%= file_list %>
</body>
</html>
```

Other Helpers

- Examples of other built-in helpers
  - <%= distance_of_time_in_words(Time.now, Time.local(2020, 12, 25) %>
    - *“about 1 month” ago*
  - <%= number_to_currency(123.45) %>
    - $123.45
  - <%= number_to_phone(2125551212) %>
    - 212-555-1212
  - <%= number_with_delimiter(12345678, “,”) %>
    - 12,345,678

- Many others - look at the documentation:
  http://guides.rubyonrails.org/action_view_overview.html#overview-of-helpers-provided-by-action-view

Form data and sessions

- User-submitted form data is available in the params hash
  - E.g., name = params[“name”]
  - E.g., age = params[“age”]
  - Same for both GET and POST
- Session variables are stored in the “session” hash
  - session[“loggedin”] = true
  - session[“loginrole”] = “admin”
  - Session ID is stored in session[:session_id]
- To reset a session
  - reset_session

Default Page

- By default, whatever doesn’t match to a controller and action would go to the public directory
- Can change default page to a controller and action
  - edit config/routes.rb

```
# You can have the root of your site routed by hooking up ''
# -- just remember to delete public/index.html.
# map.connect '', :controller => "welcome"
map.connect '', :controller => 'say', :action => 'hello'
```

  - remember to remove public/index.html

Implementing model classes with ORM and ActiveRecord

ActiveRecord

- ActiveRecord is the Object Relational Mapping (ORM) tool in Rails

- ActiveRecord maps:
  - Tables to model classes
  - Rows to objects
  - Columns to object attributes
    - determined at run time
  - Result: much less code required

The info about the database (connection) is specified at config/database.yml. E.g.,

```
development:
  adapter: mysql
  database: students_development
  username: username
  password: password
  host: localhost
```

Since Laravel has borrowed many concepts from Rails, it provides similar convenience

Observe the extensive use of "convention over configurations" and the database-centric design

For example, if a model class is called "Student":

- model (Ruby class) is assumed to be defined in app/models/student.rb
- DB table is assumed to be called "students"

Note the plural form and the small letter "s"

- controller methods are in app/controllers/student_controller.rb
- views are app/views/student/*.erb.html

ActiveRecord model class

- Model classes can be generated by “script/generate model model_name”
  - E.g., “script/generate model person” will create app/models/person.rb

```
class Person < ActiveRecord::Base
end
```

  - which would map to the ‘people’ table in database
  - If you want to map to a table with a custom name:

```
class Person < ActiveRecord::Base
    set_table_name “another_table_name”
end
```

Similar to Laravel, there is no need to declare the attributes (table columns) or object instance variables – these would be recognized automatically during runtime.

- If the name contains multiple camel-case words, the table name has underscores between the words

Class Name	Table Name	Class Name	Table Name
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

Create

- With ORM, you can create a table row by creating an object

```
an_order = Order.new
an_order.name = “Dave William”
an_order.address = “122 Main”
an_order.phone = 2125551212
an_order.save
```

or

```
an_order = Order.new(
    :name => “Dave William”,
    :address => “122 Main”,
    :phone => 2125551212 )
an_order.save
```

or

```
an_order = Order.create(
    :name => “Dave William”,
    :address => “122 Main”,
    :phone => 2125551212 )
```

Note: no need to set a primary key. Rails assumes “id” is the primary key and set autoincrement

“Convention over configuration”

The create method creates a new object and saves it automatically.
i.e., create() ~= new() + save

Read

- Use the find method with a primary key
  - an_order = Order.find(27)
  - an_order = Order.find(params[“order_id”])

- More options for find():
  - an_order = Order.find(:first, :conditions => “name = ‘Dave William’”)
  - orders = Order.find(
    :all,
    :conditions => “name = ‘Dave’”,
    :order => “pay_type, shipped_at DESC”,
    :limit => 10)

Results sorted in descending (DESC) or ascending (ASC) order

:conditions - similar to WHERE in SQL

:order - similar to ORDER BY in SQL

:limit - number of records to retrieve

- Can also write your own SQL
  - orders = Orders.find_by_sql(“select * from orders”)
  - Useful for performance-critical operations

Update and Delete

- Update
  - find the row(s) using find
  - update the fields
  - Save

```
order = Order.find(123)
order.name = "Fred"
order.save
```

- Can also use update()
  - order = Order.update(123, :name => “F”, :address => “hk”)
    - finds, updates, saves, and returns object

- Delete
  - Order.delete(123)
  - Order.delete([1,2,3,4])
  - Order.delete_all([“price > ?”, maxprice])

Other ActiveRecord's conventions

("conventions over configurations" again)

- A table is assumed to have the following columns
  - created_at, created_on, updated_at, updated_on
    - These timestamps are automatically maintained by Rails

- "Find by column_name" for a particular column
  - Dynamically associates a find_by method with column name
  - orders = Order.find_by_name("Dave William")
  - orders = Order.find_by_address("123 Main")
  - Code is much more concise and readable!

- Validators: defined in the model class, automatically run before any data is saved
  - validates_presence_of :title, :description, :image_url
  - validates_numericality_of :price
  - validates_uniqueness_of :title

```
class Person < ActiveRecord::Base
    validates_presence_of :name
end
```

  - You can also define a custom validate method
  - Errors (if any) would be saved in the "errors" instance variable

Scaffolding and Migrations

Scaffolding

- *Scaffolding* is a convenient way to generate the skeleton (e.g., the MVC components) for a new application in a single operation

The name of the model class; its plural form will be used as controller name – "convention" again

- For example,
- $ rails generate scaffold Post name:string title:string content:text

- would generate a skeleton of the application, including some empty class definitions, default layouts/views and migration files

- After the skeleton files are created, you can:
  - Run Migration ("rake") to create the database (more on this later)
  - Modify the MVC components to fit your needs

Scaffolding

Similar to Resource Controller in Laravel, but scaffolding in Rails also generate basic views for showing, editing and creating DB entries.

- $ rails generate scaffold Post name:string title:string content:text
- 15 files are created:

File	Purpose
db/migrate/20180207214725_create_posts.rb.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/fixtures/posts.yml	Dummy posts for use in testing
app/controllers/posts_controller.rb	The Posts controller
app/views/posts/index.html.erb	A view to display an index of all posts
app/views/posts/edit.html.erb	A view to edit an existing post
app/views/posts/show.html.erb	A view to display a single post
app/views/posts/new.html.erb	A view to create a new post
app/views/posts/_form.html.erb	A partial erb file for controlling the overall look and feel of the form used in edit and new views
app/helpers/posts_helper.rb	Helper functions to be used from the post views
test/unit/post_test.rb	Unit testing harness for the posts model
test/functional/posts_controller_test.rb	Functional testing harness for the posts controller
test/unit/helpers/posts_helper_test.rb	Unit testing harness for the posts helper
config/routes.rb	Edited to include routing information for posts
public/stylesheets/scaffold.css	CSS for making the scaffolded views look better

Model

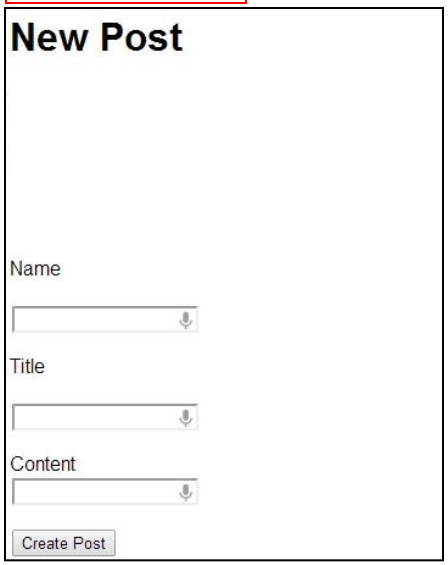
Controller

Several “views” for showing, editing and creating DB entries

The generated views

/posts

New post



/posts when 2 posts added

/posts/1

Migration

- Rails by default assumes that the database is already there and is defined in config/database.yml
- If the database has not been created yet, you may use Rails's Migration support for creating the database tables (similar to Laravel, which actually borrowed the Migration concept from Rails)
  - Advantage: hide the developer from the specific implementation details of the underlying database
- When a model class is generated by "scaffolding", a migration skeleton file would be created, which defines the structure of the corresponding database table
- Example of a migration file:
  - $ rails generate scaffold Student first_name:string last_name:string sid:int

```
class CreateStudents<ActiveRecord::Migration
 def self.up
   create_table :students do |tbl|
     tbl.column :last_name, :string
     tbl.column :first_name, :string
     tbl.column :sid, :integer
   end
 end
 def self.down
   drop_table :students
 end
end
```

You may modify the migration file to fit your needs before running the "rake" command

Rails creates the primary key ("id") automatically and assumes autoincrement

```
CREATE TABLE students (
  id INT NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(255),
  first_name VARCHAR(255),
  sid INT
);
```

"rake db:migrate" would generate a SQL statement like this and send it to the DB to create the database.

Ruby on Rails: summary

- Rails is a server-side MVC framework that advocates the "principle of least effort"
- Common features of modern frameworks (including Rails and Laravel):
  - "Convention over configuration"
    - Much coding avoided by making reasonable assumptions
    - Allows for rapid prototyping/implementation of database-centric applications
  - "Object-relational mapping" (ORM)
    - Greatly simplifies the implementation of model classes and data retrievals
    - DB table creation and management can be simplified by the Migration support
  - Scaffolding generates the skeleton of an application (MVC) in one single operation
- Rails was among the first frameworks to promote the concept of "convention over configuration"
  - After its great success, many other frameworks have adopted the same idea (e.g., Java Enterprise Edition, Apache Struts, Laravel, etc.)
  - E.g., Laravel and Rails are very similar, and in fact, most modern MVC frameworks are similar, so mastering one of them (a popular one) should enable you to easily learn the others
  - When you look for a good MVC framework, do check whether it can provide the above time-saving features (together with a good ORM support)
  - The amount of development time/efforts saved can be huge

“Socialization” in Web 2.0, and CMS

“Socialization” in web 2.0 revisited

- Lots of user-generated contents
  - In contrast to “centralized” content providers in “Web 1.0” – much contents are created by users in web 2.0
  - Photos (Flickr/Instagram), videos (Youtube), blogs (WordPress), tags, bookmarks, etc. – all shared to everyone
    - User-created metadata (e.g., tags, timestamps) of the above
  - A same piece of content might be modified by multiple authors in collaborations.

- Challenge: developing websites for supporting user-generated contents, their metadata, user communities and user collaborations can be tedious

- Solution: use development frameworks that support socialization
  - Content management systems (CMS)
  - Wiki – allows collaborative editing

Types of CMS

- Traditional Enterprise CMS or ECMS (e.g., Microsoft SharePoint):
  - Document (versioning/lifecycle) management
  - Knowledge management (Knowledge base)
  - Workflow management
  - Messaging and email management
  - More specialized CMS designed for enterprises (e.g., for ERP, CRM, etc.)

- Web CMS (our focus):
  - General-purpose CMS/portals (e.g., Drupal, WordPress, Joomla, Plone, etc.)
    - Also have modules for supporting traditional websites and e-commerce functions
  - Forums (e.g., phpBB)
  - E-commerce (e.g., Magento, OpenCart, osCommerce, Drupal Commerce, etc.)
  - Course management (e.g., Moodle)
  - Teams and collaboration, or “groupware” (e.g., Confluence, Kolab, Horde, etc.)
  - Collaborative writing systems (e.g., Wiki, etc.)

General definition

- A CMS supports the creation, management, publishing and sharing of content from multiple users
  - I.e., it manages the complete “lifecycle” of user-generated contents
- The content can be anything users create
- Also manages metadata
  - Metadata is data that “describes” data
  - E.g., when you upload a photo to Facebook, the timestamps, “tags” and textual description are all metadata of that photo.
  - Other metadata: access permissions, automatically generated indexes (of tags/keywords), tag clouds, statistics (e.g., access counts), etc.

Advantages of using CMS

- Convenience
  - Most CMS, once set up, and be configured without coding
  - Popular CMS often have hundreds of themes available
  - Powerful web tools for both administrators and users
  - Management tasks are done through the web browsers, usually through a module called “control panel”
- Lower development cost (compared to the traditional, customized approach):
  - E.g., if all you want is to set up a blog or an e-commerce website selling a handful of products, you really don’t have to develop one in Laravel / Rails by yourself, you can simply use WordPress or Joomla (or anything similar) – there are also many tutorials / books available for non-programmers.
- Flexible - many popular CMS (and their modules) can be used to build both corporate / e-commerce websites and user communities
- CMS in general encourage web standards - browser compatibility for popular CMS should have been well-tested and should not be a problem

Comprehensive supports

Typical content elements supported:

- Front/content pages
- News and articles
- Sitemaps
- Photo galleries
- Calendar and to-dos
- Surveys
- Tags and tag clouds
- Blogs
- HTML forms
- Discussion forums
- Banner advertisements

More functions:

- SEO (keywords, headers)
- Data backup and other scheduled jobs
- Traffic data and statistics
- Templates
- Authorization levels
- Data import & export
- RSS or Atom feeds
- Development tools for plugins and extensions

In most cases, developers only need to deploy the CMS (and their modules), and customize the graphics (or apply a readily-available theme), and the basic functions are ready for use

User communities and traditional websites

- CMS support features for building user communities and enabling user interactions:
  - Account management facilities
  - Direct/private messaging,
  - Discussion forums, etc.

- CMS were originally designed to manage contents and communities, but developers soon realized that they need some “traditional” features like:
  - Traditional website designs, e.g., navigation menus, etc.
  - E-commerce features like product catalogues, shopping carts, payment solutions, product check-out page, etc.
  - => These are supported through extensions / plugins / modules

Architecture and customizations

- CMS are intended to be “one-for-all” solutions. So, most CMS are component-based:
  - There is a core engine, which provides essential functions such as user account management, authentication, system configurations, themes, etc., plus a number of optional modules/components developed by the community
  - Developers can choose which module/component to compose a website
    - E.g., an online bookstore might need a “traditional” e-commerce interface with “shopping-cart” and “comments” modules, etc.
    - E.g., a website supporting an active community might include a “forum” module with private messaging, etc.

- Customizations
  - Modification of the “core engine” or any module/component is not required (nor recommended) in most cases
  - Instead, the “presentation” should be specified through the theming support
    - “Themes ⇔ CMS” is analogous to “CSS ⇔ HTML”
  - For some special websites/application logics, customizations of core engine/modules may still be required
    - The customization effort can be large
    - Alternatively, developers may develop a website with traditional MVC frameworks (e.g., Laravel, Rails, etc.), while CMS are used only in the sub-domains that need CMS’s functions e.g., discussion forums, etc.

Deployment

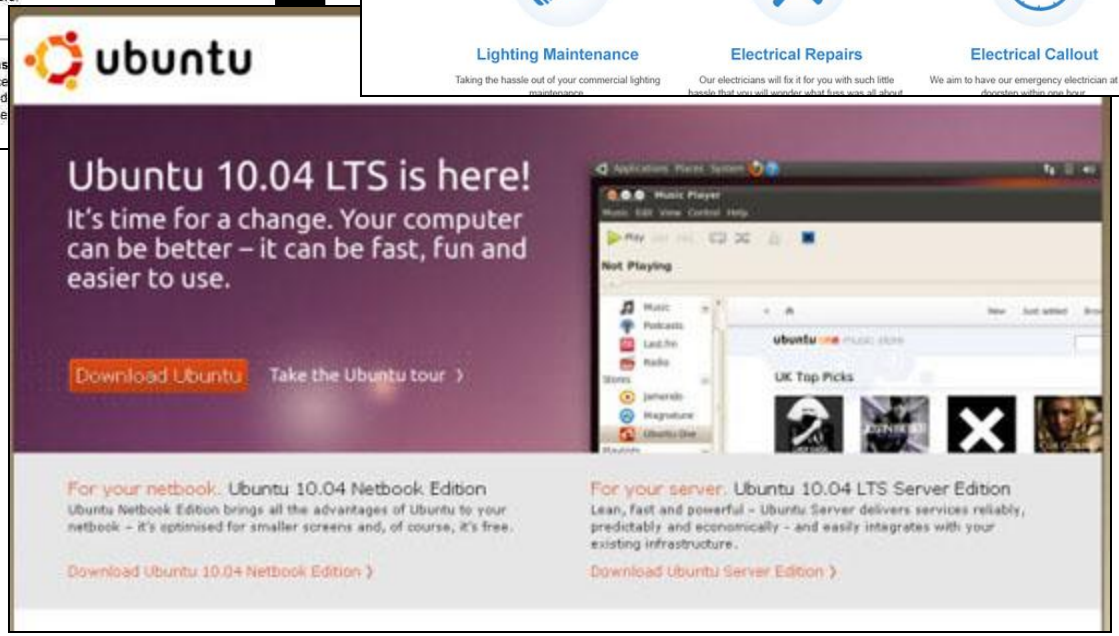
- Download the CMS package and install it on a web server (e.g., our XAMPP platform)
  - Simple - unzip the package, run a script to create the database and set the access permission for a few important files and folders
- => Basic setup is done through web forms, without coding
- Can use the default themes, but they are quite generic
- Custom designs or themes can be downloaded from the web, or implement your own with CSS
- If you have special needs, you can extend the CMS through code
  - E.g., Drupal -> PHP, WordPress -> PHP, Plone -> Python

Examples: Drupal and Wiki

Drupal

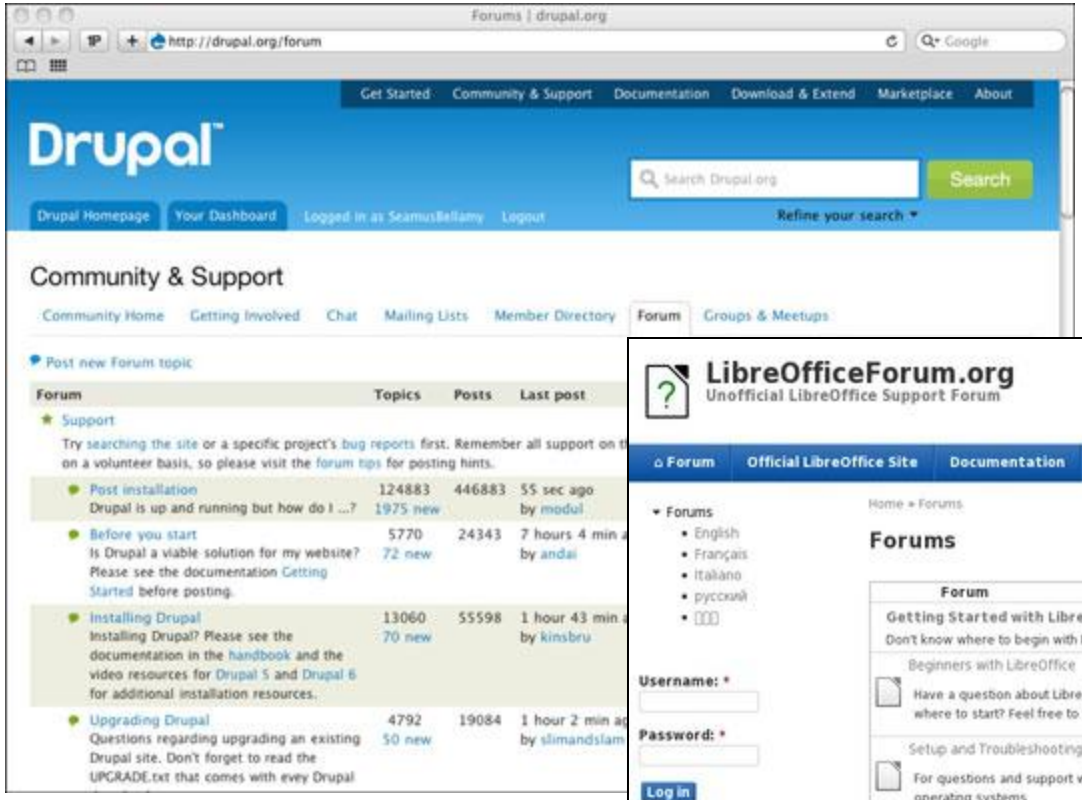
- One of the most popular web CMS
- Flexible, powerful and customizable
  - While some CMS were made for specific usage (e.g., WordPress mainly for blogs; Joomla for e-commerce/social websites, etc.), Drupal is aimed to be a “one-size-fits-all” solution
- Large and active developer community
  - Great community support
  - A large collection of community-developed modules, extensions and plugins
  - While some other CMS are easier to use for beginners (e.g., WordPress), Drupal is used by many large enterprises and governments, e.g., Cisco, Tesla, PayPal, RedHat, Greenpeace, The White House, etc.
- Drupal supports:
  - Common and also customized content elements
  - User communities
  - Customizations of website layout and design
  - Web interface for administration
  - Thousands of (free/paid) themes available

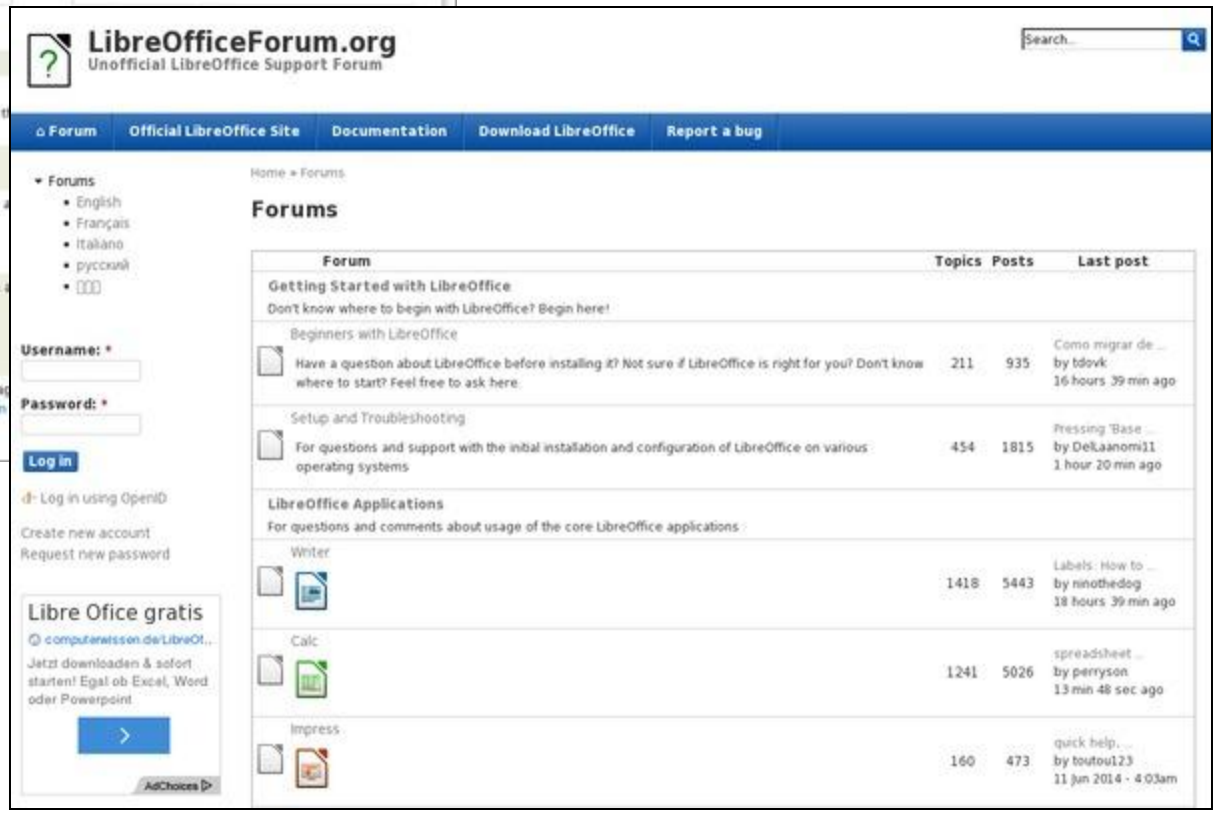
Sample Drupal websites



Organization/
Enterprise websites

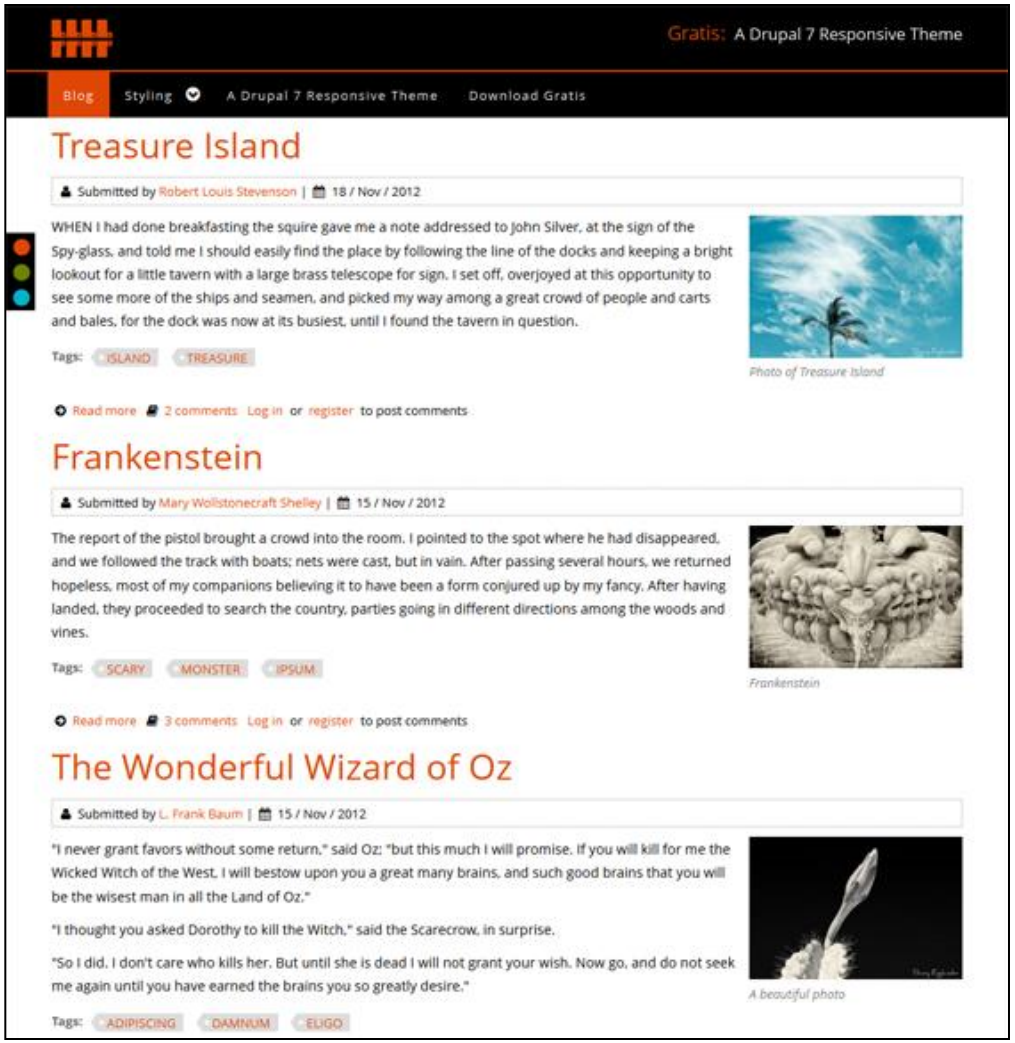
Sample Drupal websites





User communities and discussion boards

Sample Drupal websites



Multi-user blogs

i18n/Unicode supports

Sample Drupal websites

Highly customizable

Installation of Drupal (other CMS similar)

- Download the zip file from Drupal
- Run the provided database scripts to create the database tables
- Web server configurations (set up domain name, documents' path, etc.)
- Create the working/cache folders
- From this point onwards, configuration can be done through the Drupal's web interface

Wiki

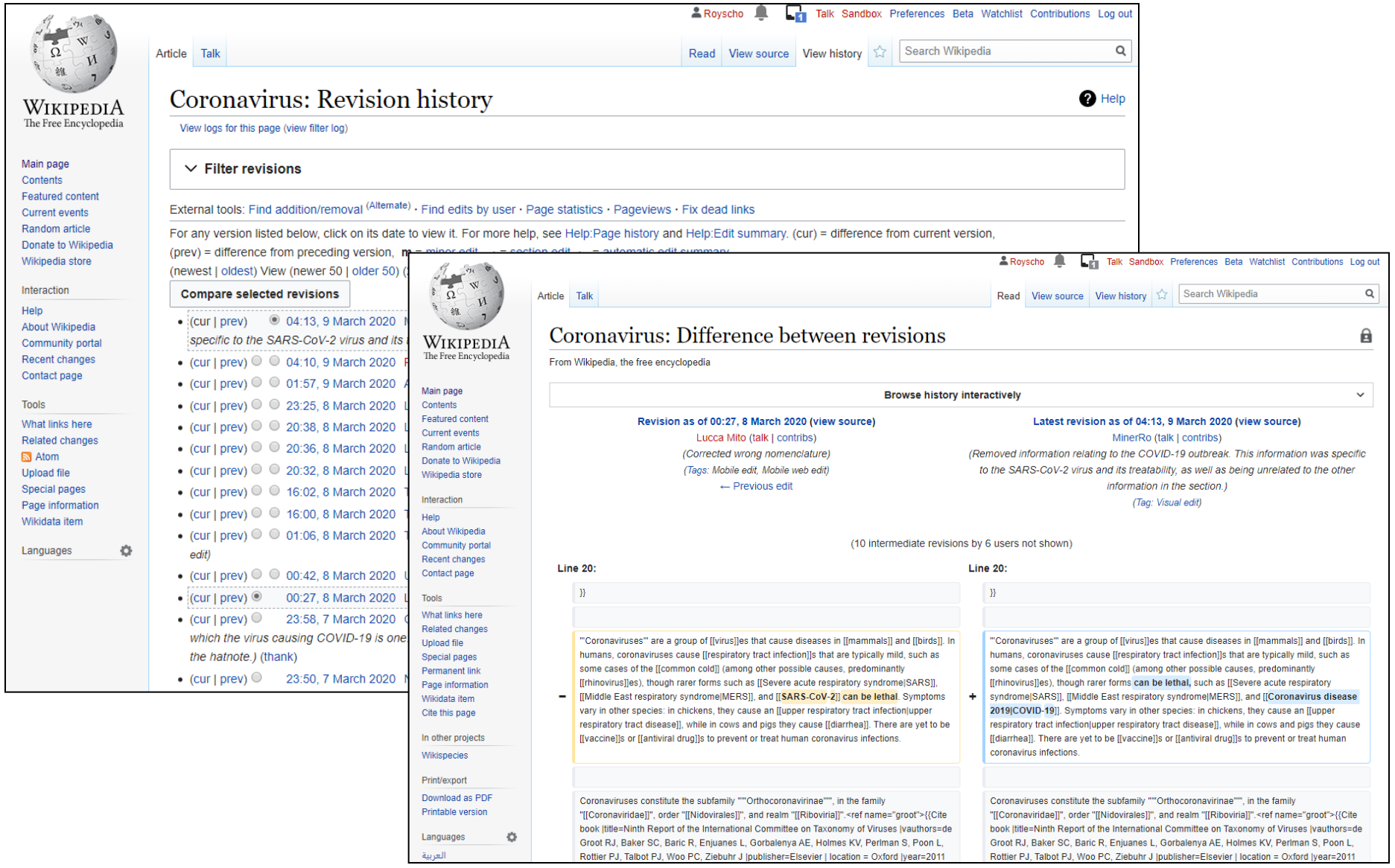
- A wiki is a collaborative authoring tool that allows users to add and edit content collaboratively
- I.e., the same piece of content (e.g., a document) can modified by multiple people together
- Wiki was invented in 1995 for online collaboration on computer programming
- Evolved in early 2000's into a way to facilitate all kinds of online collaborative writing

Wiki

- A page can be reverted to any of its previous states.
- Popular Wiki's:
  - Wikipedia
  - wikiHow: the world's largest "How-To" manual
  - wikitravel.org: a world-wide travel guide
- MediaWiki:
  - Developed for Wikipedia since 2002
  - Scalable and multilingual
  - Open-source (GPL)
  - Features:
    - Versioning of pages
    - Editing features (flexible page/content structures, markup)
    - Community features (discussion, access rights, etc.)

Built with MediaWiki -
Users can download a copy and use in their website/intranet

Revision history in MediaWiki



Summary of Part 2: A “framework-centric” approach to rapid website / webapp development

Summary of Part 2 (Sessions 4-6: design and implementation of web applications)

Covered in Part 2

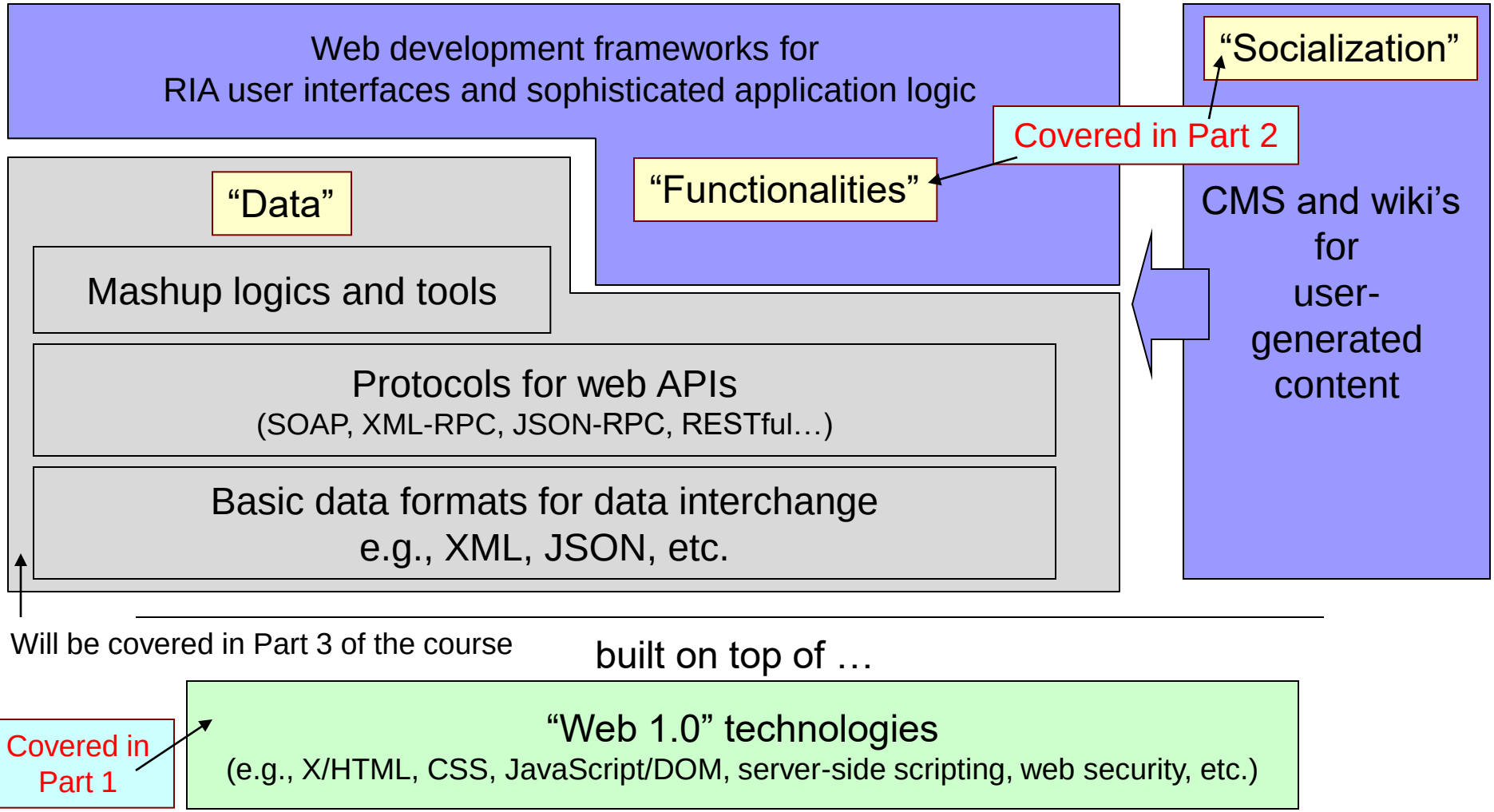
Will be covered in Part 3

- Web 2.0 websites/web apps: "functionality", "socialization" and "data"
  - Sophisticated UI (client-side) and application logic (server-side)
  - Developing these websites using traditional technologies (i.e., HTML, JS, CSS, PHP, etc.) is time-consuming – if not impractical. **Rapid** development tools much desired
  - Frameworks and CMS come to rescue:
    - Client-side (JavaScript/CSS) libraries/frameworks: greatly simplify the development of interactive, and desktop-software-like RIA UIs
    - Server-side MVC frameworks
      - MVC ensures better maintainability and extensibility of websites/apps
      - Simplify tedious tasks like data validations, authentication, database retrievals, etc.
    - CMS/Wiki good for managing user-generated contents and metadata, user communities and collaborative work

    Extensive library/ plugins/ extensions supports!
  - Concepts that further simplify development tasks:
    - Convention over configuration
    - Object-relational mapping, "scaffolding", database migration, etc.

- **Separations** of {document structure/content (HTML), behavior (JS), presentation (CSS), model, view, controller}
  - For maintainability and extensibility

Summary of Part 2 (Sessions 4-6: design and implementation of web applications)



Other popular frameworks and CMS

- We have used jQuery, Laravel, Ruby on Rails, Drupal and Wiki as examples of libraries/frameworks/CMS, but there are many other options – each has its own strengths and weaknesses. Other popular examples:

- Client-side libraries/frameworks:
  - JavaScript/CSS: jQuery, Bootstrap, Foundation, Dojo, Bulma, YAML, etc.
  - Mobile: PhoneGap, AppDeck

- Server-side frameworks (mostly MVC or variants):
  - PHP: Laravel, Symfony, CakePHP, CodeIgniter, Zend Framework, etc.
  - Java: Java Server Faces, Apache Struts, Spring Framework
  - JavaScript: Express on Node.js
  - Ruby: Ruby on Rails
  - Microsoft: ASP.NET MVC

See Moodle for the links, under the "Post-class self-learning resources" in Sessions 4-6

- CMS:
  - General-purpose CMS/portals (e.g., Drupal, WordPress, Joomla, Plone, etc.)
  - Forums (e.g., phpBB)
  - E-commerce (e.g., Magento, OpenCart, osCommerce, Drupal Commerce, etc.)
  - Class/classroom/course management (e.g., Moodle)
  - Teams and collaboration (e.g., Confluence, Kolab, Horde, etc.)
  - Collaborative writing systems (e.g., MediaWiki, etc.)

A “framework-centric” approach to rapid website / webapp development

- The idea is to “center” the implementation of a website around a set of chosen client-/server-side frameworks:

- Assuming that your project requirements, e.g., intended functions, delivery schedule, UI, etc., **have been well-defined**

- Then, choose the most suitable libraries/framework(s) -- “most suitable” in terms of:
  - Whether it fulfills the general selection criteria of web technologies (see Session 1)
  - Shortest development time (least effort)
    - Ask yourself: apart from the graphical designs (which should be customized for each project), can a library/framework (and its plugins/modules) meet most of your project’s functional needs? (note: it should, probably in 80%+ of the time)
    - You should then be able to shortlist one or two good candidates (except for very special web apps (e.g., online editors or games)
  - Rooms for future expansion, i.e., the extensibility of the tools

“Framework-centric development”

- Other factors/tradeoffs to consider when choosing the library/framework:
  - Quick prototyping vs. rigorous designs vs. interoperability:
    - Some frameworks (e.g., PHP frameworks, etc.) are better for quick prototyping (=> shorter “time-to-demo”, or “time-to-market”)
    - Some (e.g., Apache Struts, etc.) emphasize rigorous object-oriented and MVC designs (=> possibly more maintainable/extensible in the long run)
    - Some (e.g., some Java/.NET frameworks) are easier to be integrated with backend components: e.g., it may be easier for a Java-based framework to be integrated with backend Java Enterprise applications

Tradeoff like these need to be considered carefully.

  - Other requirements (e.g., multi-device support, customized reports, PDF output, HTML5 animations, connection to other web APIs, fancy widgets, e-commerce features, etc.)
    - Most of these can be satisfied through the extension/plugin supports of the libraries/frameworks (or the underlying programming language)
    - E.g., there are many PHP/Java libraries available for generating PDF documents
    - => You will need to explore the extension/plugin library of each tool and do a careful comparison before deciding which framework to use
  - Expertise of your development team
    - Can greatly affect the development time and quality of the products

“Framework-centric development”

- After a framework is chosen, use its features as much as possible and avoid re-implementing features by yourself
  - Less code = less bugs
  - Again - explore the available plugins/libraries developed by the community
    - Except UI components and customized application logic, most of the tasks we have to do have been done by the community already.
  - Implement UI components with JS/CSS libraries/frameworks (e.g., jQuery, Bootstrap, etc.) and their plugins, start with some readily-available templates whenever possible
- For some features that are only available in another framework/library, seek to use both at the same time
  - E.g., jQuery + Bootstrap
- Use CMS/Wiki for community or content-rich, frequently-updated websites (or a sub-domain of the website)
- Choosing the right framework(s) is important
  - Before deciding, try their demos, read their documentations, explore their extension/plugin libraries, check the activeness and scale of their user bases and developer communities, product roadmap, etc.
- Although new libraries/frameworks/CMS appear almost every year, *many of them do not last long*, and most of them are conceptually very similar to what we have introduced (e.g., Laravel, Rails, Drupal, Wiki, etc.).

Post-class self-learning resources

- Getting started with Rails

  http://guides.rubyonrails.org/getting_started.html

- RailsInstaller: an easy way to install Ruby on Rails on Windows and MacOS

- Take a brief look at other CMS, *and try their demos* (if available):
  - General-purpose CMS/portals (e.g., Drupal, WordPress, Joomla, Plone, etc.)
  - Forums (e.g., phpBB)
  - E-commerce (e.g., Magento, OpenCart, osCommerce, Drupal Commerce, etc.)
  - Class/classroom/course management (e.g., Moodle)
  - Teams and collaboration, or "groupware" (e.g., Confluence, Kolab, Horde, etc.)
  - Collaborative writing systems (e.g., MediaWiki, etc.)

- References:

  - B. Somerville et al. Beginning Rails 6: from novice to professional. 4th edition. Apress. 2020.
  - T. Tomlinson. Beginning Drupal 8 (e-book). Apress. 2015.

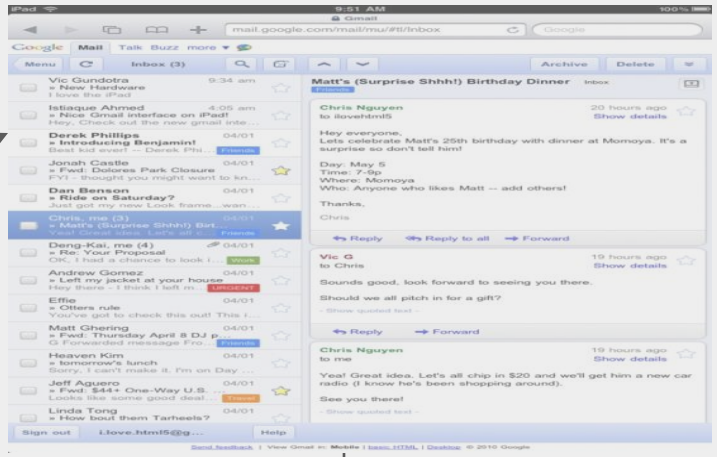
(Please see Moodle for the links)

Summary of scope

Part 1 (done):

Websites becoming "web apps" -> more sophisticated, lots of interactions with users, "Web 2.0", etc.

Part 2 (done):



Integration and interoperability issues -> how to reuse existing & remote data?

Part 3 (lectures 7-9):

YouTube, Gmail, Amazon, online databases, Maps, updated event lists, YOUR websites, ……..

The web/cloud(s)

Part 4 (lecture 10):

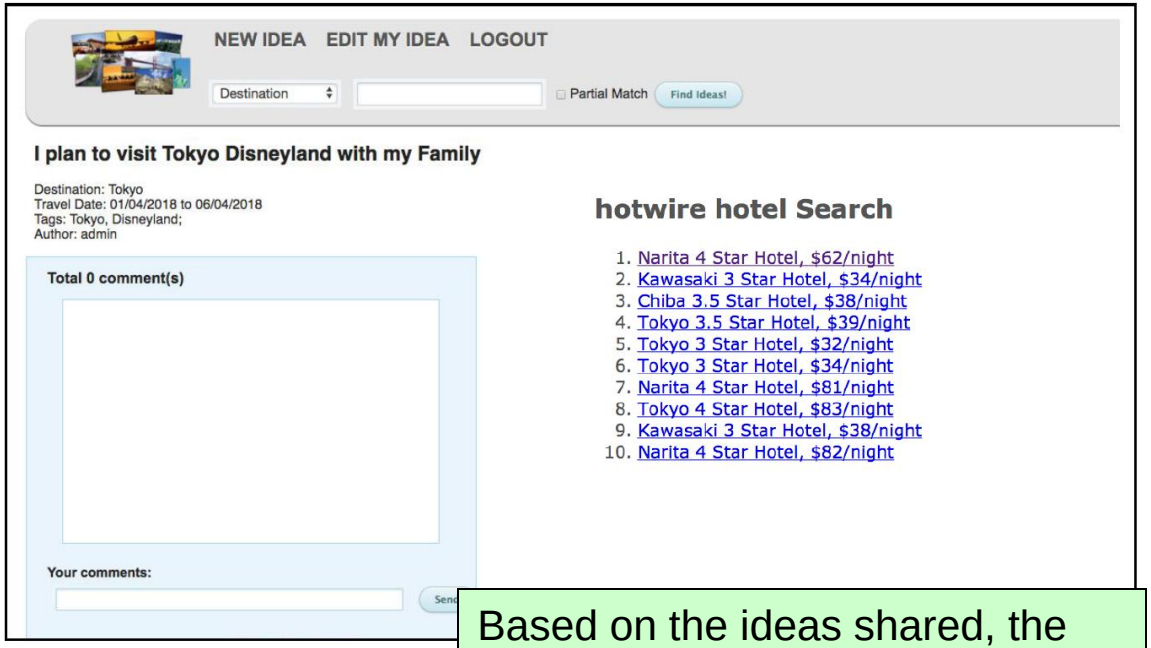
You have a great website, how to make it loaded fast at users' computers, and most important… popular?

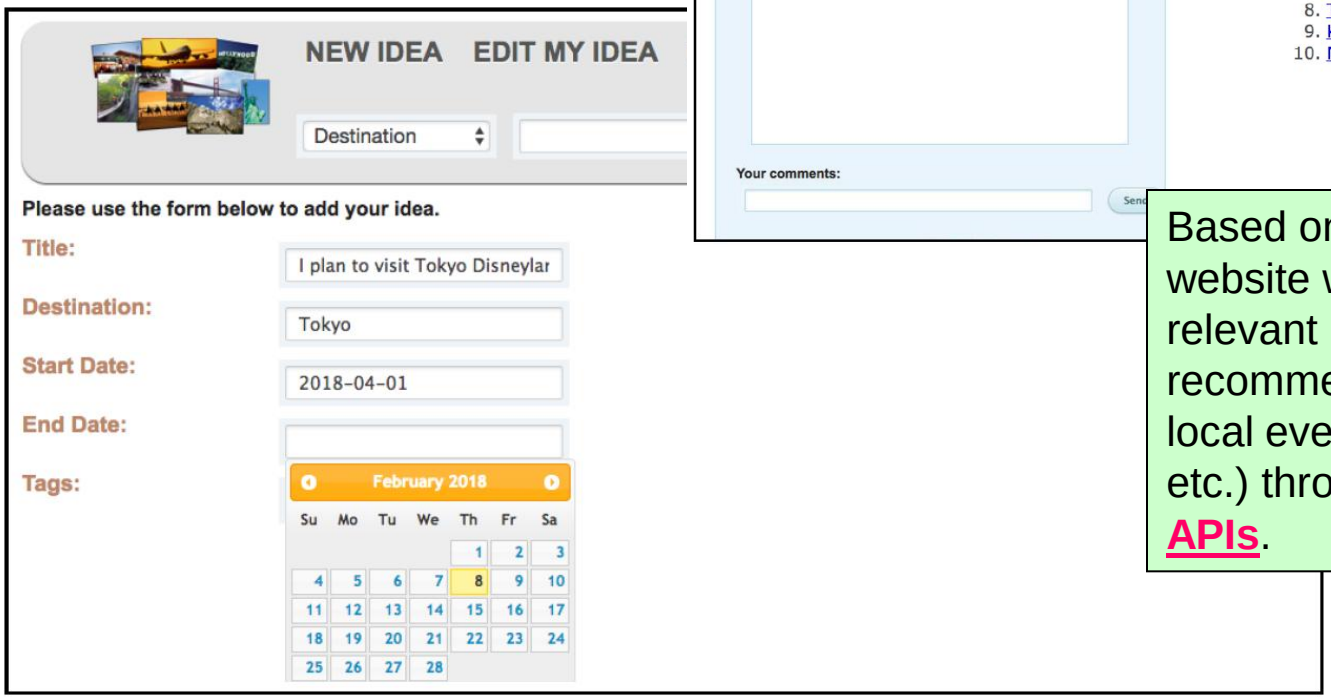
Optimizations

Group Project

Suggested topic of the project (you may propose your own)

A website for a group of friends to share, search and discuss about **travel ideas**, destinations and schedule.



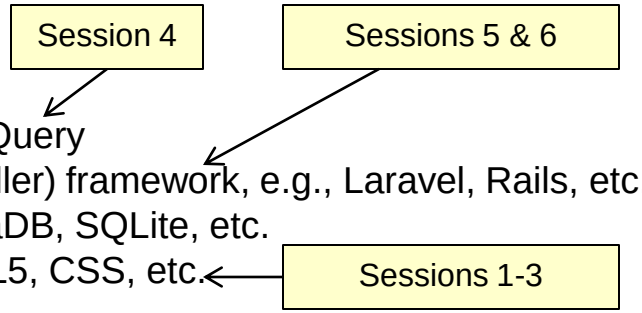


Based on the ideas shared, the website will provide further, relevant information (e.g., hotel recommendations, flight schedule, local events/festivals, landmarks, etc.) through the use of **web APIs**.

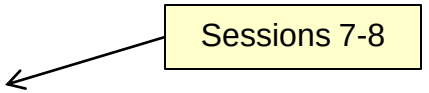
Project details

- Goal: to implement a simple but fully-functional website

- Technologies involved:
  - A client-side (Javascript) library, e.g., jQuery
  - A server-side MVC (model-view-controller) framework, e.g., Laravel, Rails, etc.
  - Any popular DBMS, e.g., MySQL/MariaDB, SQLite, etc.
  - Other standards involved include HTML5, CSS, etc.



- Recommended technologies:
  - Laravel + jQuery + MariaDB; or
  - Rails + jQuery + SQLite; or
  - Any other popular client-side (JS) library and server-side **MVC** frameworks and DBMS (please discuss with us beforehand)



- The website will make use of at least two public web APIs found at programmableweb.com - in a meaningful way

- Workload: the “complexity” of the resultant website/website components will be similar to that of the assignment. But unlike the assignment,
  - The system design has to be done by the group; and
  - All coding has to be done by individuals from scratch

Project details

- Each project group: 2 members by default
  - 3+ – please discuss with us about what functions to be added to balance the workload
  - You are encouraged to use online means for group meetings and communications
  - 1 member – possible but not recommended.
    - Please read the “one-person mode” section in the project specifications about how the project can be simplified.
    - The workload for a one-person group will be a little bit higher than half of the original workload (for 2 people).

- Approval of topic and technologies:
  - You might propose your own topic and technologies, but please discuss with us beforehand (we will assess the feasibility)
  - The choice of web APIs will also need to be confirmed with Steven before implementation

- Important dates:
  - Project: announced by Feb 6, 2021, through email
  - Names of group members sent to Steven: by Feb 20, 2021
    (Note: if you plan to work on your own, please also inform Steven. If we do not receive your names, we will form a group for you.)
  - Your confirmed project arrangements and job distributions sent to Steven for approval (or discuss with him, if you want to propose your own topic or work items): by March 6, 2021
  - Due date of deliverables: April 10, 2021
  - **Late penalty**: 1 point (out of 40), per day. *No submission would be accepted after April 30, 2021*.

- **Please start to group as soon as possible.**

Grading scheme

- Deliverables:
  - A group report (done by whole group)
  - Implementation and coding (done by individuals)
  - A short individual report (done by individuals)

- Total mark = 40 points

- [12] Project mark (would be the same for all group members)
  - [8] Group report
    - Website description, functional specifications, web APIs used, system/database designs, etc.
    - The work distribution
  - [4] Usability of the website
    - Features, interface design/layout, bugs, etc.

- [28] Individual mark
  - [5] Individual report
    - High-level system design of each module implemented, details of work done, program files developed, etc.
  - [12] Implementation of modules
    - Correctness, MVC, data validations, etc.
  - [8] Quality of implementations
    - Are “separations” done properly?, quality of module designs, etc.
  - [3] Peer evaluation (details TBA)

Please refer to the spec. for the detailed deliverable and grading scheme.

Project - Hints

- The project, if the default topic & technologies are adopted, should not be difficult if you are able to do the assignment.
  - They share a number of similar features, e.g., basic website designs, form validations, database access & searching, user authentication, etc.
- And, the assignment should be easy if you can do the labs.
- => So, please consider starting with the labs, and make sure that you understand how the model answers work.