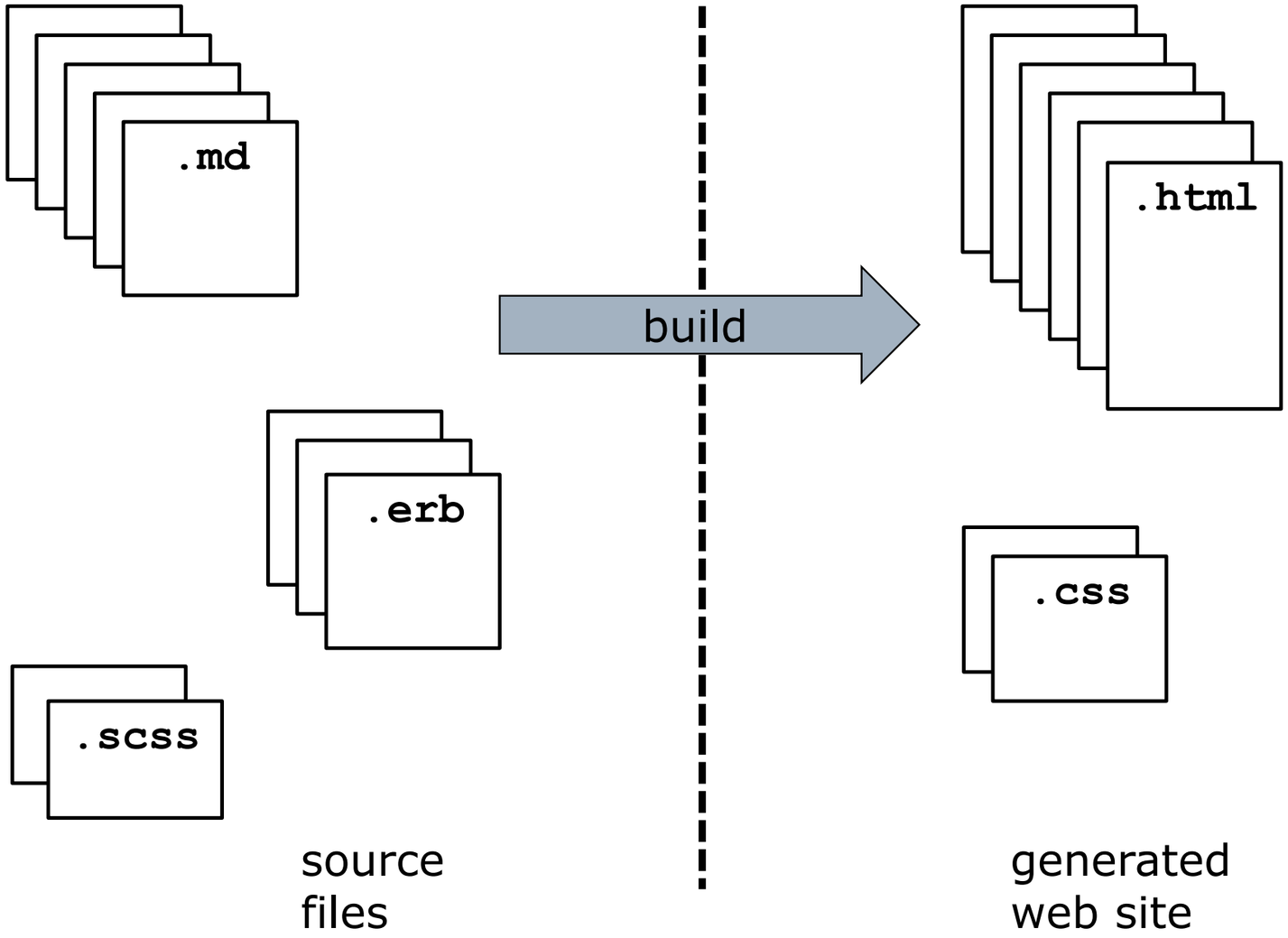
Static Site Generation

Lecture 21

What is Static Site Generation?

- Using a *program* to produce HTML pages
  - Analogous to compiling programs
  - Translation maps source code → machine code
  - SSG maps source code → html, css, and other assets for site
- Development cycle:
  - Write source
  - Compile (aka *build*)
  - Test/inspect result
- Examples of translators
  - Jekyll (used for GitHub Pages, see github.io)
  - Middleman
  - Lots more, see: staticsitegenerators.net

The Build Process



Middleman: A Ruby Gem

- Project is a directory (eg mysite)

```
mysite$ middleman init
```

  - Creates configuration files, README, Gemfile, etc
- Develop: Create/edit files in **source/**
  - Organize CSS, images, etc into subdirectories of source
- Preview locally (no build needed)

```
mysite$ bundle exec middleman server
```

- Build the site (from the source)

```
mysite$ bundle exec middleman build
```

  - Result is placed in **myproj/build**
- Deploy: copy/rsync/ftp contents to server

```
mysite$ rsync -avz --del myproj/build ~/WWW
```

Deployment Option: GitHub

- GitHub Pages: serves repo as web site
  - URL **`https://org.github.io/repo/`**
  - Settings > Pages > Build > Source > Deploy from a branch
  - Select branch name, eg gh-pages, and optionally a folder
- Alternative: GitHub Action (aka a CI/CD pipeline)
  - Defined by YAML file in **`.github/workflows/`**
  - Responds to an event (eg push on main)
  - General purpose: Runs a build process, deploys the result
- Advice: Use relative links (notice path in URL)

```
# config.rb
activate :relative_assets
set :relative_links, true
```

- Helpful: add URL to repo's About section

Why Bother?

1. Code reuse and single-point-of-control over change
2. Authoring of content in a language that is more human-friendly
3. Parameterized generation of markup and content

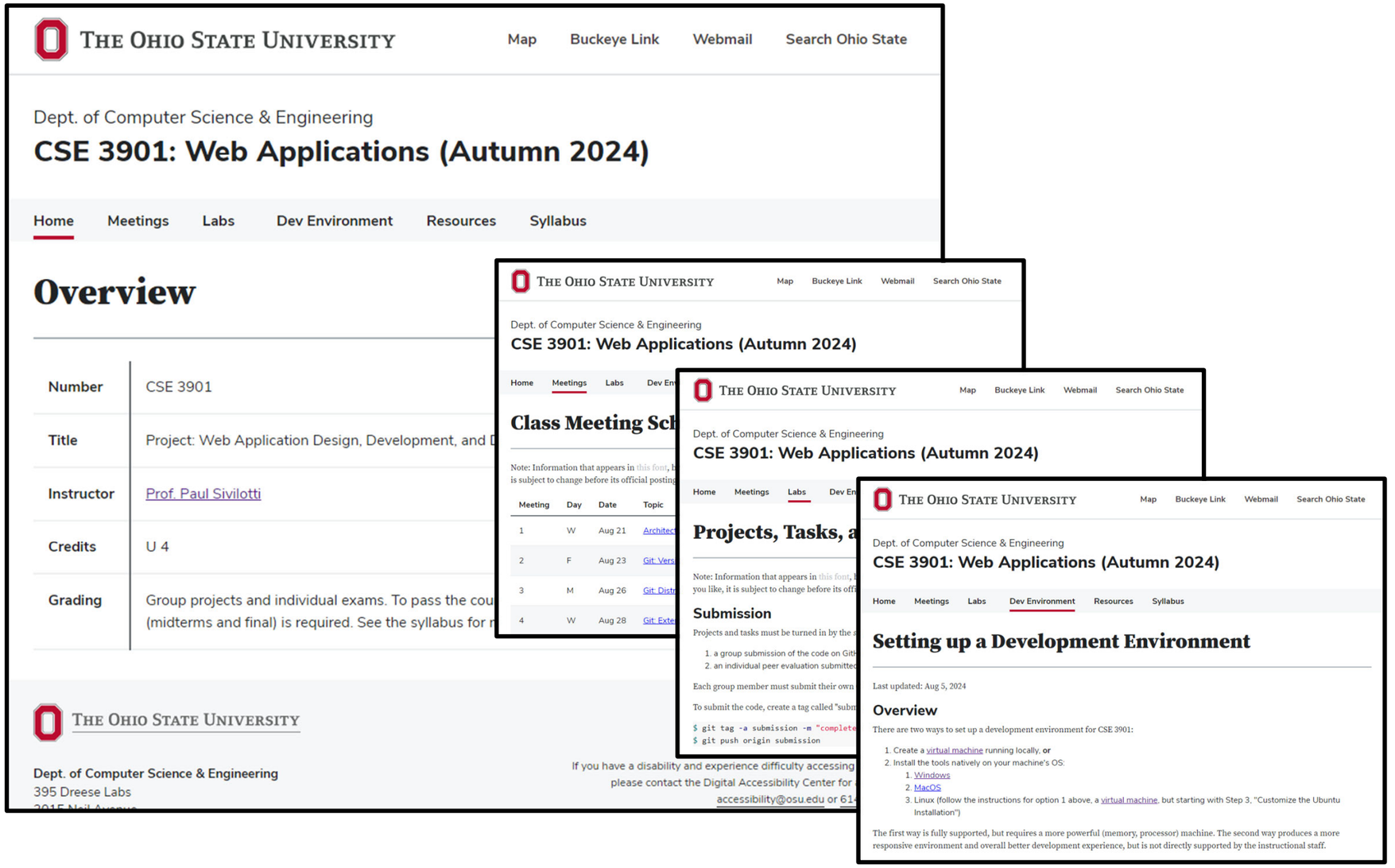
Let's look at each of these benefits in turn…

Why Bother?

1. Code reuse and single-point-of-control over change
2. Authoring of content in a language that is more human-friendly
3. Parameterized generation of markup and content

Let's look at each of these benefits in turn…

Motivation #1: Visual Identity (Example)



Motivation #1: Visual Identity

- For a unified look, use same headers & footers
  - Example: OSU web sites share nav bar
  - Example: course web site shares navigation and footer
- But duplication of code is evil
  - Corollary: cut-and-paste is evil
  - Destroys single-point-of-control over change
- Solution:
  - Put the shared HTML in one file (a *partial*)
  - Include this file in each page
  - But HTML does not have such an include mechanism…

ERb: Embedded Ruby

- General mechanism for *templating*
  - "Template" = a string (file) used to produce a final string (file)
  - Contains (escaped) bits of ruby
    - `<% ruby code %>` a scriplet: executes (ruby) code
    - `<%= ruby expr %>` an expression: replaced with its value
    - `<%# text %>` a comment: ignored
  - Example

```
<%# example.txt.erb %>
This is some text.
<% 5.times do %>
Current Time is <%= Time.now %>!
<% end %>
```

- Command line tool erb processes the template, produces result

```
$ erb example.txt.erb > example.txt
```

- Naming convention: ***filename.outputlang*.erb**
  - Example **index.html.erb**
- Many alternative templating languages exist, eg HAML

Generation of Site

- Source files in **`source/`** subdirectory

```
$ ls source
index.html.erb  syll.html.erb
meet.html.erb
```

- Compile

```
$ bundle exec middleman build
```

- Result after building

```
$ ls build
index.html meet.html syll.html
```

Partials

- A document *fragment* included in other documents
- Include in a template using the **`partial`** function

```
<body>
  <%= partial "navigation" %>
  …
  <%= partial "footer" %>
</body>
```

- Partial's *filename* begins with '_'
  - ie **`_navigation.erb`**

```
<div class="navbar">
  <ul id="site-nav"> <li> … </li> </ul>
</div>
```

- Note: '_' is omitted in the function argument!

Generation of Site with Partials

- Source files in **source/** subdirectory

```
$ ls source
_footer.erb       meet.html.erb
_navigation.erb   syll.html.erb
index.html.erb
```

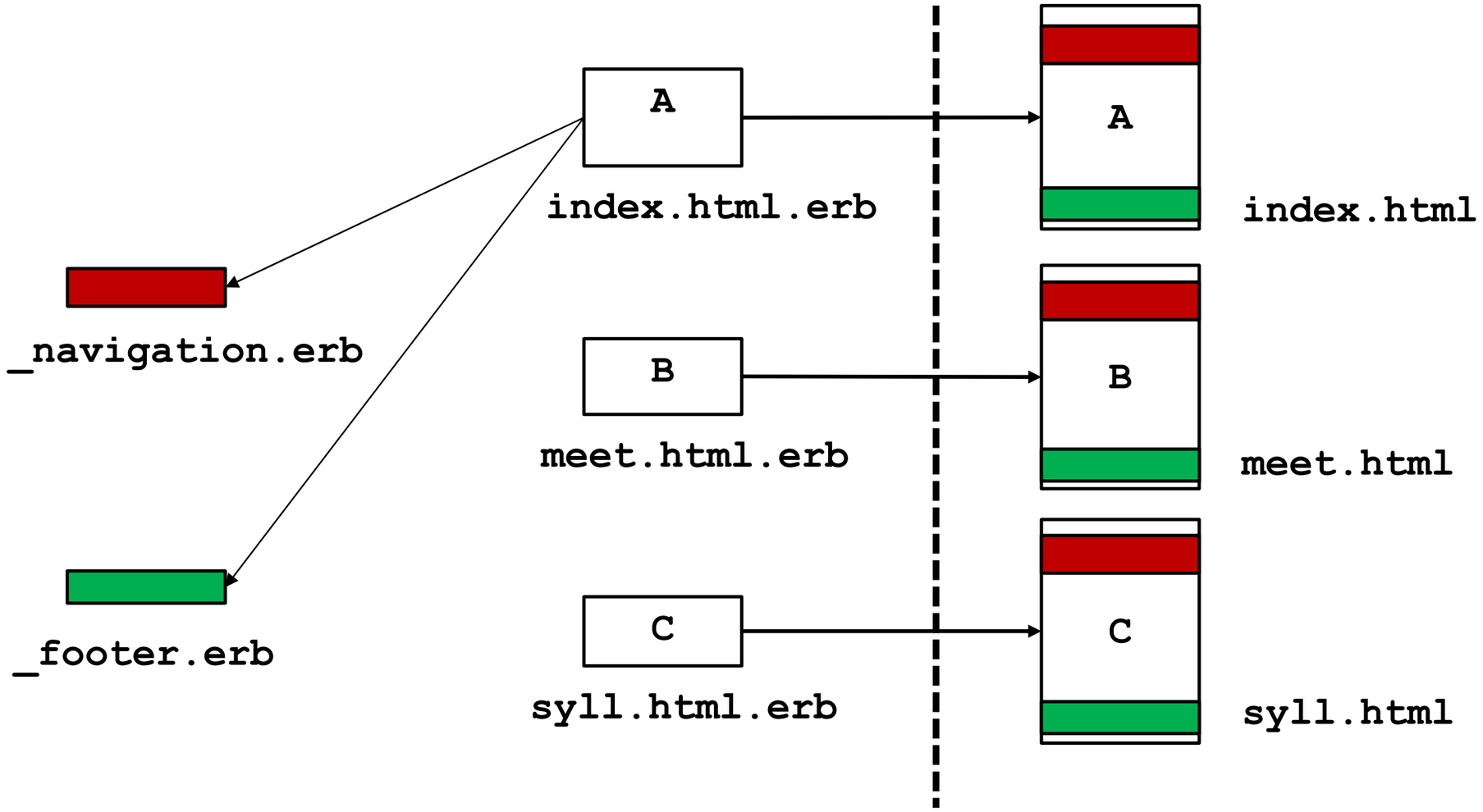
- Compile

```
$ bundle exec middleman build
```

- Result after building

```
$ ls build
index.html meet.html syll.html
```

Site Generation With Partials



Tricks with Partials

- Content of a partial can be customized by passing arguments to the call
- In call to partial, pass a hash called `:locals`

```
<%= partial "banner",
        locals: { name: "Syllabus",
                  amount: 34 } %>
```

- In the partial, access this hash using *variables*

```
<%# _banner.erb %>
<h3> <%= name %> </h3>
<p> Costs <%= "$#{amount}.00" %></p>
```

Problem

- How do we guarantee that every page includes partial(s)?
  - Partials don't ensure the same page *structure* across the site
- Every page should look like:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Class Meetings</title>
    <link rel="stylesheet" type="text/css" href="osu_style.css">
  </head>
  <body>
    <%= partial "navigation" %>
    … <!-- different for each page -->
    <%= partial "footer" %>
  </body>
</html>
```

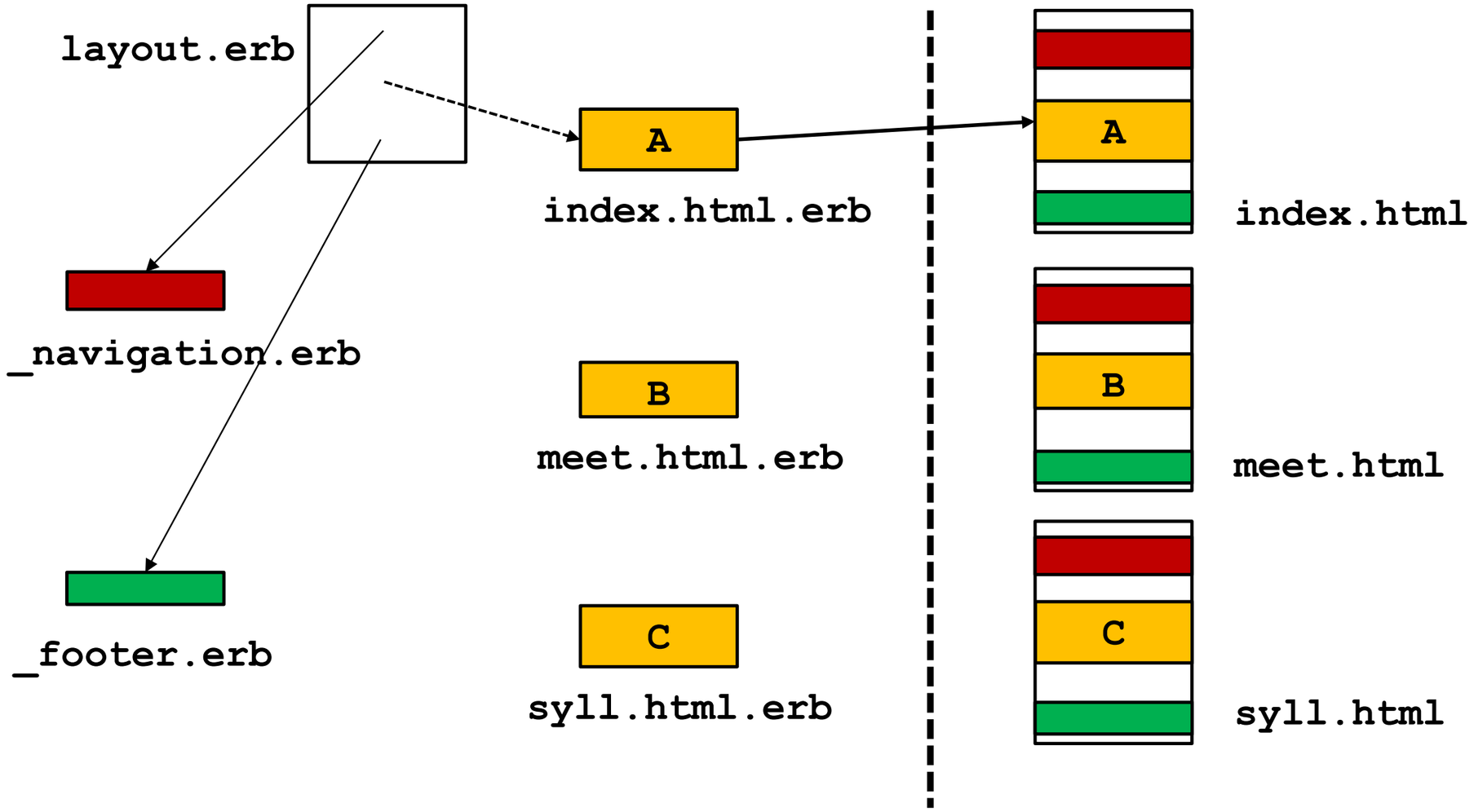
Solution: Layouts

- HTML formed from: Layout + Template
  - Layout is the common structure of HTML pages
  - Layout uses **`yield`** to include (page-specific) template
- File: layout.erb

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title> … etc
  </head>
  <body>
    <%= partial "navigation" %>
    <%= yield %>
    <%= partial "footer" %>
  </body>
<html>
```

- Layout is where you put site-wide styling
  - *e.g.*, navigation bar, div's with CSS classes, footers

Site Generation With Layouts



Generation of Site with Layouts

- Default layout in **source/layouts/layout.erb**

```
$ ls -F source
index.html.erb meet.html.erb layouts/ syll.html.erb
$ ls source/layouts
_footer.erb _navigation.erb layout.erb
```

- Result after building

```
$ ls build
index.html meet.html syll.html
```

Page-Specific Data in Layout

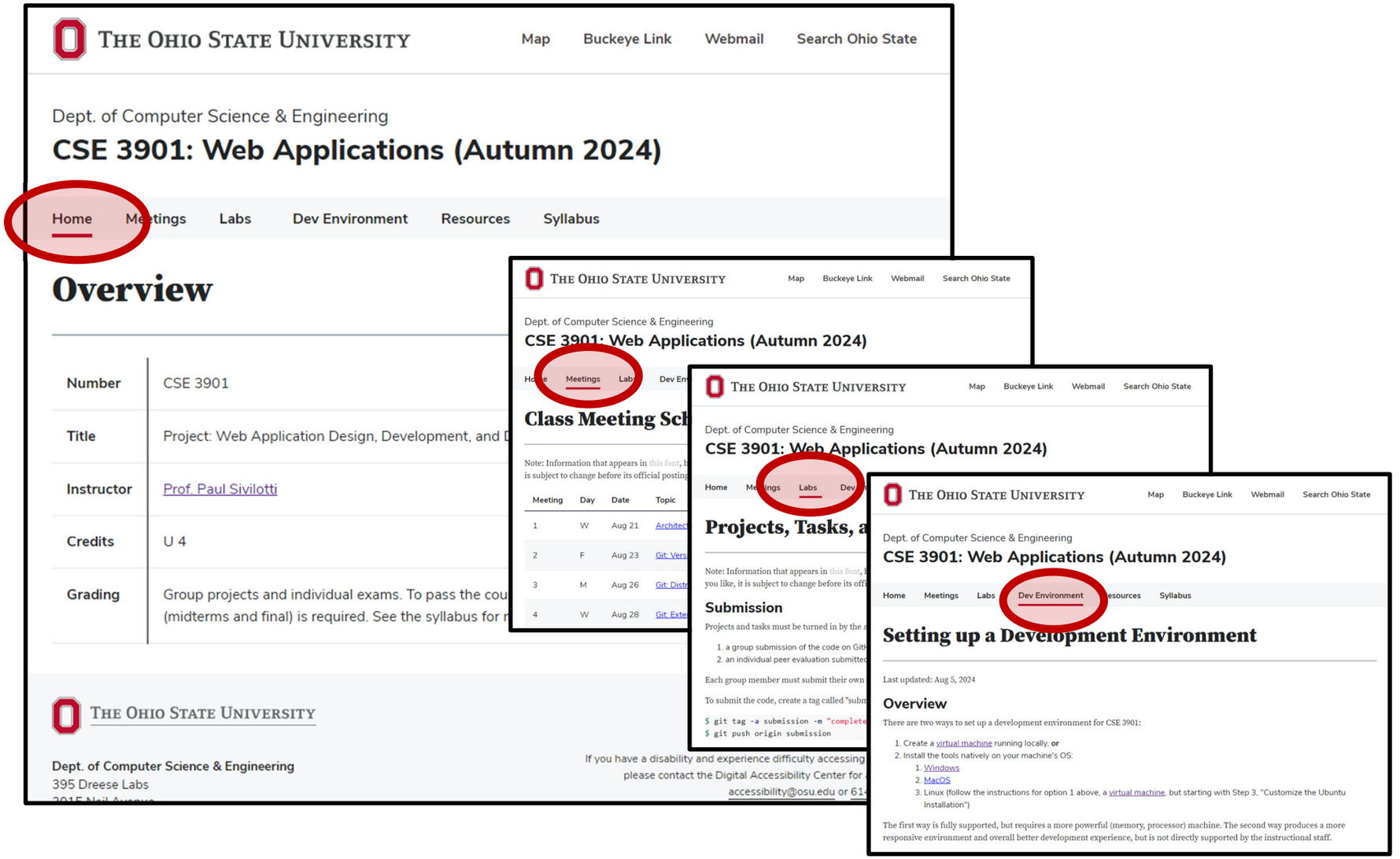
- Some layout content is page-specific
  - Example: `<title>` in document's head
- Solution: Ruby variable **`current_page`**
  - Example: **`current_page.path`**
- Template can use "frontmatter" to set the value of **`current_page.data`**
  - In template (**`meet.html.erb`**)

```
---
title: "Class Meetings"
---
```

  - In layout (**`layout.erb`**)

```
<title> <%= current_page.data.title %> </title>
```

Example: Navbar Highlights



Why Bother? (2)

1. Code reuse and single-point-of-control over change
2. Authoring of content in a language that is more human-friendly
3. Parameterized generation of markup and content

Let's look at each of these benefits in turn…

Motivation #2: Improved Syntax

- HTML tags make content hard to read
  - `<p>`, `<h2>`, `<em>`, `<a href="…">` etc
  - vs plain text, which is easier to read
- Common plain text conventions:
  - Blank lines between paragraphs
  - Underline titles with –'s or ='s
  - Emphasize *words*, _words_, **words**
  - Links as [text](url)
  - Unordered lists with bullets using * or -
  - Numbered lists with 1., 2., 3.

Why Middleman?

The last few years have seen an explosion in the amount and variety of tools developers can use to build web applications. Ruby on Rails selects a handful of these tools:

- Sass for DRY stylesheets
- CoffeeScript for safer and less verbose javascript
- Multiple asset management solutions, including Sprockets
- ERb & Haml for dynamic pages and simplified HTML syntax

Middleman gives the stand-alone developer access to all these tools and many, many more. Why would you use a

```
<h2>Why Middleman?</h2>

<p>The last few years have seen an explosion in the amount and
variety of tools developers can use to build web applications.
Ruby on Rails selects a handful of these tools:</p>

<ul>
<li><a href="http://sass-lang.com/">Sass</a> for DRY
stylesheets</li>
<li><a href="http://coffeescript.org/">CoffeeScript</a> for safer
and less verbose javascript</li>
<li>Multiple asset management solutions, including <a
href="https://github.com/sstephenson/sprockets">Sprockets</a></li
>
<li><a href="http://ruby-doc.org/stdlib-
2.0.0/libdoc/erb/rdoc/ERB.html">ERb</a> &amp;amp; <a
href="http://haml.info/">Haml</a> for dynamic pages and
simplified HTML syntax</li>
</ul>

<p><strong>Middleman</strong> gives the stand-alone developer…
```

```
## Why Middleman?

The last few years have seen an explosion in the amount and
variety of tools developers can use to build web applications.
Ruby on Rails selects a handful of these tools:

* [Sass](http://sass-lang.com/) for DRY stylesheets
* [CoffeeScript](http://coffeescript.org/) for safer and less
verbose javascript
* Multiple asset management solutions, including
[Sprockets](https://github.com/sstephenson/sprockets)
* [ERb](http://ruby-doc.org/stdlib-
2.0.0/libdoc/erb/rdoc/ERB.html) & [Haml](http://haml.info/) for
dynamic pages and simplified HTML syntax

**Middleman** gives the stand-alone developer…
```

Markdown

- Formalizes these ASCII conventions
  - Filename extension: `.md`
  - Adds some less familiar ones (eg `)
- Translator generates HTML from markdown
  - Examples: GitHub readme's, user-posted comments on web boards (StackOverflow)
  - Other target languages possible too
- See Middleman's README.md
  - Regular view
  - Raw view
- Warning: there are many Markdown dialects (and extensions)
  - Original: daringfireball.net (Gruber, 2004), stale
  - Now: CommonMark, GFM, MultiMarkdown, Pandoc, Markdown Extra
  - Choice of parser matters too! kramdown, rdiscount, redcarpet, …

CSS: Magic Numbers

- Literals are common in CSS

```
h1 { background-color: #ff14a6; }
h2 { color: #ff14a6; }
```

- Result: Lack of single-point-of-control
- Solution: SASS allows variables

```
$primary: #ff14a6;
h1 { background-color: $primary; }
h2 { color: $primary; }
```

- Translator generates CSS from SASS source
- Note: CSS custom properties can be used instead

CSS: Repeated Ancestry in Rules

- CSS requires separate selectors, even when paths overlap

```
.navbar ul { … }
.navbar ul li { … }
.navbar a { … }
```

- Problems:
  - Changing the classname requires changing all of these selectors
  - Related rules are not structurally connected (rule list is flat)
- Solution: SASS allows nested selectors

```
.navbar {
  ul { …
    li { … } }
  a { … }
}
```

Why Bother? (3)

1. Code reuse and single-point-of-control over change
2. Authoring of content in a language that is more human-friendly
3. Parameterized generation of markup and content

Let's look at each of these benefits in turn…

Motiv'n #3: Content Generation

- Problem: Repeated content
  - Example: Course offering term
  - Used in several templates, multiple places in a template
  - Lack of single point of control over change
- Solution: Define content in a separate data file
  - Put data file(s) in **`data/`** subdirectory
  - Define variables using YAML

```
# data/dates.yml
term: "Autumn 2025"
```

  - Variables are available for use in templates

```
# index.html.erb
Semester: <%= data.dates.term %>
```

Generating Repeated Structures

- Problem: Repeated structure
  - Example: Each row in table should look the same (content of each column, number of columns, …)
- Solution: Generate row structure with code
  - Define content in an iterable (eg an array)

```
# data/meetings.yml
- topic: "Architecture"
  type:  "lecture"
- topic: "Git"
  type:  "lab"
- topic: "Ruby: Basics"
  type:  "lecture"
```

Generating Repeated Structures (Cont'd)

- Solution: continued…
  - Iterate over array, creating a table row from each entry

```
<% data.meetings.each do |meet| %>
  <tr>
    <td> <%= meet.topic %> </td>
    <td> <%= meet.type %> </td>…
  </tr>
<% end %>
```

Generating Random Content

- Want placeholder content for a quick prototype
  - Example: Useful for making style/layout decisions
  - Don't care about the actual text, images, headers, etc
- Solution: use a *method* that returns an HTML string

```
<%= lorem.sentence %>
```

- Other lorem methods

```
lorem.paragraphs 2
lorem.date
lorem.last_name
lorem.image('300x400') #=> http://placehold.co/300x400
```

Helper Functions

- Used to generate common HTML snippets
- Example: hyperlinks

```
<a href="/about.html">About us</a>
```

- Generate HTML using `link_to` helper in template:

```
<%= link_to('About us', '/about.html') %>
  #=> <a href="/about.html">About us</a>
```

- Many optional arguments

```
<%= link_to('My Blog', '/blog.html',
            class: 'news') %>
  #=> <a href="/blog.html" class="news">My Blog</a>
```

(Many) More Helper Functions

- Format helpers

```
pluralize 2, 'person'                          #=> 2 people
```

- Tag helpers

```
tag :img, src: '/kittens/png'                  #=> <img src="…" />
content_tag :p, class: 'warn' do … end #=> <p class=…>…</p>
```

- Form helpers

```
form_tag '/login', method: 'post'    #=> <form action="…">…
button_tag 'cancel', class: 'clear' #=> <button name="…">…
```

- Asset helpers

```
stylesheet_link_tag 'all'          #=> <link rel="stylesheet"…
javascript_include_tag 'jquery' #=> <script src="jquery">…
favicon_tag 'images/favicon.png'#=> <link rel="icon"… />
image_tag 'padrino.png', width: '35', class: 'logo'
```

Summary

- ERb
  - Template for generating HTML
  - Scriplets and expressions
- Reuse of views with partials
  - Included with partial (eg <%= partial…)
  - Filename is prepended with underscore
  - Parameter passing from parent template
- Layouts and templates
- Markdown, SASS
- Content generation and helpers