WEEK 7: DESIGN PATTERNS FOR THE WEB

https://github.com/entr451-winter2026/web-apps-2

DESIGN PATTERNS

DESIGN PATTERNS

DESIGN PATTERNS

SOLVE A PROBLEM THAT HAS BEEN SOLVED BEFORE, IN EXACTLY THE SAME WAY

(aka, coding best-practices)

MVC
(MODEL-VIEW-CONTROLLER)

MVC
(MODEL-VIEW-CONTROLLER)

Solution to: how to build a database-backed web application

WEB APPS HAVE THREE LAYERS

- Model: talks to the DB
- View: stuff that the end-user sees
- Controller: code that connects models and views and controls traffic between the two

THE CONTROLLER

github.com/entr451-winter2026/web-apps-2

LABS

Refactor the logic in the view by moving it to the controller:

- dice
- tacos
- bitcoins

WHY MVC MAKES IT BETTER THAN BEFORE

- Our view is now clean
- Coupling our Ruby-based logic with the view just gets plain ugly
- Web applications may need to produce other types of views.
- We'll see more later...

BROWSER STATE

HTTP IS A STATELESS PROTOCOL

HTTP IS A STATELESS PROTOCOL

USER JOURNEY

1. The user visits */books* to browse a list of books for sale
2. The user clicks on a a particular title, which takes them to */books/123* where the details of the book are displayed
3. The user clicks on the "My Orders" tab, which directs them to */orders* where a list of their orders is shown
4. The user clicks on one of their open orders, which goes to */orders/42* where the details and status of order #42 can be viewed

USER JOURNEY

1. The user visits */books* to browse a list of books for sale
2. The user clicks on a a particular title, which takes them to */books/123* where the details of the book are displayed
3. The user clicks on the "My Orders" tab, which directs them to */orders* where a list of their orders is shown
4. The user clicks on one of their open orders, which goes to */orders/42* where the details and status of order #42 can be viewed

All separate HTTP request/responses!

USER JOURNEY

1. The user visits */books* to browse a list of books for sale
2. The user clicks on a a particular title, which takes them to */books/123* where the details of the book are displayed
3. The user clicks on the "My Orders" tab, which directs them to */orders* where a list of their orders is shown
4. The user clicks on one of their open orders, which goes to */orders/42* where the details and status of order #42 can be viewed

All separate HTTP request/responses!

Each don't know (or care about each other)!

USER JOURNEY

1. The user visits */books* to browse a list of books for sale
2. The user clicks on a a particular title, which takes them to */books/123* where the details of the book are displayed
3. The user clicks on the "My Orders" tab, which directs them to */orders* where a list of their orders is shown
4. The user clicks on one of their open orders, which goes to */orders/42* where the details and status of order #42 can be viewed

All separate HTTP request/responses!

Each don't know (or care about each other)!

Order doesn't matter!

The web browser doesn't maintain state between requests

The web browser doesn't maintain state between requests

HTTP IS STATELESS

THAT SUCKS.*

WE *NEED* TO MAINTAIN STATE

- A typical web app:
  - Knows who is using the application (i.e. logged-in)
  - Knows what the user did earlier (for example, added something to a shopping cart)
  - Is perfectly capable of displaying dynamic content based on the user's journey (for example, showing a "thank you" message if the user places an order)

THE TRUTH ABOUT WEB APPS!

- They are all a huge hack
- Stateless browsers tricked into thinking they have state
- A few techniques... we'll look at a couple today

THE QUERY STRING

visit amazon.com and search for "toaster ovens"

or....fake search using link:

https://www.amazon.com/s?k=toaster+ovens&crid=1MFO9ZLN6H8F2&sprefix=toaster+oven%2Caps%2C86&ref=nb_sb_noss_1

https://www.amazon.com/s?
k=toaster+ovens&crid=1MFO9ZLN6H8F2&sprefix=toas
ter+oven%2Caps%2C86&ref=nb_sb_noss_1

query string

GOOGLE MAPS EXAMPLE

https://google.com/maps

GOOGLE MAPS EXAMPLE

https://google.com/maps?q=paris

https://google.com/maps?q=las+vegas

https://google.com/maps?q=paris&hl=fr

OUR OWN APP EXAMPLE

/dice?username=Ben

/tacos?favorite=veggie

/bitcoins?amount=0.25

SERVER LOG!

KEY-VALUE PAIRS

HTTP

```
?username=Ben
```

Rails params hash

```
{ "username" => "Ben" }
```

EXAMPLE: USING THE PARAMS HASH

SYSTEMS DESIGN MOMENT

PROS AND CONS OF THE QUERY STRING

SYSTEMS DESIGN MOMENT

PROS AND CONS OF THE QUERY STRING

- Pros
  - Can be bookmarked
  - Can be more easily remembered
- Cons
  - Can be hacked

OTHER WAYS TO MAINTAIN STATE

- POST parameters (next)
- Cookies 🍪
- Other browser-based technologies (not yet commonly used)
    - Local storage
    - Session storage
    - Cache storage
    - Web SQL

REST

REPRESENTATIONAL STATE TRANSFER 🤢

- Another design pattern!
- A web app == things (resources) to CRUD
- A web app is made up of actions (we've only seen the "index" action)

A WEB APP == THINGS TO CRUD

- A CRM application is where we can CRUD accounts, contacts, sales activities, etc.
- A photo-sharing social media application, a la Tacostagram, is where we CRUD posts, likes, comments, etc.
- An e-commerce platform allows for the CRUD of products, reviews, orders, shipments, etc.
- YouTube allows users to CRUD videos, comments, likes, etc.

BUT...

- CRUD is about data (i.e. CRUD is SQL!)
- End-users don't write SQL in our app
- They use HTTP requests instead
- How are HTTP-based user actions mapped to database CRUD?
- REST = the web (HTTP) + databases (CRUD)

HTTP

HTTP

```
https://SomeWebSite.com/tacos
```

HTTP

```
https://SomeWebSite.com/tacos
```

GET request - exposed in the URL/address bar

HTTP

```
https://SomeWebSite.com/tacos
```

GET request - exposed in the URL/address bar

Other types of requests NOT exposed in the URL/address bar

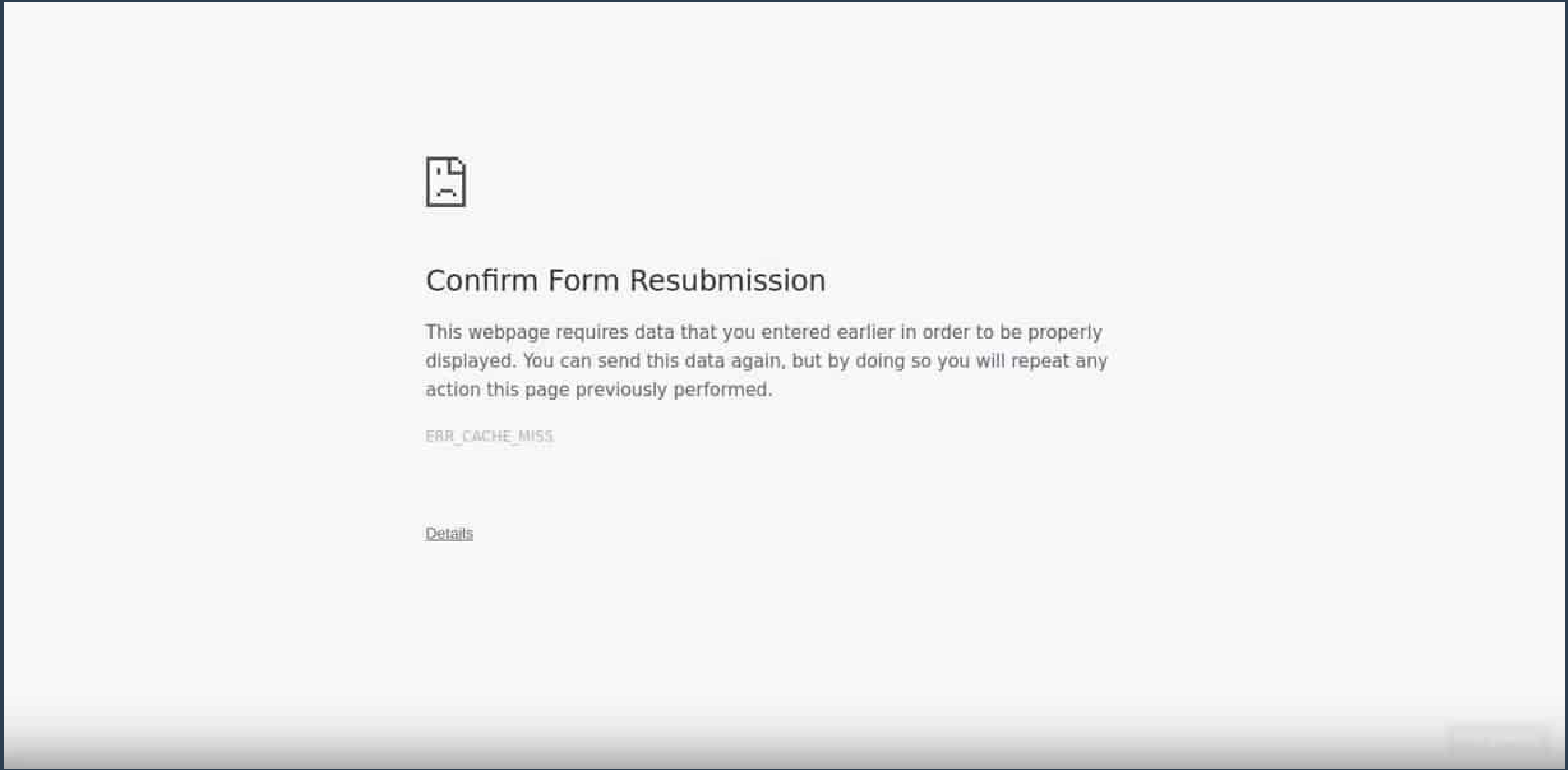
`POST` `PATCH` `DELETE`

HTTP METHODS

- Resources are the "nouns" of a web app
- HTTP request types (methods) are the "verbs"
- There are four:
  - GET
  - POST
  - PATCH *(also PUT)*
  - DELETE
- **End-users can only perform GET requests!**
  - *The other 4 are done only by developers, usually in response to form submissions.*

HTTP + CRUD = REST

HTTP Method	Use Case	CRUD
GET	Visiting a web page	Read
POST	Submitting a form that creates new data	Create
PATCH	Submitting a form that updates existing data	Update
DELETE	Submitting a form that deletes existing data	Delete



Confirm Form Resubmission

This webpage requires data that you entered earlier in order to be properly displayed. You can send this data again, but by doing so you will repeat any action this page previously performed.

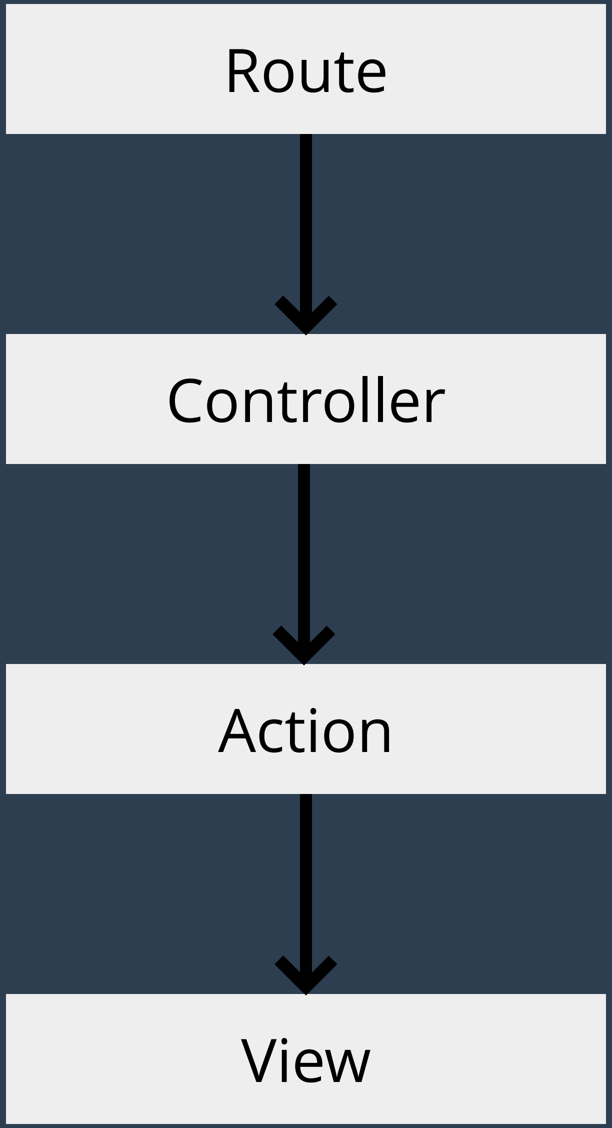
ERR_CACHE_MISS

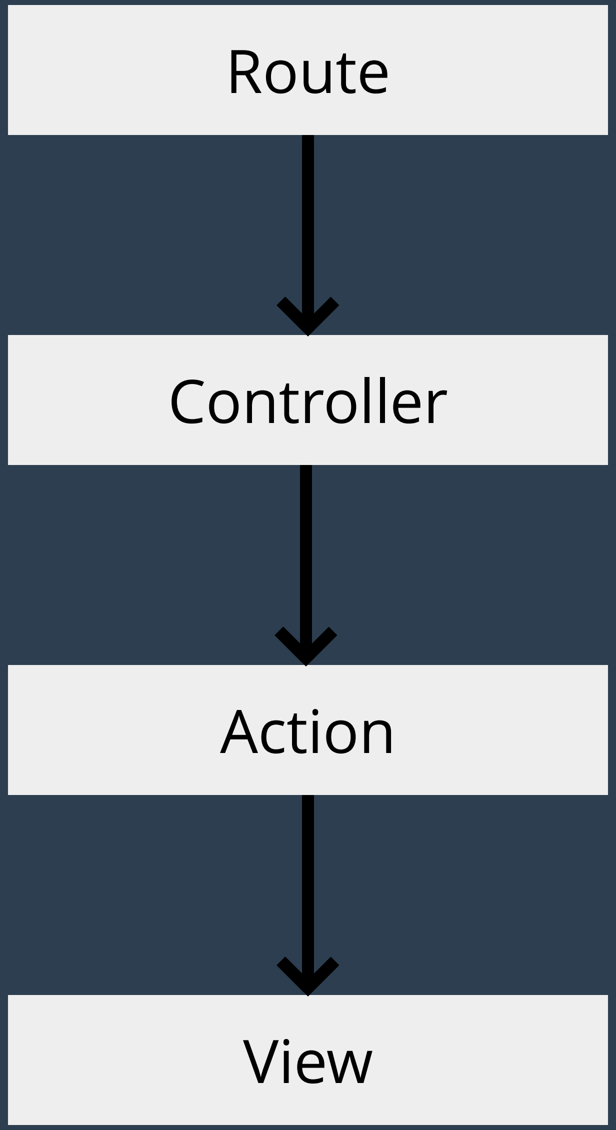
Details

THE REST DESIGN PATTERN

- There can be as many resources ("nouns") as you want
- There can only be four verbs – the 4 HTTP methods
- "REST API"

REST IN RAILS


Route
Controller
Action
View

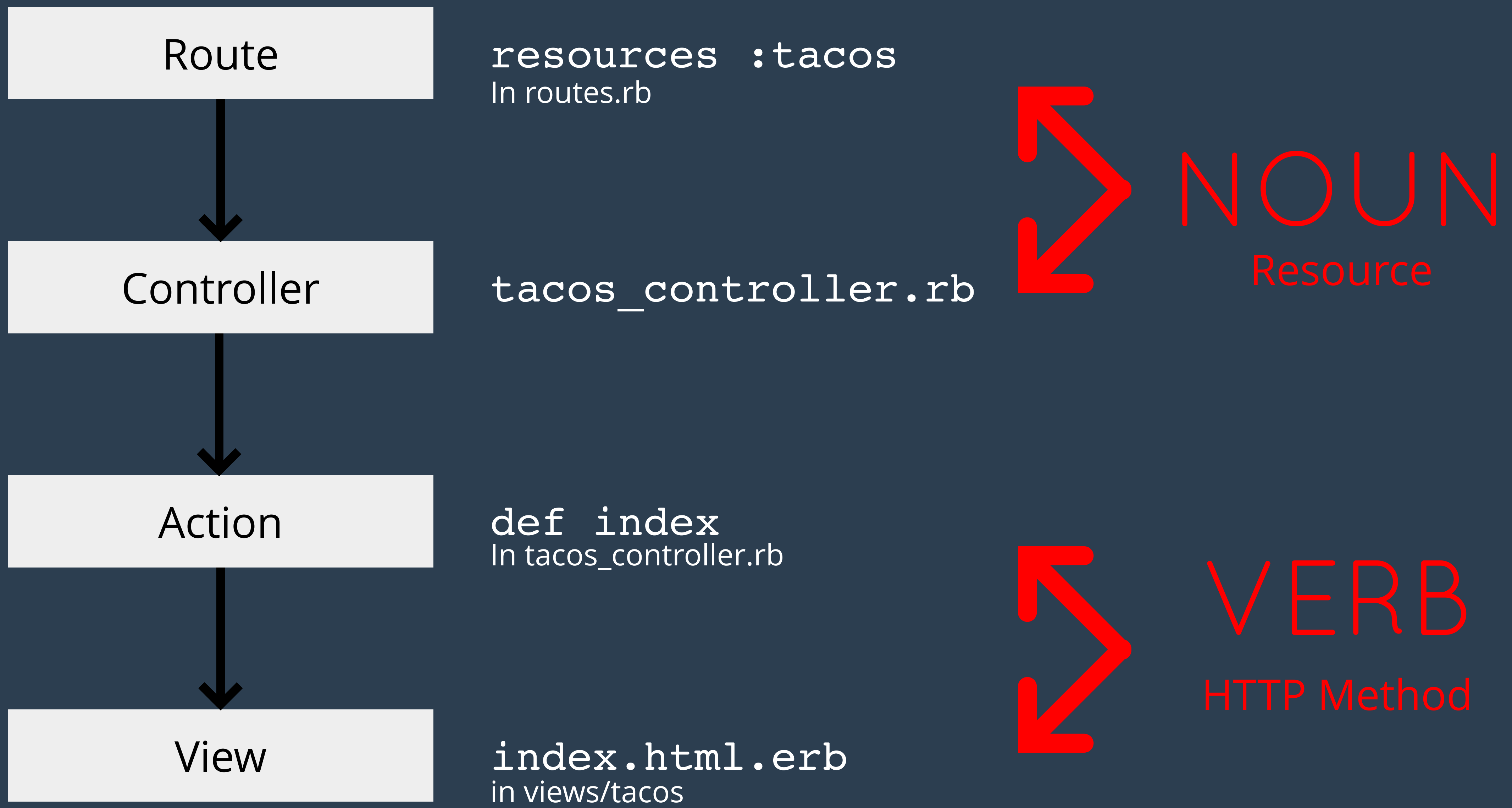


`resources "tacos"`
in routes.rb

`tacos_controller.rb`

`def index`
in tacos_controller.rb

`index.html.erb`
in views/tacos



```
resources :tacos
```
In routes.rb

```
tacos_controller.rb
```

```
def index
```
In tacos_controller.rb

```
index.html.erb
```
in views/tacos

Action

```
def index
```

In tacos_controller.rb

ACTIONS

THE ONE ACTION WE KNOW ABOUT SO FAR

Action	**CRUD Use Case**	**HTTP Method**	**Path**
index	Read (display) a list of all tacos	GET	/tacos

There are **4** letters in CRUD

There are **4** letters in CRUD

There are **4** HTTP methods

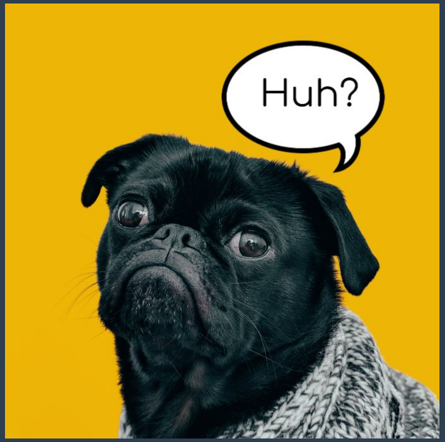
There are **4** letters in CRUD

There are **4** HTTP methods

How many actions are there?

That's right...

7



THE 7 ACTIONS

dynamic segment

Action	CRUD Use Case	HTTP Method	Path
index	Read (display) a list of all tacos	GET	/tacos
show	Read (display) information on a single taco	GET	/tacos/123 *(where 123 is the ID of the Taco)*

There are 2 GETs for the reading of data

THE 7 ACTIONS

Action	**CRUD Use Case**	**HTTP Method**	**Path**
new	A form to fill out about a new taco	GET	/tacos/new
create	Receives information from the form and creates the taco	POST	/tacos

There are 2 actions for the creation of data – one GET (the form) and one POST (the action that receives the data from the form)

Note: not all actions have views!

THE 7 ACTIONS

Action	**CRUD Use Case**	**HTTP Method**	**Path**
edit	A form about an existing taco	GET	/tacos/123/edit
update	Receives information from the form and updates the taco	PATCH	/tacos/123

There are 2 actions for the update of data – one GET (the form) and one PATCH (the action that receives the data from the form)

THE 7 ACTIONS

Action	**CRUD Use Case**	**HTTP Method**	**Path**
destroy	Destroys (deletes) a taco	DELETE	/tacos/123

And one more to receive and process a request to delete a taco

THE 7 ACTIONS

Action	Usually has view?	HTTP Method	Path
index	✅ Yes	GET	/tacos
show	✅ Yes	GET	/tacos/123
new	✅ Yes	GET	/tacos/new
create	No	POST	/tacos
edit	✅ Yes	GET	/tacos/123/edit
update	No	PATCH	/tacos/123
destroy	No	DELETE	/tacos/123

THE 7 ACTIONS

- A convention that takes a while to master
- Try to understand them – it's worth it
- Implement them for every resource = you have a complete web app
- Forget them? From the terminal: *rails routes*
    - or if your server is running, visit */rails/info/routes*

CODE-ALONG

THE 2 READS (INDEX AND SHOW)

CODE-ALONG

CREATE THINGS (NEW AND CREATE)

LAB: TACOSTAGRAM!

- You'll find that there's a *PostsController* and an *index* action/view set up already in the project
- Add a new action in the controller and corresponding *new.html.erb* view
- Create a form in *new.html.erb* that will allow the end-user to input a post's *author*, *body*, and *image* (see *db/schema.rb*)
    - *author* is the author's name (e.g. Brian)
    - *body* is the text body of the post (e.g. these tacos are delish!)
    - *image* is simply the URL address of an image on the internet
        - For example, visit https://unsplash.com/ and search for tacos – right-click and copy image address
- Add a create action to accept values entered into the form and redirect back to the page with all posts

CODE-ALONG

CREATING RELATIONSHIPS

COMPLETE CODE

github.com/entr451-winter2026/web-apps-2-complete

👆 includes *edit*, *update*, & *destroy*

HUH?

- How data gets into the database
  - i.e. connecting frontend to backend to database
- MVC
  - Separation of logic
- REST
- Began thinking about security
- Design Patterns!
  - established application and system solutions to common problems

ASSIGNMENT

- Posted in Canvas

NEXT WEEK

- Users & Security