Web development in a nutshell workshop

Hector Correa
hector@hectorcorrea.com
http://hectorcorrea.com/webdev-nutshell

January/2024

Table of Contents

What is a web application?

Broadly speaking a “web application is software that runs in your web browser”.

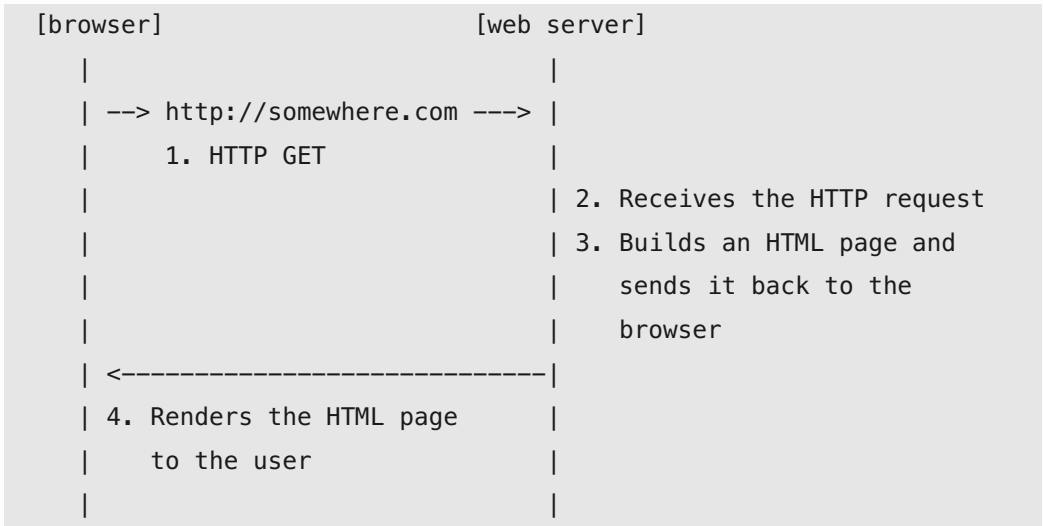
Browsers *run* these applications by making requests to a server, processing the responses from the server, and rendering them on your machine.

There are many technologies involved in a web application (e.g. HTML, CSS, JavaScript, HTTP, web servers, backend code) and each of these technologies can be a workshop of their own. The idea of this workshop is to give you a *high level* overview of how these technologies fit on the overall picture and how they interact with each other.

For the most part a web application is divided in two parts: client-side and server-side:

- The client-side (also known as front-end) is what the browser renders and it includes the HTML content of the page as well as the CSS to style it.
- The server-side (also known as back-end) is the code that produces the HTML that the browser renders. The code on the server-side can be as simple as a text file with the HTML or a complex piece of code that fetches information from a database and builds the HTML with the results.

The browser communicates with the web server via a protocol known as HTTP. When you make a request to visit a website the communication loop between your browser and the web server looks more or less as the one shown in the diagram below:



If this is your first encounter with web development the following are good pieces to learn more about the history of the internet and the world wide web:

- History of the web

- How the Internet works,
- Getting started with the web
- The book Broad Band - The Untold Story of the Women Who Made the Internet by Claire L. Evans is also a fascinating story of the people and communities that made the Internet what it is today.
- The birth of the Web at CERN where you can view a copy of the very first web site.

Client-side

We'll start this workshop by looking at the client-side, and in particular we'll focus on HTML and CSS.

HTML

HTML stands for Hypertext Markup Language and it's the language that we use to build web pages that the browser renders.

Below is an example of a very basic web page with HTML:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is my first web page</p>
  </body>
</html>
```

Notice how it has some sections (e.g. `html`, `head` and `body`) and some other elements (e.g. `<h1>` and `<p>`) surrounded by angle backets.

With your *editor* open the file `hello-world.html` that comes with the examples that you downloaded, it should look like the HTML above. Now, launch your *browser* and open the file `hello-world.html`.

The `<head>` section is where we define the metadata of the page. In the previous example we set the title which is how your browser will identity the page within your browser tabs. Go ahead and change the title of your page from `Hello` to `Hola` in your `hello-world.html` file, save it, and refresh your browser. You should see the new title.

The `<body>` section is the actual content that will be diplayed to the user. Inside the `<body>` section we can add many different kind of HTML elements: paragraphs, divs, tables, images, forms, input boxes, links, buttons, and many more.

HTML elements

The elements inside the `<body>` section of an HTML page can be repeated, for example you can have many paragraphs (`<p>`), many headers (`<h2>`), or many divs (`<div>`).

Some of these elements can also be nested. For example you can have a `<div>` with many paragraphs `<p>`. A paragraph can contain many spans (`<span>`), bold tags (`<b>`), or italic tags (`<i>`).

Most of the HTML elements on a page have an open and a close tag, for example `<p>` to indicate the beginning of a paragraph and `</p>` to end it.

The anchor tag (`<a>`) is a special element because this element allows us to create links to other web pages.

Feel free to make changes to `hello-world.html` and try different things. Remember to reload the page on your browser to see your changes.

With your editor open the file `hello-world-fancy.html`, the content will look like the HTML below:

```
<html>
  <head>
    <title>Hello (fancy)</title>
  </head>
  <body>
    <h1>Hello World (fancy edition)</h1>
    <div>
      <p>This is my first web page</p>
      <p>And it has many paragraphs</p>
      <p>
        This paragraph is so fancy it even has text
        in <b>bold</b> and <i>italics</i>.
      </p>
      <p class="important-paragraph">
        This paragraph is very important.
      </p>
      <p>
        We even have a <a href="hello-world.html">link to our original
hello-world page</a>.
```

```
      </p>
      <p>
        This paragraph even has a <button id="do-nothing-
button">button</button> that does nothing</p>
      </p>
      <p class="important-paragraph">
        This paragraph is also very important.
      </p>
      <!-- this is a comment,
        the browser ignores it -->
      <p>
        Links can also point to other websites too, for example to
        <a href="https://en.wikipedia.org/wiki/HTML"
target="_blank">Wikipedia</a>
      </p>
    </div>
    <div>
        <p>And here is more content and a photo<p>
        <img src="./public/chicago-street-art.jpg" alt="Chicago street
art"/>
    </div>
    <footer>
      This is the footer of the page.
    </footer>
  </body>
</html>
```

Now load this file in your browser, notice how there is text in **bold**, *italic*, a link to another page, and an even an image. Can you find in the HTML above where these elements are defined?

The Mozilla web site has a wealth of information to understand HTML and how to structure a web page.

HTML is what defines the *content* and the *structure* of a web page, but to style a page we use an additional language: CSS.

CSS

CSS stands for Cascading Style Sheets and it's an additional language that we use inside an HTML page. As the name implies, we use CSS to *style* a web page, that is to change the color of the elements, their margins, how they are aligned on the page (e.g. to the left, to the right), the fonts to use, and many more properties.

To define a CSS inside an HTML page we use the `<style>` tag inside the `<head>` section of the page.

Let's change the style of some of the elements of our previous page. Replace the `<head>...</head>` section of the `hello-world-fancy.html` file that you created before with the following text:

```
<head>
  <title>Hello (fancy with CSS)</title>
  <style>
    h1 {
      background-color: orange;
    }
    div {
      margin-left: 100px;
    }
    p {
      font-family: 'Courier New', Courier, monospace;
    }
    img {
      border: 2px solid black;
      box-shadow: 10px 10px orange;
    }
    footer {
      color: gray;
    }
  </style>
</head>
```

Refresh the `hello-world-fancy.html` file in your browser. Notice how the header now has an orange background, the text is using a different font, the image has a drop shadow, and the footer is grayed out.

All of these changes happened by just changing adding the CSS definitions under the `<style>` section of the page. Notice that we did not touch the HTML content inside the `<body>` at all.

CSS - selectors

In our previous example we defined a CSS style that applied to *all* paragraphs:

```
p {
  font-family: 'Courier New', Courier, monospace;
}
```

But we can be more specific and define an style that applies only to those elements that have the *class attribute* set to `important-paragraph`. For example, let's add the following style anywhere inside the `<style>...</style>` section of a page:

```
.important-paragraph {
  color: red;
  font-style: italic;
}
```

Now, when we refresh the page we will see some paragraphs in red but not others. If you look closely at the HTML of our `hello-world-fancy.html` page you'll notice that those paragraphs have the class attribute set.

We could also add an even more specific style that will target *only* the one HTML element that has the *id attribute* set to `do-nothing-button`. Let's add the following style to our page (again, put this style anywhere inside the `<style>...</style>` section of a page):

```
#do-nothing-button {
  padding: .375rem .75rem;
  font-size: 1rem;
  line-height: 1.5;
  border-radius: .25rem;
  color: #fff;
  background-color: #007bff;
  border-color: #007bff;
}
```

Refresh the page and notice how our button looks different.

This ability to target different elements on our styles is known as CSS Selectors and it's very powerful. The Mozilla page on CSS Selectors has information of many kind of selectors that you can use in your CSS.

CSS - layouts

In addition to using CSS to style individual HTML elements on a page, CSS can be used to layout how the elements are positioned in relationship to one another and how they are rearranged on the page when the user resizes their browser window.

CSS provides many ways to layout pages: normal flow, flexbox, grids, and floats to name a few. The Mozilla guide on CSS Layout gives a good overview on these different styles.

File `css-column-layout.html` shows an example of how to use the Multiple-column layout. File `css-grid-layout.html` shows an example of to use realize a common website layout using the CSS Grid Layout

The talk Everything You Know About Web Design Just Changed by Jen Simmons is another resource to see how these layouts are used and have evolved over the years.

CSS - using an external file

In our previous examples we inserted the CSS for our pages right on the `<head>...</head>` section of the HTML page.

Another common way to insert CSS inside a page is to *reference an external file* that has the CSS that we would normally put in the `<style>...</style>` section. For example, let's replace the `<head>...</head>` section of our `hello-world-fancy.html` page to look like this:

```
  <head>
    <title>Hello (external CSS)</title>
    <link rel="stylesheet"
href="https://fonts.xz.style/serve/inter.css">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/@exampledev/new.css@1.1.2/new.css">
  </head>
```

The `<link>` element tells the browser to load the CSS styles from the URL indicated. This particular example is using the CSS defined by new.css which is a very small framework for frontend development that uses "some sensible defaults and styles your HTML to look reasonable".

Refresh your browser to view the `hello-world-fancy.html` page with the new defaults.

Leaving file:// behind

If you look at the address bar in your browser while you have the `hello-world-fancy.html` loaded you'll notice that it says something along the lines of `file:///something/.../hello-world-fancy.html`.

The `file://` at the beginning of the URL indicates that the browser is accessing this file via the "file protocol". However, when you visit a website, say https://wikipedia.org, you would notice that the URL starts with `http://` (or `https://`) rather than `file://`.

The file protocol is fine for loading simple HTML pages from our local disk, but for web development we use the HTTP protocol which allows us to load HTML pages from remote servers. In the example above the URL `https://wikipedia.org` tells the browser to fetch the Wikipedia home page via the network (rather than fetching it from a file on our local disk.)

HTTP

HTTP stands for Hypertext Transfer Protocol and it's a network protocol that the browser uses to communicate with other servers.

When we visit a site, say Wikipedia, our browser issues an HTTP GET request to fetch a page from `wikipedia.org`. When the `wikipedia.org` server receives our request it builds an HTML page and sends it back to our browser (as an HTTP response). Our browser then parses the HTML and renders it for us to view. The following diagram shows this sequence:

```
[our browser]              [wikipedia server]
   |                              |
   | --> http://wikipedia.org --> |
   |     1. HTTP GET              |
   |                              | 2. Receives the HTTP request
   |                              | 3. Builds an HTML page and
   |                              |    sends it back to the
   |                              |    browser
   | <--------------------------- |
   | 4. Renders the HTML page     |
   |    to the user               |
   |                              |
```

A network protocol is a set of rules that define how computers in a network can communicate with each other. In the example above the browser could be running on a Windows laptop or an Apple iPhone, and the Wikipedia server could be a Linux computer. A network protocol, like HTTP, is what allows these computers to communicate with each other, despite the fact that they are using entirely different hardware, running different operating systems, and located in locations around the world.

To learn more about HTTP check out the tutorials on the Mozilla website and the post What is a protocol? at Cloudflare.

> Note: In most web applications these days you'll see HTTPS instead of HTTP. HTTPS is the encrypted version of HTTP, the "S" stands for Secure. For the purposes of this workshop they are interchangeable.

Server-side

So far all our examples have shown the client-side of a web application. In the next section we are going to dive into the server-side of the application.

The server-side of a web application receives the requests from the client (e.g. somebody wants to view the content of this page), figures out what is the content appropriated for that request, builds an HTML page with that content, and sends it back to the browser. On the client-side the browser (as we have seen) knows how to render an HTML page.

The code for the server-side can be written any programming language: Ruby, Python, PHP, Node.js, Java, C#, Go, and others. For this workshop we are going to use Ruby.

Keep in mind that, regardless of the programming language that you use on the server-side, the goal is to produce a string with HTML that will be send back to the browser.

Installing Ruby

Since we are going to use Ruby for our server-side code let's start by installing Ruby. We are going to use a Docker to download a container with Ruby already preinstalled.

> Note: This section of the workshop is *by far* the more complicated part because it requires that you to have Docker running, click a bunch of options within Visual Studio Code, and hope everything installs correctly. But once we get this going it will be fun again.

From within Visual Studio Code (VS Code) press `Option-Command-O` (Mac) or `Alt-Command-O` (Windows) to open the Remote Window menu. From this menu pick the option "Reopen in Container". If the option to "Reopen in Container" is not shown you will need to install the "Dev Containers" extension first by going to the "View" menu and selecting "Extensions".

Once you select "Reopen in Container" VS Code will re-launch and it will create the Docker container with Ruby. This will take a minute or two the first time you do it, it'll be a bit faster the next time you select this option. You can click on the "Starting Dev Container (show log)" link to view the progress.

> If you get an error that says "Make sure the Docker daemon is running" it means the Docker Desktop application is not running. Go ahead and launch it and select "Reopen in Container" again.

When VS Code has completed loading the container the log window will display an inconspicuous message that says “Launched Extension Host Process” and the bottom left corner of VS Code (the blue/purple label) will say “Dev Container: Existing Dockerfile @ desktop-linux”.

Once the container has finished loading, click on the “Terminal” menu and select “New Terminal”. This will open a Terminal window at the bottom of the screen and it will show a prompt that looks like this:

```
root@1a2b3c4d:/workspaces/webdev-nutshell#
```

The number `1a2b3c4d` will be different on your installation but that’s OK.

Once you are on this prompt type `ruby --version` and you should see that Ruby 3.0 is installed on this container:

```
root@1a2b3c4d:/workspaces/webdev-nutshell# ruby --version
ruby 3.0.6p216 (2023-03-30 revision 23a532679b) [x86_64-linux]
```

On this same prompt run the command `ruby hello.rb` and look at the output. You can see the code for `hello.rb` file from VS Code, it will be a single line of code:

```
puts "Hello world from a Ruby program"
```

Then look at the code of `hello_again.rb`, what do you think this one does? Run it via `ruby hello_again.rb` to find out.

If you are new to Ruby the Ruby user’s guide might be a good place for you to start and learn a little it about the language.

Server-side code with Ruby

The goal of this workshop is to show you how to build a web application using Ruby. To do this we are going to install a few additional Ruby libraries (or `gems` as they are known in the Ruby parlance) that provide the plumbing needed for making a Ruby application available via the web:

- Sinatra - library for creating web applications in Ruby
- Rackup - library that provides the web server interface
- Webrick - library that provides the actual web server implementation
- Byebug - a debugger for Ruby

To these gems run the following command from your VS Code Terminal window:

```
gem install sinatra rackup webrick
```

You'll see the following output:

```
  > Fetching ...
  > Fetching rack-3.1.8.gem
  > Fetching sinatra-4.0.0.gem
  > Successfully installed rack-3.1.8
  > Successfully installed sinatra-4.0.0
  > Fetching rackup-2.2.0.gem
  > Successfully installed rackup-2.2.0
  > Fetching webrick-1.9.0.gem
  > Successfully installed webrick-1.9.0
  > 9 gems installed
```

To install **Byebug** run the following command on your Terminal window:

```
gem install byebug
```

You'll see the following output

```
  > Fetching byebug-11.1.3.gem
  > Building native extensions. This could take a while...
  > Successfully installed byebug-11.1.3
  > 1 gem installed
```

Now that we have these `gems` installed we can run our first web application in Ruby:

```
ruby webdemo1.rb
```

You'll see the following output

```
  > [2023-12-15 01:30:52] INFO  WEBrick 1.8.1
  > [2023-12-15 01:30:52] INFO  ruby 3.0.6 (2023-03-30) [x86_64-linux]
  > == Sinatra (v3.1.0) has taken the stage on 3000 for development
with backup from WEBrick
  > [2023-12-15 01:30:52] INFO  WEBrick::HTTPServer#start: pid=7584
port=3000
```

while the application is running we can access it with our browser by going to http://localhost:3000/ - notice that this time we are using an `http://` URL instead of a `file://` URL.

This application is rather underwhelming but it is showing a Ruby program is accepting HTTP requests from a browser and returning valid HTML that the browser can render. You can view the code for this tiny application in `webdemo1.rb`:

```
require "sinatra"

set :port, 3000
set :bind, '0.0.0.0','localhost'
disable :strict_paths

get("/") do
  # Render a hard-coded HTML string
  html = "<h1>Welcome to your first web app</h1>"
  html += "<p>You are on your way to beat Google :)</p>"
  return html
end
```

The important thing about this application is that we could put it on a different server (rather than on our own laptop) and make it available for the world to access, say via a URL like `http://my-first-ruby-application.org`. We are not going to this now, but the fact that this application uses HTTP as its communication protocol and produces HTML means that we could.

> The 3000 in the URL http://localhost:3000/ designates the port the browser will use to talk to our `localhost` server. If we don't indicate a port in the URL (e.g. http://wikipedia.org) the browser by default will use port 80 and will issue http://wikipedia.org:80 behind the scenes.
>
> Our little Ruby program is hard-coded to listen on port `3000`, see line `set :port, 3000` in `webdemo1.rb`.
>
> Port numbers below 1024 are reserved in most operating systems and require administrative privileges to be accessed. To work around that security restriction developers typically use port numbers like 3000 or 8080 during development.

For now, type `CTRL-C` from your terminal to shut down this application. You'll see the following output in the terminal:

```
^C== Sinatra has ended his set (crowd applauds)
[2023-12-15 02:01:57] INFO  going to shutdown ...
[2023-12-15 02:01:57] INFO  WEBrick::HTTPServer#start done.
```

if you refresh your browser pointing to http://localhost:3000/ the browser will report that it cannot connect to it anymore, which makes sense, we just shut it down.

Sinatra views

Our previous example has a hard-code string with HTML right in our Ruby code. This is OK for a simple demo but not very convenient in the long run. Sinatra allows us to move our HTML outside the Ruby code into an ERB file, which is similar to an HTML file but with some extra features.

If we look at the code in `webdemo2.rb` you'll notice that in addition to having the HTML hard-coded like the previous demo, there is also *a second block of code* that loads an ERB view called `fancy.erb`:

```
get("/fancy") do
  # Render the HTML on ./views/fancy.erb
  erb(:fancy)
end
```

By default the ERB file is expected to be found in the `./views` folder. In fact you can open this file in VS Code and you'll notice that is an almost identical copy to the `hello-world-fancy.html` that we used at the beginning of the workshop.

We can run this new demo by running `ruby webdemo2.rb` from the Terminal window and pointing our browser again to http://localhost:3000/fancy

Notice how it looks just like our previous `hello-world-fancy.html` page, but again, this time the page is served via HTTP (rather than via the file system).

Another thing that is not obvious in the code but that is important to notice is the page rendered at http://localhost:3000/fancy is using some CSS styling, but there is no `<style>...</style>` section defined in our `fancy.erb` file. Where is this styling coming from?

Sinatra uses the concept of a *layout* page that is used as the base for any other ERB file rendered. You can view the code for it under `./views/layout.erb`. Notice that this page is an skeleton of a page with everything but a `<body>`, in fact the body looks like this:

```
<body>
  <%= yield %>
</body>
```

Sinatra will render everything in the `layout.erb` file and when it sees the `<%= yield %>` directive it will embed the content of our `fancy.erb` as part of output. For more information about layouts in Sinatra checkout this blog post.

The `<%= ... %>` syntax in the layout file is something unique to ERB files (i.e. it's not HTML even though it looks like it). When Sinatra sees this syntax is expects the code inside of it to be Ruby code.

For example, let's add the following line to the `./views/fancy.erb` file. You can

put it anywhere you want, but a good place is immediately after the `<h1>...</h1>` line:

```
<p>Today is: <%= DateTime.now.to_date %></p>
```

This line will render an HTML paragraph `<p>...</p>` and inside of it will insert *the result of the Ruby expression* `DateTime.now.to_date` that is inside the `<%= ... %>`. The actual value that will be rendered will look like this:

```
<p>Today is: 2023-12-30</p>
```

Try it, refresh your browser and the http://localhost:3000/fancy page should show you today's date.

Sinatra routes (get)

An important part of a web application is the code that figures out how to handle each different URL that is requested. In `webdemo2.rb` we saw two blocks of code to handle two different URLs. There was one block of code to handle requests to http://localhost:3000/ and a different block of code to handle requests for http://localhost:3000/fancy

Notice the `get("/")` and `get("/fancy")` in the code below:

```
get("/") do
  # Render a hard-coded HTML string
  html = "<h1>Welcome to your first web app</h1>"
  html += "<p>You are on your way to beat Google :)</p>"
  html += "<p>and it even has a link to our <a href=/fancy>fancy</a>
page"
  return html
end

get("/fancy") do
  # Render the HTML found on ./views/fancy.erb
  erb(:fancy)
end
```

These calls are known as routes and there are many kind of routes that we can use in a Sinatra application.

Let's go ahead and look at `webdemo3_books.rb`. Notice how this example has several additional routes, for example there is one route that looks like this:

```
get("/books/:id") do
  book_id = params["id"]
  @book = BookDatabase.find(book_id)
  erb(:book_show)
end
```

The `:id` on the previous route indicates that there is a "named value" that will vary. For example in the URL `/books/1` the `1` will be considered the id while on the URL `/books/759` the `759` will be considered the id. We can access the `:id` of the URL via `params["id"]` as you can see in the code above.

Another important thing to notice in this route is that it declares an instance variable: `@book`. The value of this variable will be visible when Sinatra executes `erb(:book_show)` which means that we can reference `@book` in `book_show.erb`. If we look inside `book_show.erb` there is an HTML fragment that looks like this:

```
  <p>
    <b>Title:</b>
    <%= @book.title %>
  </p>
```

Notice the line `<%= @book.title %>`. Remember that anything between the `<%= ... %>` is Ruby code that will be executed. In this case `@book.title` is the title of the book in the `@book` variable that we declared in our router.

So far we have only seen `get()` routes in Sinatra. In the next section we'll talk about `post()` routes.

HTTP POST and HTML FORMs

When a browser wants to push information to a web server, for example when we hit "Save" after entering the information of a book, it issues an HTTP POST request to pass the information.

In order for us to tell the browser what information must be passed in the HTTP POST request we use HTML FORM and HTML INPUT elements.

The way this works in our `webdemo3_books.rb` example is that when the user clicks the "New Book" button the browser issues an HTTP GET to render an HTML FORM that allows the user to enter the values for a new book, the code for this route is below:

```
get("/new-book") do
  @book = BookDatabase.create_new
  erb(:book_new)
end
```

There is nothing extraordinary on this route. In fact it looks very similar to the other routes that we have reviewed. What is new in this code is the HTML that it renders inside the `book_new.erb` view. Below is a section of that HTML:

```
<body>
  <form action="/new-book-save" method="post" >
    <p>
      <b>Title</b>
      <input type="text" id="title" name="title" />
    </p>
    ...
    <p>
      <input type="submit" id="saveButton" name="saveButton"
value="Save" />
    </p>
  </FORM>
</body>
```

This view introduces two new HTML elements: `<form>` and `<input>`.

HTML FORMs are a way to group the values that we want to pass to the server. These values are captured via `<input>` elements inside the `<form>`. The HTML FORM element itself has two *attributes* that are important:

- The `action` tells the browser the URL on the server *where* the information will be send to when the user clicks the "submit" button on their browser. In our example we are telling the browser to use the `/new-book-save` URL.
- The `method` tells the browser *how* to submit the information. In our case we are using the POST method which means that the browser will issue an `HTTP POST /new-book-save` when the user clicks the "submit" button.

The values that the browser will push to the server are captured via the `<input>` elements inside our HTML FORM. There are many kind of `<input>` elements but in this workshop we will only use two of them: text boxes and submit buttons.

Text boxes are used to allow the user to enter information in a form and they are defined as `<input type="text" ... />`, notice the type is "text".

Submit buttons are used to give the user a button to indicate they are ready to

send the information to the server and they are defined as follow: `<input type="submit" ... />`, notice that the type is "submit".

The `id` and `name` attributes in `<input>` elements are important since this is the way the *server* will recognize each of the data elements that it receives.

All of the above means that when the users clicks the "submit button" in our form the browser will issue an `HTTP POST /new-book-save` request and it will pass to the server the values in the `<input>` elements, in our case *title*, *author*, and *year*.

Our `webdemo3_books.rb` has a route to handle this particular HTTP POST request. The code is below:

```
post("/new-book-save") do
  # Get the values submitted on the HTML FORM...
  title = params["title"]
  author = params["author"]
  year = params["year"].to_i

  # ...add the new the book to our database
  new_book_id = BookDatabase.add(title, author, year)

  # ...and send the user to the show page for our new book
  redirect "/books/#{new_book_id}"
end
```

This method does three things:

1. It gathers the values that the browser pushed to the server, these values are in the `params` variable.
2. Then it calls the `BookDatabase.add()` method to add a new the record to our database.
3. And at the end it sends the user to the "details page" for the particular book that they just added (in HTTP lingo, it redirects them).

JavaScript (on the client-side)

Another common technology that is used in most web pages is JavaScript (JS). JavaScript is a programming language that we can embed in our HMTL files to add dynamic functionality to the page.

For example, we might want to validate that the value in a particular textbox is not empty on a page or that it matches a specific type of format (e.g. say a phone number). This kind of dynamic behavior on the page can be implemented with

JavaScript.

The topic of JavaScript on the client-side is large and we won't cover the details on this workshop. The Mozilla JavaScript documentation is a great place to learn more about JavaScript.

> Note: This section of the workshop is about using JavaScript on the client-side of a web application. It is also possible to use JavaScript on the backend of a web application via Node.js but that topic is outside of the scope of this workshop.

Let's modify our web application to validate the `title` of the books when somebody *edits* a book. To do this we are going modify the controller to render a different page, instead of using our original `book_edit.erb` page we are going to use `book_edit_with_js.erb`:

```
# Display the edit form for a single book
get("/books/:id/edit") do
  id = params[:id].to_i
  @book = BookDatabase.find(id)
  erb(:book_edit_with_js)
end
```

The content of the `./views/book_edit_with_js.erb` view is similar to the one that we had before, but at the bottom of this page we have a new `<script>...</script>` section. This is the JavaScript code that will make the validation:

```
  <script>
    /*
     * When the "saveButton" is clicked...
     */
    document.getElementById("saveButton").addEventListener("click",
(event) => {

      const titleEl = document.getElementById("title");
      if (titleEl.value.trim() === "") {
        const messageEl = document.getElementById("title-required");
        messageEl.innerHTML = "This field is required";
        event.preventDefault();
      } else {
        messageEl.innerHTML = "";
      }

    });
  </script>
```

This JavaScript code does two things. First it attaches an event to the `click` event of the `saveButton`. With this code attached, when the button is clicked it will call the code that we defined to validate that the title is not empty.

The code to perform the validation looks at the value of the `title` field. If it is empty it sets the text "This field is required" to another element on the page (`title-required`) and tells the button to not execute its default behavior via the call to `event.preventDefault()`. The default behavior of the submit button is to issue the HTTP POST defined in the HTML FORM, the call to `event.preventDefault()` stops this call.

When writing JavaScript on the client-side is common to make references to the HTML elements on the page, these elements are commonly known as the Document Object Model (DOM). The `document` referenced in the previous code is how we access the DOM in JavaScript.

Although JavaScript is a programming language like Ruby or Python, the code that we write in JavaScript on the client-side tends to look different from code in these other languages because (1) JavaScript is a completely different programming language and (2) the code in JavaScript tends to run asynchronous and asynchronous programming is hard. But asynchronous programming is also what allows the code in JavaScript to run in your browser without freezing your browser while the code executes. Again the Mozilla Developer Network has a good introduction to this topic.

Where do we go from here?

All the work that we have done in this workshop creates a website that runs on our personal computer, and although it could be deployed to a server environment for the world to access and enjoy, we do not cover website deployment and hosting in this workshop.

However, if you want to experiment creating a site that is available for the world to see you can use some of the free hosting platforms like Glitch or GitHub pages. These platforms allow you to build simple sites with just HTML, CSS, and JavaScript and make it available to the world relatively easily.

If you want to take it a bit further you could also buy your own domain and own your little corner of the web.

This workshop uses Ruby for the backend portion of web development, but a lot of the ideas and concepts presented here apply to any backend programming language like Python, PHP, Java, or C#. For example, take a look at the documentation on how to build a minimal web application with Python using

Flask (Flask is similar to Sinatra but is for Python applications) and notice the similarities in the concepts to what we learned in this workshop.

Acknowledgements

A special thank you goes to Claudia Lee for helping me test the initial examples and validate that they work on Windows environments and to Robert-Anthony Lee-Faison for helping me prepare the workshop and present it at Wintersession 2025.