HTML learning roadmap

Perfect 👌 Now let's go **deeper into the Rendering Pipeline**, step by step, with *more detailing + real examples + visuals in mind*.

We'll cover **everything from raw HTML bytes → fully rendered webpage**.
Think of it like a *factory line*:

> Bytes (materials) → Tokens (blocks) → Nodes (parts) → Trees (blueprints) → Styles → Layout → Paint → Composite → Final Screen.

🌐 Browser Rendering Pipeline (Full Depth)

---

1. HTML Bytes → Decoder

- When the browser receives an **HTML file** from the server, it's in **raw bytes** (0s and 1s).
- Browsers must interpret these bytes using a **character encoding scheme** (UTF-8, ISO-8859-1, etc.).
- This decoding converts raw bytes → **human-readable characters** (text).

📌 Example:

- Bytes: `01001000 01010100 01001101 01001100`
- With UTF-8 decoding → `"HTML"`

⚠ If wrong encoding is chosen (e.g., server says ISO but actual file is UTF-8), you'll see **garbled text**.

---

2. Tokenization

- After decoding, the browser runs an **HTML Tokenizer**.
- Tokenizer scans the character stream and groups them into **tokens**.
- Tokens represent the smallest meaningful parts of HTML:
  - **Start tag**: `<html>`, `<p>`
  - **End tag**: `</p>`
  - **Attributes**: `class="title"`

- **Text tokens:** `Hello World`

📌 Example:

Input:

```
html

<p class="msg">Hi</p>
```

Tokens:

- `StartTag: p` (with attribute `class="msg"`)
- `Text: Hi`
- `EndTag: p`

---

3. Node Builder

- Tokens → **DOM Nodes** (objects in memory).
- Each tag becomes a **node object** (tree element).
- Text becomes **text nodes**.

📌 Example:

```
html

<p class="msg">Hi</p>
```

Becomes:

- `ElementNode: <p>` (with `class=msg`)
- `TextNode: "Hi"`

---

4. Tree Builder

- Nodes are arranged into a **DOM Tree** (Document Object Model).
- This represents the **structure of the page**.

📌 Example:

```
html

<html>
 <body>
```

```
  <p class="msg">Hi</p>
 </body>
</html>
```

DOM Tree (visual):

```
cpp

Document
 └── html
     └── body
         └── p (class="msg")
             └── "Hi"
```

⚠ **Errors are handled gracefully** → browsers auto-close tags if missing.

```
html

<p>Hello
```

→ Browser still makes a valid DOM with `<p>Hello</p>`.

---

5. CSSOM Construction

- Parallelly, browser fetches CSS files.
- CSS also goes through **tokenization, node building, tree building**.
- Result = **CSSOM (CSS Object Model)** → rules structured in tree format.

📌 Example CSS:

```
css

p.msg { color: red; font-size: 20px; }
```

CSSOM:

```
yaml

CSSRuleList
 └── Rule: selector = p.msg
     ├── color: red
     └── font-size: 20px
```

6. Render Tree

- **DOM + CSSOM** are combined → **Render Tree**.
- The Render Tree contains only **visible elements** with **final styles applied**.
- Invisible elements (`<head>`, `display: none`) are skipped.

📌 Example:

DOM: `<p class="msg">Hi</p>`

CSS: `p.msg { color: red; font-size: 20px; }`

Render Tree Node:

```arduino
RenderObject (type: p)
  ├── color: red
  ├── font-size: 20px
  └── text: "Hi"
```

---

7. Layout (Reflow)

- Browser calculates **geometry** (size, position) of each element.
- Uses **box model**: content → padding → border → margin.
- Depends on parent-child relationships and viewport size.

📌 Example:

```css
p.msg { font-size: 20px; margin: 10px; }
```

Layout result:

- `<p>` width = 300px, height = 40px, margin-top = 10px.

⚠ Any DOM/CSS change can **trigger reflow** → expensive for performance.

---

8. Painting

- Once layout is done, browser paints pixels (colors, borders, shadows, images).
- Each node → rasterized into bitmaps.

📌 Example:

`<p style="color:red; background:yellow">Hi</p>`

→ Paints a yellow rectangle, then draws red text "Hi".

---

9. Compositing

- Complex pages are split into **layers** (like Photoshop).
- Layers are composited together in the correct order (using **z-index**, stacking contexts, transforms, opacity).
- GPU helps merge layers → final frame is pushed to screen.

📌 Example:

- Background layer → main content layer → floating modal layer → cursor overlay.

🔥 Example Walkthrough

Take this code:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Test</title>
  <style>
    p { color: blue; }
  </style>
</head>
<body>
  <p>Hello World</p>
</body>
</html>
```

➡ Step by Step:

1. Bytes decoded → `"<!DOCTYPE html>..."`
2. Tokenizer sees `<html>`, `<head>`, `<p>...`
3. DOM Nodes built (`html`, `head`, `body`, `p`, text).
4. CSS parsed → CSSOM with rule `p { color: blue }`.
5. Render Tree created with visible `<p>` + computed style (color: blue).

6. Layout calculates `<p>` position/size.
7. Paint draws white background + blue text.
8. Composite merges into final frame.
   ✅ Browser displays → "Hello World" (blue).

👉 So the **Rendering Pipeline** is not one step, but a chain:
Decode → Tokenize → DOM → CSSOM → Render Tree → Layout → Paint → Composite

---

Would you like me to now **deep dive separately into DOM Tree + CSSOM + Render Tree differences** with more real MERN examples (like a dashboard page),
or move directly into **performance issues (Reflow vs Repaint vs Composite only)**?