# Day 9

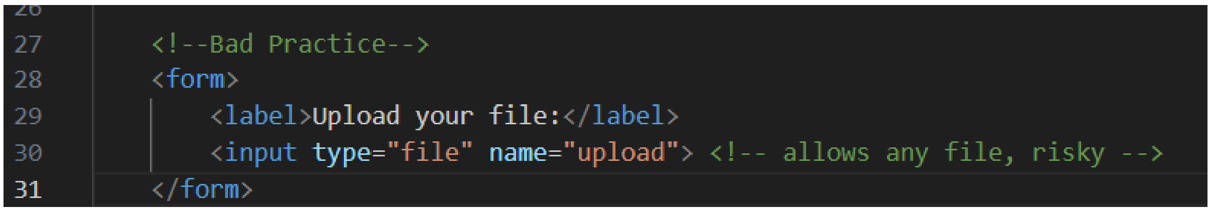
# “Web Development + Security”

## Security Practices for Forms and Input tags:

## Security practices for <input> tag:

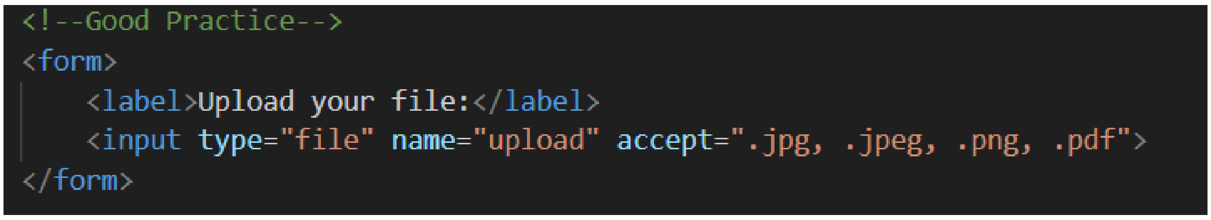
1. Restrict file types using accept.
2. Limit text length with maxlength.
3. Always validate input server-side.

Bad Practice example: (1)



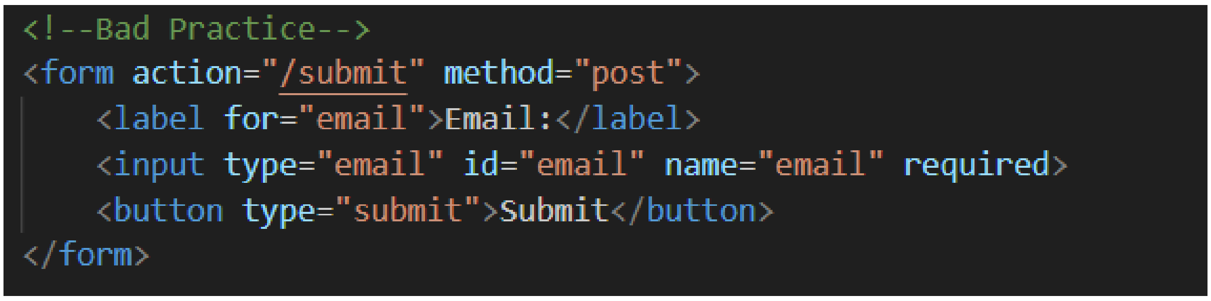
```html
<!--Bad Practice-->
<form>
    <label>Upload your file:</label>
    <input type="file" name="upload"> <!-- allows any file, risky -->
</form>
```

Good Practice example: (1)



```html
<!--Good Practice-->
<form>
    <label>Upload your file:</label>
    <input type="file" name="upload" accept=".jpg, .jpeg, .png, .pdf">
</form>
```

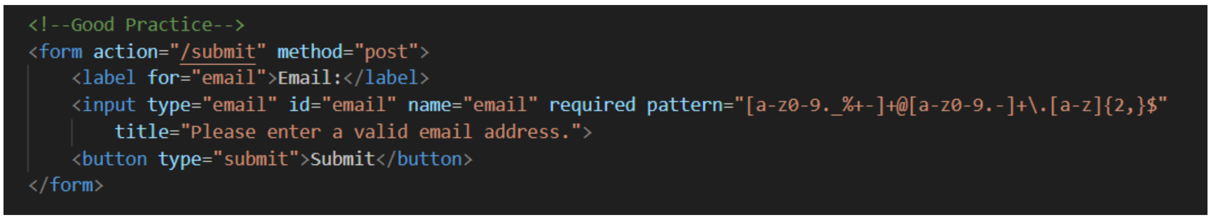
Bad Practice example: (2)



```html
<!--Bad Practice-->
<form action="/submit" method="post">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
    <button type="submit">Submit</button>
</form>
```

The browser checks if it looks like an email, but an attacker can disable JS, use dev tools, or send a direct POST request with invalid data like not-an-email or malicious scripts. If the server doesn’t check it, it can store invalid or dangerous data.

Good Practice example: (2) (simple HTML fix)

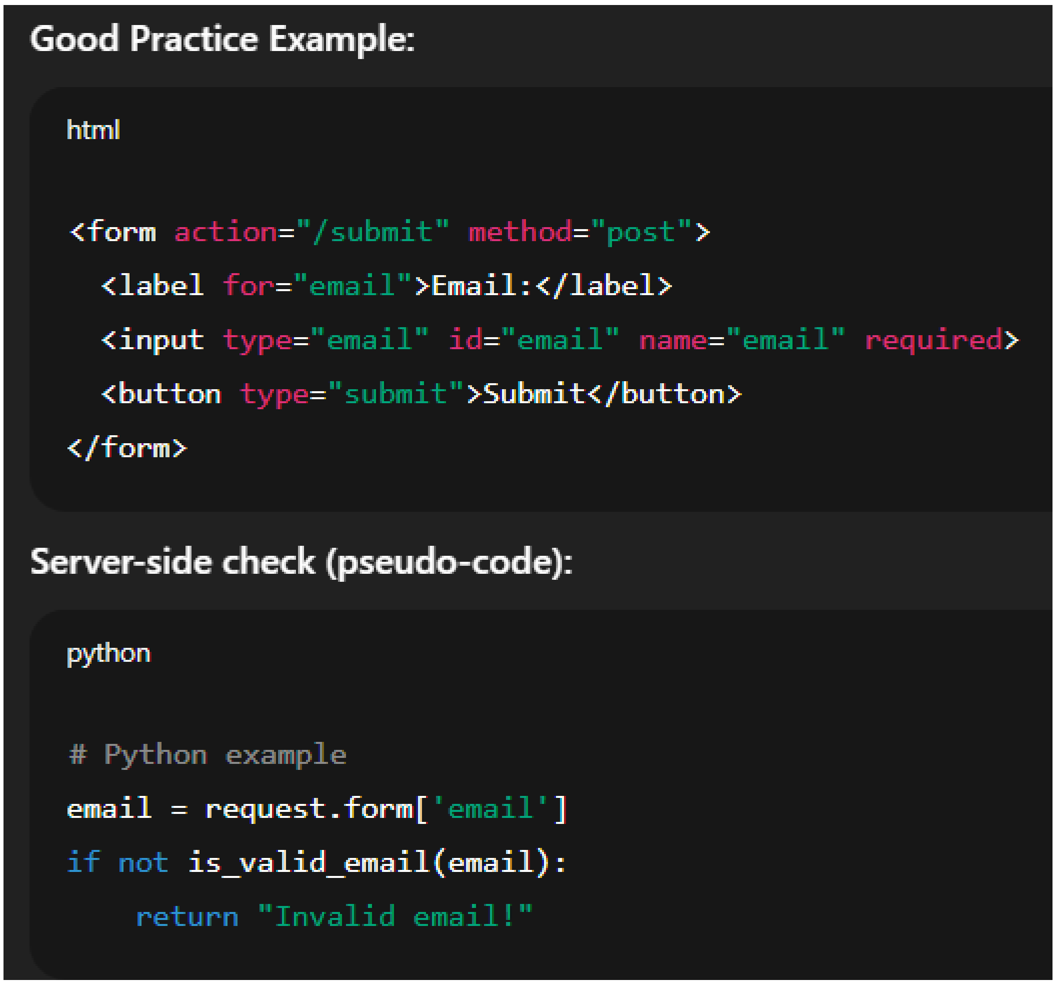


```html
<!--Good Practice-->
<form action="/submit" method="post">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"
        title="Please enter a valid email address.">
    <button type="submit">Submit</button>
</form>
```

**Why it's good:**

1. **type="email"** → basic HTML5 email validation.
2. **required** → ensures the user cannot submit the form empty.
3. **pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"** → enforces a stricter email format.
4. **title="..."** → gives a helpful message if the pattern doesn't match.
5. **label for="email"** → improves usability and accessibility.

Good Practice example: (2) (Server-side fix)



Good Practice Example:

html

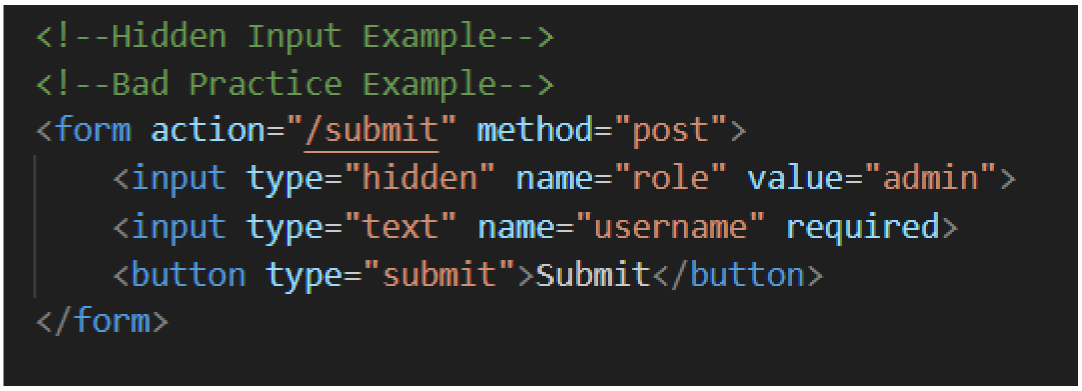
```html
<form action="/submit" method="post">
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <button type="submit">Submit</button>
</form>
```

Server-side check (pseudo-code):

python

```python
# Python example
email = request.form['email']
if not is_valid_email(email):
    return "Invalid email!"
```

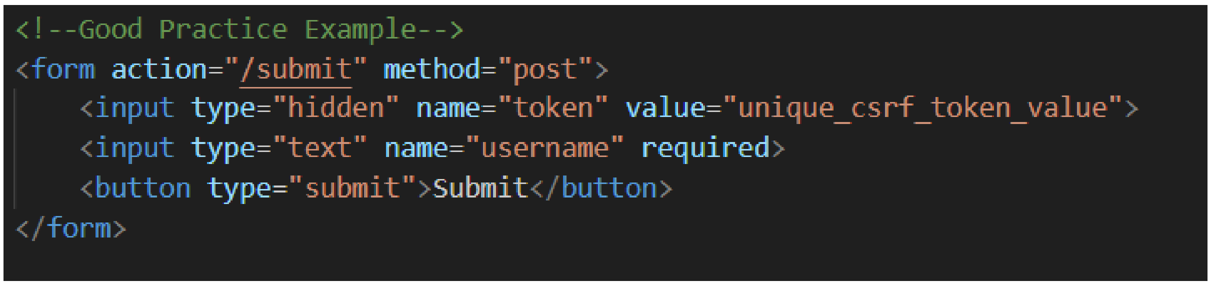
Bad Practice example: (3)



```html
<!--Hidden Input Example-->
<!--Bad Practice Example-->
<form action="/submit" method="post">
    <input type="hidden" name="role" value="admin">
    <input type="text" name="username" required>
    <button type="submit">Submit</button>
</form>
```

It is not secure because it stores sensitive information, like the user’s role, in a hidden field. Hidden fields can be easily modified by anyone using browser developer tools or by sending a custom request, which means a normal user could change their role to “admin” and gain unauthorized access. Additionally, if the server blindly trusts this hidden value without validation, it creates a serious security risk. In short, sensitive data should never be stored or relied on in client-side hidden fields; all critical logic must be handled securely on the server.

Good Practice example: (3)

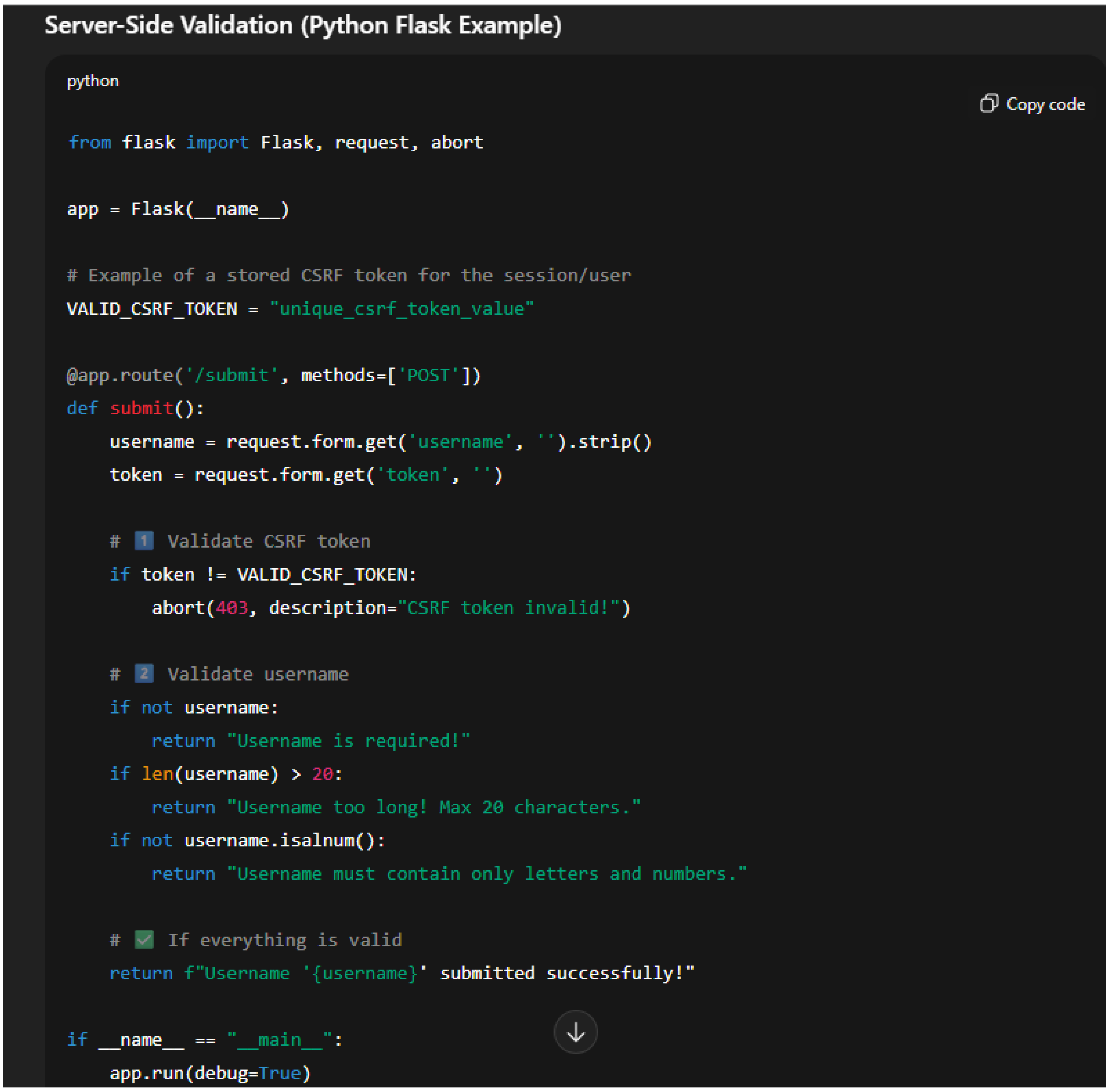


```html
<!--Good Practice Example-->
<form action="/submit" method="post">
    <input type="hidden" name="token" value="unique_csrf_token_value">
    <input type="text" name="username" required>
    <button type="submit">Submit</button>
</form>
```

The hidden field contains a CSRF token, which is a unique, randomly generated value for each user session. When the form is submitted, the server checks this token to ensure the request is coming from a legitimate user and not from a malicious site trying to perform actions on the user’s behalf. This prevents Cross-Site Request Forgery (CSRF) attacks. The form does not include sensitive data like roles or passwords in hidden fields, keeping critical logic on the server side.

But, If the token is static, predictable, or poorly implemented, attackers could guess it and bypass CSRF protection. Also, if developers start adding sensitive information (like roles, permissions, or passwords) in hidden fields alongside the token, it would create a security risk similar to the first bad example.

Good Practice example: (3) (Server-side check)



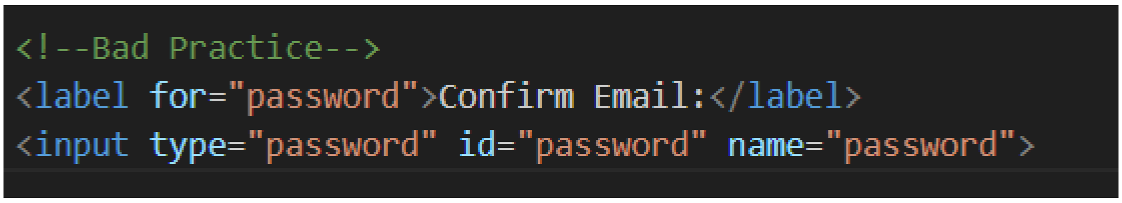
Server-Side Validation (Python Flask Example)

```python
from flask import Flask, request, abort

app = Flask(__name__)

# Example of a stored CSRF token for the session/user
VALID_CSRF_TOKEN = "unique_csrf_token_value"

@app.route('/submit', methods=['POST'])
def submit():
    username = request.form.get('username', '').strip()
    token = request.form.get('token', '')

    # 1️⃣ Validate CSRF token
    if token != VALID_CSRF_TOKEN:
        abort(403, description="CSRF token invalid!")

    # 2️⃣ Validate username
    if not username:
        return "Username is required!"
    if len(username) > 20:
        return "Username too long! Max 20 characters."
    if not username.isalnum():
        return "Username must contain only letters and numbers."

    # ✅ If everything is valid
    return f"Username '{username}' submitted successfully!"

if __name__ == "__main__":
    app.run(debug=True)
```

## <u>Security practices for &lt;label&gt; tag:</u>

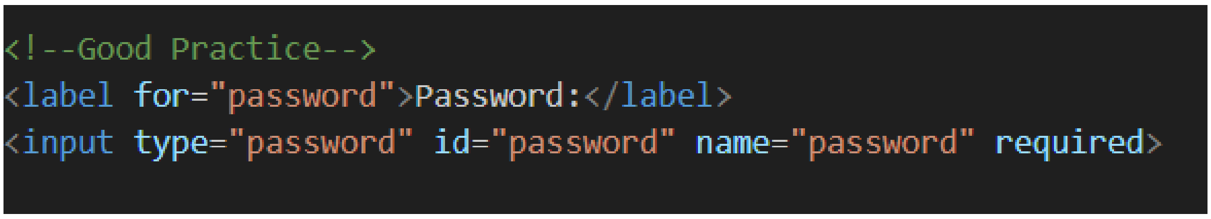
1. Label clearly describes the input.
2. Always link for to input id.

Bad practice:



```html
<!--Bad Practice-->
<label for="password">Confirm Email:</label>
<input type="password" id="password" name="password">
```

Good Practice:



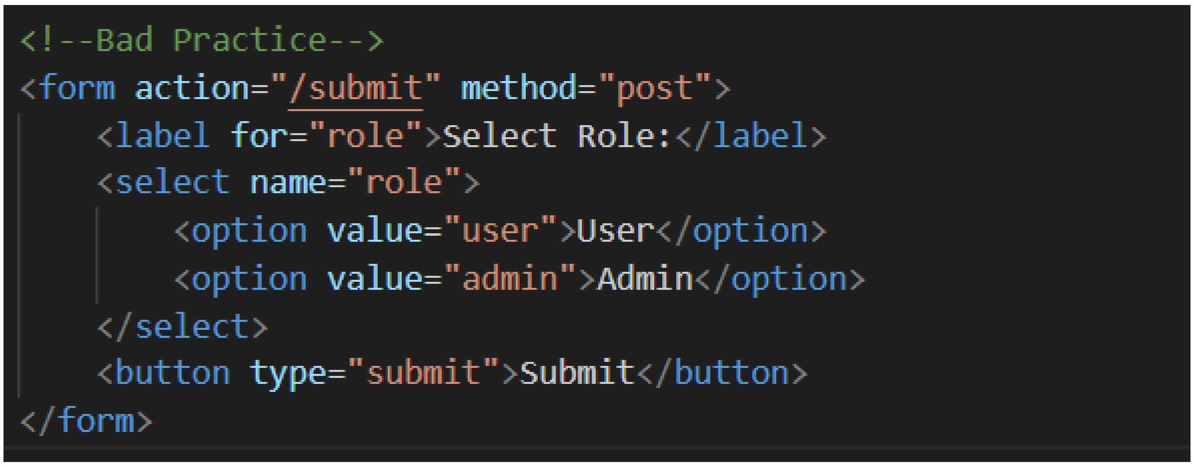
```html
<!--Good Practice-->
<label for="password">Password:</label>
<input type="password" id="password" name="password" required>
```

## <u>Security practices for &lt;select&gt; tag:</u>

1. Assume dropdown value is safe
2. No default placeholder
3. Rely on required

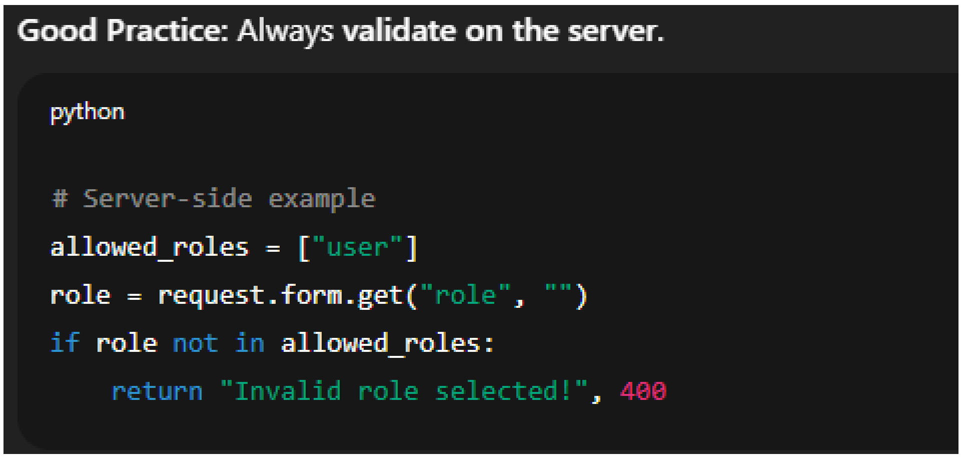
Bad Practice example 1:

Relying on the <select> values on the client side without server validation. Attackers can manipulate the value before submitting. A user can manually change role=admin and gain unauthorized privileges if the server trusts the value blindly.



```html
<!--Bad Practice-->
<form action="/submit" method="post">
    <label for="role">Select Role:</label>
    <select name="role">
        <option value="user">User</option>
        <option value="admin">Admin</option>
    </select>
    <button type="submit">Submit</button>
</form>
```

Good Practice example 1:



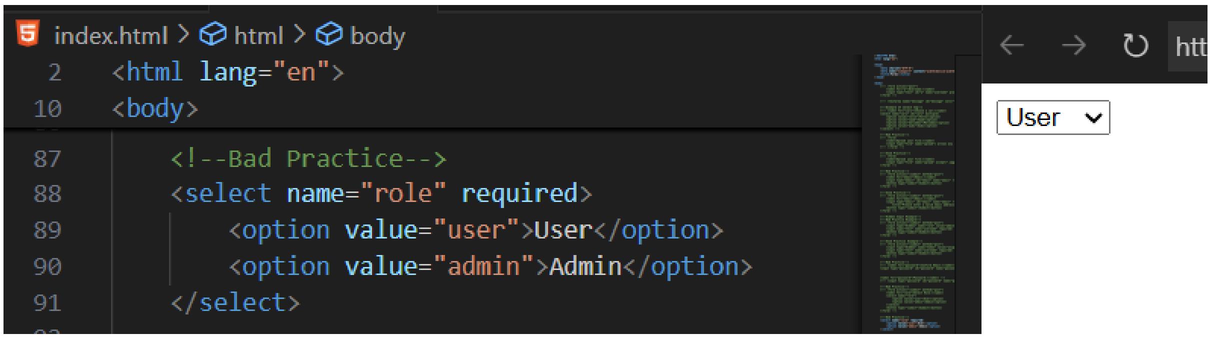
**Good Practice:** Always validate on the server.

python

```python
# Server-side example
allowed_roles = ["user"]
role = request.form.get("role", "")
if role not in allowed_roles:
    return "Invalid role selected!", 400
```

Bad Practice example 2:

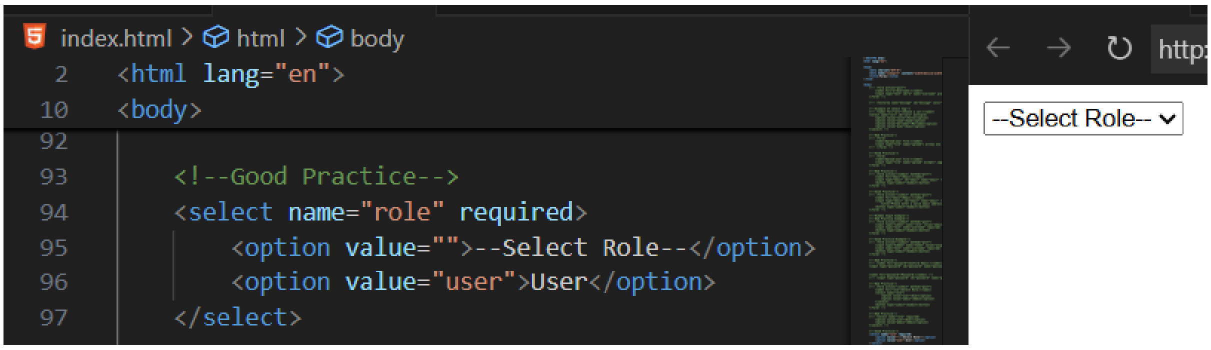
Not having a placeholder can cause the first option to be submitted unintentionally. If the user doesn’t actively select, user is automatically chosen, which may not reflect their intent.



index.html > html > body

```html
<html lang="en">
<body>
    <!--Bad Practice-->
    <select name="role" required>
        <option value="user">User</option>
        <option value="admin">Admin</option>
    </select>
```

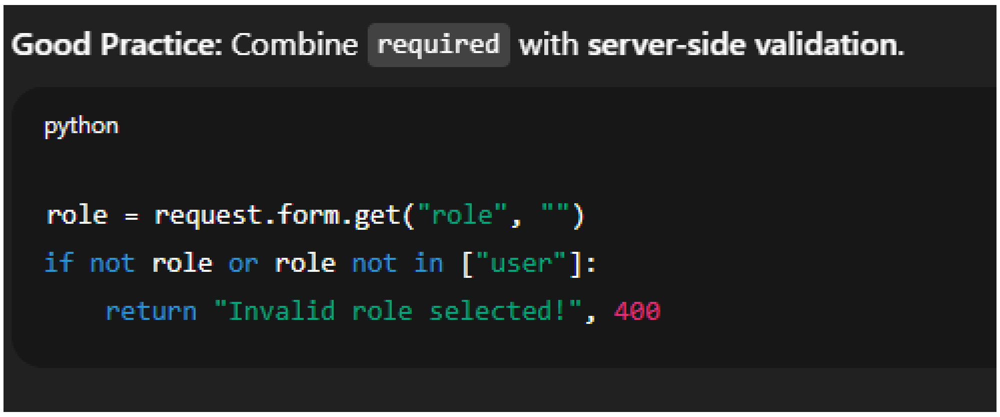
Good Practice example 2:



index.html > html > body

```html
<html lang="en">
<body>

    <!--Good Practice-->
    <select name="role" required>
        <option value="">--Select Role--</option>
        <option value="user">User</option>
    </select>
```

Good Practice example 3:


**Good Practice:** Combine `required` with **server-side validation.**

python

```python
role = request.form.get("role", "")
if not role or role not in ["user"]:
    return "Invalid role selected!", 400
```

--The End--