



Block 1 – Client Server Interaction

Maaike van Putten
Software developer & trainer
www.brightboost.nl





Day 4: PRG, validation & error handling

Overview



HTTP & HTML
FORMS



State
management



AJAX &
refresh
components



PRG,
validation
& error
handling

What we'll cover today

- Understanding idempotent operations
- The POST-Redirect-GET pattern
- AJAX returning HTML
- Form validation strategies
- Error handling in production
- Real-world edge cases





Understanding idempotent operations

(Recap)



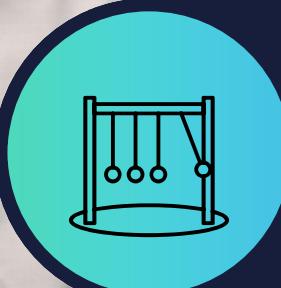
What is idempotency?

An operation is **idempotent** if performing it multiple times has the **same effect** as performing it once.

A photograph of a young Black man with short hair, wearing a white button-down shirt, sitting at a wooden desk. He is looking down at a piece of paper and a calculator, appearing to be in deep thought or calculation. The background shows some greenery and a window.

Idempotency

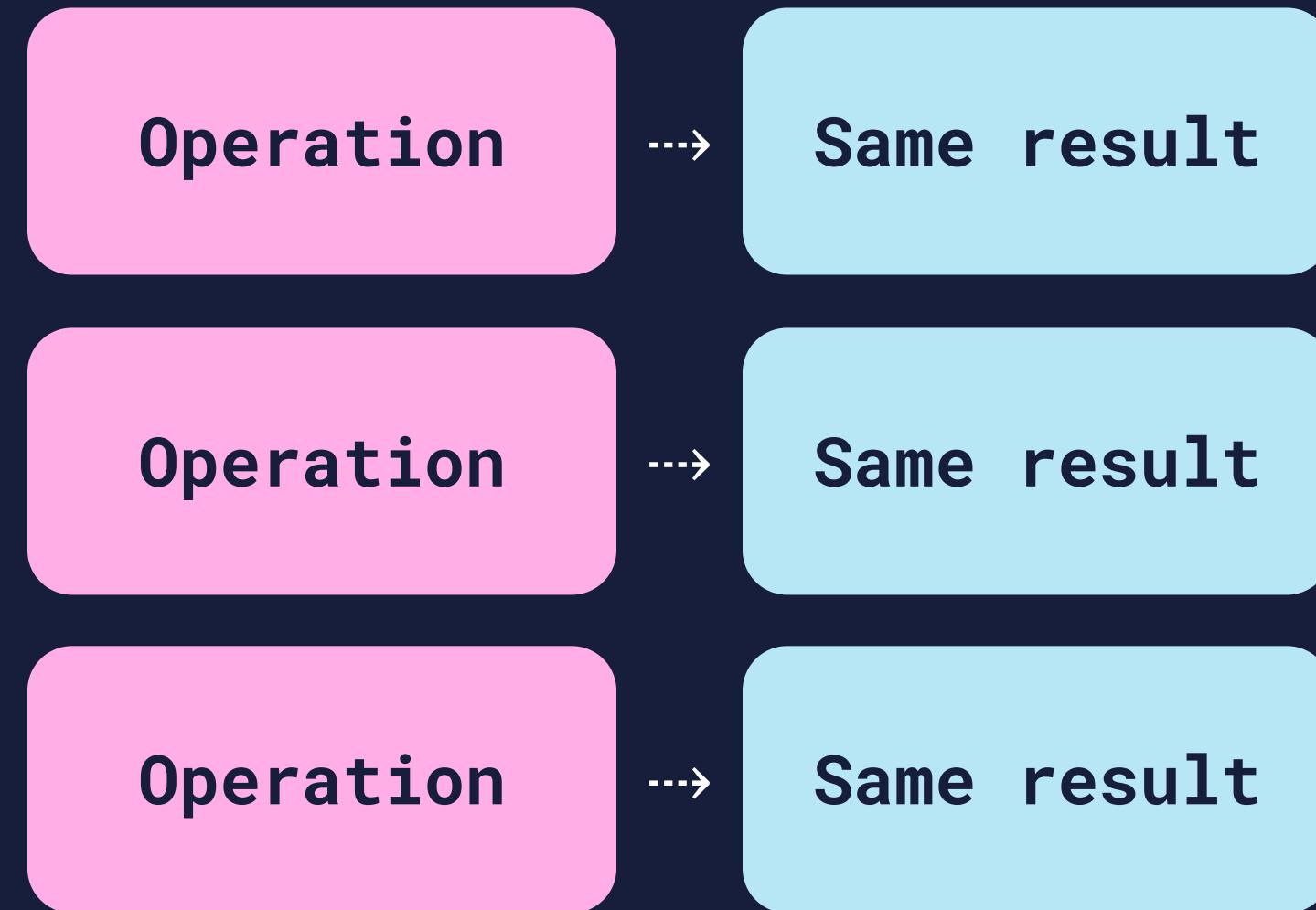
Not about the same response



About the same effect on the system

Math example: $x \times 1 = x$ (no matter how many times)

Idempotency



HTTP methods: which are idempotent?



Idempotent methods

- ✓ **GET /products**

Read data (safe + idempotent)

- ✓ **PUT /product/123**

Set product 123 to state X

- ✓ **DELETE /product/123**

Remove product 123



Not idempotent

- ✗ **POST /order**

Each call creates a new order

- ✗ **POST /cart/add**

Each call adds another item

- ✗ **POST /counter/increment**

Each call increments counter

What stands out here if we compare the left and right column?



POST is designed to be
non-idempotent

Idempotency in action: banking

This operation creates problems if accidentally repeated!



Idempotent operation

“Set account balance”

Do it once:

Balance =

Do it 5 times:

Balance = €100



Non-idempotent operation

“Deposit €100”

Do it once:

Balance = starting + €100

Do it 5 times:

Balance = starting + €500

A blurred background image of a person's hands using a laptop keyboard and holding a credit card, with a coffee cup on the desk.

Idempotency in e-commerce

Scenario 1: Idempotent

Operation: "Set quantity of item X to 3"

Result after 1x: Quantity = 3

Result after 10x: Quantity = 3

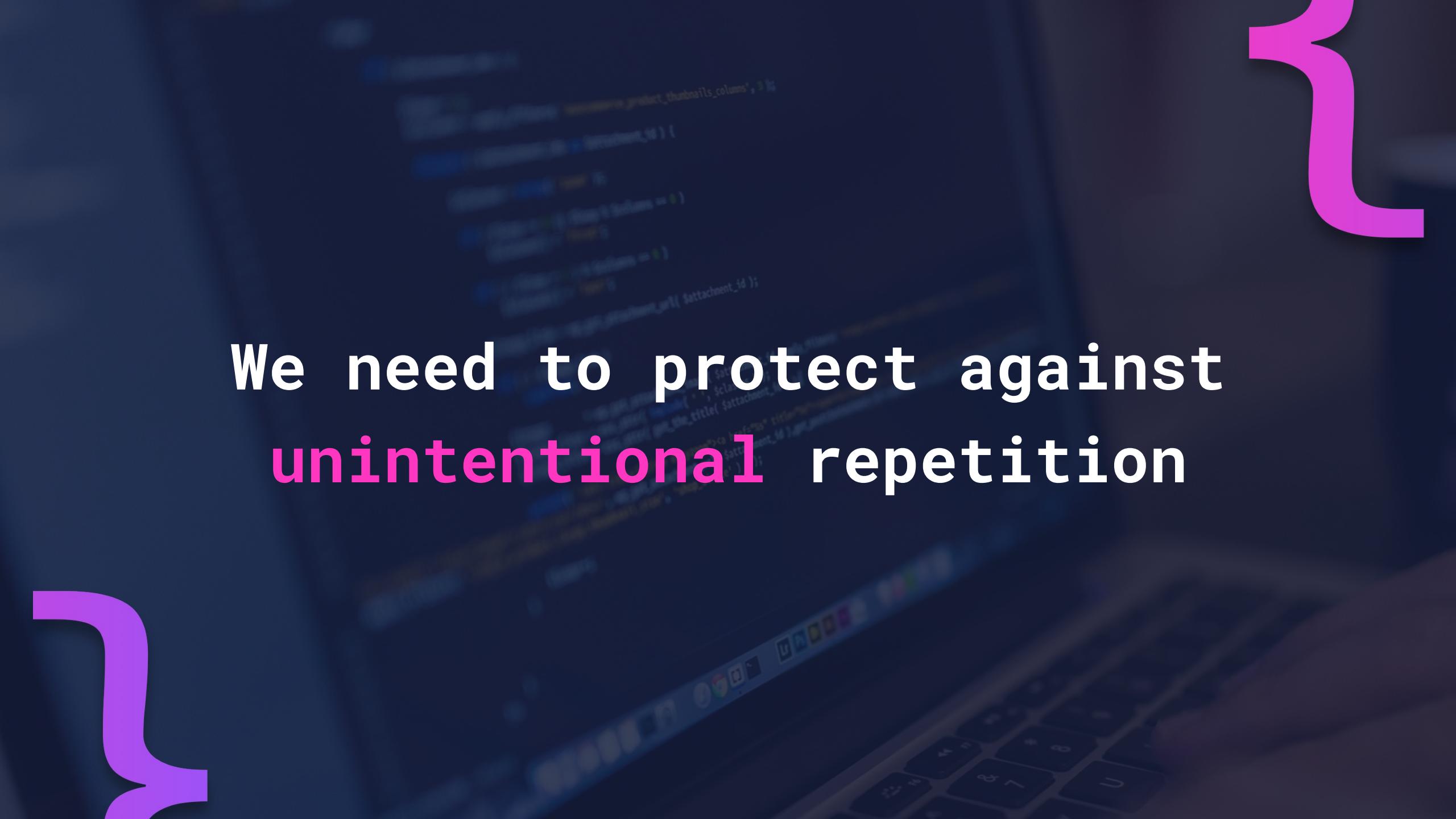


Scenario 2: Non-idempotent

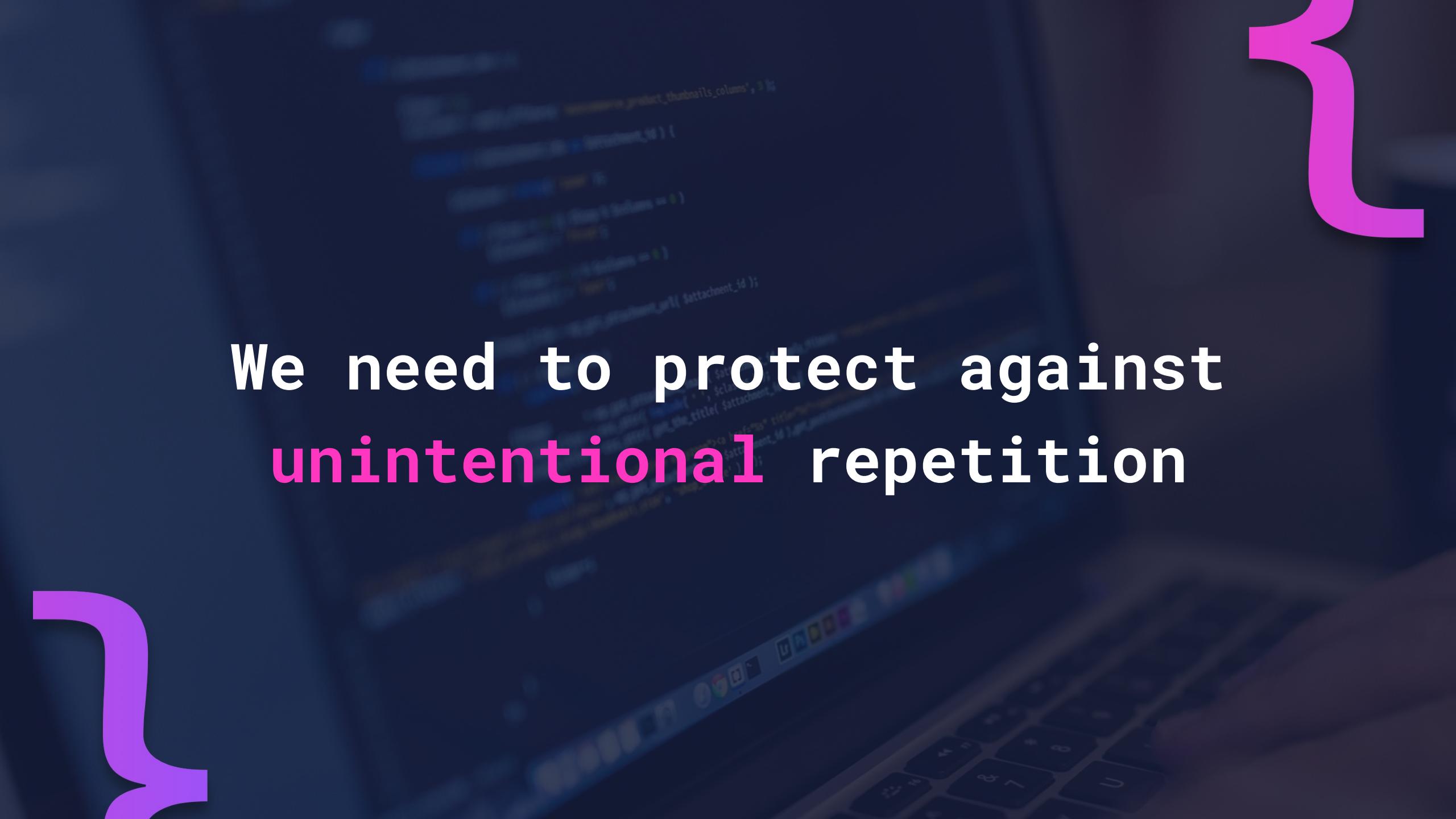
Operation: "Add 1 item X to cart"

Result after 1x: 1 item in cart

Result after 10x: 10 items in cart!



We need to protect against
unintentional repetition



The reality of HTTP communication



Networks are unreliable (timeouts, dropped packets)



Users are impatient (double-clicks, rapid refreshes)

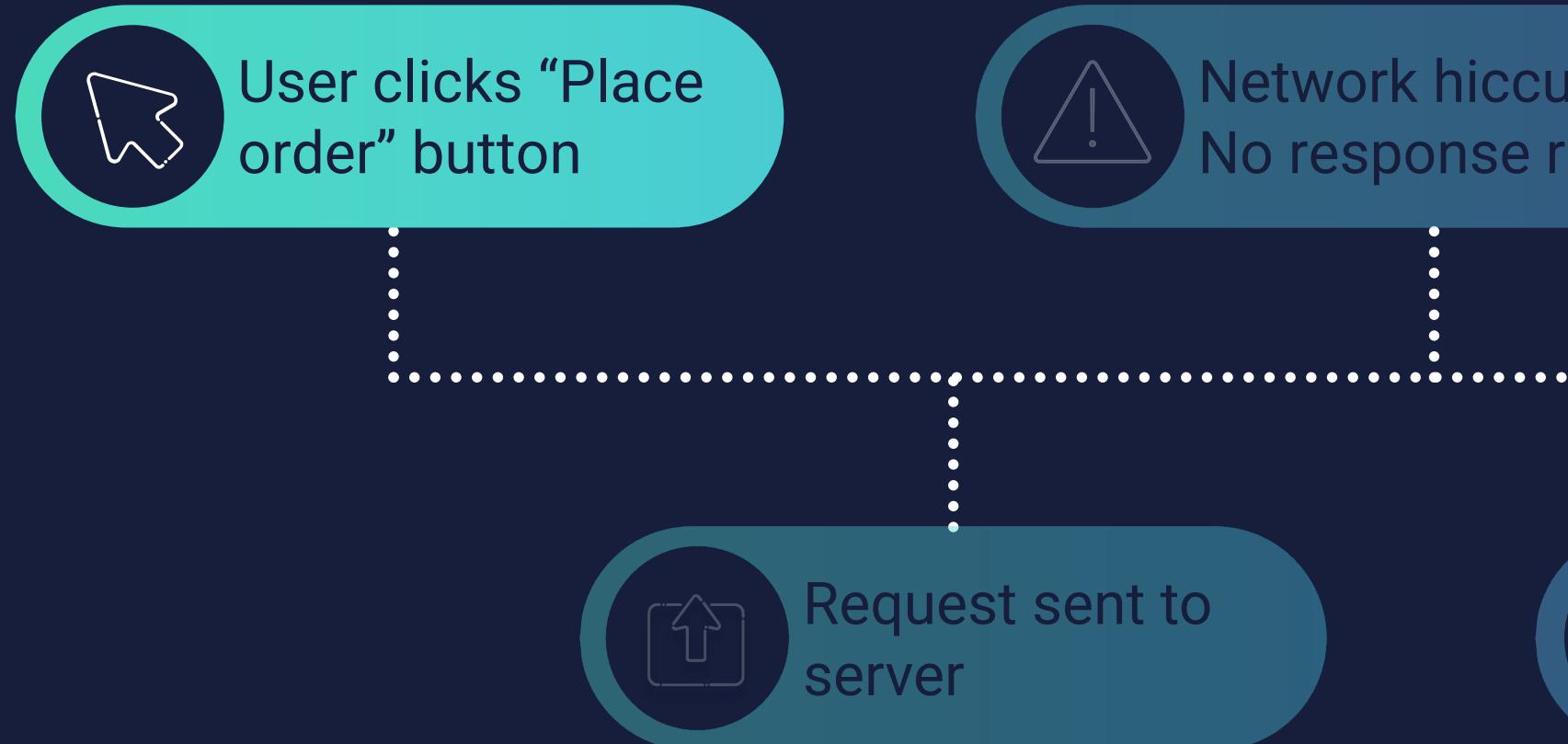


Browsers retry failed requests automatically

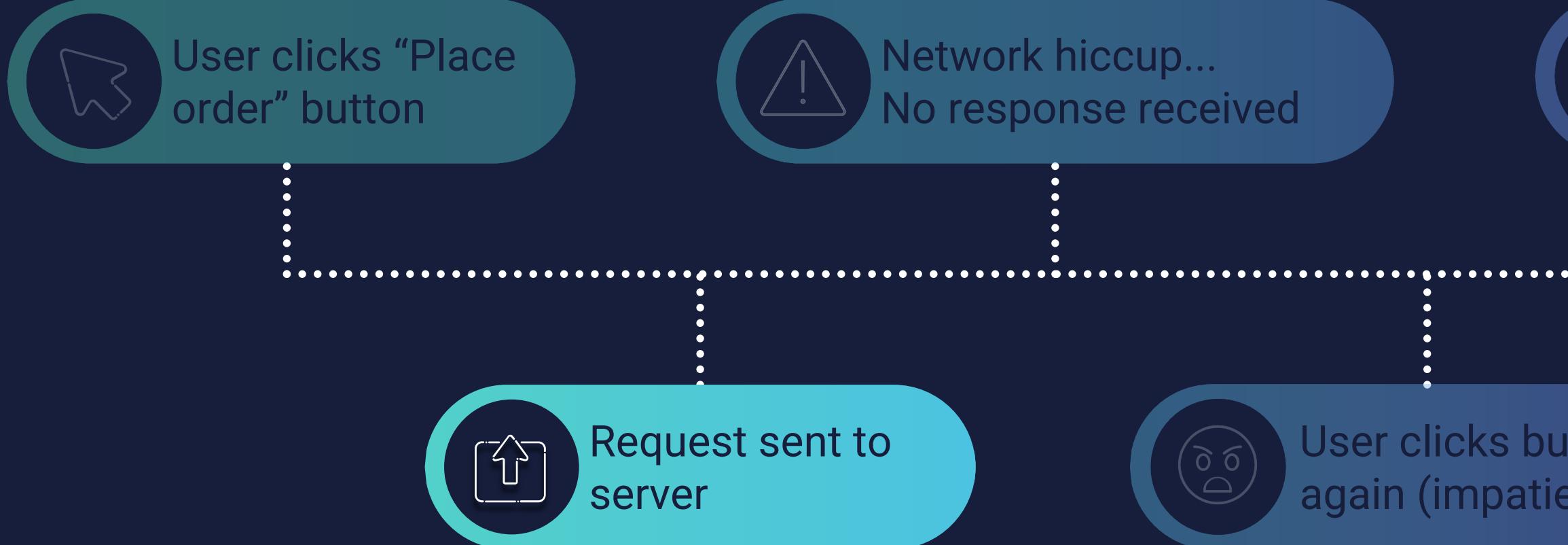


Mobile connections drop and reconnect

The scenario



The scenario



The scenario

User clicks “Place order” button



Network hiccup...
No response received



Second requ...
sent

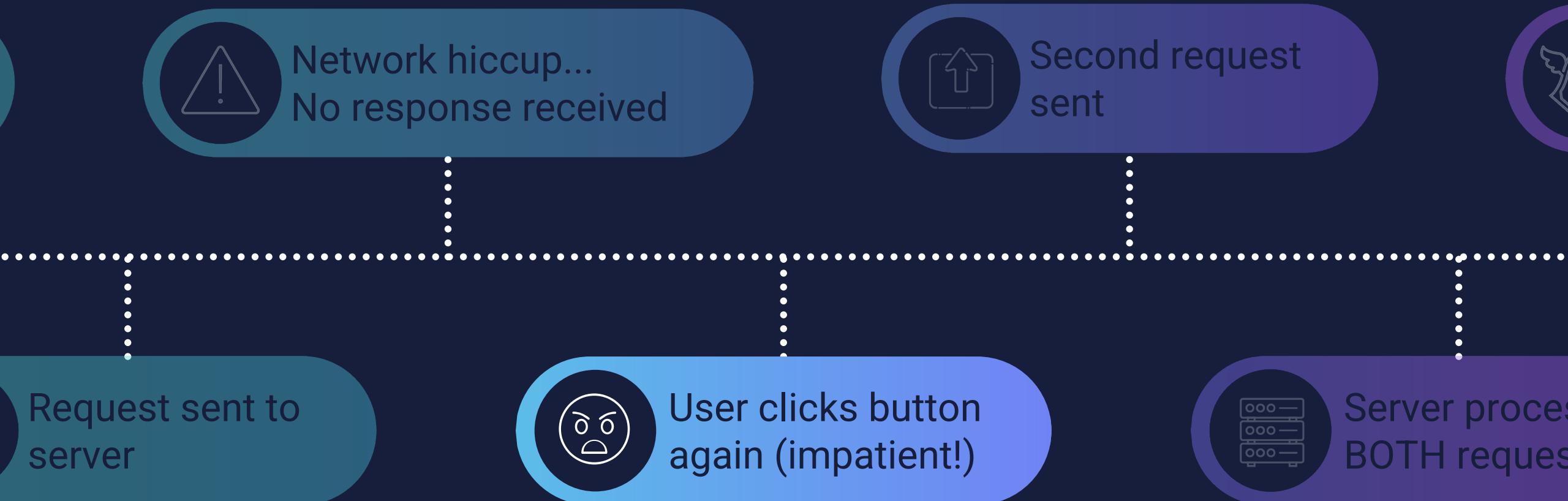


Request sent to
server



User clicks button
again (impatient!)

The scenario



The scenario

Network hiccup...
response received



Second request
sent



Result: User
charged twice!

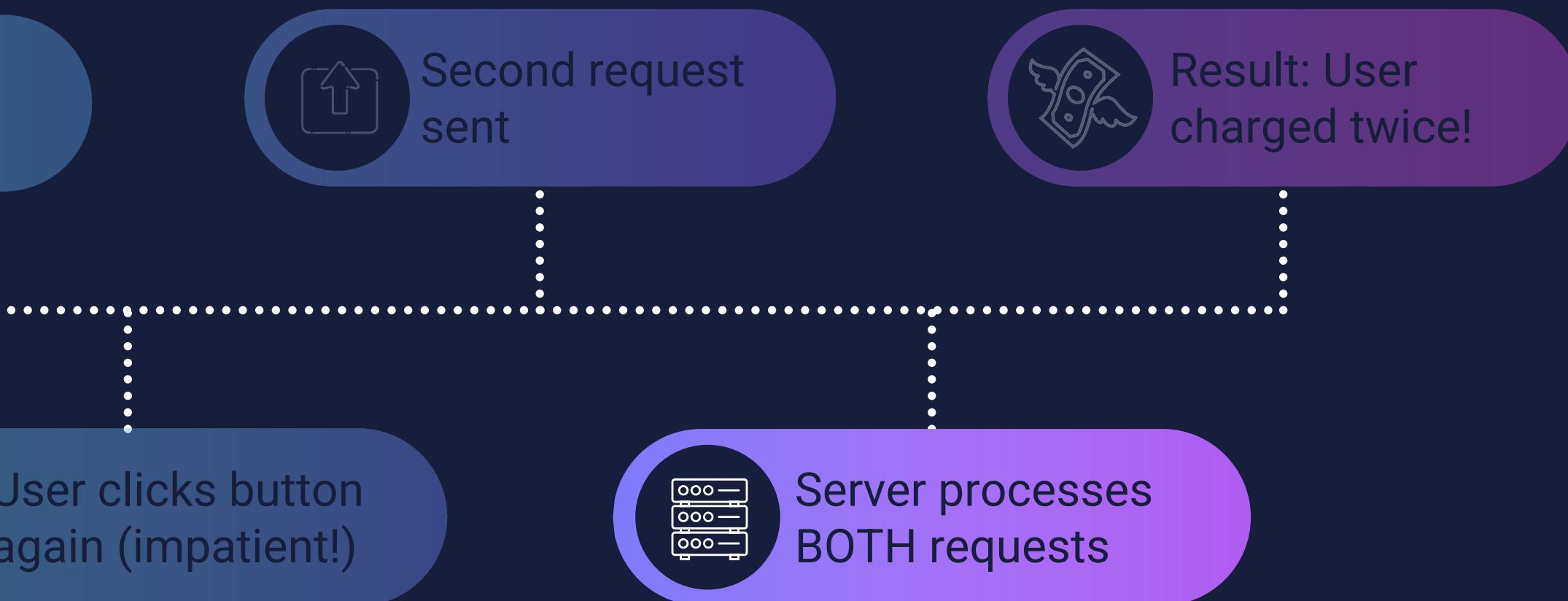


User clicks button
again (impatient!)

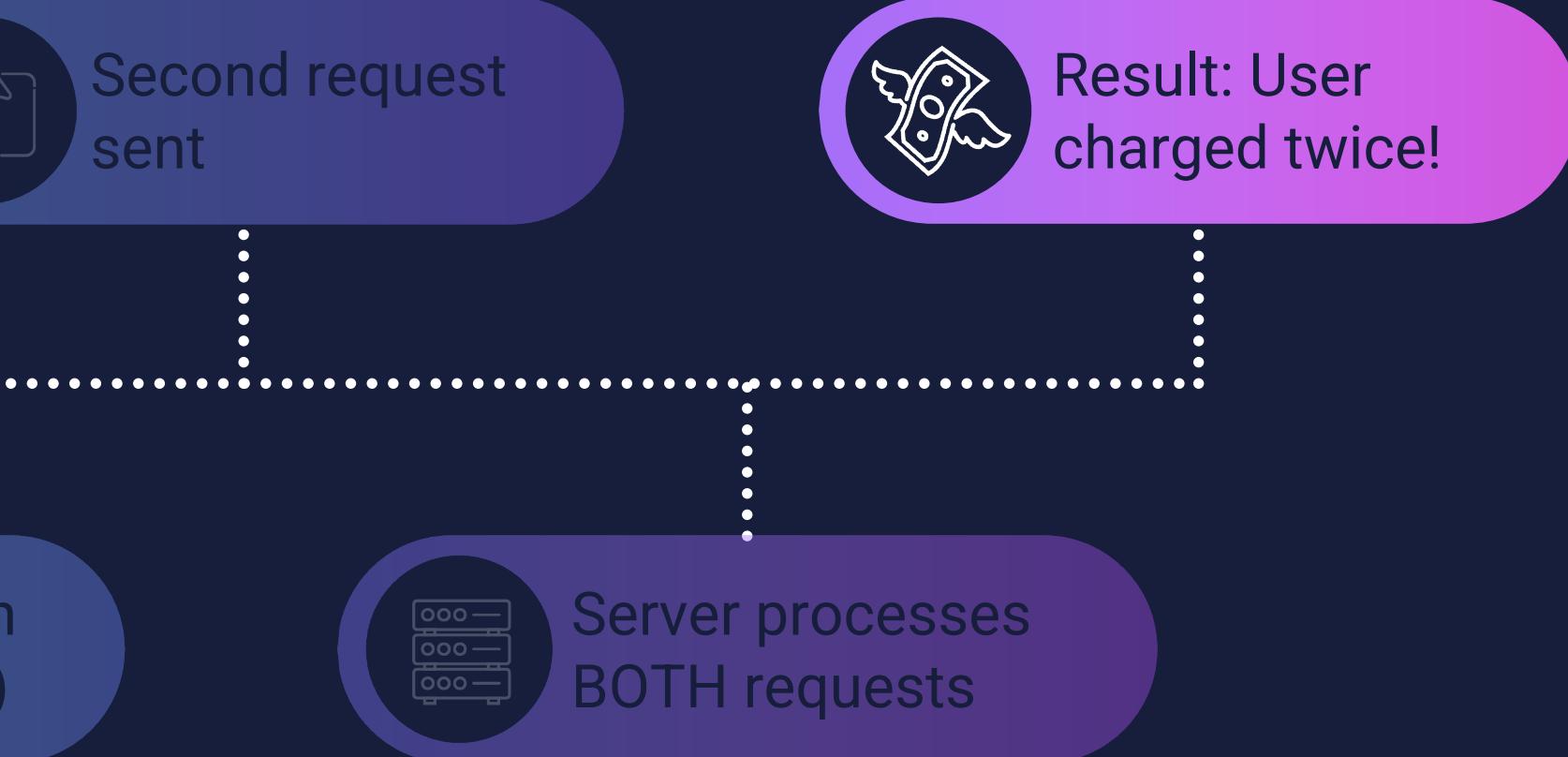


Server processes
BOTH requests

The scenario



The scenario





Why POST is special

By design:

- POST is **specifically designed** to be non-idempotent
- Creating resources **SHOULD** generate new ones each time
- This is correct behavior for the HTTP specification

The problem:

- Not POST itself, but **accidental repetition**



How do we prevent
unintentional duplicate
submissions?







The POST-Redirect-GET pattern

Solving the double submit problem

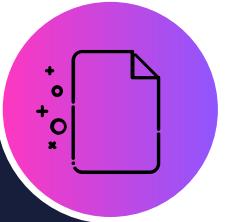
Traditional approach (the wrong way)

Simple HTML form

```
<form action="/order" method="post">
  <input type="text" name="item" value="burger" />
  <button>Place Order</button>
</form>
```



Traditional approach (the wrong way)

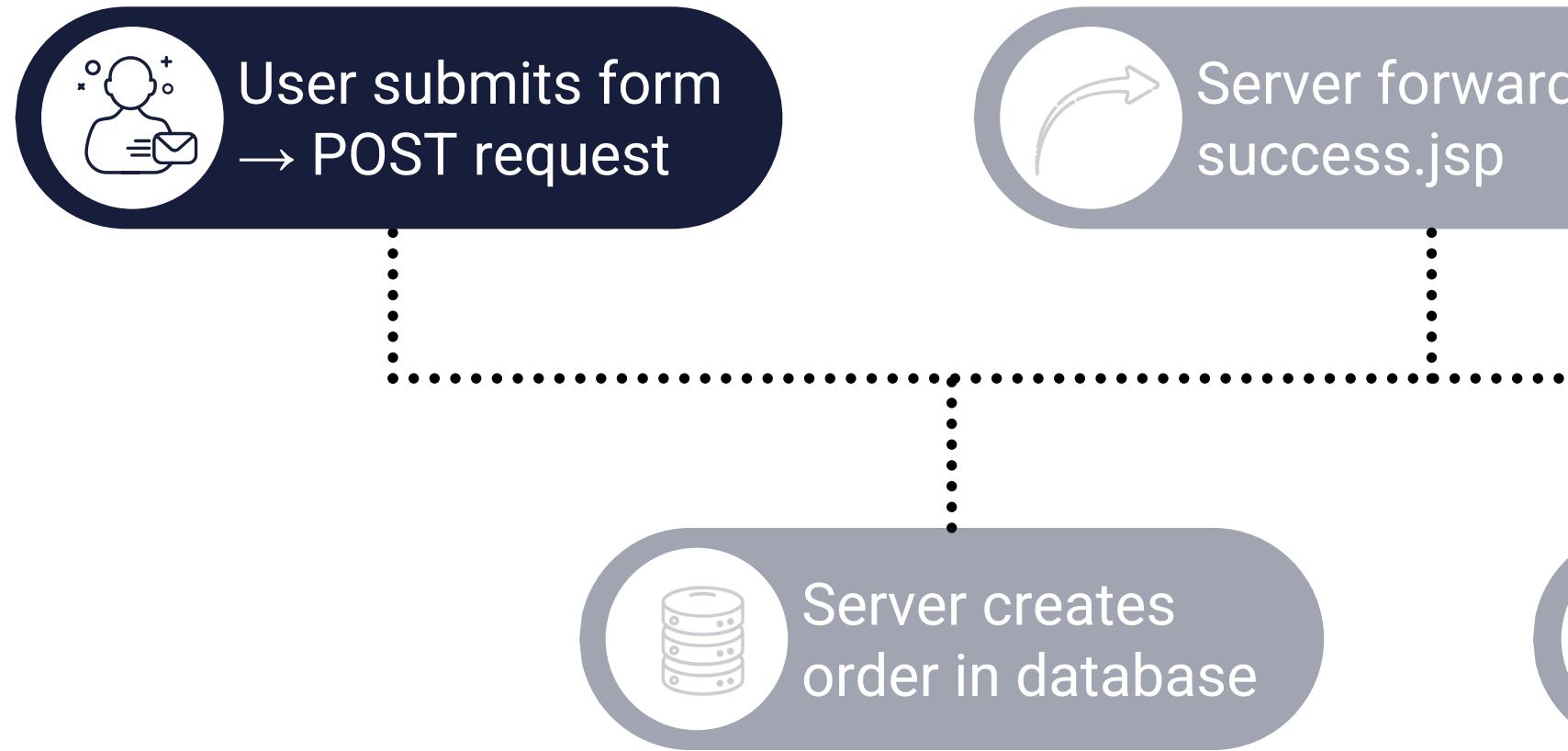


Servlet code (wrong)

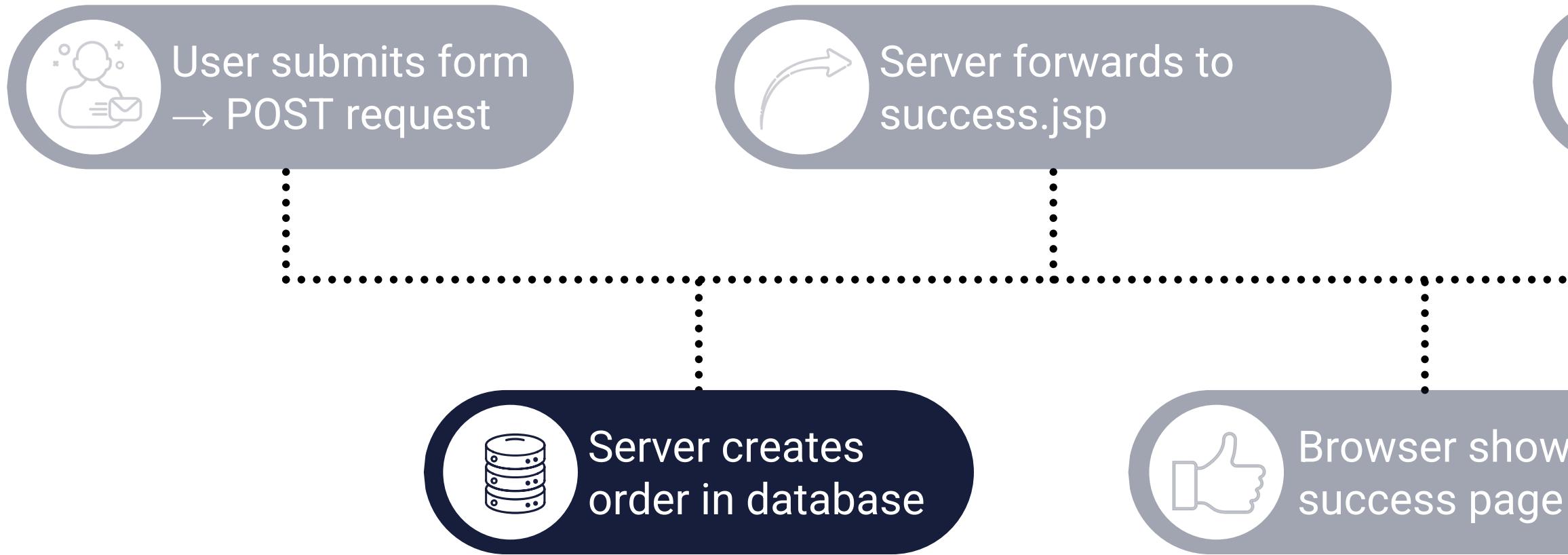
```
protected void doPost(...) {  
    // 1. Process the order  
    String item = request.getParameter("item");  
    Order order = orderService.createOrder(item);  
  
    // 2. Show success page directly: PROBLEM!  
    request.getRequestDispatcher("/success.jsp")  
        .forward(request, response);  
}
```

Let's see what goes wrong...

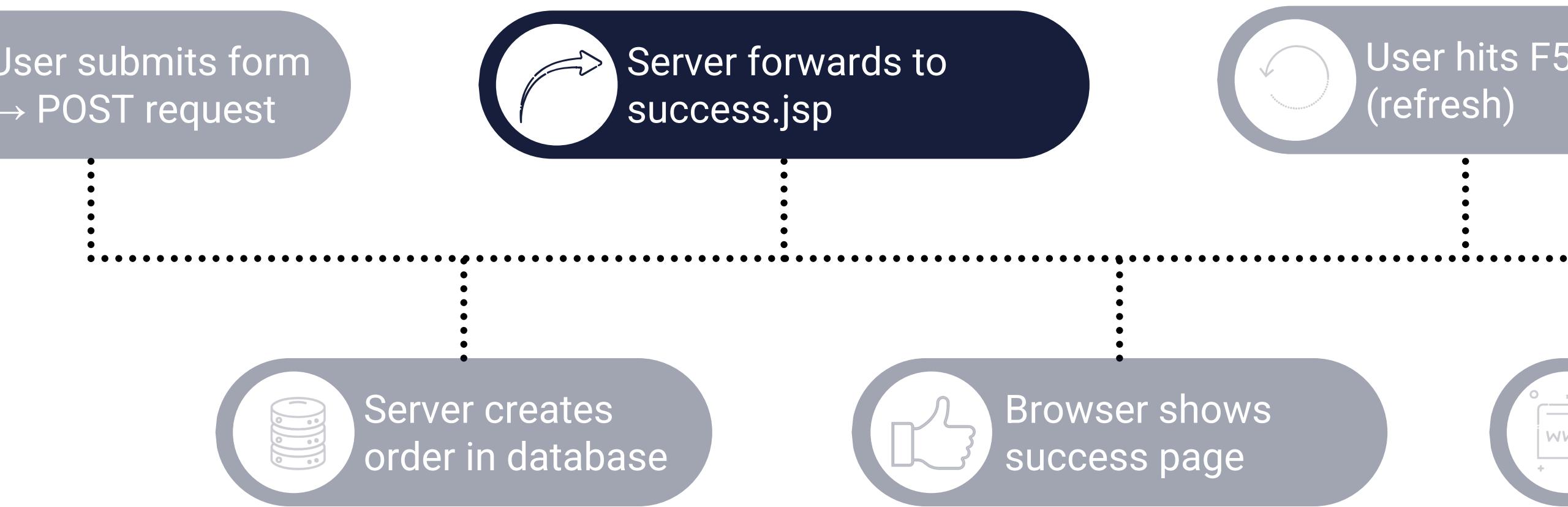
The double submit problem



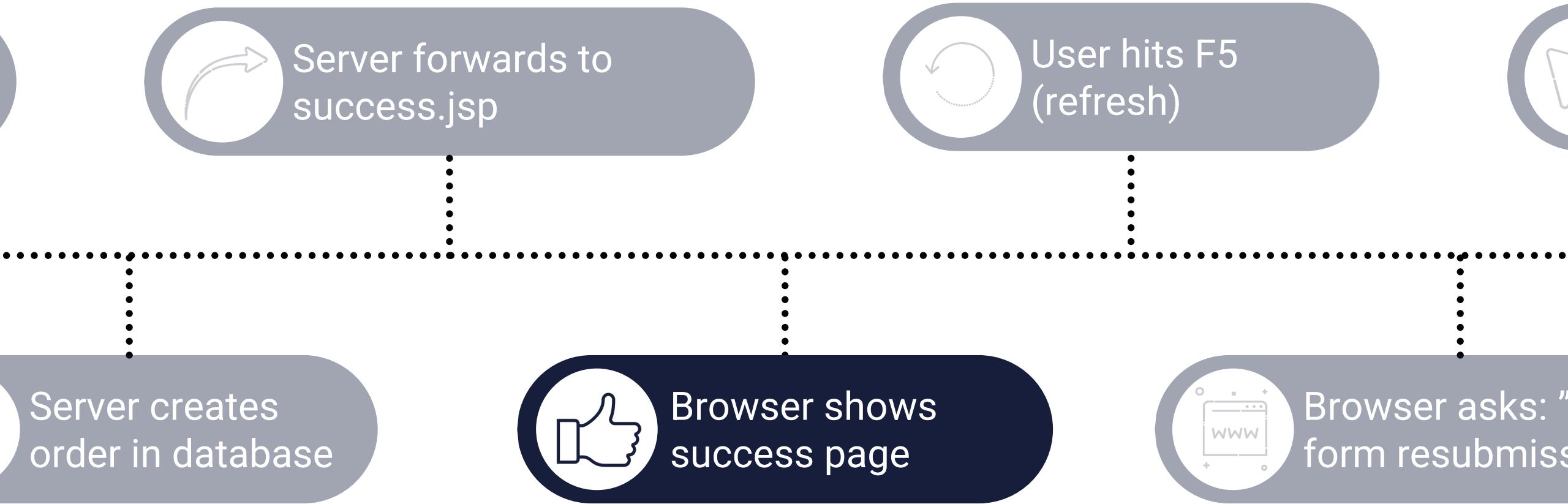
The double submit problem



The double submit problem

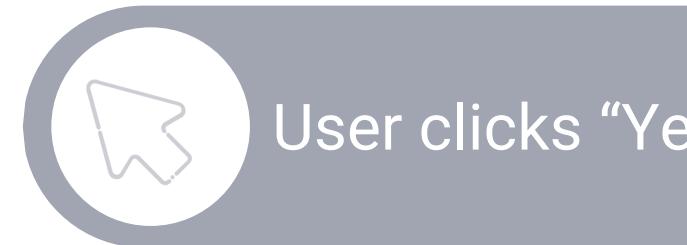
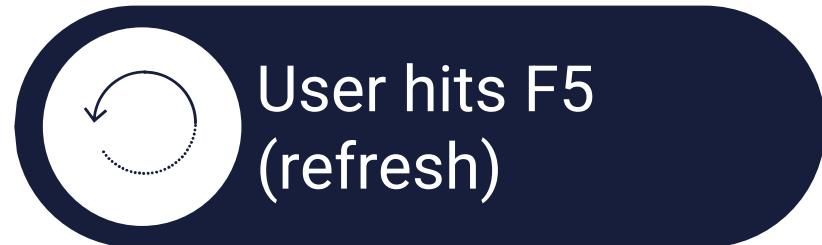


The double submit problem

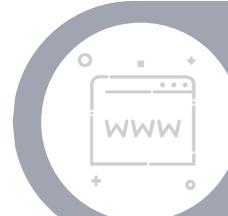


The double submit problem

Server forwards to success.jsp

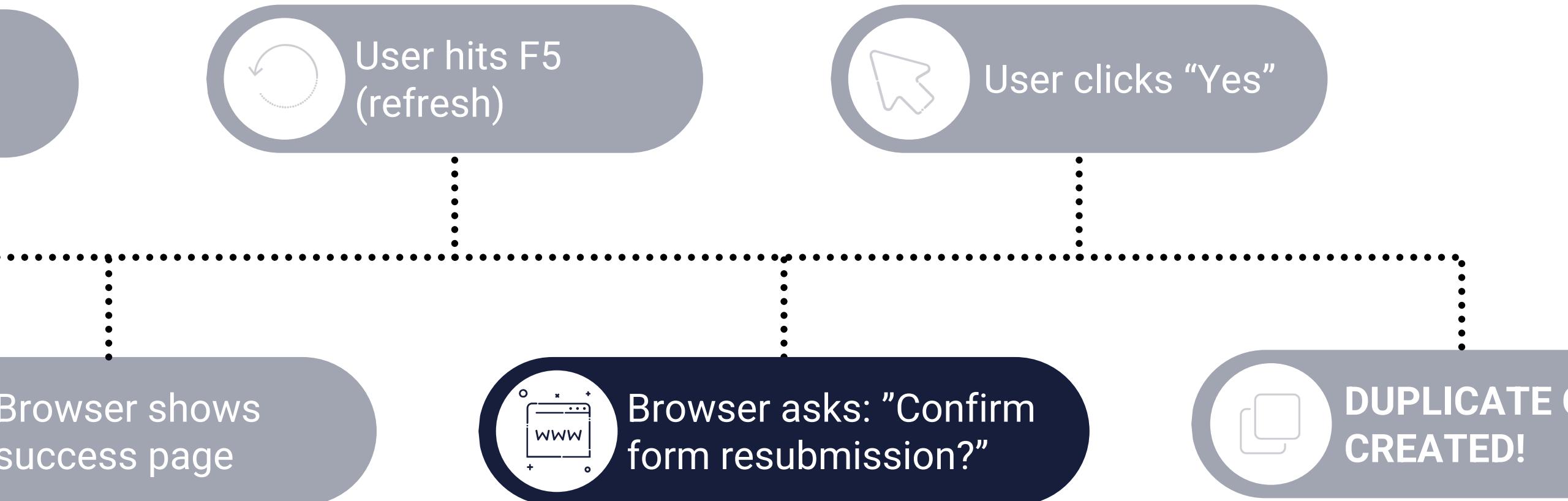


Browser shows success page

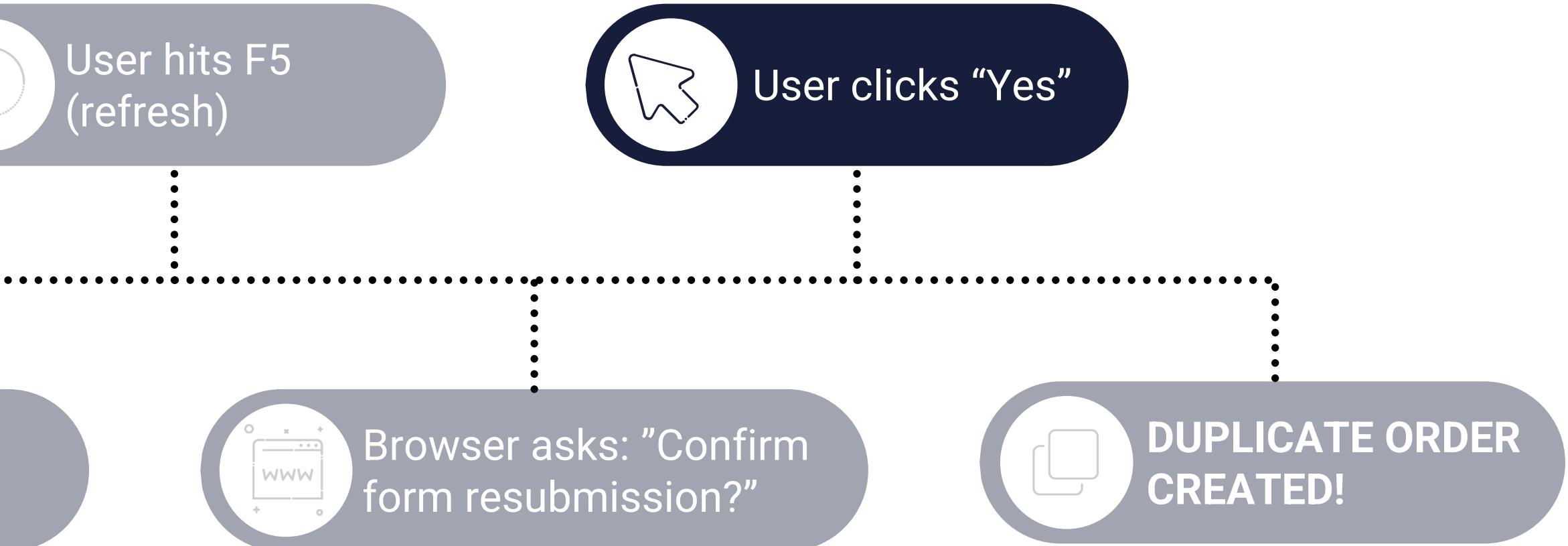


Browser asks: "Confirm form resubmission?"

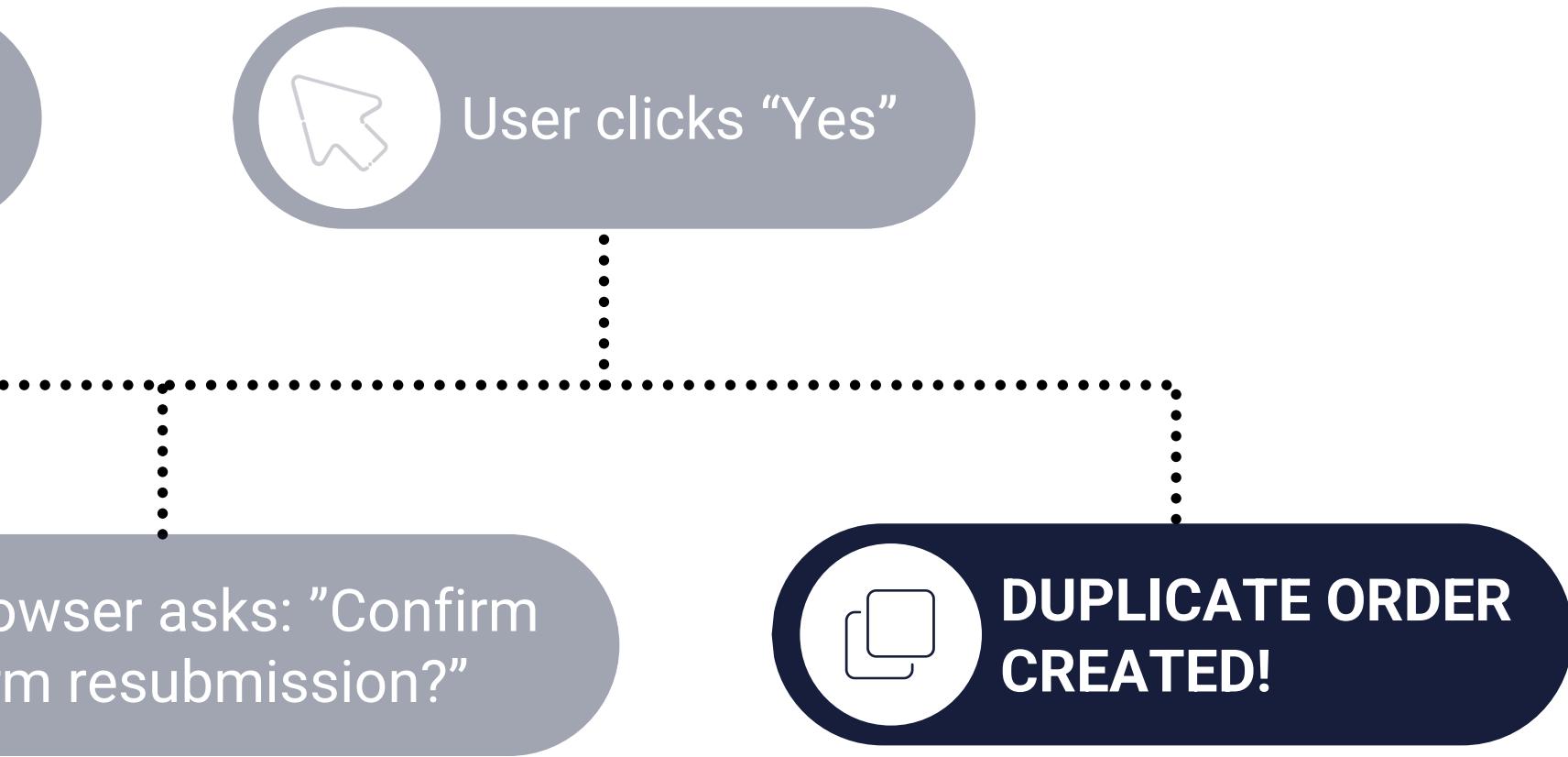
The double submit problem



The double submit problem



The double submit problem





Why this happens



Browser remembers last action was a POST

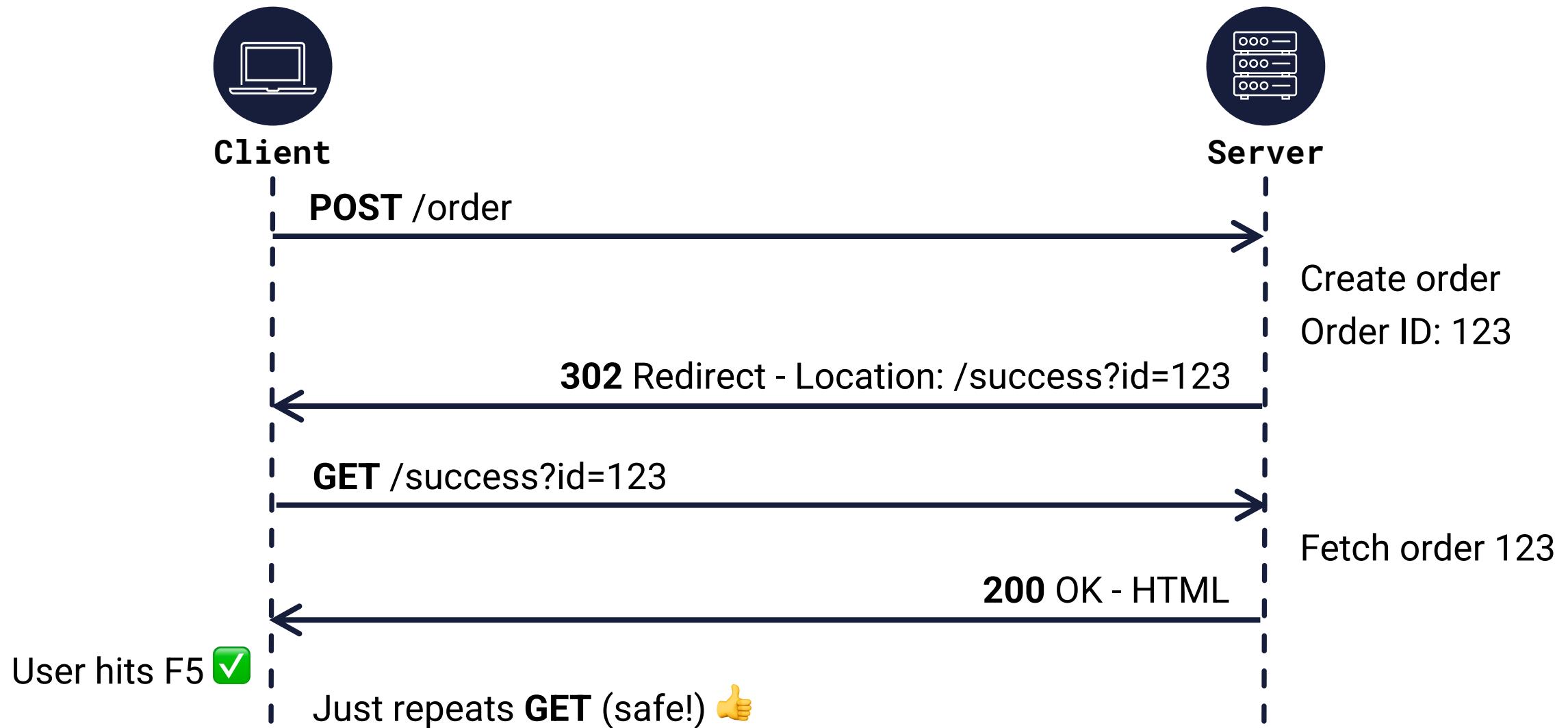


Refresh means “repeat last action”



Browser warns user, but can't prevent it

The solution: POST-Redirect-GET pattern



A photograph of a young man with dark, curly hair and a beard, wearing a light blue sweatshirt. He is sitting at a desk, looking down at a laptop screen. A pair of glasses rests on a book in front of him. The background shows a window and some plants.

PRG pattern

- POST does the work
- Redirect separates action from display
- GET shows result (safe to repeat)

PRG pattern: three clear steps



POST does the work

```
protected void doPost(...) {  
    // Extract and validate  
    String item = request.getParameter("item");  
  
    // THIS CHANGES THE DATABASE  
    Order order = orderService.createOrder(item);  
  
    // REDIRECT Don't render here!  
    response.sendRedirect("/success?orderId=" + order.getId());  
}
```



R

•
•
•

PRG pattern: three clear steps



Redirect separates

- HTTP 302/303 status code
- Browser automatically follows
- URL changes in address bar



GET shows result

```
protected void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
{  
    Long orderId = Long.parseLong(request.getParameter("id"));  
    Order order = orderService.findById(orderId);  
    // Forward to view  
}
```

```
m");  
item);  
  
= " + order.getId());
```

PRG pattern: three clear steps



GET shows result

```
protected void doGet(...) {  
    Long orderId = Long.parseLong(request.getParameter("orderId"));  
    Order order = orderService.getOrder(orderId);  
    // Forward to another page (read-only, safe)  
}
```

What actually gets sent

POST request

```
POST /order HTTP/1.1  
Host: localhost:8080  
Content-Type: application/x-www-form-urlencoded
```

item=burger&quantity=2

GET request (automatic)

```
GET /success?orderId=123 HTTP/1.1  
Host: localhost:8080
```

Browser automatically follows the redirect

Redirect response

```
HTTP/1.1 302 Found  
Location: /success?orderId=123  
Content-Length: 0
```

How the browser behaves

Aspect	Without PRG ✗	With PRG ✓
Address bar	Shows <code>/order</code> (POST URL)	Shows <code>/success?orderId=123</code> (GET URL)
Refresh (F5)	“Confirm form resubmission?”	Just re-fetches GET (safe)
Back button	Confusing behavior	Clean, predictable navigation
Bookmark	Can't bookmark POST	Shareable URL to success page
Browser history	Messy	Clean and logical

Demo 0, 1, 2

PRG



Exercise 0

Fix the double submit bug

Given:

Servlet that forwards after POST

Task:

Convert to PRG pattern

Time: 15 minutes



Common mistake: writing after redirect

```
response.sendRedirect("/success");
response.getWriter().println("Redirecting..."); // Won't work!
```

Why it fails

- `sendRedirect()` sets status 302 and Location header
- Browser immediately follows redirect
- Any output after is **ignored**
- May throw `IllegalStateException`



Correct code

```
response.sendRedirect("/success");  
// NO CODE AFTER sendRedirect!
```



Rule

After redirect, return immediately.

Common mistake: forgetting URL encoding

```
// What if item has spaces or special characters?  
response.sendRedirect("/success?item=" + item)
```

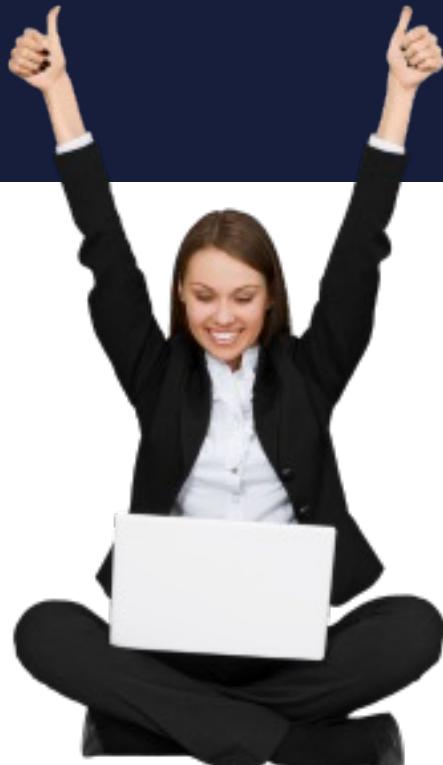
Problems

- **item = "John Doe"** >> Invalid URL (space)
- **item = "A&B"** >> Parsed as two parameters
- **item = "åäö"** >> Encoding errors



Correct code

```
String encodedItem = URLEncoder.encode(item, StandardCharsets.UTF_8);  
response.sendRedirect("/success?item=" + encodedItem);
```



Result

"John Doe" >> "John+Doe" (valid URL)

Common mistake: passing too much data

```
response.sendRedirect("/success?item=" + item +  
    "&quantity=" + quantity +  
    "&price=" + price +  
    "&customer=" + customer +  
    "&address=" + address + ...);  
// URL gets too long!
```



Problems

- URLs have length limits (2000 chars in some browsers)
- Hard to read and maintain
- Security concerns (sensitive data in URL)

Better approach

```
// Pass only ID, fetch details in GET handler  
response.sendRedirect("/success?orderId=" + order.getId())
```



Common mistake: forward vs redirect



Forward

```
request.getRequestDispatcher("/success")
    .forward(request, response);
```

- Happens server-side
- URL doesn't change
- Still has double-submit problem



Redirect

```
response.sendRedirect("/success");
```

- Browser makes new request
- URL changes
- Solves double submit problem

When **not** to use PRG



Search forms: already using GET



Filters and sorting: Already using GET



Read-only operations: Already using GET



Always use PRG for

Creating records (orders, users, posts)

Updating records (profile edits, settings)

Deleting records

Any operation that changes server state

PRG pattern summary



Prevents double submission

User can refresh safely



Clean browser history

Back button works predictably



Bookmarkable URLs

Success pages can be shared



Better user experience

No “Confirm resubmission” dialogs



Industry standard

Used by all major web frameworks



If POST changes data, always
redirect to GET





AJAX returning HTML instead of JSON

Modern enhancement without full JSON API

AJAX can return different things

Approach A: return JSON

```
fetch("/api/cart/add")
  .then((response) => response.json())
  .then((data) => {
    // JavaScript builds HTML
    element.innerHTML = `
      <div>${data.itemName}</div>
      <div>€${data.price}</div>
    `;
  });
});
```

Problems

- HTML duplicated (server JSP + JS)
- Styling duplicated (CSS classes)
- Hard to maintain
- More JavaScript code

AJAX can return different things

Approach B: return HTML

```
fetch("/cart/add")
  .then((response) => response.text())
  .then((html) => {
    // Just insert - server did the work!
    element.innerHTML = html;
  });
}
```

Benefits

- Server keeps control of HTML
- Reuse existing JSP templates
- Consistent styling
- Less JavaScript

Why return HTML from server?

- ✓ **Single source of truth**

- HTML structure defined once (in JSP)
- No duplication between server and client

- ✓ **Consistent styling: same look everywhere**

- CSS classes applied server-side

- ✓ **Easier maintenance**

- Change layout → Update only JSP (no need to update JS)

- ✓ **Less JavaScript code**

- Simple insertion, no templating logic
- Fewer bugs to debug

- ✓ **Works for both**

- Same endpoint serves traditional and AJAX requests
- Progressive enhancement



Demo 3

**Partial updates without
JSON**

A dark blue rectangular area containing a blurred screenshot of a computer screen. The screen shows several lines of code, likely PHP, with syntax highlighting. The code includes variables like \$attachment_id, \$attachment_url, and \$attachment_type, and functions like get_the_title() and get_the_content(). The background of the slide features a light gray diagonal grid pattern.

Exercise 1

Add AJAX enhancement

Given:

Working traditional form with PRG

Task:

Form that works with AND without
JavaScript

Time: 20 minutes



Best practice: progressive enhancement

Start with working HTML/forms, add AJAX on top

Step 1: HTML that works without JavaScript

```
<form action="/cart/add" method="post" id="cart-form">
  <input type="hidden" name="itemId" value="123" />
  <button type="submit">Add to Cart</button>
</form>
```

```
if
/
f
}
}
```

Best practice: progressive enhancement

Start with working HTML/forms, add AJAX on top

Script

Step 2: Enhance with JavaScript

```
if (window.fetch) {  
    // Check if AJAX available  
    form.addEventListener("submit", function (e) {  
        e.preventDefault(); // Stop normal submission  
        // Try AJAX...  
        // If fails, fall back: this.submit();  
    });  
}
```

Benefit

✓ Works

✓ Works

✓ Same s

Best practice: progressive enhancement

Start with working HTML/forms, add AJAX on top

Benefits

- ✓ Works without JavaScript (accessibility!)
- ✓ Works with JavaScript (better UX!)
- ✓ Same server code handles both

Behind the scenes: HTTP messages

Scenario 1: Traditional request (no JS)

POST /cart/add HTTP/1.1
(no X-Requested-With header)

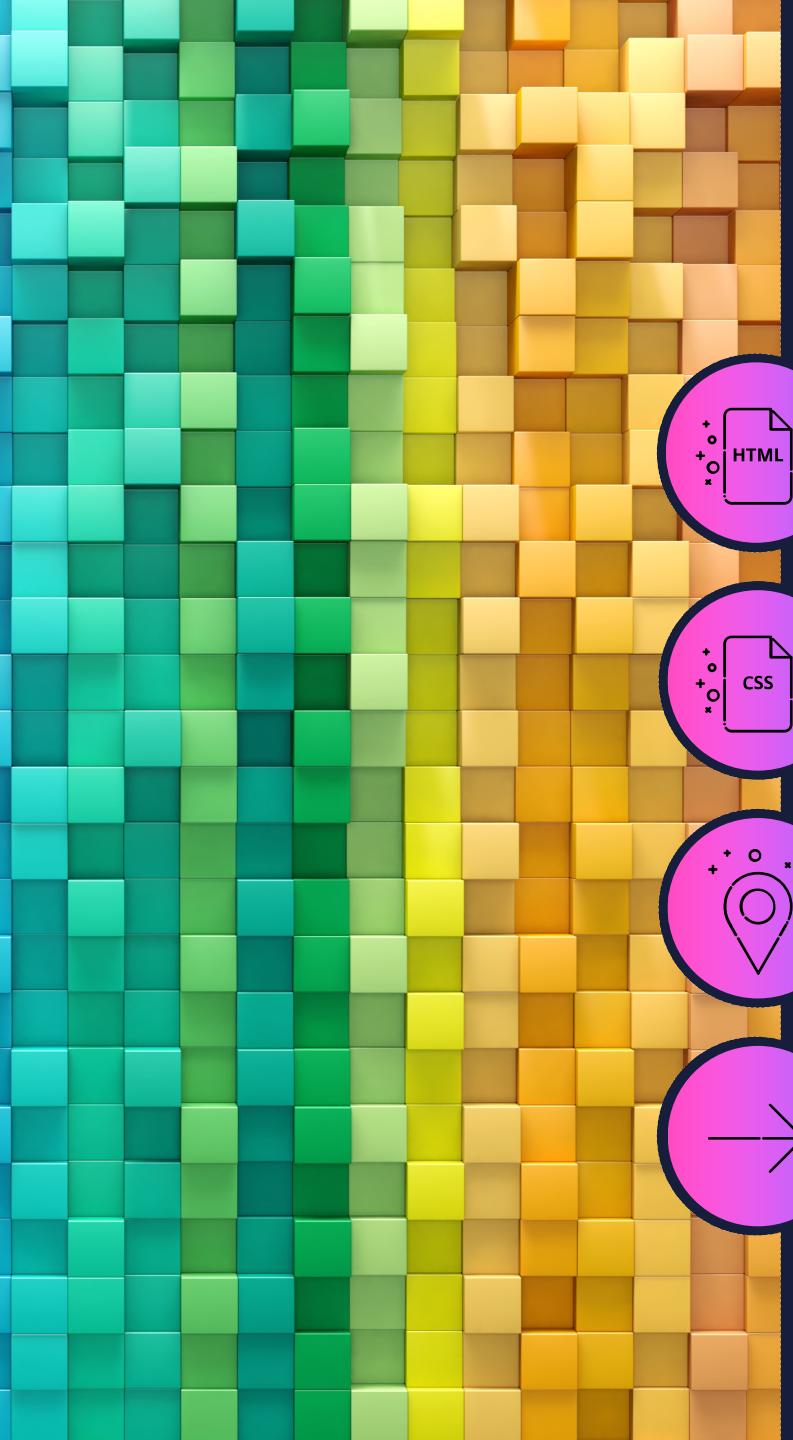
- > Server: Detects traditional request
- > Responds with 302 redirect
- > Browser follows to GET /cart
- > Full page loads

Scenario 2: AJAX requests

POST /cart/add HTTP/1.1
With: XMLHttpRequest

- > Server: Detects AJAX request
- > Responds with 200 and HTML fragment
- > JavaScript updates page
- > No navigation, no reload

Same endpoint, different behavior based on header



Benefits of reusing fragments



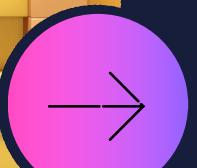
Same HTML structure everywhere



Same CSS styling



One place to maintain



Consistent behavior

AJAX returning HTML summary



Server keeps control

HTML rendering stays server-side, business logic in one place



Reuse JSP templates

Create fragments once, use for full page and AJAX updates



Progressive enhancement

Works without JavaScript, enhanced with JavaScript



Less complexity

Less JavaScript code, fewer bugs to debug



Consistent design

Same CSS, same structure - One source of truth



When to use AJAX with HTML:
Server-rendered apps that want smooth
partial updates



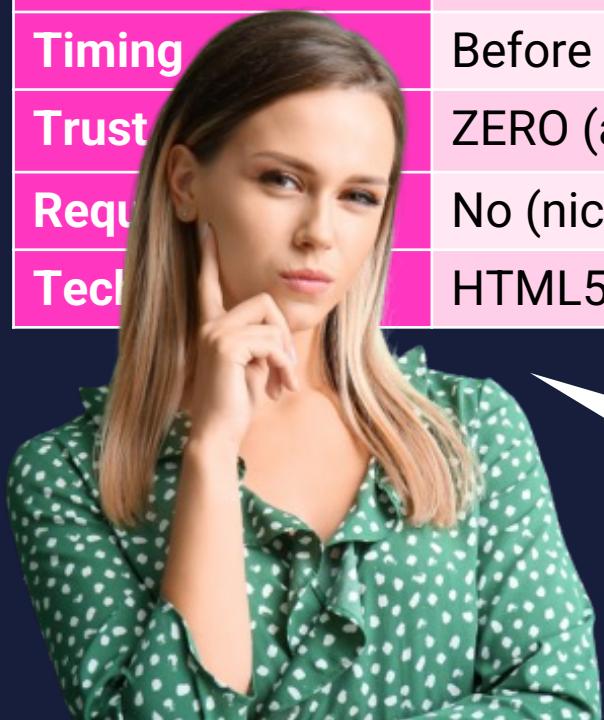
Form validation & error handling

Handling invalid input gracefully

Two types of validation

Client-side vs server-side validation

Aspect	Client-side	Server-side
Purpose	Quick feedback, better UX	Security, business rules
Timing	Before sending to server	After receiving request
Trust	ZERO (always bypass-able)	ABSOLUTE (only one that counts)
Requirements	No (nice to have)	YES (mandatory)
Technology	HTML5 + JavaScript	Java servlet code



NEVER trust client-side validation alone!

Client-side validation: HTML5 built-in



```
<form action="/order" method="post">
  <input type="text" name="name" required /> Required field

  <input type="email" name="email" required />

  <input type="number" name="quantity" min="1" max="100" required />

  <input
    type="text"
    name="zipcode"
    pattern="[0-9]{4}[A-Z]{2}"
    title="Dutch zipcode (e.g., 1234AB)"
  />

  <input type="text" name="username" minlength="3" maxlength="20" required />
  <button type="submit">Order</button>
</form>
```

Client-side validation: HTML5 built-in



```
<form action="/order" method="post">  
  <input type="text" name="name" required />  
  
  <input type="email" name="email" required />   <---- Email format  
  
  <input type="number" name="quantity" min="1" max="100" required />  
  
  <input  
    type="text"  
    name="zipcode"  
    pattern="[0-9]{4}[A-Z]{2}"  
    title="Dutch zipcode (e.g., 1234AB)"  
  />  
  
  <input type="text" name="username" minlength="3" maxlength="20" required />  
  <button type="submit">Order</button>  
</form>
```

Client-side validation: HTML5 built-in



```
<form action="/order" method="post">  
  <input type="text" name="name" required />  
  
  <input type="email" name="email" required />  
  
  <input type="number" name="quantity" min="1" max="100" required />  
  
  <input  
    type="text"  
    name="zipcode"  
    pattern="[0-9]{4}[A-Z]{2}"  
    title="Dutch zipcode (e.g., 1234AB)"  
  />  
  
  <input type="text" name="username" minlength="3" maxlength="20" required />  
  <button type="submit">Order</button>  
</form>
```

Number with range

Client-side validation: HTML5 built-in



```
<form action="/order" method="post">  
  <input type="text" name="name" required />  
  
  <input type="email" name="email" required />  
  
  <input type="number" name="quantity" min="1" max="100" required />  
  
  <input  
    type="text"  
    name="zipcode"  
    pattern="[0-9]{4}[A-Z]{2}"  
    title="Dutch zipcode (e.g., 1234AB)"  
  />  
  
  <input type="text" name="username" minlength="3" maxlength="20" required />  
  <button type="submit">Order</button>  
</form>
```

Pattern matching

Client-side validation: HTML5 built-in

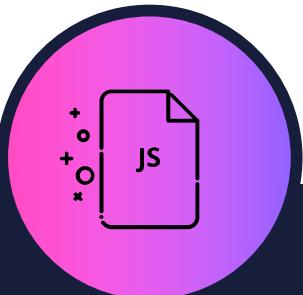


```
<form action="/order" method="post">  
  <input type="text" name="name" required />  
  
  <input type="email" name="email" required />  
  
  <input type="number" name="quantity" min="1" max="100" required />  
  
  <input  
    type="text"  
    name="zipcode"  
    pattern="[0-9]{4}[A-Z]{2}"  
    title="Dutch zipcode (e.g., 1234AB)"  
  />  
  
  <input type="text" name="username" minlength="3" maxlength="20" required />  
  <button type="submit">Order</button>  
</form>
```



Length constraints

Custom JavaScript validation



```
document.getElementById("order-form").addEventListener("submit", function (e) {
  const quantity = parseInt(this.querySelector('[name="quantity"]').value);
  const email = this.querySelector('[name="email"]').value;

  // Custom business rule
  if (quantity > 10) {
    e.preventDefault();
    alert("Maximum 10 items per order. " + "Contact sales for bulk orders.");
    return;
  }

  // Custom format check
  if (!email.includes("@company.com")) {
    e.preventDefault();
    alert("Please use your company email.");
    return;
  }
});
```

When to use custom JS validation



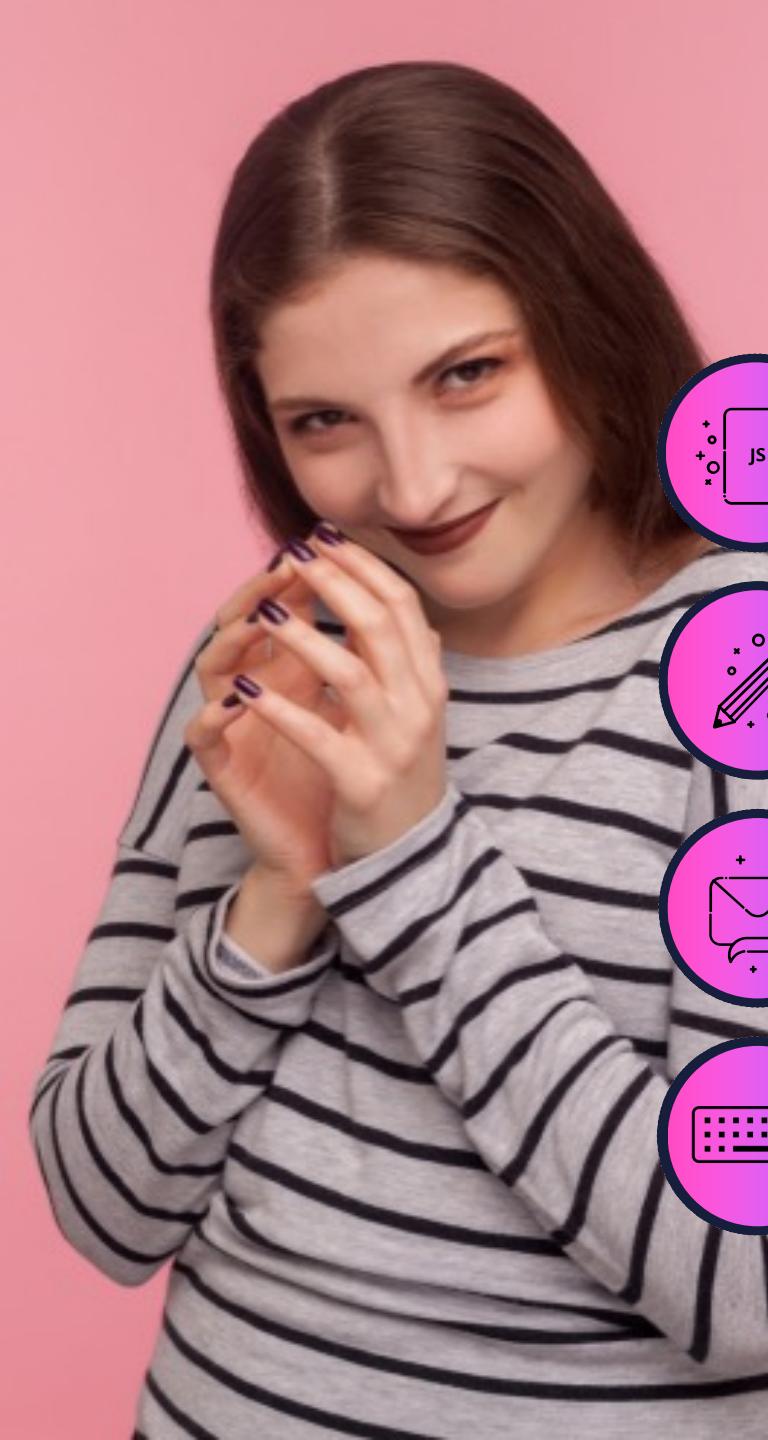
Business rules



Complex validation



Better error messages

A photograph of a young woman with long brown hair, smiling and resting her chin on her hand. She is wearing a grey and white horizontally striped long-sleeved shirt. The background is a solid pink color.

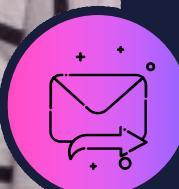
Why client-side is not enough



Users can disable JavaScript



Users can edit HTML in DevTools

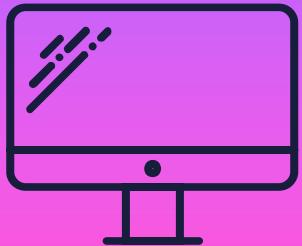


Users can send direct HTTP requests



Users can use browser console

Client-side vs server-side validation



Client-side

UX enhancement only



Server-side

Security requirement

Demo 4

Server-side validation

Collect all errors, not just the first one



Validation errors: three strategies

We'll examine each in detail



Approach A: **Direct render**

Forward directly to form

- ✓ Simple, fast
- ✗ Back button broken
- ✗ Refresh shows warning

Use for: Prototypes only



Approach B: **PRG + flash**

Redirect to form, errors in session

- ✓ Best UX
- ✓ No navigation issues
- ✗ More complex

Use for: Production apps



Approach C: **AJAX**

Return JSON, show errors inline

- ✓ Best UX
- ✓ No navigation issues
- ✗ Requires JavaScript

Use for: Modern apps

Approach A: forward directly



```
private void handleValidationErrors(...) {  
    // Pass errors and original input to JSP  
    request.setAttribute("errors", errors);  
    request.setAttribute("name", name);  
    request.setAttribute("email", email);  
  
    // Forward back to form  
    request.getRequestDispatcher("/WEB-INF/order-form.jsp")  
        .forward(request, response);  
}
```

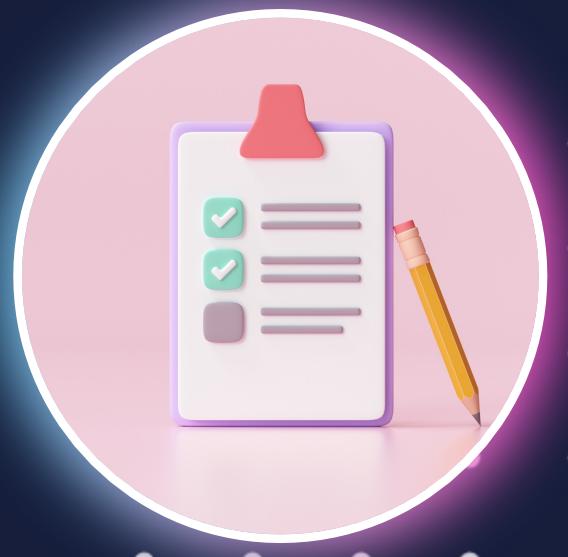
Approach A: forward directly

What happens:

1. User submits invalid form >> POST /order
2. Server validates, finds errors
3. Forwards to order-form.jsp
4. Browser URL still shows “/order”
5. User hits F5 >> “Confirm resubmission?”

Problems:

- ✗ Browser thinks last action was POST
- ✗ Refresh attempts resubmission
- ✗ Back button confusion



Approach B: RPG + Flash attributes (production ready)

Step 1: Store in session and Redirect



```
private void handleValidationErrors(...) {  
    HttpSession session = request.getSession();  
  
    // Store errors temporarily in session  
    session.setAttribute("flash_errors", errors);  
  
    // Store form data to repopulate  
    Map<String, String> formData = new HashMap<>();  
    formData.put("name", name != null ? name : "");  
    formData.put("email", email != null ? email : "");  
    formData.put("quantity", quantity != null ? quantity : "");  
    session.setAttribute("flashFormData", formData);  
  
    // Redirect back to form page  
    response.sendRedirect("/order-form");  
}
```

Approach B: RPG + Flash attributes (production ready)

Step 2: Display form with flash data



```
@WebServlet("/order-form")
public class OrderFormServlet extends HttpServlet {

    protected void doGet(...) {
        HttpSession session = request.getSession();

        // Retrieve flash data
        List<String> errors =
            (List<String>) session.getAttribute("flash_errors");
        Map<String, String> formData =
            (Map<String, String>) session.getAttribute("flash_formData");

        // CRITICAL: Remove immediately (flash pattern!)
        session.removeAttribute("flash_errors");
        session.removeAttribute("flashFormData");
```

```
// Pass to JSP
if (errors != null) {
    request.setAttribute("errors", errors);
}
if (formData != null) {
    request.setAttribute("formData", formData);
}

request.getRequestDispatcher("/WEB-INF/order-form.jsp")
    .forward(request, response);
}
```



What is flash?



Data stored in session **temporarily**



Retrieved once and **immediately deleted**



Won't reappear on subsequent refreshes

Complete flow of the “flash” pattern

1. POST /order (invalid data)
2. Store errors in session
3. Redirect to GET /order-form
4. GET /order-form
5. Retrieve flash data from session
6. DELETE flash data from session immediately
7. Show form with errors
8. User refreshes >> GET /order-form again
9. No flash data anymore >> Clean form



Why this works: Data consumed on first display only

Approach B: benefits



Clean user experience:

- **Submit invalid form:**
 - See errors with their input preserved
- **Hit F5 (refresh):**
 - No “Resubmit form” warning
 - No errors shows (flash consumed)
 - Clean form displayed
- **Hit back button**
 - Goes to previous page
 - Clean navigation history
- **Hit forward button**
 - Returns to form
 - No weird behavior

Comparison

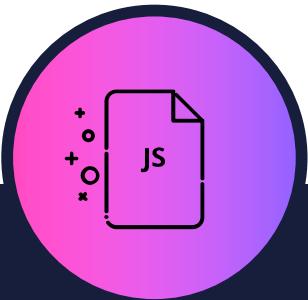
Feature	Approach A (direct render)	Approach B (PRG + flash)
Refresh	Warning	Clean
Back button	Broken	Works
History	Messy	Clean

Approach C: AJAX validation

Client-side

```
form.addEventListener("submit", function (e) {
  e.preventDefault(); // Always prevent default

  fetch("/order", {
    method: "POST",
    body: new FormData(this),
  })
    .then((response) => {
      if (response.status === 400) {
        // Validation errors
        return response.json().then((data) => {
          displayValidationErrors(data.errors);
        });
      }
      return response.json();
    })
    .then((data) => {
      // Success
      window.location.href = "/success?orderId=" + data.orderId;
    });
});
```



```
function displayValidationErrors(errors) {  
  // Show errors next to fields  
  errors.forEach((error) => {  
    const field = document.querySelector(`[name="${error.field}"]`);  
    // Add error message next to field  
  });  
}
```

Approach C: Server returns JSON errors



```
@WebServlet("/order")
public class OrderServlet extends HttpServlet {

    protected void doPost(...) {
        // Detect AJAX
        boolean isAjax = "XMLHttpRequest".equals(
            request.getHeader("X-Requested-With"))
    }

    List<ValidationError> errors = validateOrder(request);

    if (!errors.isEmpty()) {
        if (isAjax) {
            // Return JSON for AJAX
            response.setStatus(400);
            response.setContentType("application/json");
        }
    }
}
```

```
        // Build JSON with errors
        // { "errors": [ {"field": "email", "message": "..."}, ...] }
    } else {
        // Traditional - use PRG with flash
        handleValidationErrors(...);
    }
    return;
}

// Process order...
}
}
```



Same endpoint handles both AJAX and traditional

Which approach to use?

Criteria	Approach A: Direct render	Approach B: PRG + flash	Approach C: AJAX validation
Complexity	Simple	Medium	Complex
Back button	Broken	Clean	Perfect
Refresh	Warning	Clean	Perfect
Accessibility	Works	Works	Needs JS
User experience	Poor	Good	Excellent
Production ready?	No	Yes	Yes

Recommendations



Internal tools/prototypes >> Approach A (direct render)



Public-facing traditional apps >> Approach B (PRG + flash)



Modern SPAs >> Approach C (AJAX validation)



Best of both worlds >> Implement B (PRG + flash), enhance with C (AJAX validation)

Preserving user input



Always preserve user input on errors

Why it matters:

- Users spent time filling out form
- Don't make them do it again!

Rule: Always pre-fill form fields with user's previous input

Exercise 2

Implement flash pattern

Given:

Direct rendering of validation errors

Task:

Convert to PRG with flash attributes

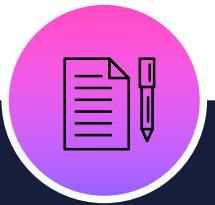




Error handling in production

Handling things when they go wrong

Five types of errors



Validation errors **400**

- User input doesn't meet requirements
- User's responsibility
- Show friendly error, allow correction



Business logic errors **409/422**

- Input valid but business rules prevent action
- E.g., out of stock, insufficient balance
- Explain why, suggest alternatives



Auth errors **401/403**

- Not logged in or no permission
- Redirect to login or show access denied

Five types of errors



Not found

404

- Resource doesn't exist
- Show helpful 404 page



Server errors

500

- Something broke server-side
- Generic message, log details

Validation errors: quick recap

What we already learned:

- Client-side validation (UX)
- Server-side validation (security)
- Three approaches to display errors:
 - Direct render
 - PRG + flash
 - AJAX



Validation errors: key principles



Collect ALL errors, not just first



Preserve user input



Show friendly, specific messages



Return HTTP 400 status code

Business logic error example

Scenario:

- User tries to order 10 items, but only 3 available

This is **NOT** a validation error:

- Input format is correct
- Business rule prevents the action



How to handle out of stock scenario

-  Check stock level after validation
-  If insufficient, create friendly error message
-  Redirect back with error (PRG pattern)
-  Show available quantity
-  Let user adjust and retry
-  **HTTP status code:** 409 (conflict) or 422 (unprocessable entity)

Handling out of stock (traditional)



```
protected void doPost(...) {  
    String productId = request.getParameter("productId");  
    int quantity = Integer.parseInt(request.getParameter("quantity"));  
  
    // Validation passed, check business rules  
    Product product = productService.getProduct(productId);  
  
    if (!product isInStock(quantity)) {  
        // Business logic error  
        HttpSession session = request.getSession();  
        session.setAttribute("flash_error",  
            "Sorry, only " + product.getStockLevel() +  
            " items available. Please reduce quantity.");  
        session.setAttribute("flash_requestedQuantity", quantity);  
    }  
}
```

```
// Redirect back to product page
response.sendRedirect("/product/" + productId);
return;
}

// Proceed with order
orderService.createOrder(productId, quantity);
response.sendRedirect("/success");
}
```

Business logic errors with AJAX

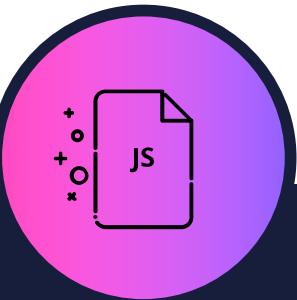
Server response



```
if (isAjax) {  
    if (!product isInStock(quantity)) {  
        response.setStatus(409); // Conflict  
        response.setContentType("application/json");  
  
        PrintWriter out = response.getWriter();  
        out.println("{");  
        out.println("  \"error\": \"Product out of stock\",");  
        out.println("  \"availableQuantity\": " + product.getStockLevel());  
        out.println("  \"message\": \"Only " + product.getStockLevel() +  
                  " items available\"");  
        out.println("}");  
        return;  
    }  
}
```

Business logic errors with AJAX

Client handling



```
.then(response => {
  if (response.status === 409) {
    return response.json().then(data => {
      showError(data.message);
      // Update max quantity in form
      document.querySelector(' [name="quantity"] ').max =
        data.availableQuantity;
    }) ;
  }
  // ...
})
```

Handling server errors: global configuration

web.xml configuration



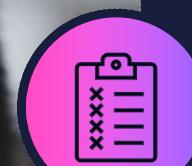
```
<error-page>
    <error-code>500</error-code>
    <location>/WEB-INF/error-pages/500.jsp</location>
</error-page>
```

```
<error-page>
    <error-code>404</error-code>
    <location>/WEB-INF/error-pages/404.jsp</location>
</error-page>
```

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/WEB-INF/error-pages/500.jsp</location>
</error-page>
```



Purpose of global config



Catch all unhandled exceptions



Show user-friendly error page



Hide technical details



Log for debugging



User-friendly error page

Friendly, apologetic tone

Clear next steps

No technical jargon

Details only in development

Demo 5

Servlet-level error handling



Exercise 3

Servlet-level error handling

Time: 30 minutes





Error handling best practices

Golden rules

Never expose technical details



"NullPointerException at OrderService.java:142"



“Something went wrong. Please try again.”

Always log with context



```
logger.error("Order failed. User: {}, Product: {}, Qty: {}",  
    userId, productId, quantity, exception)
```

Use correct HTTP status codes

400: Validation error

401: Not authenticated

403: Not authorized

404: Not found

409: Business logic conflict

500: Server error



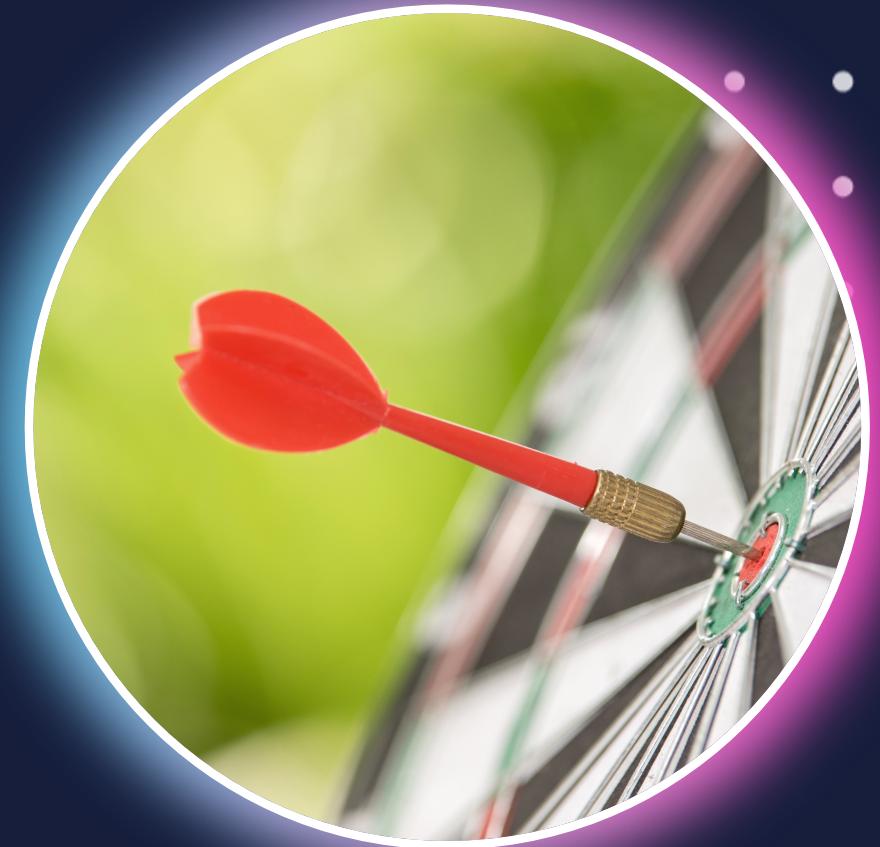


Preserve user input

Don't make the user re-type everything!

Differentiate error types

Different errors need different handling



Provide helpful context

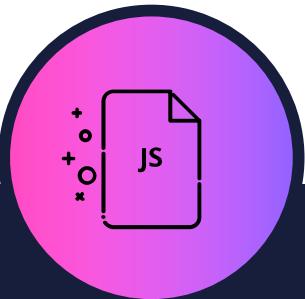
- ✗ "Error occurred"
- ✓ "Unable to process payment. Please check your card details."

- ✗ "Invalid input"
- ✓ "Email address must contain an @ symbol"

- ✗ "Operation failed"
- ✓ "This product is out of stock. Expected back in 3 days."



Have fallbacks for AJAX



```
.catch(error => {
  // If AJAX completely fails
  if (error.message === 'Failed to fetch') {
    // Fall back to traditional form submission
    formElement.submit();
  }
}) ;
```



Test error scenarios



Network failures



Invalid input



Session expiry



Database down



Out of stock



Next up:

Kaasus time!

What would you do?



Scenario 1: The impatient user

User submits order form, waits
30 seconds, sees nothing,
submits again.

**How would you prevent
duplicate orders?**

What would you do?

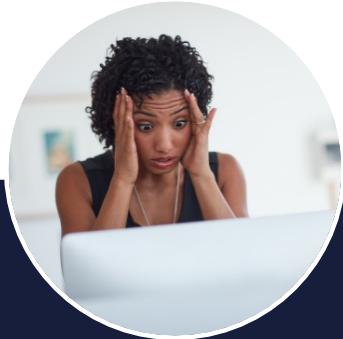


Scenario 2: The multi-tab user

User opens product page in 3 tabs, adds to cart in all 3.

Should cart show 1 item
or 3?

What would you do?



Scenario 3: The browser crash

Browser crashes after POST but
before redirect.

**Order created, but user
never saw confirmation.
Now what?**

What would you do?



Scenario 4: The network failure

Payment processed but
response never reaches user.

User tries again. How to
avoid double charge?

What would you do?



Scenario 5: The session expiry

User fills out long form, session expires, submits.

How to preserve their
work?



Questions or suggestions?

maaike.vanputten@brightboost.nl

See you next time! 💕