

Client Actions in React Router v7 Framework Mode

1. Understanding Client Actions

What are Client Actions?

Client Actions in React Router v7 are functions that handle user interactions and mutations on the client side. They process form submissions, handle data mutations, and manage side effects when users interact with your application. Think of them as event handlers that can perform complex operations asynchronously.

Aviation Analogy: Client Actions are like the pilot's pre-flight checklist procedures. When a pilot needs to take off (user interaction), they follow a series of standardized steps to ensure everything is ready (data validation, processing, and submission).

2. Server Actions

Understanding Server Actions

Server Actions are backend functions that Client Actions can call. They handle actual data mutations, database operations, and business logic on the server. Client Actions can trigger these Server Actions to perform work that requires server-side processing.

Aviation Analogy: Server Actions are like the airport's air traffic control center. While the pilot (client) initiates the action, the control center (server) handles the complex coordination, databases, and critical operations that keep the system safe and functional.

Key Characteristics:

- Executed on the server
- Handle database operations, authentication, and business logic
- Marked with "use server" directive in the action file
- Return data that flows back to the client
- Provide security by keeping sensitive logic server-side

Example - Server Action:

```
// actions/booking.server.ts
```

```
"use server"
```

```
export async function createFlightBooking(formData: FormData) {  
  const passengerId = formData.get("passengerId")  
  const flightNumber = formData.get("flightNumber")  
  const seatClass = formData.get("seatClass")  
  
  try {  
    // Database operation - creating booking in server  
    const booking = await db.bookings.create({  
      passengerId,  
      flightNumber,  
      seatClass,  
      bookingDate: new Date(),  
      status: "confirmed"  
    })  
  
    return {  
      success: true,  
      bookingId: booking.id,  
      confirmationNumber: booking.confirmationNumber  
    }  
  } catch (error) {  
    return {  
      success: false,  
      error: error.message  
    }  
  }  
}
```

3. Calling Actions

Overview

Calling Actions refers to triggering server-side or client-side actions from your React components. React Router v7 provides several methods to call actions, each suited for different scenarios.

Aviation Analogy: Calling actions is like a pilot communicating with air traffic control using different methods - radio communication (forms), direct request systems (useSubmit), or auxiliary systems (fetcher). Each method serves a specific purpose in the aviation workflow.

Methods to Call Actions

There are three primary methods to call actions in React Router v7:

1. **Calling actions with Forms** - Standard HTML form submission
 2. **Calling actions with useSubmit** - Programmatic submission
 3. **Calling actions with fetcher** - Non-navigation data fetching and mutations
-

4. Calling Actions with a Form

How Forms Work

When you use HTML `<Form>` component from React Router, submitting the form automatically calls the action associated with the route. The form data is collected, validated, and sent to the action.

Aviation Analogy: Using a Form is like filing a standard flight plan through official channels. You fill out all required information on the official form (Form component), submit it to the right authority (the action), and the system automatically processes it according to established procedures.

Characteristics:

- Automatic form data collection
- Browser-native form submission behavior
- Standard request/response cycle
- URL changes (navigation)
- Best for traditional form submissions

Detailed Example - Booking Form:

```
// routes/booking.tsx
```

```
import { Form, useActionData } from "react-router-dom"
```

```
import { createFlightBooking } from "../booking.server"
```

```
export async function action({ request }: ActionFunctionArgs) {  
  if (request.method !== "POST") {  
    return null  
  }  
}
```

```
const formData = await request.formData()  
const result = await createFlightBooking(formData)
```

```
if (!result.success) {  
  return { error: result.error }  
}
```

```
return result  
}
```

```
export default function BookingPage() {  
  const actionData = useActionData()
```

```
  return (  
    <div className="booking-container">  
      <h1> ✈️ Flight Booking System </h1>
```

```
      {actionData?.error && (  
        <div className="error-banner">  
          ⚠️ Booking Failed: {actionData.error}  
        </div>  
      )}
```

```
      {actionData?.success && (  
        <div className="success-banner">  
          ✅ Booking Confirmed!  
          <p>Confirmation Number: {actionData.confirmationNumber}</p>  
          <p>Booking ID: {actionData.bookingId}</p>  
        </div>  
      )}
```


```
      <Form method="post" className="booking-form">
```

```
<fieldset>
  <legend>Passenger Information</legend>
```

```
<label htmlFor="passengerId">
  Passenger ID:
  <input
    type="text"
    id="passengerId"
    name="passengerId"
    required
    placeholder="Enter your passenger ID"
  />
</label>
```

```
<label htmlFor="flightNumber">
  Flight Number:
  <input
    type="text"
    id="flightNumber"
    name="flightNumber"
    required
    placeholder="e.g., AA-101"
  />
</label>
```

```
<label htmlFor="seatClass">
  Seat Class:
  <select id="seatClass" name="seatClass" required>
    <option value="">Select seat class</option>
    <option value="economy">Economy</option>
    <option value="business">Business</option>
    <option value="first">First Class</option>
  </select>
</label>
</fieldset>
```

```
<button type="submit" className="submit-btn">
   Complete Booking
</button>
```

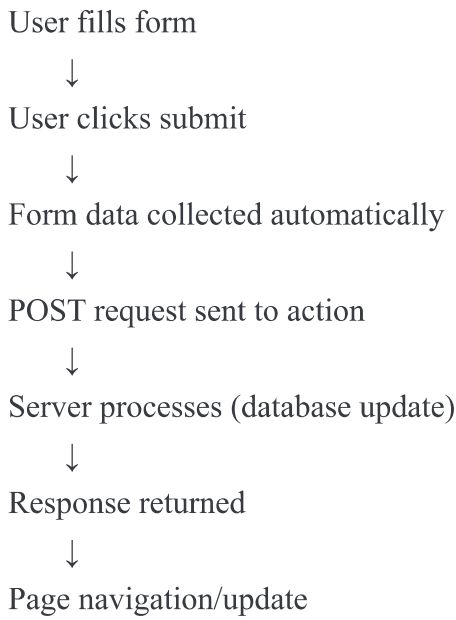
```
</Form>
```

```

<div className="form-explanation">
  <h3>How This Works (Form Method)</h3>
  <p>
    When you click "Complete Booking," the form data is automatically collected
    and sent to the server action. This is like filing a flight plan through
    official channels - the entire process follows a predictable path.
  </p>
</div>
</div>
)
}

```

Form Method - Visual Flow:



5. Calling Actions with useSubmit

How useSubmit Works

The `useSubmit` hook allows you to programmatically submit forms without relying on standard form submission. This gives you more control over when and how actions are called, enabling custom logic, validation, or timing.

Aviation Analogy: Using `useSubmit` is like a pilot having a direct communication line with air traffic control. Instead of following the standard flight plan procedures, the pilot can make immediate requests based on real-time conditions and make decisions programmatically.

Characteristics:

- Programmatic control over submission
- Custom validation before submission
- Can submit without a form element
- Conditional submission logic
- Better for dynamic or complex workflows
- No automatic URL navigation

Detailed Example - Real-Time Booking with Validation:

```
// routes/advanced-booking.tsx
```

```
import { Form, useActionData, useNavigation } from "react-router-dom"
import { useSubmit } from "react-router-dom"
import { useState } from "react"
```

```
export async function action({ request }: ActionFunctionArgs) {
  if (request.method !== "POST") {
    return null
  }
```

```
  const formData = await request.formData()
  const passengerId = formData.get("passengerId") as string
  const flightNumber = formData.get("flightNumber") as string
```

```
// Server-side validation
```

```
if (!passengerId.match(/^P\d{6}$/)) {
  return { error: "Invalid passenger ID format (must be P followed by 6 digits)" }
}
```

```
// Check if flight exists and has available seats
```

```
const flight = await db.flights.findOne({ flightNumber })
if (!flight) {
  return { error: "Flight not found" }
}
```

```
if (flight.availableSeats === 0) {
  return { error: "No available seats on this flight" }
}
```

```
const booking = await db.bookings.create({
  passengerId,
  flightNumber,
  seatClass: formData.get("seatClass"),
  bookingDate: new Date(),
  status: "confirmed"
})
```

```
return {
  success: true,
  bookingId: booking.id,
  message: "Booking confirmed successfully"
}
```



```
}  
}
```

```
export default function AdvancedBookingPage() {
```

```
  const submit = useSubmit()
```

```
  const navigation = useNavigation()
```

```
  const actionData = useActionData()
```

```
  const [formData, setFormData] = useState({
```

```
    passengerId: "",
```

```
    flightNumber: "",
```

```
    seatClass: "economy"
```

```
  })
```

```
  const [clientErrors, setClientErrors] = useState<string[]>([])
```

```
  const isSubmitting = navigation.state === "submitting"
```

```
  // Client-side validation
```

```
  const validateForm = () => {
```

```
    const errors: string[] = []
```

```
    if (!formData.passengerId.trim()) {
```

```
      errors.push("Passenger ID is required")
```

```
    } else if (!formData.passengerId.match(/^P\d{6}$/)) {
```

```
      errors.push("Passenger ID must be P followed by 6 digits (e.g., P123456)")
```

```
    }
```

```
    if (!formData.flightNumber.trim()) {
```

```
      errors.push("Flight number is required")
```

```
    } else if (!formData.flightNumber.match(/^([A-Z]{2})-\d{3,4}$/)) {
```

```
      errors.push("Flight number format should be XX-### (e.g., AA-101)")
```

```
    }
```

```
    return errors
```

```
  }
```

```
  const handleInputChange = (
```

```
    e: React.ChangeEvent<HTMLInputElement | HTMLSelectElement>
```

```
  ) => {
```

```
    const { name, value } = e.target
```

```
    setFormData(prev => ({ ...prev, [name]: value })))
```

```

    setClientErrors([]) // Clear errors on input
  }

const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault()

  // Client-side validation before submission
  const errors = validateForm()
  if (errors.length > 0) {
    setClientErrors(errors)
    return
  }

  // Create FormData and submit
  const fd = new FormData()
  fd.append("passengerId", formData.passengerId)
  fd.append("flightNumber", formData.flightNumber)
  fd.append("seatClass", formData.seatClass)

  submit(fd, { method: "post" })
}

return (
  <div className="advanced-booking-container">
    <h1>  Advanced Flight Booking</h1>

    {clientErrors.length > 0 && (
      <div className="error-box">
        <h3>  Please fix these errors:</h3>
        <ul>
          {clientErrors.map((error, idx) => (
            <li key={idx}>{error}</li>
          ))}
        </ul>
      </div>
    )}

    {actionData?.error && (
      <div className="server-error">
        <h3>  Server Error:</h3>
        <p>{actionData.error}</p>
      </div>
    )}
  </div>
)

```

```
</div>
```

```
)}
```

```
{actionData?.success && (
```

```
<div className="success-box">
```

```
<h3> ✓ Success!</h3>
```

```
<p>{actionData.message}</p>
```

```
<p>Booking ID: {actionData.bookingId}</p>
```

```
</div>
```

```
)}
```

```
<form onSubmit={handleSubmit} className="booking-form">
```

```
<label htmlFor="passengerId">
```

```
Passenger ID (format: P123456):
```

```
<input
```

```
type="text"
```

```
id="passengerId"
```

```
name="passengerId"
```

```
value={formData.passengerId}
```

```
onChange={handleInputChange}
```

```
placeholder="P123456"
```

```
disabled={isSubmitting}
```

```
</label>
```

```
<label htmlFor="flightNumber">
```

```
Flight Number (format: AA-101):
```

```
<input
```

```
type="text"
```

```
id="flightNumber"
```

```
name="flightNumber"
```

```
value={formData.flightNumber}
```

```
onChange={handleInputChange}
```

```
placeholder="AA-101"
```

```
disabled={isSubmitting}
```

```
</label>
```

```
<label htmlFor="seatClass">
```

```
Seat Class:
```

```
<select
```

```

    id="seatClass"
    name="seatClass"
    value={formData.seatClass}
    onChange={handleInputChange}
    disabled={isSubmitting}
  >
    <option value="economy">Economy</option>
    <option value="business">Business</option>
    <option value="first">First Class</option>
  </select>
</label>

<button type="submit" disabled={isSubmitting}>
  {isSubmitting ? "🔄 Processing..." : "✈️ Book Flight"}
</button>
</form>

```

```

<div className="explanation">

```

```

  <h3>How This Works (useSubmit Method)</h3>

```

```

  <p>

```

This implementation uses useSubmit to give us control over when and how the booking is submitted. Think of it as a pilot having direct control over communications - we validate information locally first (like a pre-flight checklist), then submit to the server when everything is correct.

```

  </p>

```

```

  <ul>

```

```

    <li>Client-side validation happens before submission</li>

```

```

    <li>User gets immediate feedback on errors</li>

```

```

    <li>Server can still validate and reject requests</li>

```

```

    <li>Better UX with custom loading states</li>

```

```

  </ul>

```

```

</div>

```

```

</div>

```

```

)

```

```

}

```

useSubmit Method - Visual Flow:

User triggers action (button click, etc.)



Custom JavaScript logic executes



Client-side validation



Form data prepared programmatically



submit() called with data



POST request sent to action



Server processes



Response returned (no automatic navigation)

6. Calling Actions with a Fetcher

How Fetcher Works

A fetcher allows you to call actions without navigating away from the current page. It's perfect for isolated mutations like liking a post, updating a status, or performing actions without a full page reload.

Aviation Analogy: Using a fetcher is like cabin crew using an auxiliary communication system while the plane is already in flight. They can make requests to ground control, get responses, and update internal systems without the pilot changing the flight path or navigating to a different destination.

Characteristics:

- No page navigation
- Non-blocking data fetching
- Ideal for isolated mutations
- Can be used multiple times on a page
- Great for real-time updates
- Multiple fetchers can coexist
- State is managed independently

Detailed Example - Seat Selection with Fetcher:

// routes/seat-selection.tsx

```
import { useFetcher, useLoaderData } from "react-router-dom"
import { useState } from "react"

export async function loader() {
  const flight = await db.flights.findOne({ flightNumber: "AA-101" })
  const bookedSeats = await db.bookings.find(
    { flightNumber: "AA-101" },
    { projection: { seatNumber: 1 } }
  )

  return {
    flight: {
      flightNumber: flight.flightNumber,
      departure: flight.departure,
      destination: flight.destination,
      aircraft: flight.aircraft,
      totalSeats: flight.totalSeats
    },
    bookedSeats: bookedSeats.map(b => b.seatNumber)
  }
}

export async function action({ request }: ActionFunctionArgs) {
  if (request.method !== "POST") {
    return null
  }

  const formData = await request.formData()
  const action = formData.get("_action")

  if (action === "reserveSeat") {
    const seatNumber = formData.get("seatNumber") as string
    const passengerId = formData.get("passengerId") as string

    // Check if seat is already booked
    const existingBooking = await db.bookings.findOne({
      flightNumber: "AA-101",
      seatNumber
    })
  }
}
```

```
if (existingBooking) {  
  return {  
    success: false,  
    error: "Seat is already booked",  
    seatNumber  
  }  
}
```

// Reserve the seat

```
await db.seatReservations.create({  
  passengerId,  
  flightNumber: "AA-101",  
  seatNumber,  
  reservedAt: new Date()  
})
```

```
return {  
  success: true,  
  message: `Seat ${seatNumber} reserved successfully`,  
  seatNumber  
}  
}
```

```
return { success: false, error: "Unknown action" }  
}
```

```
export default function SeatSelectionPage() {  
  const { flight, bookedSeats } = useLoaderData()  
  const fetcher = useFetcher()  
  const [selectedSeat, setSelectedSeat] = useState(null)  
  const [passengerId, setPassengerId] = useState("")
```

// Generate seat grid (typical aircraft layout)

```
const rows = 20  
const columns = ["A", "B", "C", "D", "E", "F"]  
const seats = []
```

```
for (let i = 1; i <= rows; i++) {  
  for (const col of columns) {  
    seats.push(`${i}${col}`)  
  }  
}
```

```
}
```

```
const isSubmitting = fetcher.state === "submitting"
```

```
const lastSubmission = fetcher.data
```

```
return (
```

```
<div className="seat-selection-container">
```

```
<h1> 🛫 Select Your Seat</h1>
```

```
<div className="flight-info">
```

```
<h2> {flight.flightNumber}</h2>
```

```
<p>
```

```
{flight.departure} → {flight.destination}
```

```
</p>
```

```
<p>Aircraft: {flight.aircraft}</p>
```

```
</div>
```

```
<div className="seat-info">
```

```
<p>Enter your Passenger ID:</p>
```

```
<input
```

```
type="text"
```

```
value={passengerId}
```

```
onChange={e => setPassengerId(e.target.value)}
```

```
placeholder="P123456"
```

```
disabled={isSubmitting}
```

```
/>
```

```
</div>
```

```
<div className="seat-grid">
```

```
<div className="legend">
```

```
<span className="seat available">
```

```
<span className="dot"></span> Available
```

```
</span>
```

```
<span className="seat booked">
```

```
<span className="dot"></span> Booked
```

```
</span>
```

```
<span className="seat selected">
```

```
<span className="dot"></span> Selected
```

```
</span>
```

```
</div>
```



```

<div className="seats">
  {seats.map(seat => {
    const isBooked = bookedSeats.includes(seat)
    const isSelected = selectedSeat === seat
    const isFailed =
      lastSubmission?.error && lastSubmission?.seatNumber === seat

    return (
      <button
        key={seat}
        className={`seat-button ${
          isBooked ? "booked" : "available"
        } ${
          isSelected ? "selected" : ""
        } ${
          isFailed ? "failed" : ""
        }`}
        onClick={() => {
          if (!isBooked) {
            setSelectedSeat(seat)
          }
        }}
        disabled={isBooked || isSubmitting}
      >
        {seat}
      </button>
    )
  })}
</div>
</div>

```

```

{selectedSeat && (
  <div className="booking-section">
    <h3>Confirm Seat Selection</h3>
    <p>Selected Seat: <strong>{selectedSeat}</strong></p>

    <fetcher.Form method="post">
      <input type="hidden" name="_action" value="reserveSeat" />
      <input type="hidden" name="seatNumber" value={selectedSeat} />
      <input type="hidden" name="passengerId" value={passengerId} />

      <button
        type="submit"

```

```

disabled={
  isSubmitting ||
  !passengerId ||
  !selectedSeat
}
>
{isSubmitting ? "🔄 Reserving..." : "✅ Reserve Seat"}
</button>
</fetcher.Form>

```

```

{lastSubmission?.success && (
  <div className="success-message">
    ✅ {lastSubmission.message}
  </div>
)}

```

```

{lastSubmission?.error && (
  <div className="error-message">
    ❌ {lastSubmission.error}
  </div>
)}
</div>
)}

```

```

<div className="explanation">
  <h3>How This Works (Fetcher Method)</h3>
  <p>

```

This example uses a fetcher to handle seat reservations without navigating away from the page. Imagine it as cabin crew handling passenger requests via an auxiliary system while the flight continues normally.

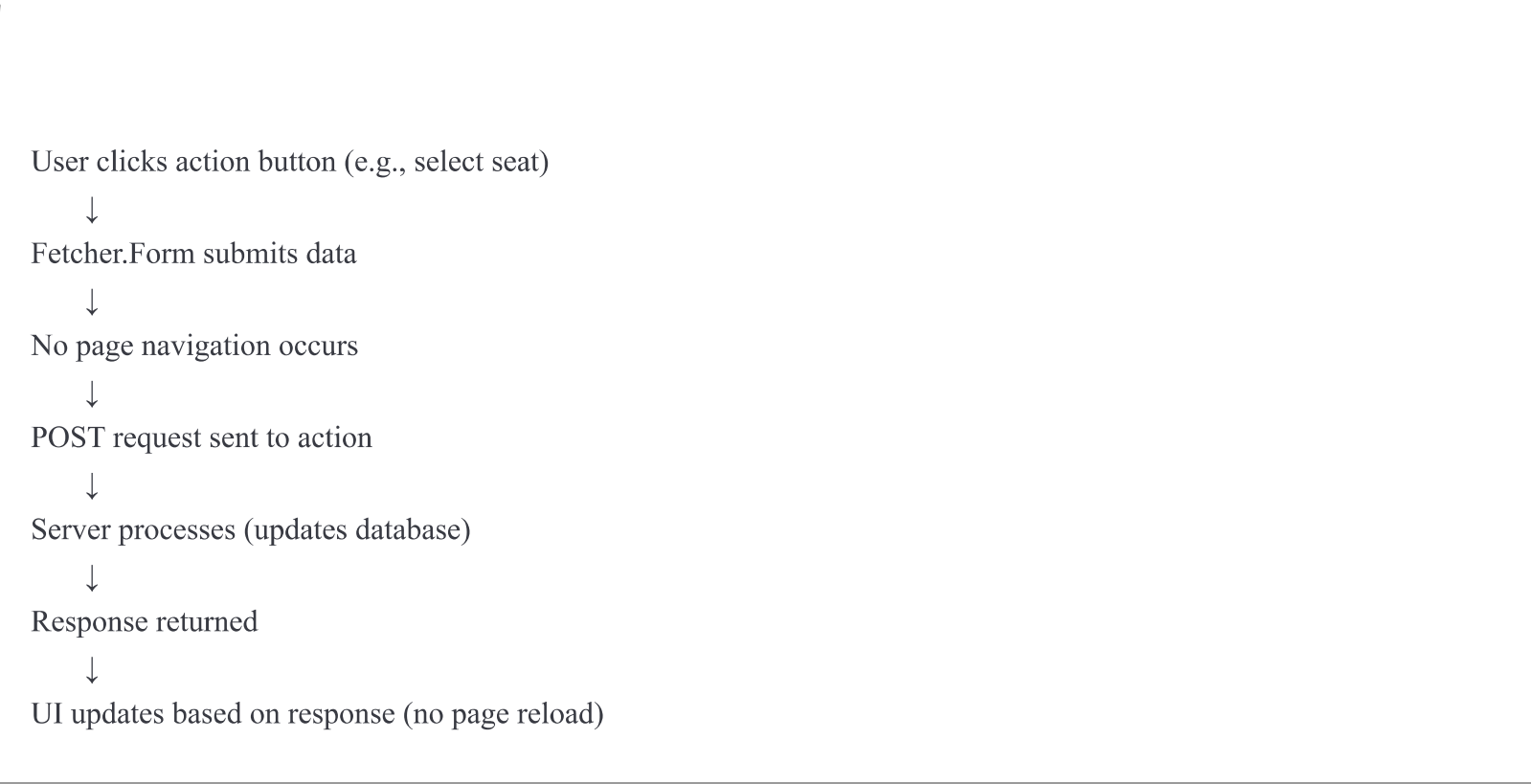
```

</p>
<ul>
  <li>Each seat button click triggers a fetcher submission</li>
  <li>The page doesn't reload or navigate</li>
  <li>Response data updates the UI immediately</li>
  <li>Multiple fetchers can work simultaneously</li>
  <li>State is managed within the fetcher</li>
</ul>
</div>
</div>

```

```
)  
}
```

Fetcher Method - Visual Flow:



7. Comparison Table: Three Methods

Aspect	Form	useSubmit	Fetcher
Navigation	Yes (full page)	Yes (full page)	No
Use Case	Traditional forms	Complex flows	Isolated mutations
Validation	Server-side	Client + server	Server-side
Data Collection	Automatic	Programmatic	Programmatic
User Experience	Page reload	Full nav	Seamless
Multiple Actions	One per form	Sequential	Multiple simultaneous
Real-time Updates	No	No	Yes
State Management	Browser history	Browser history	Internal to fetcher

8. Aviation Analogy Summary

Understanding the three methods through aviation:

Form Method (Standard Flight Plan): Like a commercial airline filing a standard flight plan. All procedures are predetermined, data is collected on standardized forms, submitted through official channels, and the process is predictable.

useSubmit Method (Direct Pilot Control): Like a military pilot who has direct communication with air traffic control. They can validate conditions, make decisions on the fly, and communicate with the command center programmatically

rather than following rigid procedures.

Fetcher Method (Auxiliary Communication): Like cabin crew using internal communication systems while the plane is flying. They can request services, update systems, and handle passenger needs without affecting the flight path. Multiple crew members can communicate simultaneously without disrupting the main flight operations.

Key Takeaways

- 1. **Server Actions** execute on the server and handle sensitive operations
 - 2. **Forms** are best for traditional page-based submissions with navigation
 - 3. **useSubmit** gives you programmatic control for complex validation and workflows
 - 4. **Fetchers** enable non-navigational mutations perfect for real-time updates
 - 5. Choose the method based on your UX requirements and workflow complexity
 - 6. Always validate on both client and server for security and reliability
-

Practice Exercises

- 1. Create a Form-based booking system with error handling
- 2. Build a useSubmit example with client-side validation
- 3. Implement a fetcher for rating/liking functionality
- 4. Combine multiple fetchers on one page
- 5. Create a complex multi-step booking flow using useSubmit
- 6. Implement optimistic UI updates with fetcher