Next.js 15+ Learning Guide: RSC, Server Actions & UI Diffs

📚 Complete Learning Resource

This document explains the three core concepts demonstrated in the EduShop app.

Quick Concepts Overview 🎯

This project demonstrates **9 core Next.js concepts** working together:

#	Concept	What It Does	Where	Priority
1️⃣	**React Server Components (RSC)**	Components that run on server, fetch data securely	`src/lib/products.ts`	⭐⭐⭐ Essential
2️⃣	**Server Actions**	Mutations without API routes, callable from client	`src/lib/actions.ts`	⭐⭐⭐ Essential
3️⃣	**Client Components**	Browser-based React with hooks and events	`src/components/SearchBar.tsx`	⭐⭐⭐ Essential
4️⃣	**UI Diffs & State**	React efficiently updates UI based on state changes	`src/components/SearchBar.tsx`	⭐⭐ Important
5️⃣	**Suspense & Streaming**	Progressive rendering, components load independently	`src/app/page.tsx`	⭐⭐ Important

#	Concept	What It Does	Where	Priority
6️⃣	**App Router**	File-based routing, nested layouts, route groups	`src/app/` folder	⭐⭐ Important
7️⃣	**TypeScript**	Full type safety across client-server boundary	`src/types/`	⭐ Nice to have
8️⃣	**Tailwind CSS**	Utility-first styling framework	Throughout	⭐ Nice to have
9️⃣	**Next.js Features**	Metadata API, image optimization, code splitting	Throughout	⭐ Nice to have

The Three Core Concepts (Deep Dive)

📖 **Read these sections in this order:**

1. **Section 0: Foundations** - How server/client code separation works
2. **Section 1: RSC** - Server-side data fetching
3. **Section 2: Server Actions** - Server mutations from client
4. **Section 3: UI Diffs & Streaming** - Reactive UI updates
5. **Integration** - How they work together
6. **Architecture Overview** - Visual diagrams and decision trees

How This Guide Flows

The first three sections build your mental model of how Next.js separates concerns. Then, Sections 4-6 show you the **production patterns** you'll use every day: handling errors, capturing forms, and validating data. Finally, we tie everything together with architecture diagrams and decision trees.

0. Foundations: How Next.js Separates Server & Client Code 🏗️

The Key Concept

In Next.js, **server code and React code are NOT completely separate** — they're **mixed in the same folder but distinguished by directives**. This is the revolutionary part of modern Next.js!

```
src/
├── lib/
│   ├── products.ts        ← Server only (no directive)
│   └── actions.ts         ← "use server" directive
└── components/
    ├── SearchBar.tsx      ← "use client" directive
    └── ProductCard.tsx    ← Server by default
```

The Directive System

Next.js uses **simple text directives** to tell the compiler where code runs:

Directive	Where	DB Access?	Browser JS?	Use For
None (default)	Server	✅ Yes	❌ No	Data fetching, Server Components
`"use server"`	Server	✅ Yes	❌ No	Mutations, Server Actions
`"use client"`	Browser	❌ No	✅ Yes	UI state, events, interactivity

Real Examples from Your EduShop

Server Component (default - no directive):

```
// src/lib/products.ts - runs on server only
export async function getAllProducts(): Promise<Product[]> {
  return PRODUCTS; // Safe to access any data
}
```

Server Action (`"use server"`):

```
// src/lib/actions.ts
"use server";

export async function searchProducts(query: string) {
  const allProducts = await getAllProducts(); // Can call server functions!
  return allProducts.filter(/*...*/);
}
```

Client Component (`"use client"`):

```
// src/components/SearchBar.tsx
"use client";

export function SearchBar() {
  const [query, setQuery] = useState(""); // React hooks here!
  const results = await searchProducts(query); // Calls server action
  return <input onChange={handleSearch} />; // Browser events
}
```

How the `.next` Folder Works

When you run `npm run build` :

1. **Next.js reads all your source files** in `src/`
2. **Analyzes directives** to determine what runs where
3. **Creates two separate bundles:**
   - **Server bundle** (`.next/server/`) - all functions marked `"use server"` + RSC functions
   - **Client bundle** (`.next/static/`) - only code with `"use client"`
4. **Secrets stay safe** - anything server-side never reaches the browser

Quick Mental Model

Think of it like:

- `src/` **= Your source code** (mixed server & client)
- `.next/server/` **= Server JavaScript** (runs on your Node.js server)
- `.next/static/` **= Client JavaScript** (runs in user's browser)
- **The directive (** `"use server"` **,** `"use client"` **) = The router** that decides where each file goes

1. React Server Components (RSC) 🖥️

What Are They?

React Server Components are React components that **run exclusively on the server**. They're the default in Next.js App Router.

```
// This is a Server Component (default)
export default async function ProductList() {
  const products = await fetch('database...'); // Secure DB query
  return <div>{/* render products */}</div>;
}
```

Key Characteristics

Feature	Description
Execution	Runs only on the server, never in the browser
JavaScript	Zero JavaScript sent to browser for this component
Database Access	Can directly query databases
Secrets	API keys, tokens never exposed to client
Bundle Size	Reduces client-side JavaScript significantly
Async/Await	Can use async operations directly
No Hooks	Cannot use useState, useEffect, useContext, etc.

Real Examples in EduShop

File: `src/lib/products.ts`

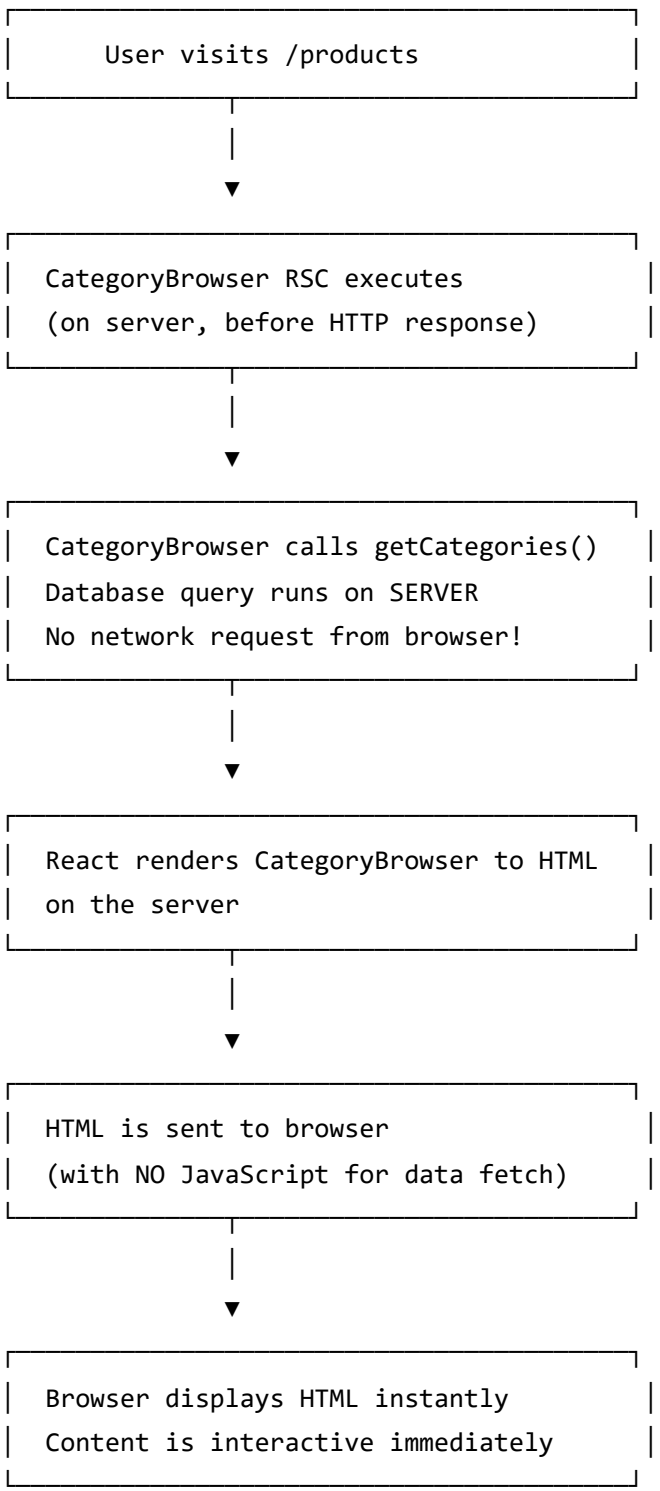
```
// RSC function - runs on server only
export async function getAllProducts(): Promise<Product[]> {
  await delay(500); // Simulate database query
  return PRODUCTS; // Directly return data
}
```

File: `src/components/CategoryBrowser.tsx`

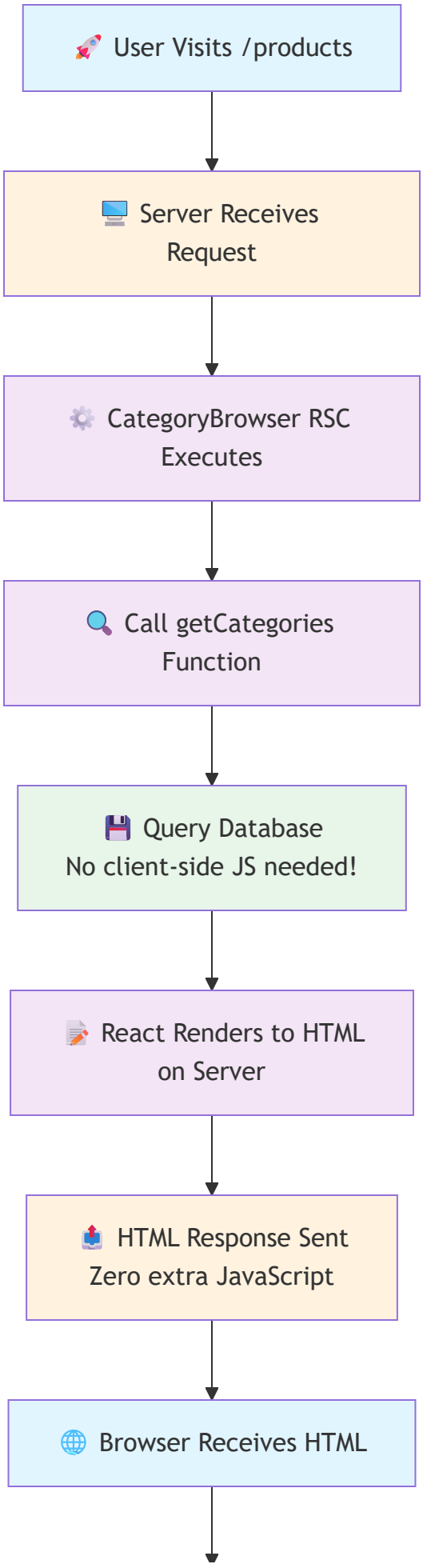
```tsx
// RSC component - no "use client" needed
export async function CategoryBrowser() {
  // This is a Server Component
  const categories = await getCategories(); // Runs on server

  return (
    <div>
      {categories.map(category => (
        <CategorySection key={category} categoryName={category} />
      ))}
    </div>
  );
}

// Nested RSC
async function CategorySection({ categoryName }: { categoryName: string }) {
  const products = await getProductsByCategory(categoryName); // Server DB query

  return (
    <section>
      {products.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </section>
  );
}
```

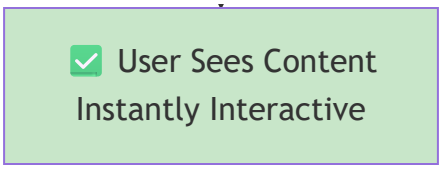
Data Flow

Text Diagram



Architecture Diagram (Mermaid)


🚀 User Visits /products
🖥️ Server Receives Request
⚙️ CategoryBrowser RSC Executes
🔍 Call getCategories Function
💾 Query Database
No client-side JS needed!
📝 React Renders to HTML on Server
📤 HTML Response Sent
Zero extra JavaScript
🌐 Browser Receives HTML



Benefits Summarized

1. **Security**: Database queries never exposed to client
2. **Performance**: Data fetching doesn't require client-side JavaScript
3. **Bundle Size**: Less code shipped to browser
4. **Always Fresh**: Each page request fetches latest data
5. **SEO Friendly**: All content rendered to HTML server-side

Key Takeaway: RSC is Your Data Layer

Think of RSC as a **secure, zero-JavaScript data layer**. Users see content instantly because data fetching happens server-side before HTML is sent. This is the foundation everything else builds on.

2. Server Actions ⚡

What Are They?

Server Actions are asynchronous functions that run on the server and can be called from Client Components. They're marked with the `"use server"` directive.

```
"use server"; // Marks this as server code

export async function addToCart(productId: string, quantity: number) {
  // This function runs on the server when called from client
  const product = await getProductById(productId);
  // Validate, update database, etc.
  return { success: true, product };
}
```

Key Characteristics

Feature	Description
Directive	Marked with `"use server"` at file/function level
Callable From	Client Components, Server Components, forms
Security	Runs on server, secrets stay safe
No API	No need to create `/api/` routes
Serialization	Automatic data serialization
Type Safety	Full TypeScript support
Form Integration	Works with HTML `<form>` action prop

Real Examples in EduShop

File: `src/lib/actions.ts`

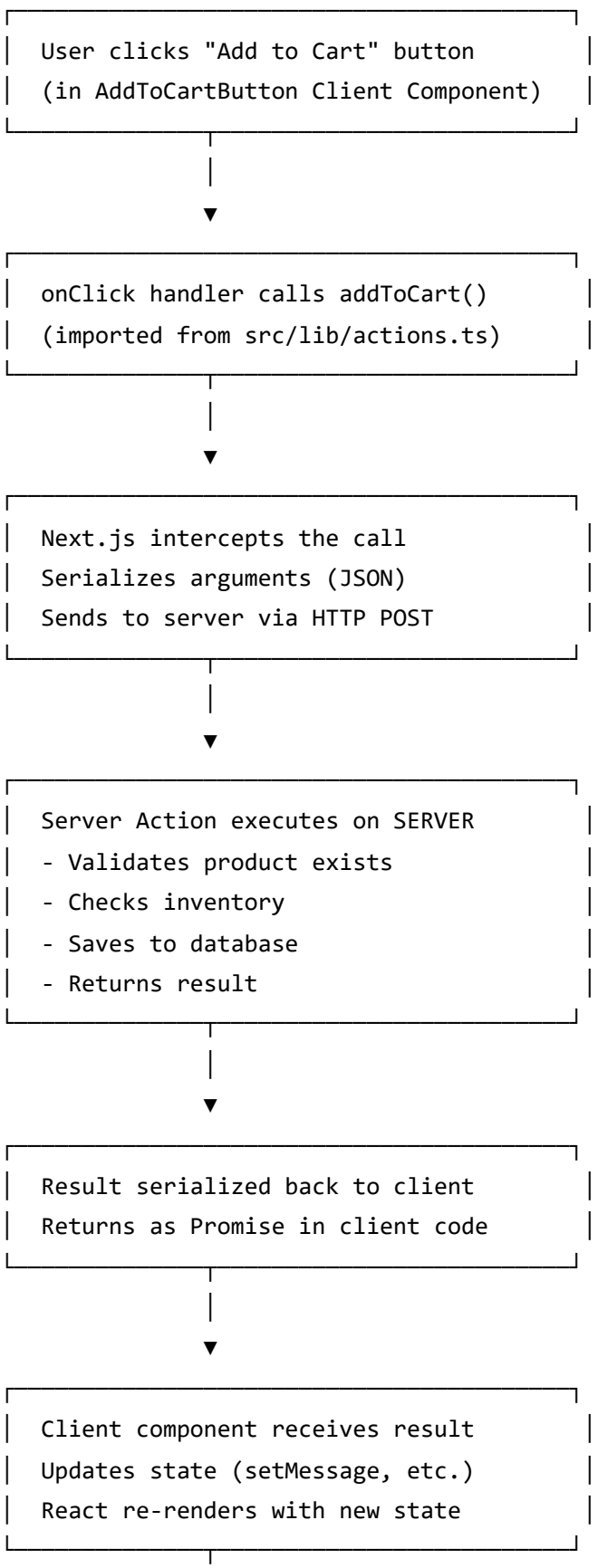
```tsx
"use server"; // All functions in this file are Server Actions

// Server Action 1: Search products
export async function searchProducts(query: string): Promise<Product[]> {
  await new Promise((resolve) => setTimeout(resolve, 300)); // Simulate DB delay

  const allProducts = await getAllProducts();

  const searchTerm = query.toLowerCase();
  return allProducts.filter(
    (product) =>
      product.name.toLowerCase().includes(searchTerm) ||
      product.description.toLowerCase().includes(searchTerm)
  );
}

// Server Action 2: Add to cart (with validation)
export async function addToCart(productId: string, quantity: number) {
  const allProducts = await getAllProducts();
  const product = allProducts.find((p) => p.id === productId);

  if (!product) {
    throw new Error("Product not found");
  }

  if (!product.inStock) {
    throw new Error("Product is out of stock");
  }

  // In real app: Save to database here
  return {
    success: true,
    message: `Added ${quantity} x ${product.name} to cart`,
    product,
    quantity,
  };
}
```

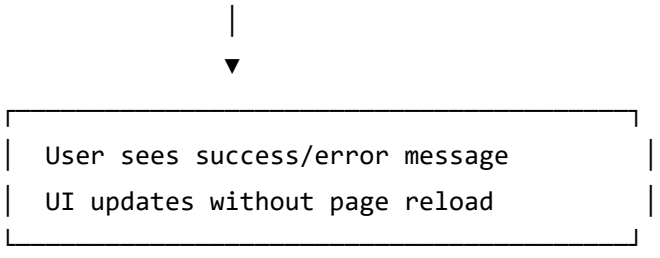
File: `src/components/AddToCartButton.tsx`

```tsx
"use client"; // This is a Client Component

import { addToCart } from "@/lib/actions"; // Import Server Action
import { useState } from "react";

export function AddToCartButton({ productId }: { productId: string }) {
  const [quantity, setQuantity] = useState(1);
  const [loading, setLoading] = useState(false);
  const [message, setMessage] = useState("");

  async function handleAddToCart() {
    setLoading(true);

    try {
      // Call Server Action directly from client
      const result = await addToCart(productId, quantity);
      setMessage(result.message);
    } catch (error) {
      setMessage("Error: " + error.message);
    } finally {
      setLoading(false);
    }
  }

  return (
    <div>
      <input
        type="number"
        value={quantity}
        onChange={(e) => setQuantity(parseInt(e.target.value))}
      />
      <button onClick={handleAddToCart} disabled={loading}>
        {loading ? "Adding..." : "Add to Cart"}
      </button>
      {message && <p>{message}</p>}
    </div>
  );
}
```

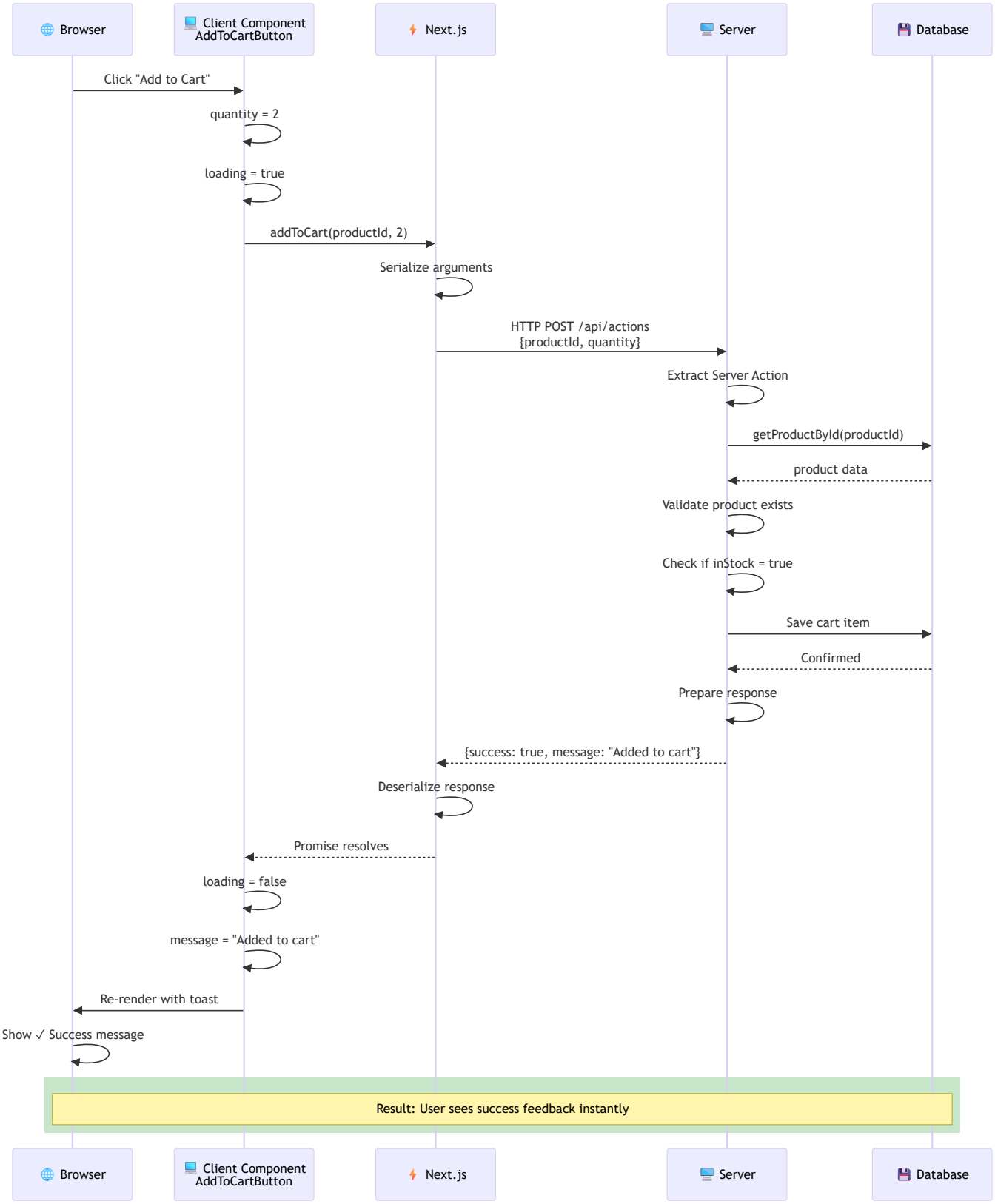
Data Flow

Text Diagram




User sees success/error message
UI updates without page reload

Sequence Diagram (Mermaid)



Using Server Actions with Forms

Server Actions shine with HTML forms:

```tsx
"use client";

import { submitContactForm } from "@/lib/actions";

export function ContactForm() {
  return (
    <form action={submitContactForm}>
      <input name="email" type="email" required />
      <input name="message" type="text" required />
      <button type="submit">Send Message</button>
    </form>
  );
}
```

Server Action:

```tsx
"use server";

export async function submitContactForm(formData: FormData) {
  const email = formData.get("email");
  const message = formData.get("message");

  // Save to database, send email, etc.

  // Revalidate page to show new message
  revalidatePath("/contact");
}
```

Benefits Summarized

1. **No API Routes**: Forget about creating `/api/` endpoints
2. **Type Safe**: Full TypeScript inference across client-server boundary
3. **Secure**: Can safely access databases and private APIs
4. **Automatic Serialization**: Complex objects work automatically
5. **Progressive Enhancement**: Works even if JavaScript fails

Key Takeaway: Server Actions are Your Mutation Layer

Server Actions let **client components safely modify data** without creating API routes. When a user interacts with your app, Server Actions handle the heavy lifting on the server side. Combined with

RSC, you now have both reading (RSC) and writing (Server Actions) covered.

3. UI Diffs & Streaming 🎨

What Are They?

UI Diffs refer to how React updates the UI when state changes. In Next.js, Streaming and Suspense allow different parts of a page to load and update independently.

Key Concepts

State-Based UI Changes

```
"use client";

import { useState } from "react";

export function SearchResults() {
  const [query, setQuery] = useState("");
  const [results, setResults] = useState([]);
  const [loading, setLoading] = useState(false);

  async function handleSearch(e) {
    const searchQuery = e.target.value;
    setQuery(searchQuery);

    if (!searchQuery) {
      setResults([]);
      return;
    }

    setLoading(true);
    const results = await searchProducts(searchQuery);
    setResults(results);
    setLoading(false);
  }

  // Different UI based on state
  return (
    <div>
      <input onChange={handleSearch} />

      {/* UI DIFF 1: Show loading while fetching */}
      {loading && <p>Searching...</p>}

      {/* UI DIFF 2: Show results when ready */}
      {results.length > 0 && (
        <div>
          Found {results.length} products
          {results.map(product => (
            <div key={product.id}>{product.name}</div>
          ))}
```

```
        </div>
      )}

      {/* UI DIFF 3: Show empty state */}
      {!loading && results.length === 0 && query && (
        <p>No products found for "{query}"</p>
      )}
    </div>
  );
}
```

Streaming with Suspense

Streaming allows parts of your page to render and be sent to the browser independently:

```
// src/app/page.tsx (Server Component)
import { Suspense } from "react";
import { CategoryBrowser } from "@/components/CategoryBrowser";

export default function Home() {
  return (
    <div>
      {/* This renders immediately */}
      <h1>Welcome to EduShop</h1>

      {/* This has a fallback while loading */}
      <Suspense fallback={<div>Loading categories...</div>}>
        <CategoryBrowser />
      </Suspense>
    </div>
  );
}
```

What happens:

1. Browser loads page
2. Gets HTML for <h1> immediately
3. Gets loading fallback: "Loading categories..."
4. Browser shows: "Welcome to EduShop" + "Loading categories..."
5. Server finishes fetching categories
6. Streams actual category HTML to browser
7. Browser replaces loading fallback with real content
8. User sees final page fully loaded

Real Example in EduShop

File: `src/app/page.tsx`

```tsx
import { Suspense } from "react";
import { CategoryBrowser } from "@/components/CategoryBrowser";

export default function Home() {
  return (
    <div className="space-y-12 py-8">
      {/* Section 1: Hero - renders immediately */}
      <section>
        <h1>Welcome to EduShop</h1>
        <p>Learn Next.js 15+ patterns</p>
      </section>

      {/* Section 2: Search - Client Component, renders immediately */}
      <section>
        <SearchBar />
      </section>

      {/* Section 3: Categories - RSC with Suspense boundary */}
      <section>
        <h2>Browse by Category</h2>

        <Suspense fallback={<LoadingSkeletons />}>
          <CategoryBrowser /> {/* Fetches products here */}
        </Suspense>
      </section>
    </div>
  );
}
```

Rendering Timeline

```
Time 0ms:   Browser requests /
Time 50ms:  Server sends HTML for hero section
Time 100ms: Browser shows hero section
Time 150ms: Server still fetching categories (Suspense fallback being shown)
Time 200ms: Browser shows loading skeleton
Time 500ms: Server finishes fetching categories from "database"
Time 550ms: Server sends category HTML to browser
Time 600ms: Browser replaces loading skeleton with categories
Time 650ms: User sees complete page
```

Key Benefit: User sees content progressively instead of waiting for the slowest component.

Search Component - UI Diffs Example

File: `src/components/SearchBar.tsx`

```tsx
"use client";

import { useState } from "react";
import { searchProducts } from "@/lib/actions";

export function SearchBar() {
  const [query, setQuery] = useState("");
  const [results, setResults] = useState([]);
  const [loading, setLoading] = useState(false);
  const [searched, setSearched] = useState(false);

  async function handleSearch(e) {
    const searchQuery = e.target.value;
    setQuery(searchQuery);

    if (searchQuery.trim() === "") {
      setResults([]);
      setSearched(false);
      return;
    }

    setLoading(true);
    setSearched(true);

    try {
      const searchResults = await searchProducts(searchQuery);
      setResults(searchResults);
    } finally {
      setLoading(false);
    }
  }

  return (
    <div>
      <input
        type="text"
        placeholder="Search products..."
        value={query}
        onChange={handleSearch}
      />

      {/* 4 Different UIs rendered based on state */}
```

```
      {/* State 1: Not searched yet */}
      {!searched && <p>Start typing to search...</p>}

      {/* State 2: Searching (loading) */}
      {loading && <p>🔄 Searching...</p>}

      {/* State 3: Found results */}
      {!loading && searched && results.length > 0 && (
        <div>
          <p>Found {results.length} products</p>
          {results.map(product => (
            <ProductCard key={product.id} product={product} />
          ))}
        </div>
      )}

      {/* State 4: No results */}
      {!loading && searched && results.length === 0 && (
        <p>No products found for "{query}"</p>
      )}
    </div>
  );
}
```

How React optimizes these diffs:

1. When `query` changes → Re-render input (efficient)
2. When `loading` becomes true → Re-render loading message (efficient)
3. When `results` updates → Reconcile product list (only updates changed items)
4. React only updates the DOM nodes that changed

🔗 Integration: How RSC, Server Actions & UI Diffs Work Together

Complete Flow Example: Product Search

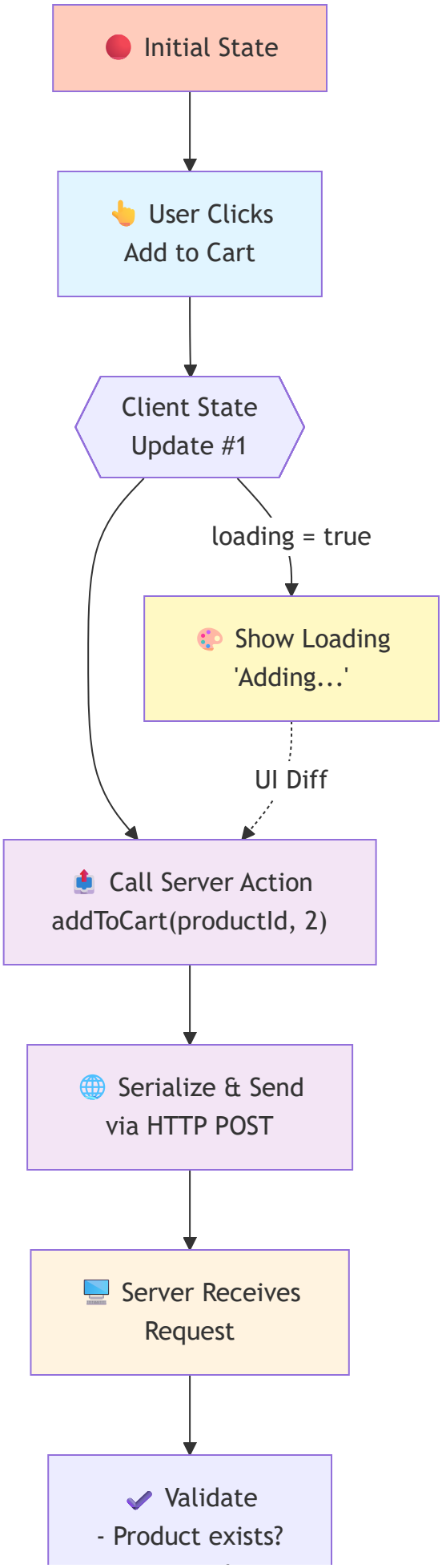
Text Diagram

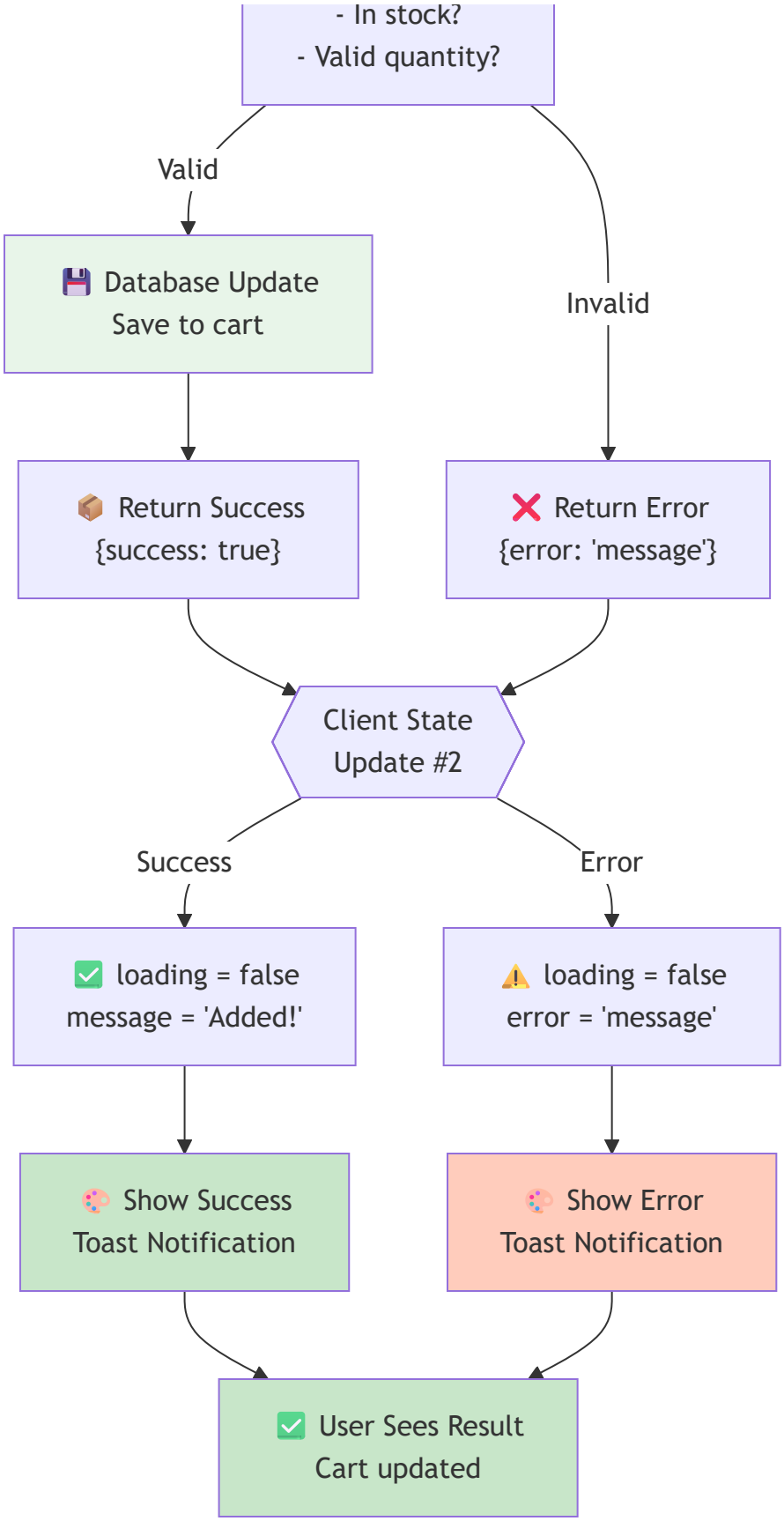
```
User types in search box
         |
         ▼
SearchBar component (Client) updates state
         |
         ▼
Input renders with new query value
         |
         ▼
onChange handler calls searchProducts() Server Action
         |
         ▼
State: loading = true
Component re-renders with loading message
         |
         ▼
Server Action runs on server:
- Gets all products (RSC function)
- Filters by search query
- Returns matching products
         |
         ▼
State: loading = false, results = [products]
Component re-renders with results
         |
         ▼
User sees matching products
```

Architecture Diagram (Mermaid)

Complete State Management Flow: Add to Cart

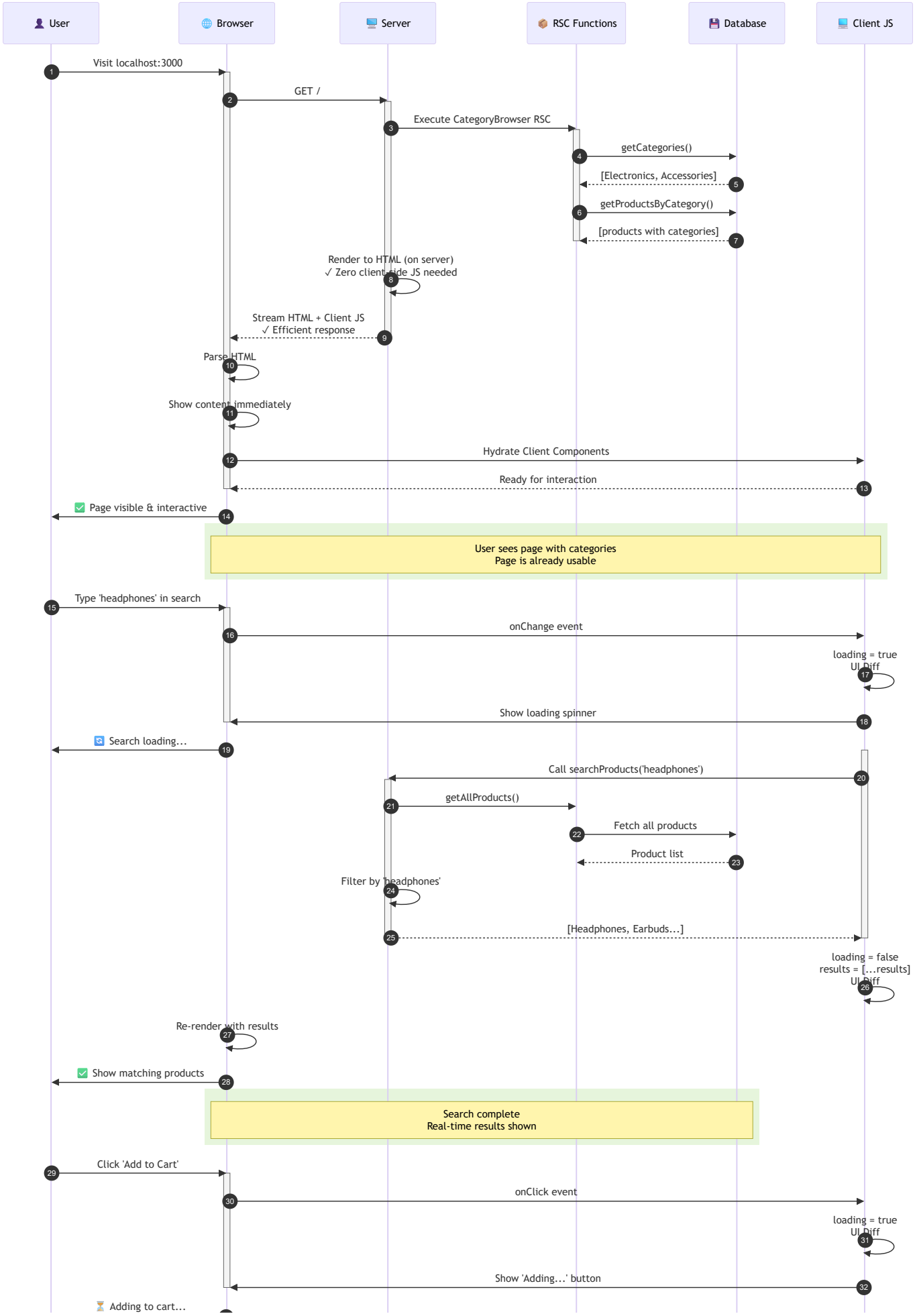
Complete State Diagram (Mermaid)

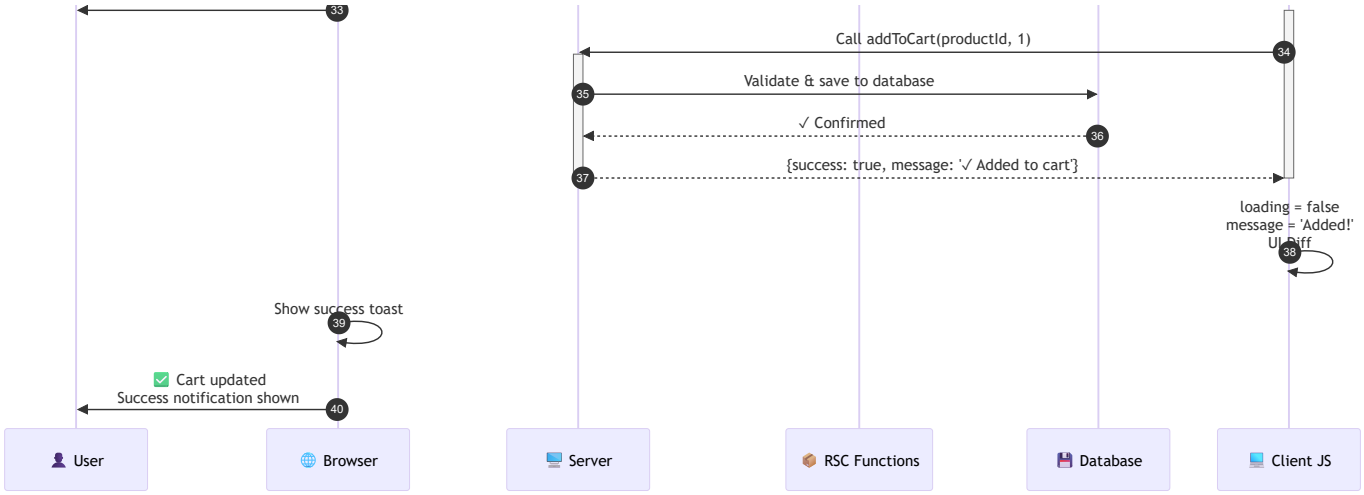

🔴 Initial State
👆 User Clicks Add to Cart
Client State Update #1
loading = true
🎨 Show Loading 'Adding...'
UI Diff
📤 Call Server Action addToCart(productId, 2)
🌐 Serialize & Send via HTTP POST
🖥️ Server Receives Request
✔ Validate - Product exists?



Complete Homepage Load: End-to-End Workflow

This diagram shows the entire lifecycle from user visiting the page to interacting with it.


User
Browser
Server
RSC Functions
Database
Client JS
Visit localhost:3000
GET /
Execute CategoryBrowser RSC
getCategories()
[Electronics, Accessories]
getProductsByCategory()
[products with categories]
Render to HTML (on server)
✓ Zero client-side JS needed
Stream HTML + Client JS
✓ Efficient response
Parse HTML
Show content immediately
Hydrate Client Components
Ready for interaction
✅ Page visible & interactive
User sees page with categories
Page is already usable
Type 'headphones' in search
onChange event
loading = true
UI Diff
Show loading spinner
🔄 Search loading...
Call searchProducts('headphones')
getAllProducts()
Fetch all products
Product list
Filter by 'headphones'
[Headphones, Earbuds...]
loading = false
results = [...results]
UI Diff
Re-render with results
✅ Show matching products
Search complete
Real-time results shown
Click 'Add to Cart'
onClick event
loading = true
UI Diff
Show 'Adding...' button
⏳ Adding to cart...



Key Insights from the Workflow:

1. **Initial Page Load (RSC)**
   - Server executes RSC functions
   - Database queries happen on server
   - HTML rendered server-side
   - Zero JavaScript overhead for data fetching
   - User sees content instantly
2. **Search Interaction (Server Action + UI Diffs)**
   - Client component detects input
   - State updates trigger UI diff
   - Loading state shown immediately
   - Server Action filters products
   - Results update state
   - UI re-renders with new data
3. **Add to Cart (Complete Flow)**
   - Button click triggers action
   - Loading state prevents double-clicks
   - Server validates data safely
   - Database updates happen server-side
   - Success/error feedback shows instantly
   - UI diffs make only necessary updates
4. **Performance Benefits**
   - No waterfall requests (parallel execution)
   - Progressive rendering (content appears gradually)
   - Minimal JavaScript for data fetching
   - Efficient UI updates (only changed parts)
   - Immediate visual feedback (UI diffs)

🎓 Learning Checklist

Understand RSC

- [ ] Read `src/lib/products.ts`
- [ ] See how no "use client" is needed
- [ ] Understand data fetches on server
- [ ] Note: No useState, useEffect, etc.

Understand Server Actions

- [ ] Read `src/lib/actions.ts`
- [ ] See "use server" directive
- [ ] Note the async functions
- [ ] Check how they're imported in components

Understand Integration

- [ ] Read `src/components/AddToCartButton.tsx`
- [ ] See how Client Component calls Server Action
- [ ] Note state management (useState)
- [ ] See how result updates UI

Understand UI Diffs

- [ ] Read `src/components/SearchBar.tsx`
- [ ] See different renders based on state
- [ ] Note loading, error, and result states
- [ ] Observe conditional rendering

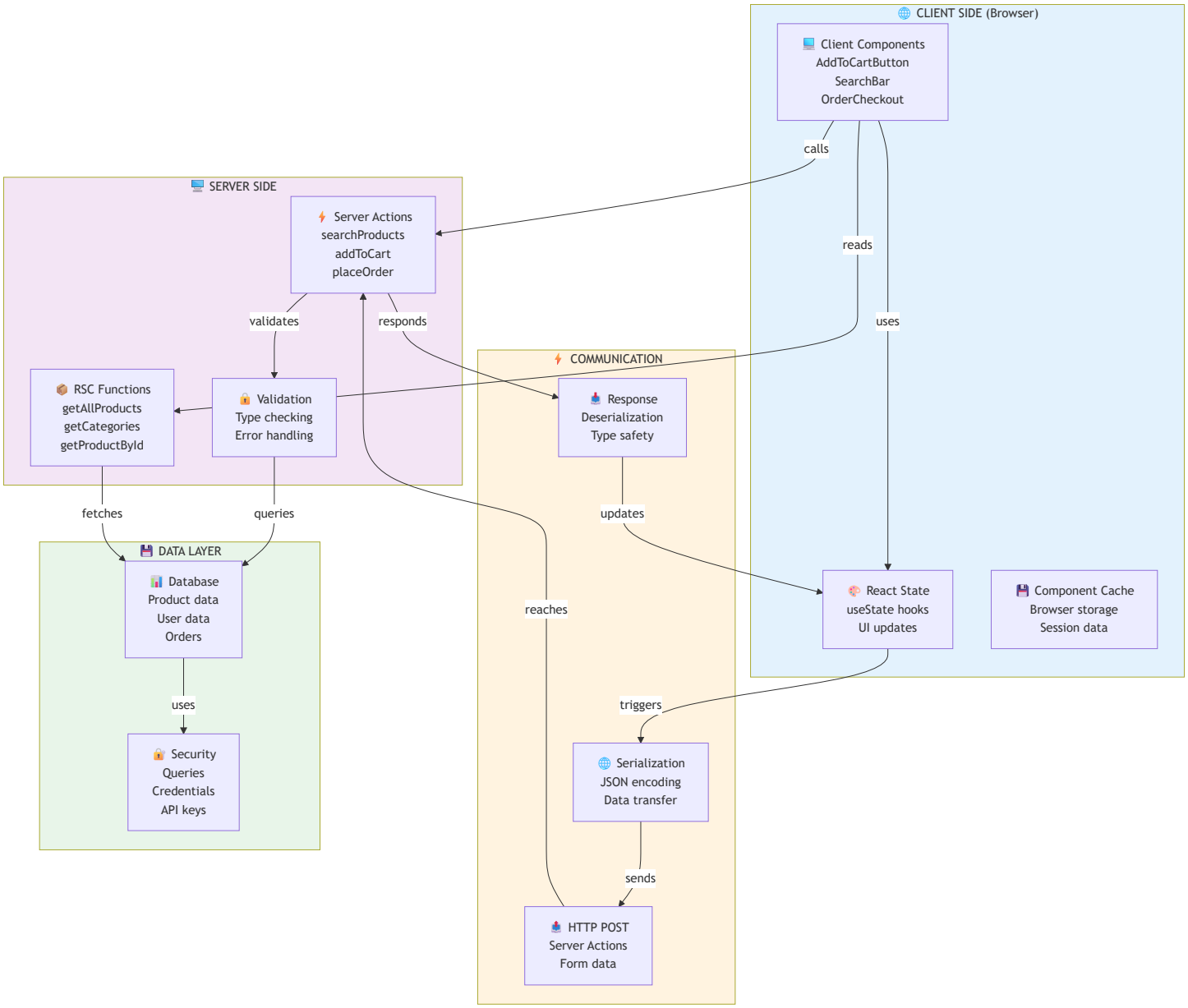
Advanced: Understand Streaming

- [ ] Read `src/app/page.tsx`
- [ ] See Suspense boundaries
- [ ] Understand fallbacks
- [ ] See how different sections load independently

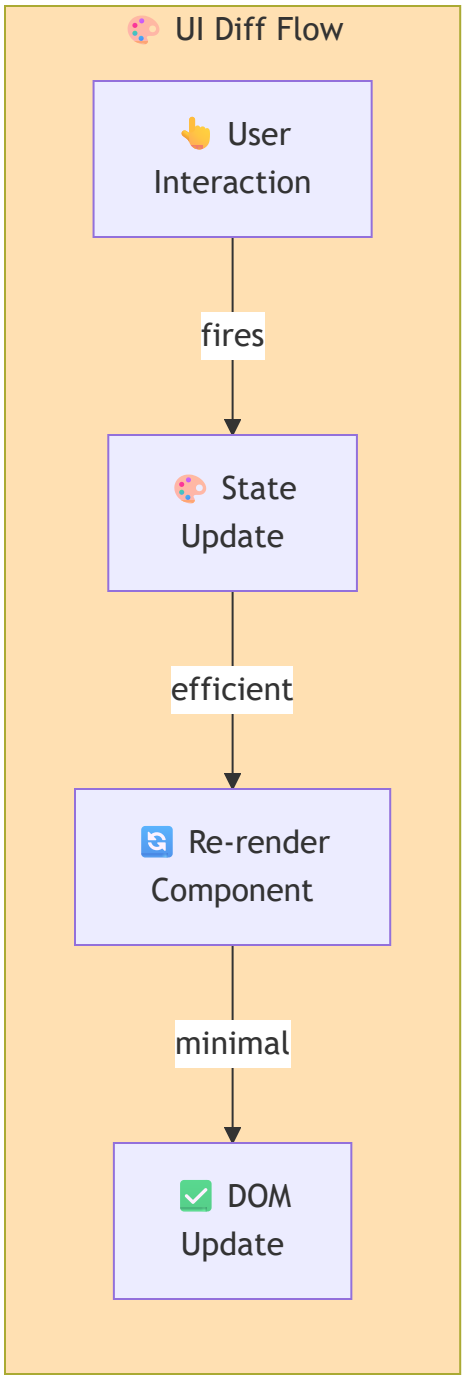
Complete Architecture Overview

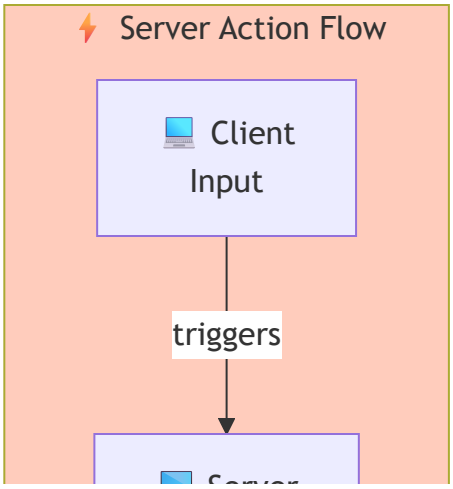
Component Interaction Architecture

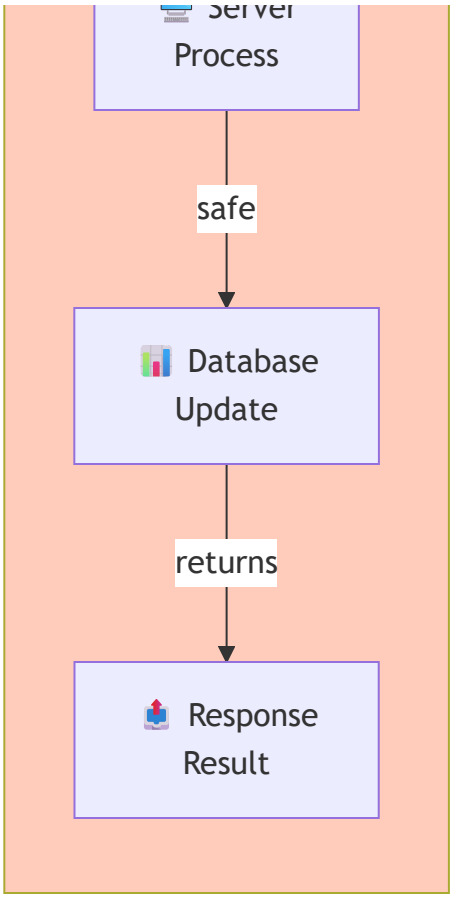
This diagram shows how all three concepts work together in the complete system:

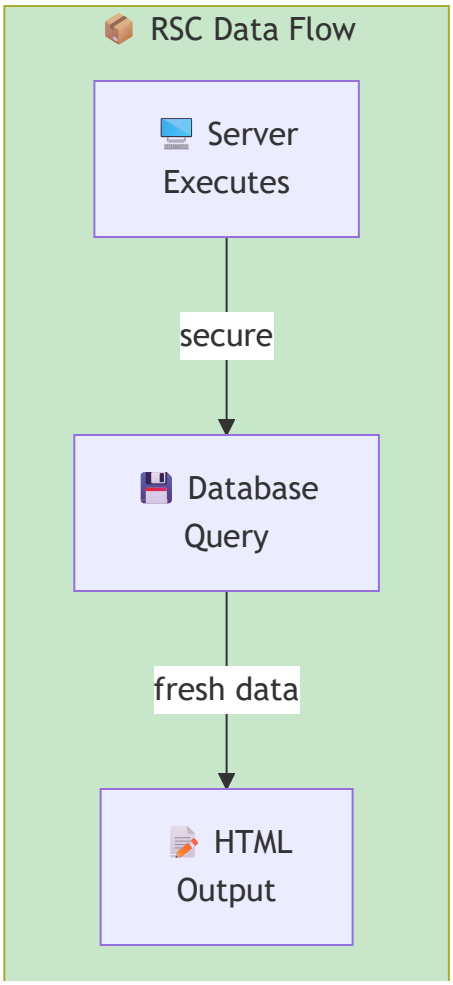


Data Types & Flow Visualization

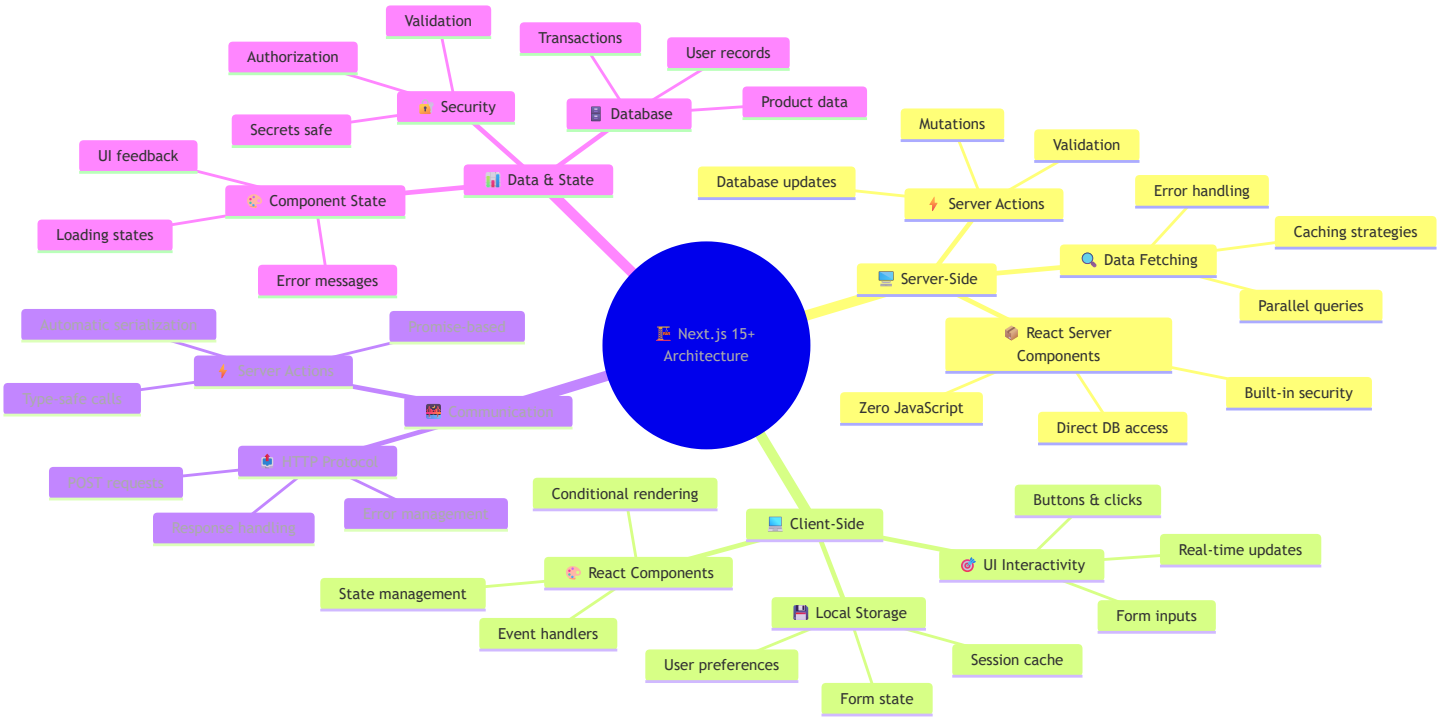

UI Diff Flow
User Interaction
fires
State Update
efficient
Re-render Component
minimal
DOM Update


Server Action Flow
Client Input
triggers

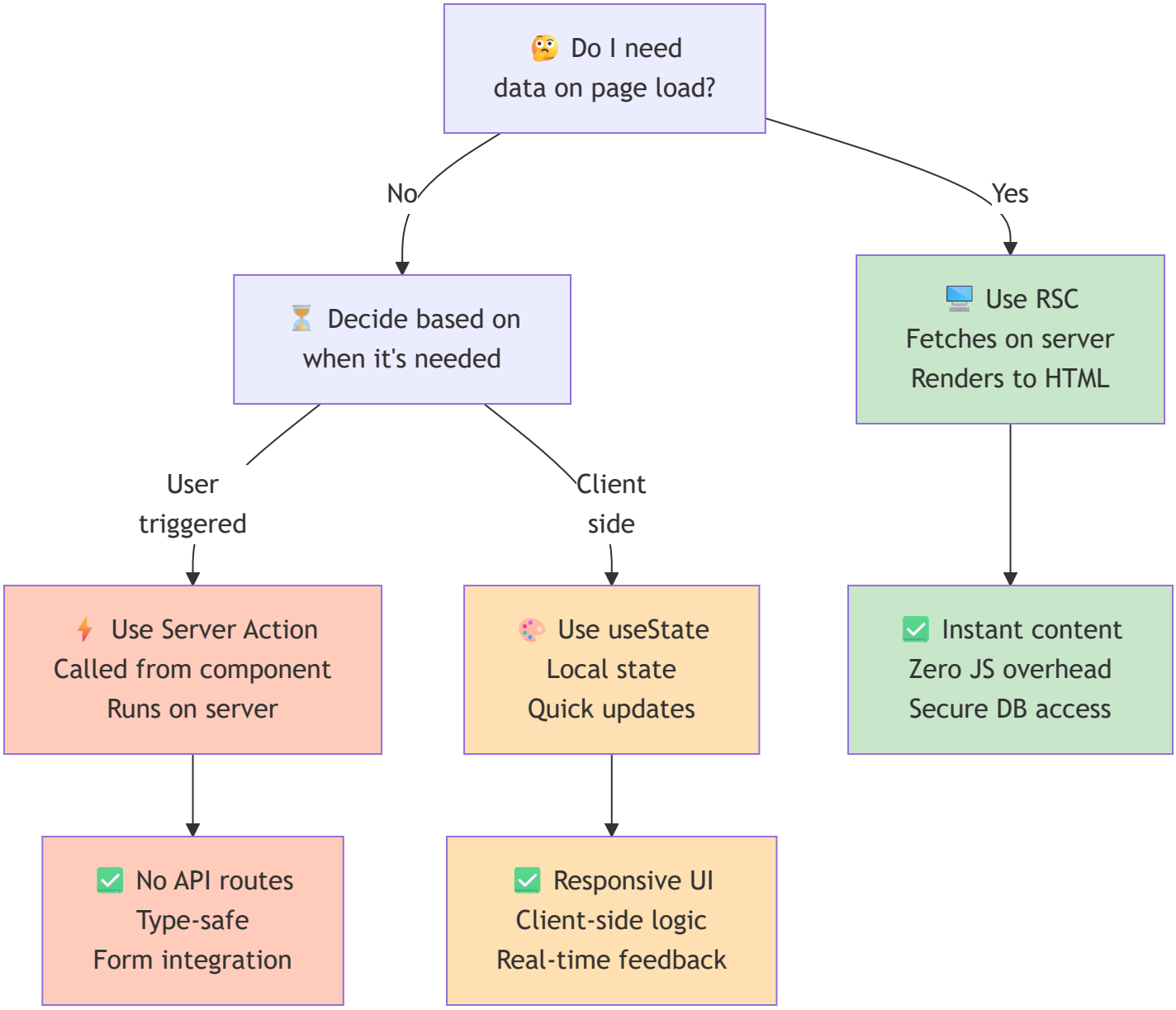

Server
Process
safe
Database
Update
returns
Response
Result


RSC Data Flow
Server
Executes
secure
Database
Query
fresh data
HTML
Output

Technology Stack Breakdown



When to Use Each Concept



From Theory to Practice: Production Patterns

You've learned the **foundational concepts** — now let's add the **production patterns** that separate hobby projects from robust applications. The next three sections show you real-world scenarios and how to handle them correctly.

4. Error Handling & Exception Handling 🛡️

What Is Error Handling?

Error handling is the practice of **catching, managing, and recovering from errors** in your application. Now that you understand RSC, Server Actions, and UI Diffs, it's time to make them **production-ready** by handling what goes wrong.

Errors can happen at three levels:

- **Server-side** (database errors, API failures)
- **Client-side** (invalid user input, network failures)
- **Network-level** (timeouts, disconnections)

Your job as an architect is to **handle errors gracefully** so users see helpful messages instead of crashes.

The Error Handling Mindset

Without error handling (BAD):

```
export async function addToCart(productId: string) {
  const product = await getProductById(productId);
  // If getProductById fails → entire app crashes ❌
  return product;
}
```

With error handling (GOOD):

```
export async function addToCart(productId: string) {
  try {
    const product = await getProductById(productId);
    if (!product) {
      return { success: false, error: "Product not found" };
    }
    return { success: true, data: product };
  } catch (error) {
    // Error caught, app continues ✅
    return { success: false, error: "Database error" };
  }
}
```

Error Handling in Server Actions

File: `src/lib/actions.ts` - Best Practice Pattern

```
"use server";

import { Product } from "@/types";

// Custom error class for better error types
class AppError extends Error {
  constructor(public statusCode: number, message: string) {
    super(message);
    this.name = "AppError";
  }
}

// Server Action with proper error handling
export async function addToCart(productId: string, quantity: number) {
  try {
    // Step 1: Validate input
    if (!productId) {
      throw new AppError(400, "Product ID is required");
    }

    if (quantity < 1) {
      throw new AppError(400, "Quantity must be at least 1");
    }

    // Step 2: Get product data
    const product = await getProductById(productId);
    if (!product) {
      throw new AppError(404, "Product not found");
    }

    // Step 3: Check business logic
    if (!product.inStock) {
      throw new AppError(409, "Product is out of stock");
    }

    if (quantity > product.stock) {
      throw new AppError(409, `Only ${product.stock} available`);
    }

    // Step 4: Perform action (update database)
    const result = await saveToCart(productId, quantity);

    // Step 5: Return success
```

```
    return {
      success: true,
      data: result,
      message: `Added ${quantity} x ${product.name} to cart`,
    };
  } catch (error) {
    // Handle known errors
    if (error instanceof AppError) {
      return {
        success: false,
        error: error.message,
        statusCode: error.statusCode,
      };
    }

    // Handle unexpected errors
    console.error("Unexpected error in addToCart:", error);
    return {
      success: false,
      error: "An unexpected error occurred. Please try again.",
      statusCode: 500,
    };
  }
}
```

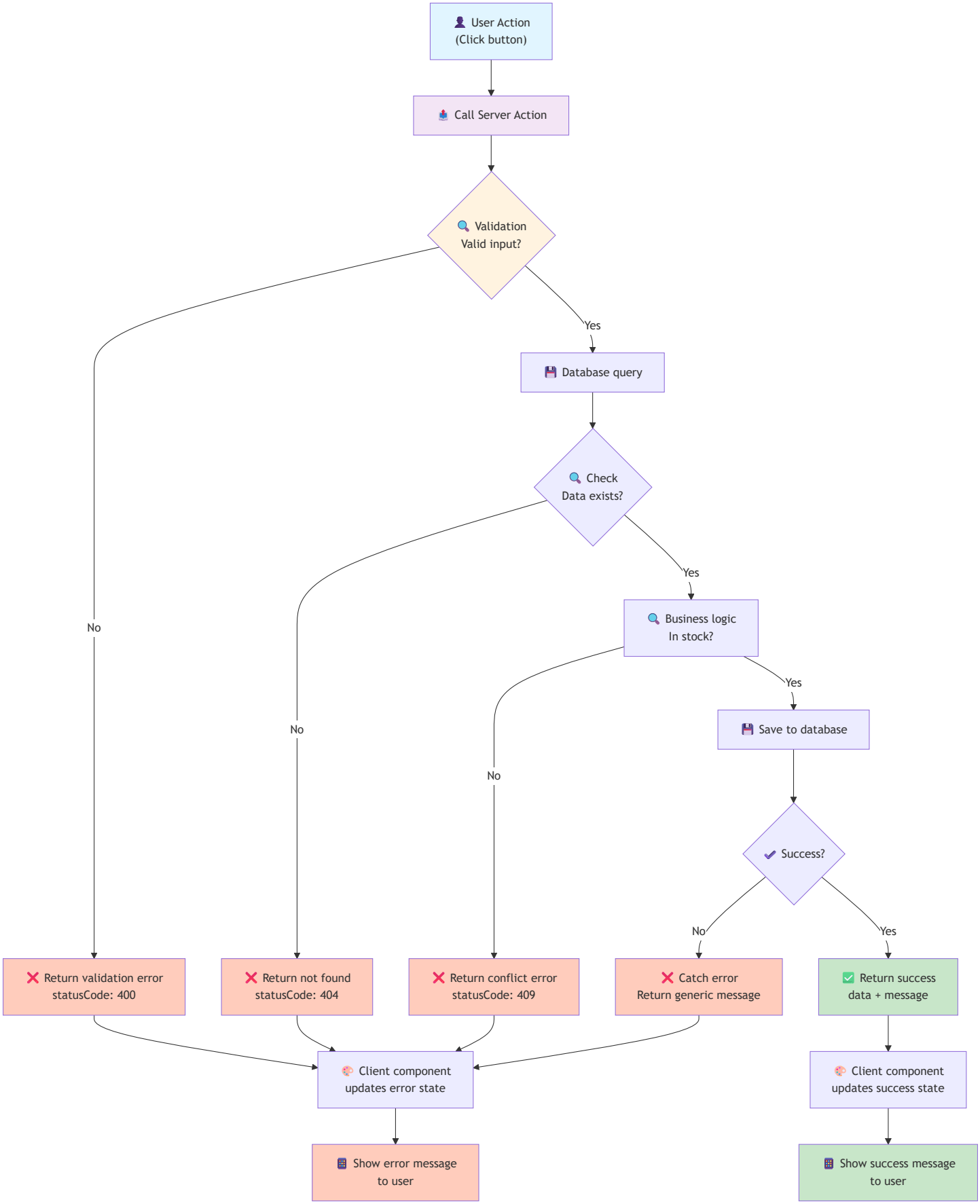
Error Handling in Client Components

File: `src/components/AddToCartButton.tsx`

```tsx
"use client";

import { addToCart } from "@/lib/actions";
import { useState } from "react";

export function AddToCartButton({ productId }: { productId: string }) {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);
  const [success, setSuccess] = useState(false);

  async function handleAddToCart() {
    setLoading(true);
    setError(null);
    setSuccess(false);

    try {
      // Call Server Action
      const result = await addToCart(productId, 1);

      // Handle Server Action result
      if (result.success) {
        setSuccess(true);
        setTimeout(() => setSuccess(false), 3000); // Hide after 3s
      } else {
        setError(result.error);
      }
    } catch (error) {
      // Handle unexpected client-side errors
      setError("Failed to add to cart. Please try again.");
      console.error("Error in handleAddToCart:", error);
    } finally {
      setLoading(false);
    }
  }

  return (
    <div>
      {/* Show error message */}
      {error && (
        <div className="bg-red-100 text-red-800 p-3 rounded">
          ⚠️ {error}
        </div>
      )}
```

```
      {/* Show success message */}
      {success && (
        <div className="bg-green-100 text-green-800 p-3 rounded">
          ✅ Added to cart!
        </div>
      )}

      {/* Button */}
      <button
        onClick={handleAddToCart}
        disabled={loading || success}
        className="bg-blue-500 text-white px-4 py-2 rounded"
      >
        {loading ? "Adding..." : success ? "Added! ✓" : "Add to Cart"}
      </button>
    </div>
  );
}
```

Error Handling Flow Diagram



Error Handling Best Practices

✅ Do	❌ Don't
Catch specific errors	Catch all errors silently
Return structured responses	Throw errors to client
Log errors for debugging	Ignore errors completely
Show user-friendly messages	Show technical stack traces
Validate input early	Assume input is valid
Handle errors in Server Actions	Let errors propagate
Test error scenarios	Only test happy paths
Use custom error classes	Throw generic strings

5. Form Handling Patterns 📝

What Is Form Handling?

Forms are **the primary way users interact with your app**. Form handling is about **capturing user input, validating it, submitting it to the server, and handling responses**. With Server Actions, this becomes elegantly simple — but you still need the right patterns for a smooth user experience.

Traditional vs Next.js Approach

Traditional (Old Way):

```
// 1. Create /api/addProduct route
// 2. Handle POST request
// 3. Call from client with fetch()
// 4. Parse response
// Lots of boilerplate! 😞
```

Next.js with Server Actions (Modern Way):

```
"use server";

export async function addProduct(formData: FormData) {
  // Just handle the form, no API route needed!
}
```

Form Patterns in EduShop

Pattern 1: Simple Form with Server Action

File: `src/components/AddProductForm.tsx`

```tsx
"use client";

import { addProduct } from "@/lib/actions";
import { useState } from "react";

export function AddProductForm() {
  const [error, setError] = useState<string | null>(null);
  const [success, setSuccess] = useState(false);
  const [loading, setLoading] = useState(false);

  async function handleSubmit(formData: FormData) {
    setLoading(true);
    setError(null);

    try {
      const result = await addProduct(formData);

      if (result.success) {
        setSuccess(true);
        // Reset form or redirect
      } else {
        setError(result.error);
      }
    } catch (error) {
      setError("Failed to add product");
    } finally {
      setLoading(false);
    }
  }

  return (
    <form action={handleSubmit}>
      {/* Text input */}
      <input
        type="text"
        name="productName"
        placeholder="Product name"
        required
      />

      {/* Number input */}
      <input
        type="number"
```

```
        name="price"
        placeholder="Price"
        step="0.01"
        required
      />

      {/* Select */}
      <select name="category" required>
        <option value="">Choose category</option>
        <option value="electronics">Electronics</option>
        <option value="accessories">Accessories</option>
      </select>

      {/* Textarea */}
      <textarea
        name="description"
        placeholder="Description"
        required
      ></textarea>

      {/* Error display */}
      {error && <div className="text-red-600">{error}</div>}

      {/* Success display */}
      {success && <div className="text-green-600">Product added!</div>}

      {/* Submit button */}
      <button type="submit" disabled={loading}>
        {loading ? "Adding..." : "Add Product"}
      </button>
    </form>
  );
}
```

Server Action:

```ts
"use server";

export async function addProduct(formData: FormData) {
  try {
    // Extract fields from FormData
    const name = formData.get("productName") as string;
    const price = parseFloat(formData.get("price") as string);
    const category = formData.get("category") as string;
    const description = formData.get("description") as string;

    // Validate on server (even though client validated!)
    if (!name || !price || !category || !description) {
      return { success: false, error: "All fields required" };
    }

    if (price <= 0) {
      return { success: false, error: "Price must be positive" };
    }

    // Save to database
    const product = {
      id: crypto.randomUUID(),
      name,
      price,
      category,
      description,
      createdAt: new Date(),
    };

    // In real app: await db.products.create(product);

    return { success: true, data: product };
  } catch (error) {
    return { success: false, error: "Failed to add product" };
  }
}
```

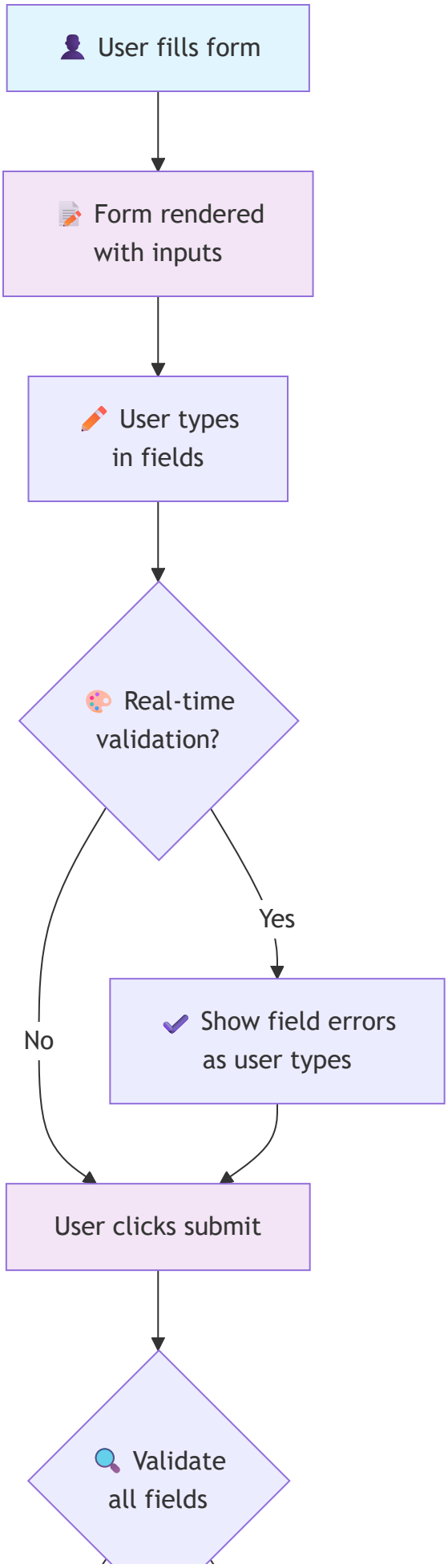
Pattern 2: Form with Client-Side Validation

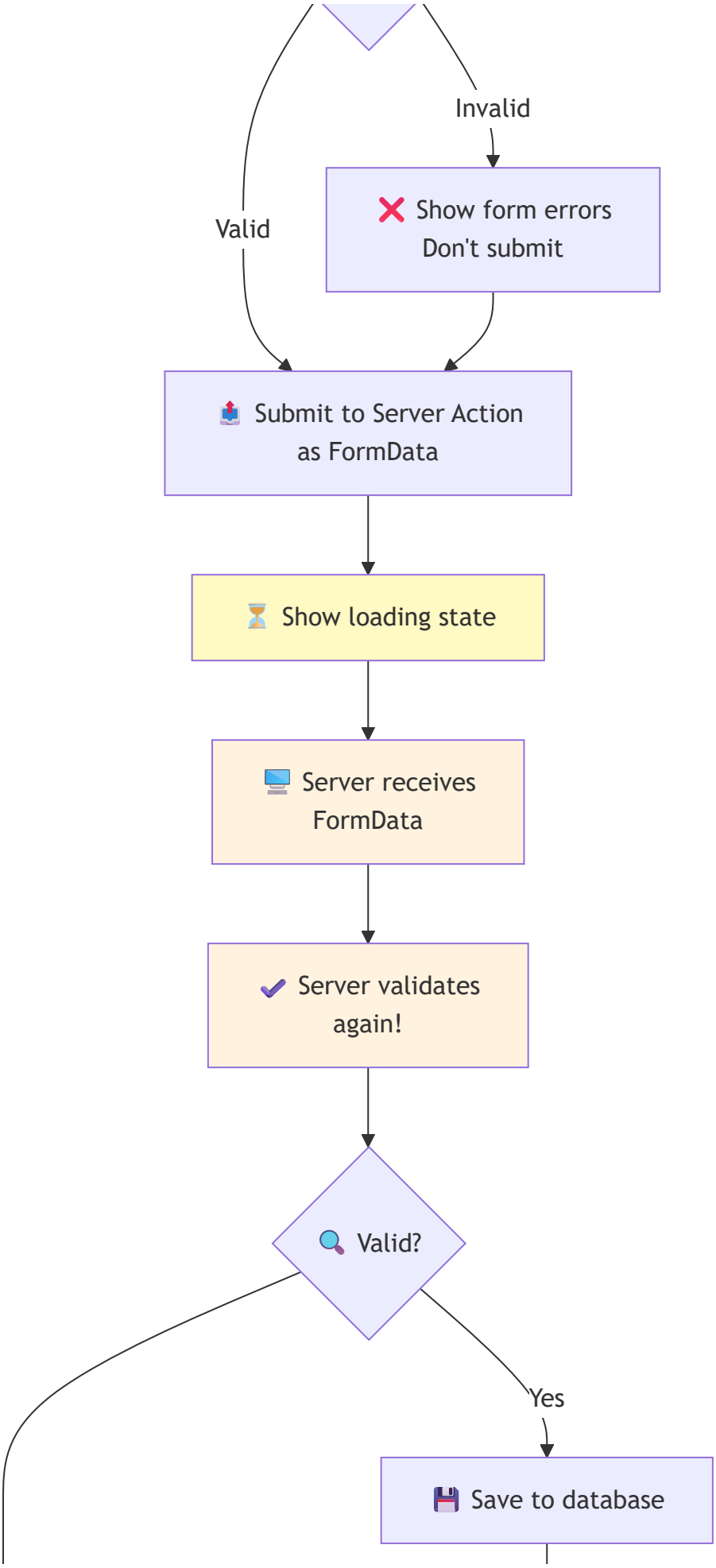
```tsx
"use client";

import { updateRating } from "@/lib/actions";
import { useState } from "react";

export function RatingForm({ productId }: { productId: string }) {
  const [rating, setRating] = useState(5);
  const [comment, setComment] = useState("");
  const [errors, setErrors] = useState<Record<string, string>>({});
  const [loading, setLoading] = useState(false);

  // Client-side validation function
  function validateForm() {
    const newErrors: Record<string, string> = {};

    if (rating < 1 || rating > 5) {
      newErrors.rating = "Rating must be between 1 and 5";
    }

    if (comment.trim().length < 10) {
      newErrors.comment = "Comment must be at least 10 characters";
    }

    if (comment.length > 500) {
      newErrors.comment = "Comment must be less than 500 characters";
    }

    return newErrors;
  }

  async function handleSubmit(e: React.FormEvent) {
    e.preventDefault();

    // Validate
    const newErrors = validateForm();
    if (Object.keys(newErrors).length > 0) {
      setErrors(newErrors);
      return;
    }

    setLoading(true);
```

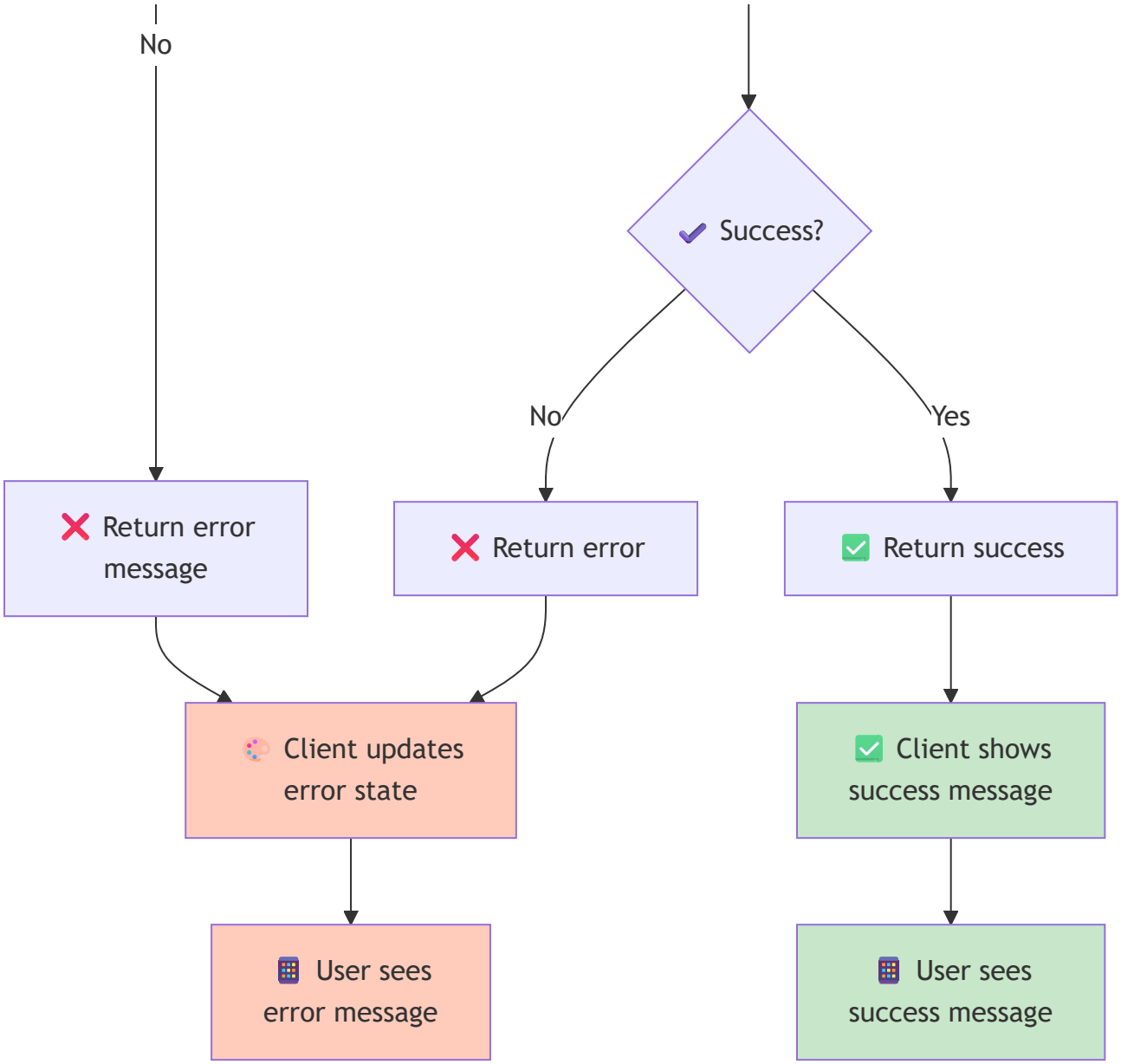
```
    setErrors({});

    try {
      // Call Server Action
      const formData = new FormData();
      formData.append("productId", productId);
      formData.append("rating", rating.toString());
      formData.append("comment", comment);

      const result = await updateRating(formData);

      if (result.success) {
        // Reset form
        setRating(5);
        setComment("");
        alert("Rating submitted!");
      } else {
        setErrors({ form: result.error });
      }
    } finally {
      setLoading(false);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* Rating input */}
      <div>
        <label>Rating (1-5):</label>
        <input
          type="range"
          min="1"
          max="5"
          value={rating}
          onChange={(e) => setRating(parseInt(e.target.value))}
        />
        <span>{rating} stars</span>
        {errors.rating && <p className="text-red-600">{errors.rating}</p>}
      </div>

      {/* Comment input */}
      <div>
        <label>Comment:</label>
```

```
        <textarea
          value={comment}
          onChange={(e) => setComment(e.target.value)}
          placeholder="Share your thoughts..."
        />
        <p className="text-sm">{comment.length}/500</p>
        {errors.comment && <p className="text-red-600">{errors.comment}</p>}
      </div>

      {/* Form-level errors */}
      {errors.form && <div className="text-red-600">{errors.form}</div>}

      {/* Submit */}
      <button type="submit" disabled={loading}>
        {loading ? "Submitting..." : "Submit Rating"}
      </button>
    </form>
  );
}
```

Form Handling Flow Diagram


User fills form
Form rendered with inputs
User types in fields
Real-time validation?
Yes
No
Show field errors as user types
User clicks submit
Validate all fields


Invalid
Valid
❌ Show form errors
Don't submit
📤 Submit to Server Action
as FormData
⏳ Show loading state
🖥️ Server receives
FormData
✔️ Server validates
again!
🔍 Valid?
Yes
💾 Save to database



Form Handling Best Practices

Area	Best Practice
Validation	Validate on client AND server
Error display	Show field-level and form-level errors
Loading state	Disable submit button while loading
Success feedback	Show success message or redirect
Security	Sanitize input on server
Accessibility	Use labels and semantic HTML
UX	Show real-time validation feedback

Area	Best Practice
Testing	Test both success and error paths

6. Data Validation 🔍

What Is Data Validation?

Data validation is the **last line of defense** before user input becomes permanent data. It happens in two places:

1. **Client-side** - For instant user feedback (fast, improves UX)
2. **Server-side** - For security (required, cannot be bypassed)

Validation ties together everything you've learned: Server Actions to process data, forms to capture it, error handling to report problems. Skipping validation is like leaving your database unlocked.

Types of Validation

```
User Input
    │
    ├─ Format validation    (email format, phone format)
    ├─ Type validation      (is it a number? string?)
    ├─ Length validation    (min/max length)
    ├─ Range validation     (between 0-100?)
    ├─ Pattern validation   (matches regex?)
    └─ Business logic       (in stock? user exists?)
```

Validation in EduShop

Pattern 1: Simple Field Validation

Client-side + Server-side:

```
// src/lib/validation.ts - Shared validation logic
export function validateEmail(email: string): string | null {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!emailRegex.test(email)) {
    return "Invalid email format";
  }
  return null;
}

export function validatePrice(price: number): string | null {
  if (price < 0) return "Price must be positive";
  if (price > 1000000) return "Price is too high";
  return null;
}

export function validateProductName(name: string): string | null {
  if (name.length < 3) return "Name must be at least 3 characters";
  if (name.length > 100) return "Name must be less than 100 characters";
  if (!/^[a-zA-Z0-9\s-]+$/.test(name)) {
    return "Name can only contain letters, numbers, spaces, and hyphens";
  }
  return null;
}

export function validateQuantity(quantity: number): string | null {
  if (!Number.isInteger(quantity)) return "Quantity must be a whole number";
  if (quantity < 1) return "Quantity must be at least 1";
  if (quantity > 999) return "Quantity is too high";
  return null;
}
```

Client Component with Validation:

```tsx
"use client";

import { validateEmail, validateQuantity } from "@/lib/validation";
import { useState } from "react";

export function CheckoutForm() {
  const [formData, setFormData] = useState({
    email: "",
    quantity: 1,
  });

  const [fieldErrors, setFieldErrors] = useState<Record<string, string>>({});

  // Validate field as user types
  function handleFieldChange(field: string, value: string | number) {
    setFormData((prev) => ({ ...prev, [field]: value }));

    // Clear error for this field
    setFieldErrors((prev) => ({
      ...prev,
      [field]: "",
    }));

    // Validate immediately
    let error = null;
    if (field === "email") {
      error = validateEmail(value as string);
    } else if (field === "quantity") {
      error = validateQuantity(value as number);
    }

    if (error) {
      setFieldErrors((prev) => ({ ...prev, [field]: error }));
    }
  }

  // Validate all fields before submit
  function validateAllFields(): boolean {
    const errors: Record<string, string> = {};

    const emailError = validateEmail(formData.email);
    if (emailError) errors.email = emailError;
```

```
  const quantityError = validateQuantity(formData.quantity);
  if (quantityError) errors.quantity = quantityError;

  setFieldErrors(errors);
  return Object.keys(errors).length === 0;
}

async function handleSubmit(e: React.FormEvent) {
  e.preventDefault();

  if (!validateAllFields()) {
    return; // Don't submit if validation fails
  }

  // Submit to server
  const formDataObj = new FormData();
  formDataObj.append("email", formData.email);
  formDataObj.append("quantity", formData.quantity.toString());

  const result = await checkout(formDataObj);
  // Handle result...
}

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Email:</label>
      <input
        type="email"
        value={formData.email}
        onChange={(e) => handleFieldChange("email", e.target.value)}
        placeholder="your@email.com"
      />
      {fieldErrors.email && (
        <span className="text-red-600">{fieldErrors.email}</span>
      )}
    </div>

    <div>
      <label>Quantity:</label>
      <input
        type="number"
        value={formData.quantity}
```

```
            onChange={(e) => handleFieldChange("quantity", parseInt(e.target.value))}
            min="1"
            max="999"
          />
          {fieldErrors.quantity && (
            <span className="text-red-600">{fieldErrors.quantity}</span>
          )}
        </div>

        <button type="submit">Checkout</button>
      </form>
    );
  }
```

Server-side Validation (Required!):

```tsx
"use server";

import {
  validateEmail,
  validateQuantity,
  validatePrice,
} from "@/lib/validation";

export async function checkout(formData: FormData) {
  try {
    const email = formData.get("email") as string;
    const quantity = parseInt(formData.get("quantity") as string);
    const price = parseFloat(formData.get("price") as string);

    // ALWAYS validate on server, even if client validated
    const emailError = validateEmail(email);
    if (emailError) {
      return { success: false, error: emailError };
    }

    const quantityError = validateQuantity(quantity);
    if (quantityError) {
      return { success: false, error: quantityError };
    }

    const priceError = validatePrice(price);
    if (priceError) {
      return { success: false, error: priceError };
    }

    // Additional server-only validation
    // (things only server can check)
    const user = await db.users.findByEmail(email);
    if (!user) {
      return { success: false, error: "User not found" };
    }

    // Perform action
    const order = await db.orders.create({
      userId: user.id,
      quantity,
      price,
    });
```

```
    return { success: true, data: order };
  } catch (error) {
    return { success: false, error: "Checkout failed" };
  }
}
```

Validation Patterns Diagram

Validation Checklist

For every input:

- [ ] Define what valid data looks like
- [ ] Validate format on client (instant feedback)
- [ ] Validate format on server (security)
- [ ] Check length/range on both sides
- [ ] Check business logic on server only
- [ ] Show specific error messages
- [ ] Test with invalid data
- [ ] Test with extreme values

Security Considerations

⚠️ Risk	🛡️ Solution
SQL injection	Use parameterized queries, sanitize input
XSS attacks	Escape output, validate input
Invalid state	Validate all inputs on server
Business logic bypass	Check business rules on server
Type coercion	Explicitly convert types

⚠️ Risk	🛡️ Solution
Empty/null values	Check for missing fields

💡 Key Takeaways

1. **RSCs** = Data fetching on server, zero JS overhead
2. **Server Actions** = Mutations without API routes
3. **UI Diffs** = React efficiently updates only changed parts
4. **Streaming** = Progressive rendering for better UX
5. **Together** = The most efficient architecture for modern web apps

Remember: Every request follows this pattern:

- 🖥️ RSCs prepare data on server
- ⚡ Server Actions handle mutations
- 🎨 UI Diffs make updates responsive
- 👤 User sees instant feedback

🔑 Key Takeaways

The Three Pillars Work Together

You've now learned the complete Next.js architecture:

1. **React Server Components** = Secure data fetching, zero client-side JS
2. **Server Actions** = Mutations without API routes
3. **UI Diffs** = React efficiently updates only changed parts
4. **Streaming** = Progressive rendering for better UX
5. **Error Handling** = Graceful failure, user-friendly messages
6. **Form Handling** = Simple, type-safe form submission
7. **Data Validation** = Security at every layer

The Mental Model

```
User Request
    ↓
RSCs fetch data on server (secure, fast)
    ↓
HTML sent to browser with content
    ↓
User interacts (clicks, types, submits)
    ↓
Server Actions process mutations safely
    ↓
UI Diffs update only what changed
    ↓
User sees instant feedback
```

Pattern Recognition

- **Need data on page load?** → Use RSC
- **User triggered action?** → Use Server Action
- **Real-time feedback?** → Use UI Diffs & loading states
- **Something might fail?** → Use try-catch & custom errors
- **Capturing user input?** → Use Server Action forms
- **Protecting data?** → Validate on both client & server

Your Path Forward

Remember: Every request follows this pattern:

- 🖥️ RSCs prepare data on server
- ⚡ Server Actions handle mutations
- 🎨 UI Diffs make updates responsive
- ✔️ Validation secures everything
- 🛡️ Error handling ensures stability

🚀 Next Steps

1. Modify `src/lib/products.ts` - Add a new filter function

2. Create a new Server Action in `src/lib/actions.ts` - Add a wishlist feature
3. Create a new Client Component - Try building a product filter
4. Add error handling - Use try-catch in Server Actions
5. Add form-based Server Action - Create a contact form

Remember: The best way to learn is by experimenting. Change the code, break things, and see what happens!