* **Server Actions**:
  + Enable users to perform actions in Next.js apps.
  + Part of the **React Server Components (RSC) Architecture**.
  + Complement **Server Components** to build **Interactive Full-Stack Applications**.
* **Purpose of Server Actions**:
  + Handle **user interactions** (e.g., creating, updating, or deleting data).
  + Perform **data mutations** (e.g., CRUD operations).(data mutation from client but in server)
  + Essential for building interactive apps beyond just **data fetching**.
* **How Server Actions Work**:
  + **1.Server Actions are Asynchronous functions** that run exclusively on the server.
  + **2.**Created using the **use server** directive:
    - Can be defined inside a **Server Component** or in a **standalone file**.
    - **Functions exported from a standalone file with use server directive become Server Actions**.(Important)
  + **Recommended practice**: Store Server Actions in a dedicated module for centralized management.
* **use server Directive**:
  + Used exclusively for **Server Actions**, not for Server Components.
  + Bridges the gap from **client to server**, allowing client-side code to communicate with the server.
  + Acts like an **API endpoint** for frontend applications.
* **Comparison with use client Directive**:
  + **use client**: Bridges server-to-client communication (e.g., for Client Components).
  + **use server**: Bridges client-to-server communication (e.g., for Server Actions).
* **Behind the Scenes**:
  + **Next.js automatically creates an API endpoint for each Server Action**.
  + Only the **URL** of the Server Action is sent to the client; the code remains on the server.
  + Ensures security (e.g., database connections, API keys) as code never reaches the browser.
  + When invoked, a **POST request** is made to the endpoint, and inputs are serialized.
* **Benefits of Server Actions**:
  + Eliminates the need for separate API or **Next.js route handlers** for data mutations.
  + Tightly integrated with **Next.js caching and revalidation system**:
    - Use **revalidatePath** and **revalidateTag** to update the UI after data mutations.
  + Simplifies **form submissions**:
    - Can be used as the action attribute in forms (works in both Server and Client Components).
    - Automatically serializes form data and sends it to the Server Action.
* **Use Cases for Server Actions**:
  + **Form submissions**: Handle form data without additional frontend code.
  + **Event handlers** and **useEffect**: Can be called like regular functions in Client Components.
  + **Data mutations**: Create, update, or delete data.
  + **Cookies**: Can work with cookies and run any backend-relevant code.
* **Key Considerations**:
  + Server Actions require a **running web server** (unlike Server Components, which can run at build time).
  + Code runs on the backend, so:
    - Ensure **user authorization** for actions.
    - Treat inputs as **unsafe** (similar to backend development).
* **Summary**:
  + Server Actions are **automatically created API endpoints** for data mutations.
  + Simplify building interactive apps by integrating **data fetching** and **mutations** within React.
  + Tightly integrated with Next.js features like **caching**, **revalidation**, and **forms**.

**Updating Profile using Server Action**

**Step1: Fetch Guest Data in page.js (Server Component)**

* **Retrieve session data to get the guest’s email.**
* **Fetch guest details from the database using their email.**
* **Pass guest data as props to the UpdateProfile client component.**

**Code:**

**const session = await auth(); // Get session data**

**const email = session?.user?.email;**

**const guest = await getGuest(email);**

**//pass it as props to updateProfile frorm**

**<UpdateProfileForm guest={guest} />;**

**//Inside UpdateProfileForm.js**

** Prepopulate fields with guest data.**

** Use a <form> with the server action as the action attribute.**

** Ensure input fields have name attributes matching database fields**

**//inside actions.js**

**xport async function updateProfile(formData) {**

**const session = await auth();**

**if (!session?.user?.email)**

**throw new Error("Unauthorized");**

**const guestID = session.user.guestId;**

**console.log(`guestID ${guestID}`);**

**// const [nationality, countryFlag] = [null, null];**

**const [nationality, countryFlag] = formData**

**.get("nationality")**

**.split("%");**

**console.log(`nationality ${nationality}`);**

**const nationalID = formData.get("nationalID");**

**// Validate National ID (6-12 alphanumeric characters)**

**if (!/^[a-zA-Z0-9]{6,12}$/.test(nationalID)) {**

**throw new Error(**

**"Invalid National ID format."**

**);**

**}**

**// Update guest data in Supabase**

**const updateData = {**

**nationality,**

**countryFlag,**

**nationalID,**

**};**

**console.log(updateData);**

**const { data, error } = await supabase**

**.from("guests")**

**.update(updateData)**

**.eq("id", guestID)**

**.select()**

**.single();**

**if (error) {**

**console.error(error);**

**throw new Error("Guest could not be updated");**

**}**

**}**

**Problem: Stale Data in UI**

* **When updating data, navigating away, and returning, old data still appears.**
* **The updated data is eventually shown after 30 seconds due to browser cache duration for dynamic pages.**
* **Goal: Ensure fresh data is displayed immediately.**

** Solution: Cache Revalidation**

* **Revalidate Cache: Clear outdated cache and fetch fresh data.**
* **Two types of revalidation:**
  + **Time-based Revalidation – Cache updates after a set time.**
  + **Manual (On-Demand) Revalidation – Clears and refetches data immediately.**
* **Preferred Approach: Use manual revalidation after data is updated in Supabase.**

** Implementation: Using revalidatePath Function**

* **Ensures all data tied to a specific route is refreshed.**
* **Can pass general paths (/account) or specific paths (/account/profile).**
* **Avoids unnecessary revalidation (e.g., reservations).**
* **Import Statement:**

**javascript**

**CopyEdit**

**import { revalidatePath } from 'next/cache';**

* **Steps:**
  1. **Call revalidatePath('/account/profile'); after data update.**
  2. **Ensures fresh data is immediately displayed.**

**React useFormStatus Hook Summary**

* **Purpose**: Provides form submission status, allowing UI updates (e.g., disabling a button, showing a loading indicator).
* **React Version**: Experimental, part of ReactDOM, expected in React 19. Already adopted by **Next.js**.
* **Key Restriction**:
  + Must be used inside a **component rendered within the form** (not in a parent component containing the form).
* **Implementation Steps**:
  + Create a new **button component** inside the form.
  + Import useFormStatus from ReactDOM.
  + Destructure returned object to access **pending** status (indicates if form is submitting).
  + Conditionally update UI:
    - Change button text (e.g., "Updating..." when pending).
    - Disable button while submitting (disabled={pending}).
    - Apply Tailwind styles for visual feedback.
* **Client Component Requirement**:
  + Since useFormStatus is a **hook**, it must be used in a **client component**.
  + If in a **server component**, extract button into a separate file with "use client" directive at the top.

**1️⃣ Step-by-Step Execution of deleteReservation**

**1️⃣ Authenticate the User**

javascript

CopyEdit

const session = await auth();

if (!session) throw new Error("You must be logged in");

* Calls auth() to get the **logged-in user's session**.
* If the user is **not logged in**, an error is thrown.

**2️⃣ Fetch User’s Bookings**

javascript

CopyEdit

const guestBookings = await getBookings(session.user.guestId);

* Retrieves **all bookings** associated with the current user using session.user.guestId.
* This ensures we have a list of **only the user's bookings**.

**3️⃣ Extract User’s Booking IDs**

javascript

CopyEdit

const guestBookingIds = guestBookings.map(booking => booking.id);

* Creates a list (guestBookingIds) containing **only booking IDs** that belong to the user.
* This helps in **quick validation** in the next step.

**4️⃣ Check if the User Owns the Booking**

javascript

CopyEdit

if (!guestBookingIds.includes(bookingId)) {

throw new Error("You are not allowed to delete this booking");

}

* **Prevents unauthorized deletions** by checking:  
  ✅ Does bookingId exist in guestBookingIds?  
  ❌ If not, an **error is thrown**, blocking deletion.

**5️⃣ Delete the Booking (Only if Authorized)**

javascript

CopyEdit

const { error } = await supabase

.from("bookings")

.delete()

.eq("id", bookingId);

* Calls **Supabase** to delete the booking **only if ownership is verified**.
* If deletion fails, an error is thrown.

**Refresh the UI Cache**

javascript

CopyEdit

revalidatePath("account/reservations");

* **Ensures the UI updates** so that the deleted booking is removed from the reservations page.

**2️⃣ Why This Fix Works?**

✅ **Prevents unauthorized deletions** → Ensures the user **owns** the booking before deleting.  
✅ **Blocks cURL-based attacks** → Even if a hacker guesses a bookingId, they can’t delete others’ reservations.  
✅ **Enhances security** → Only **authenticated** users can access their reservations.

**3️⃣ Next Steps**

* **Improve Error Messages**: Show user-friendly messages in the UI.
* **Log Failed Deletion Attempts**: Track unauthorized access attempts for security monitoring.

**🚀 Summary: Secure deleteReservation Flow**

1️Authenticate the user ✅  
2️Fetch **only the user’s bookings** ✅  
3️Extract their **booking IDs** ✅  
4️ **Verify ownership** before deletion ✅  
5️ **Delete the booking** if authorized ✅  
6️ **Revalidate UI cache** to update the page ✅

Now, even if a **malicious user tries to delete via cURL**, they **cannot delete someone else’s reservation**. 🚀

 **Rendering a Loading Indicator During Server Actions**

* In **forms**, the useFormStatus hook provides a loading indicator.
* When calling a **server action from a button**, useFormStatus cannot be used.
* Instead, use the **useTransition** hook.

 **Introduction to useTransition**

* Introduced in **React 18** as a **concurrent feature**.
* Marks a state update as a **transition** to prevent UI blocking.
* Ensures UI stays **responsive** during re-renders.
* Useful for **state updates that might block the UI**.

 **Using useTransition in Next.js**

* Can mark a **server action** as a transition.
* Provides **visual feedback** while an operation is in progress.
* Helps maintain a **smooth user experience**.

 **How useTransition Works**

* useTransition() returns:
  1. **isPending** → Boolean flag indicating if a transition is in progress.
  2. **startTransition** → Function to wrap state updates.
* Example usage:
  1. Create a handleDelete function.
  2. Wrap the operation inside startTransition().
  3. Use isPending to show a **spinner** while the action is in progress.

 **Adding a Confirmation Alert**

* A confirm() alert ensures users don’t accidentally delete reservations.

 **Final Implementation**

* If isPending is **true**, display a **spinner** inside a centered container (mx-auto).
* If **false**, render the regular button content.

 **Behind the Scenes**

* Next.js uses **Suspense boundaries** to handle transitions.
* All **navigations in Next.js** are automatically wrapped in transitions.
* **Transitions** play a key role in **modern React applications**.

Code:

* const [isPending, startTransition] =
* useTransition();
* const handleDelete = (bookingId) => {
* startTransition(() =>
* deleteReservation(bookingId)
* );
* };

Using that isPending conditionally render

Updating Reservation

**Step 1: Create the Route**

Define a new dynamic route for editing reservations in **app/account/reservations/edit/[bookingId]/page.jsx**.

export default async function Page({ params }) {

  // CHANGE

  const { bookingId } = params;

  const { numGuests, observations, cabinId } =

    await getBooking(bookingId);

  const { maxCapacity } = await getCabin(cabinId);

return jsx

**2. Implement the Server Action**

**3. Reusable Submit Button**

**Refactor the submit button into a reusable component.**

**File: components/SubmitButton.js**

**javascript**

**Copy**

**'use client';**

**import { useFormStatus } from 'react-dom';**

**export default function SubmitButton({ children, pendingLabel }) {**

**const { pending } = useFormStatus();**

**return (**

**<button type="submit" disabled={pending}>**

**{pending ? pendingLabel : children}**

**</button>**

**);**

**}**

**4. Final Integration**

Integrate the SubmitButton component into the for

**Notes on Optimistic UI and useOptimistic Hook in React**

* **Optimistic UI**:
  + A technique to improve **perceived performance** by assuming an **asynchronous operation** (e.g., deletion) will succeed before it completes.
  + The UI updates immediately upon user action (e.g., clicking delete), while the operation runs in the background.
  + If the operation fails, the UI reverts to its previous state.
  + Goal: Enhances **user experience** by making the app feel faster and more responsive, avoiding loading spinners.
* **useOptimistic Hook**:
  + A modern React hook for implementing **optimistic updates**.
  + Allows displaying a new state while an **async action** is in progress.
  + Works only in **client components** (hooks are client-side).
  + **Arguments**:
    1. **Current state**: The initial state (e.g., bookings).
    2. **Update function**: Determines the **optimistic state** based on the current state and new data (e.g., booking ID for deletion).
  + **Returns**:
    1. **Optimistic state**: The state to display during the async operation.
    2. **Setter function**: Triggers the optimistic update (e.g., optimisticDelete).
* **Implementation Steps**:
  1. Move the **reservation list** from a **server component** to a **client component** (ReservationList.js).
  2. Pass **bookings** as a prop from the server to the client.
  3. Import and use the useOptimistic hook:
     + Pass the current state (bookings) and an **update function**.
     + Destructure the returned values: optimisticBookings and optimisticDelete.
  4. **Update function**:
     + Filters out the deleted booking optimistically.
     + Returns the new state to render immediately.
  5. Combine **optimistic deletion** with the actual **server action**:
     + Call optimisticDelete to update the UI.
     + Trigger the async server action (deleteReservation) in the background.
  6. Handle **failure**:
     + If the async operation fails, the state reverts to the original bookings.
* **Example Code**:

javascript

Copy

const [optimisticBookings, optimisticDelete] = useOptimistic(bookings, (currentBookings, bookingId) => {

return currentBookings.filter(booking => booking.id !== bookingId);

});

* + **Usage**:
    1. Call optimisticDelete(bookingId) to remove a booking optimistically.
    2. Call deleteReservation(bookingId) to perform the actual deletion.
* **Testing**:
  + Add **artificial delay** to simulate async operations.
  + Manually throw an error to test **state rollback** on failure.
* **Key Concepts**:
  + **Optimistic state**: Temporary state displayed during async operations.
  + **State rollback**: Reverting to the original state if the operation fails.
  + **Prop drilling**: Passing props through multiple components (e.g., handleDelete).
* **Comparison to useReducer**:
  + Similar to useReducer in managing state transitions.
  + optimisticDelete acts like a **dispatch function**, triggering state updates.
* **Additional Improvements**:
  + Add **loading indicators** for better user experience during navigation.
  + Use **loading.js** files in nested routes to display spinners.
* **Next Steps**:
  + Implement **reservation functionality** for cabins.

 The **automatic rollback** of the optimistic update when the actual update fails is a key feature of the useOptimistic hook. Let me explain **how this works** in detail.

**How Rollback Works in useOptimistic**

When you use the useOptimistic hook, React internally keeps track of two states:

1. **Actual State**: The real state of the data (e.g., the original bookings list).
2. **Optimistic State**: The temporary state that is displayed to the user during the async operation.

Here’s the step-by-step process of how rollback works:

**1. Initial State**

* The useOptimistic hook is initialized with the **actual state** (e.g., bookings).
* Initially, the **optimistic state** is the same as the actual state.

javascript

Copy

const [optimisticBookings, optimisticDelete] = useOptimistic(bookings, updateFunction);

**2. Optimistic Update**

* When the user triggers an action (e.g., clicking the delete button), the optimisticDelete function is called.
* This updates the **optimistic state** immediately, removing the booking from the UI.

javascript

Copy

optimisticDelete(bookingId); // Optimistically removes the booking

* At this point:
  + **Actual State**: Still contains the booking (e.g., bookings remains unchanged).
  + **Optimistic State**: The booking is removed temporarily.

**3. Async Operation (Actual Update)**

* After the optimistic update, the async operation (e.g., deleteReservation) is triggered.
* This operation runs in the background and attempts to update the actual state (e.g., delete the booking from the server).

javascript

Copy

await deleteReservation(bookingId); // Simulated server action

**4. Success Case**

* If the async operation succeeds:
  + The **actual state** is updated (e.g., the booking is removed from the server).
  + The **optimistic state** remains in sync with the actual state.
  + The UI continues to reflect the updated state.

**5. Failure Case**

* If the async operation fails (e.g., due to a network error or server issue):
  + React automatically **reverts the optimistic state** to match the **actual state**.
  + This means the booking that was optimistically removed will reappear in the UI.

**Why Does This Happen?**

* React internally compares the **optimistic state** with the **actual state** after the async operation completes.
* If the async operation fails, React knows that the optimistic update was incorrect and **rolls back** to the actual state