BEGINNER FRIENDLY

PHP + MySQL + AJAX + JSON

Build a Live Data CRUD Application — Step by Step

PHP 8 | MySQL | AJAX (Fetch API) | JSON | Bootstrap 5 | XAMPP

What you will build: A complete Employee Manager that adds, reads, updates, and deletes records — instantly, without page reloads.

Audience: Absolute beginners who know basic HTML and want to build their first dynamic, live web app.

Time to complete: ~2 hours, working through one page at a time.

Published by EGOTECHWORLD (PV00352315) | egotechworld.com | PHP / Python / Web Development Tutorials

TOC Table of Contents

Each chapter is self-contained. Read in order if you are new.

HOW TO USE THIS TUTORIAL

Open XAMPP, open a code editor (VS Code recommended), and type each code block by hand. Typing — not copying — is how your brain remembers syntax.

01 What is Live Data & Why AJAX?

Understanding the problem before we write any code.

The old way — full page reloads

In a traditional website, every time you click a button (like "Save" or "Delete"), the browser sends a request to the server and the entire page is rebuilt and re-downloaded. The screen flashes white. You lose your scroll position. It feels slow.

The live data way — AJAX

AJAX stands for Asynchronous JavaScript and XML, but today we use it with JSON instead of XML. The idea is simple:

1. JavaScript silently sends a request to a PHP file in the background.
2. PHP talks to MySQL and returns a small piece of data (in JSON format).
3. JavaScript updates only the part of the page that changed.
4. No reload. No flash. Instant feedback.

Visual comparison

Where you see this every day

Facebook's "Like" button, Gmail's inbox, Google search suggestions, YouTube comments — all use AJAX. After this tutorial, you can build the same kind of experience.

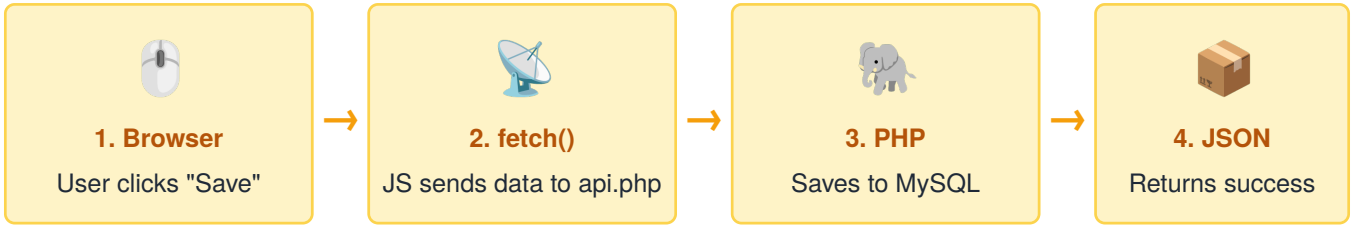
> **KEY TAKEAWAY**
>
> AJAX = JavaScript talks to PHP behind the scenes. PHP returns JSON. JavaScript updates the page. That's the whole pattern.

02 The Tech Stack Explained Visually

Four pieces, one job each.

Piece	Job	What it looks like
HTML + JS	Lives in the browser. Shows the form, captures clicks, sends AJAX requests.	`index.html` with a form and a table
PHP	Lives on the server. Receives AJAX requests, runs SQL, returns JSON.	`api.php`
MySQL	Lives on the server. Stores the actual data in tables.	A table called `employees`
JSON	The "language" PHP and JavaScript use to talk to each other.	`{"id":1,"name":"Suranjith"}`

The full request lifecycle



Then JavaScript receives the JSON and updates the page. Memorise this loop — every chapter from here on is just a variation of it.

What you need installed

- **XAMPP** — gives you Apache (web server), PHP, and MySQL all in one click.
- **A code editor** — VS Code is free and excellent.
- **A browser** — Chrome or Firefox. We will use the built-in DevTools to inspect AJAX traffic.

> **NO INTERNET HOSTING NEEDED**
>
> Everything in this tutorial runs on your own laptop. You only need a hosting account when you are ready to share your site with the world.

03 Installing XAMPP — Step by Step

A web server, PHP, and MySQL in one installer.

Step 1 — Download

Go to `apachefriends.org` and download the version for your operating system (Windows, Mac, or Linux). Pick the latest PHP 8 version.

Step 2 — Install

1. Run the installer. Accept all defaults.
2. When asked which components to install, tick at minimum: **Apache**, **MySQL**, **PHP**, and **phpMyAdmin**. You can untick the rest.
3. Install location: keep the default `C:\xampp` on Windows. Do not install inside `Program Files` — Windows permissions there cause problems.

Step 3 — Start the services

Open the XAMPP Control Panel. You will see this:

XAMPP Control Panel

Module	Status	Action
Apache	● Running	Stop
MySQL	● Running	Stop
FileZilla	○ Stopped	Start

Click **Start** next to *Apache* and *MySQL*. Both should turn green. If they do not, see Chapter 13.

Step 4 — Test it works

Open your browser and visit:

```
http://localhost/
```

You should see the XAMPP welcome dashboard. If you do — congratulations, you now have a full web server running on your computer.

Step 5 — Find your project folder

XAMPP serves files from a folder called `htdocs`. On Windows it is at:

```
C:\xampp\htdocs\
```

Anything you put inside this folder is reachable from your browser at `http://localhost/your-folder/`. We will create our project there in Chapter 5.

PORT 80 ALREADY IN USE?

If Apache will not start, another program (often Skype or IIS) is using port 80. In XAMPP, click Config → Apache (httpd.conf) and change `Listen 80` to `Listen 8080`. Then visit `http://localhost:8080/` instead.

04 Setting Up the MySQL Database

Create the table that will hold our employee records.

Step 1 — Open phpMyAdmin

With Apache and MySQL running, visit:

```
http://localhost/phpmyadmin
```

phpMyAdmin is a friendly web interface for managing MySQL. You will use it constantly.

Step 2 — Create a database

1. Click **New** in the left sidebar.
2. Database name: `live_crud`
3. Collation: `utf8mb4_unicode_ci`
4. Click **Create**.

Step 3 — Create the employees table

Click on the new `live_crud` database in the sidebar, then click the **SQL** tab at the top. Paste this SQL and click **Go**:

SQL — paste into phpMyAdmin SQL tab

```sql
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(150) NOT NULL,
    position VARCHAR(100) NOT NULL,
    salary DECIMAL(10,2) DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Step 4 — Add some test data

Still in the SQL tab, run:

SQL — sample data

```sql
INSERT INTO employees (name, email, position, salary) VALUES
('Suranjith', 'suranjith@example.com', 'Founder', 120000),
('Nimal Perera', 'nimal@example.com', 'Developer', 85000),
('Kamala Silva', 'kamala@example.com', 'Designer', 75000);
```

Click the **Browse** tab on the table — you should see your three rows.

WHAT THIS LOOKS LIKE IN PHPMYADMIN

id	name	email	position	salary
1	Suranjith	suranjith@example.com	Founder	120000.00
2	Nimal Perera	nimal@example.com	Developer	85000.00
3	Kamala Silva	kamala@example.com	Designer	75000.00

04b Understanding Each Column

Why we chose those data types.

Column	Type	What it means
`id`	INT AUTO_INCREMENT	A unique number for every row. MySQL fills it in automatically: 1, 2, 3, 4… You never set this yourself.
`name`	VARCHAR(100)	A short text field, up to 100 characters. NOT NULL means it cannot be empty.
`email`	VARCHAR(150)	Longer text for email addresses.
`position`	VARCHAR(100)	Job title — Developer, Designer, etc.
`salary`	DECIMAL(10,2)	A number with 2 decimal places. Use DECIMAL for money — never FLOAT, because float loses precision (75000.00 might become 74999.9999).
`created_at`	TIMESTAMP	Set automatically to the moment the row was inserted. Useful for sorting newest-first.

Why InnoDB and utf8mb4?

InnoDB is the modern MySQL engine — it supports transactions and foreign keys. Always use it.

utf8mb4 is a character set that supports every language, including Sinhala, Tamil, Chinese, and emojis 🎉. The older `utf8` in MySQL is broken (it cannot store emojis); always use `utf8mb4`.

SAVE THE SQL FOR LATER

Open VS Code, paste the CREATE TABLE statement into a file called `schema.sql`, and keep it with your project. You will reuse it when you move to a real hosting account.

05 Project Folder Structure

Organise files now to save pain later.

Inside `C:\xampp\htdocs\`, create a new folder called `live-crud`. Then create the files shown below — they will all be empty for now; we will fill them chapter by chapter.

```
📁 C:\xampp\htdocs\live-crud\
   ├── index.html          // the form + table the user sees
   ├── style.css           // our small bit of custom styling
   ├── app.js              // all AJAX / fetch logic
   ├── db.php              // database connection (used by api.php)
   ├── api.php             // receives AJAX, returns JSON
   └── schema.sql          // backup of the CREATE TABLE statement
```

Why this layout?

- **Separation of concerns.** HTML for structure, CSS for looks, JS for behaviour, PHP for the backend. Each file has one job.
- **One API file.** Beginners often create `add.php`, `edit.php`, `delete.php` — that scatters logic. We will use one `api.php` that decides what to do based on an `action` parameter. Cleaner and easier to debug.
- **Connection in its own file.** If the password ever changes, you edit one place.

Test the URL

Visit this in your browser:

```
http://localhost/live-crud/
```

Right now you will see a blank page or a directory listing — that is fine. Once we add `index.html`, the page will appear.

VS CODE SHORTCUT

In VS Code, open the entire `live-crud` folder with File → Open Folder. Then all six files appear in the left sidebar and you can switch between them with one click.

06 The Database Connection File

One small file that every other PHP file will include.

Open `db.php` in VS Code and type this exactly:

db.php

```php
<?php
// db.php — central database connection
// Included by api.php whenever we need to talk to MySQL

$host     = 'localhost';
$user     = 'root';       // XAMPP default
$pass     = '';           // XAMPP default — empty password
$database = 'live_crud';

try {
    $pdo = new PDO(
        "mysql:host=$host;dbname=$database;charset=utf8mb4",
        $user,
        $pass,
        [
            PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false,
        ]
    );
} catch (PDOException $e) {
    http_response_code(500);
    echo json_encode(['error' => 'DB connection failed: ' . $e->getMessage()]);
    exit;
}
```

What every line does

- **$host = 'localhost'** — the database is on the same machine.
- **$user = 'root', $pass = ''** — XAMPP's default credentials. On a real server you will change these.
- **PDO** — PHP Data Objects. Modern, secure way to talk to MySQL. We use it because it supports *prepared statements*, which prevent SQL injection (a major security hole).
- **ERRMODE_EXCEPTION** — if SQL fails, throw an exception we can catch. Without this, errors fail silently.
- **FETCH_ASSOC** — when we read rows, return them as associative arrays like `$row['name']`, not numeric keys.
- **EMULATE_PREPARES => false** — forces real prepared statements at the MySQL level. Safer.

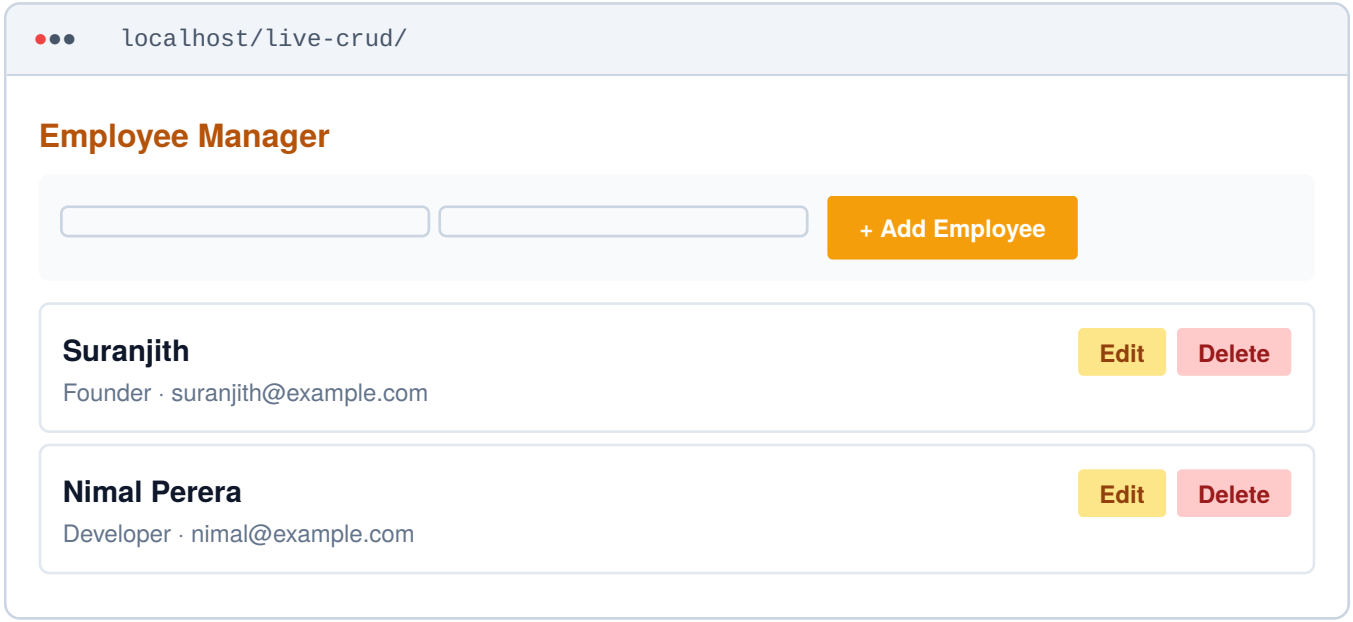
NEVER COMMIT REAL PASSWORDS TO GITHUB

If you push this project to GitHub later, change the password to something fake and add the real one in a separate `config.php` file that is listed in `.gitignore`.

07 Building the HTML Front-End

The page the user actually sees.

This is what we are building visually:



Open `index.html` and type this skeleton:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Employee Manager</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
  <link rel="stylesheet" href="style.css">
</head>
<body class="bg-light">

<div class="container py-4">
  <h1 class="mb-4 text-warning">Employee Manager</h1>

  <!-- Add / Edit form -->
  <div class="card mb-4">
    <div class="card-body">
      <form id="empForm" class="row g-2">
        <input type="hidden" id="empId">
        <div class="col-md-3"><input id="name" class="form-control" placeholder="Name"
required></div>
        <div class="col-md-3"><input id="email" type="email" class="form-control"
placeholder="Email" required></div>
        <div class="col-md-2"><input id="position" class="form-control"
placeholder="Position"></div>
        <div class="col-md-2"><input id="salary" type="number" class="form-control"
```

```
placeholder="Salary"></div>
        <div class="col-md-2"><button class="btn btn-warning w-100">Save</button></div>
      </form>
    </div>
  </div>

  <!-- Live data table — JS will fill this -->
  <div id="empTable">Loading…</div>
</div>

<script src="app.js"></script>
</body>
</html>
```

07b HTML Walk-Through

Why each piece is there.

The form

Notice every input has an **id** like `id="name"`. JavaScript will use these IDs to read the values. The `empId` hidden input is empty when adding a new employee, but holds the ID number when editing.

The empty div

`<div id="empTable">Loading…</div>` is intentionally empty. JavaScript will fetch employees from `api.php` and inject HTML into this div. The "Loading…" text is what the user sees for the half-second before the fetch completes.

Bootstrap from CDN

We load Bootstrap 5 from a CDN so we get a professional-looking table, form, and buttons with zero CSS work. The `btn-warning` and `text-warning` classes use Bootstrap's amber colour, matching your brand.

Add a tiny custom style

Open `style.css` and add:

style.css

```
body { background: #f8fafc; }
h1   { color: #b45309; font-weight: 700; }
.card { border: 0; box-shadow: 0 2px 8px rgba(0,0,0,.05); }
.spin { animation: spin 0.6s linear infinite; display: inline-block; }
@keyframes spin { to { transform: rotate(360deg); } }
```

Test the page now

Visit `http://localhost/live-crud/`. You should see the form and the word "Loading…". Nothing happens yet because we have not written the JavaScript. That is next.

> **YOU SHOULD SEE THIS**
>
> A clean Bootstrap form with four input boxes, an amber "Save" button, and grey "Loading…" text below it.

08 CREATE — Add a New Employee

The C in CRUD. Our first AJAX call.

Step 1 — Add the PHP endpoint

Open `api.php` and start with this structure:

api.php — initial structure

```php
<?php
require 'db.php';
header('Content-Type: application/json');

$action = $_REQUEST['action'] ?? '';

switch ($action) {

    case 'create':
        $stmt = $pdo->prepare(
            "INSERT INTO employees (name, email, position, salary)
             VALUES (?, ?, ?, ?)"
        );
        $stmt->execute([
            $_POST['name'],
            $_POST['email'],
            $_POST['position'],
            $_POST['salary'],
        ]);
        echo json_encode(['success' => true, 'id' => $pdo->lastInsertId()]);
        break;

    default:
        echo json_encode(['error' => 'Unknown action']);
}
```

Step 2 — The JavaScript that calls it

Open `app.js` and add:

app.js — handle form submit

```javascript
const form = document.getElementById('empForm');

form.addEventListener('submit', async (e) => {
    e.preventDefault();  // stop the normal page reload

    const data = new FormData();
    data.append('action',   'create');
    data.append('name',     name.value);
    data.append('email',    email.value);
    data.append('position', position.value);
    data.append('salary',   salary.value);
```

```
    const res  = await fetch('api.php', { method: 'POST', body: data });
    const json = await res.json();

    if (json.success) {
        form.reset();
        loadEmployees();   // refresh the table — coming in Chapter 9
    } else {
        alert('Error: ' + json.error);
    }
});
```

08b What Just Happened — Line by Line

On the PHP side

- **require 'db.php'** — gives us the `$pdo` connection.
- **header('Content-Type: application/json')** — tells the browser "what I am about to send you is JSON".
- **$_REQUEST['action']** — reads the `action` parameter from either GET or POST. We will use `switch` to route to different blocks.
- **prepare(...) with ? marks** — this is a prepared statement. The values you pass to `execute([...])` are sent *separately* from the SQL, so an attacker cannot inject malicious SQL through the form. **Always use this pattern.**
- **json_encode([...])** — turns a PHP array into a JSON string the browser can read.

On the JavaScript side

- **e.preventDefault()** — without this line, the form would do a normal POST and reload the page. We do not want that.
- **FormData** — a built-in browser object that builds a POST body for us. Each `append()` adds one field.
- **await fetch(...)** — the modern AJAX call. `await` pauses until the server responds, but only this function pauses; the rest of the page stays responsive.
- **await res.json()** — parses the JSON string into a JavaScript object.
- **form.reset()** — empties all the inputs after a successful save.

Test it now

1. Reload the page.
2. Fill in the form and click Save.
3. Open phpMyAdmin → employees table → Browse. The new row should be there!

USE DEVTOOLS TO WATCH AJAX

Press F12 → Network tab → tick "Fetch/XHR". Now click Save and you will see `api.php` appear in the list. Click it to inspect what was sent and what came back. This is your number-one debugging tool.

VALIDATION MATTERS

Right now PHP trusts whatever the form sends. In a real app you would check the email is valid, the salary is a number, etc. We will add validation in a later mini-chapter.

09 READ — Display Live Data

Pull rows from MySQL and render them as a table.

Step 1 — Add the read action to api.php

Inside the `switch` block, add a new `case` above the `default`:

api.php — add read action

```php
    case 'read':
        $stmt = $pdo->query(
            "SELECT * FROM employees ORDER BY id DESC"
        );
        echo json_encode($stmt->fetchAll());
        break;
```

That is it. Three lines. `fetchAll()` returns every row as an array of associative arrays, and `json_encode()` turns that into JSON like:

```
[
  {"id":3, "name":"Kamala", "email":"kamala@...", "position":"Designer",
"salary":"75000.00"},
  {"id":2, "name":"Nimal",  ...},
  {"id":1, "name":"Suranjith", ...}
]
```

Step 2 — Test the API directly

Before writing any JavaScript, visit this URL in your browser:

```
http://localhost/live-crud/api.php?action=read
```

You should see raw JSON. **If you see this, your PHP and database are working correctly.** If you see an error or blank page, fix it now before going further (see Chapter 13).

Step 3 — JavaScript to render the table

Add this to `app.js`:

app.js — load and render

```javascript
async function loadEmployees() {
    const res  = await fetch('api.php?action=read');
    const rows = await res.json();

    if (rows.length === 0) {
        empTable.innerHTML = '<p class="text-muted">No employees yet.</p>';
```

```
        return;
    }

    let html = `<table class="table table-striped">
        <thead><tr><th>ID</th><th>Name</th><th>Email</th>
                  <th>Position</th><th>Salary</th><th></th></tr></thead><tbody>`;

    rows.forEach(r => {
        html += `<tr>
            <td>${r.id}</td>
            <td>${r.name}</td>
            <td>${r.email}</td>
            <td>${r.position}</td>
            <td>Rs. ${r.salary}</td>
            <td>
              <button class="btn btn-sm btn-warning" onclick="editEmp(${r.id})">Edit</button>
              <button class="btn btn-sm btn-danger"  onclick="deleteEmp(${r.id})">Delete</
button>
            </td></tr>`;
    });

    html += '</tbody></table>';
    empTable.innerHTML = html;
}

// load the table the moment the page opens
loadEmployees();
```

09b How the Render Works

Template literals

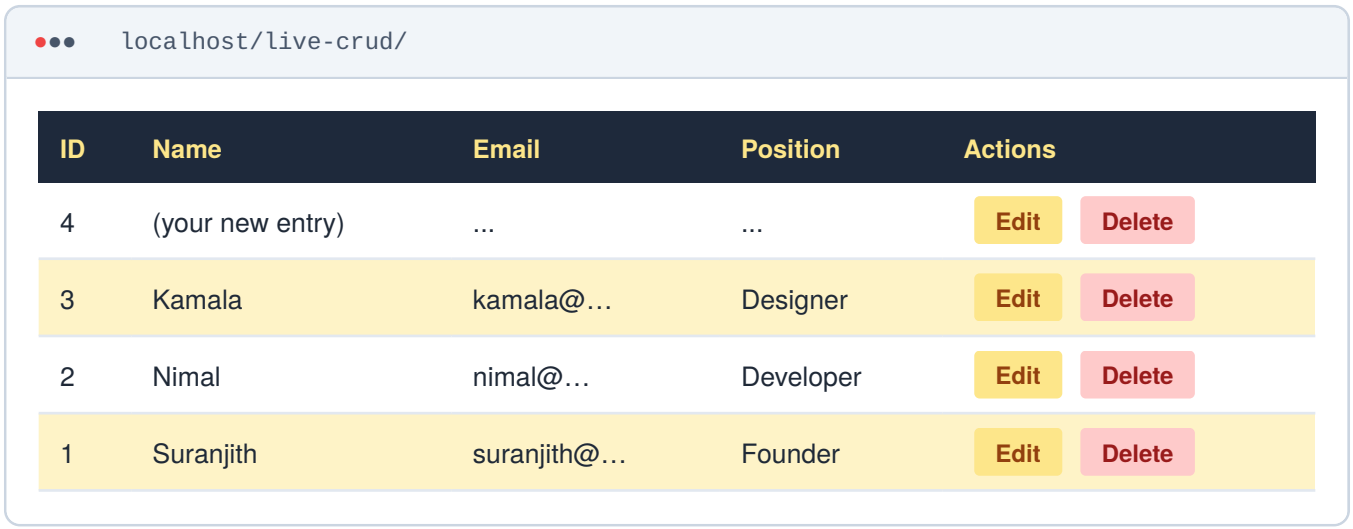
The backticks `` `...` `` create a "template literal" — a string that lets you embed variables with `${variable}`. Much cleaner than the old `'a' + b + 'c'` style.

Building HTML in a string

We start with the table opening tags, loop through every row appending a `<tr>`, then close the table. Finally we set `empTable.innerHTML` in one shot. Setting innerHTML once is faster than appending row by row.

Reload the browser now

You should see your employees in a Bootstrap-styled striped table, newest first. Try adding a new one with the form — it appears at the top instantly. **That is live data.**



SECURITY NOTE: XSS

We are inserting database content directly into innerHTML. If a user typed `<script>alert('hi')</script>` as their name, it would execute. In production, escape the values with a small helper like `esc(s)` that replaces `<` with `&lt;` before inserting. We will keep things simple for this beginner tutorial, but remember this for live sites.

WHY DESC ORDERING?

`ORDER BY id DESC` puts the newest row at the top. When the user adds an employee, they immediately see it at position #1 — instant visual feedback that the save worked.

10 UPDATE — Edit Records Without Reload

Click "Edit" → form fills with existing data → Save updates the row.

Step 1 — Add two PHP cases

We need *two* backend actions: one to fetch a single row's data into the form, and one to save the changes.

api.php — add these cases

```php
    case 'get':
        $stmt = $pdo->prepare("SELECT * FROM employees WHERE id = ?");
        $stmt->execute([$_GET['id']]);
        echo json_encode($stmt->fetch());
        break;

    case 'update':
        $stmt = $pdo->prepare(
            "UPDATE employees
             SET name=?, email=?, position=?, salary=?
             WHERE id = ?"
        );
        $stmt->execute([
            $_POST['name'],
            $_POST['email'],
            $_POST['position'],
            $_POST['salary'],
            $_POST['id'],
        ]);
        echo json_encode(['success' => true]);
        break;
```

Step 2 — JavaScript to load a row into the form

app.js — editEmp function

```javascript
async function editEmp(id) {
    const res = await fetch('api.php?action=get&id=' + id);
    const r   = await res.json();

    empId.value    = r.id;
    name.value     = r.name;
    email.value    = r.email;
    position.value = r.position;
    salary.value   = r.salary;

    name.focus();   // nice UX touch
    window.scrollTo({ top: 0, behavior: 'smooth' });
}
```

Step 3 — Make the Save button decide: create or update?

Replace the original submit handler with this smarter version:

app.js — updated submit handler

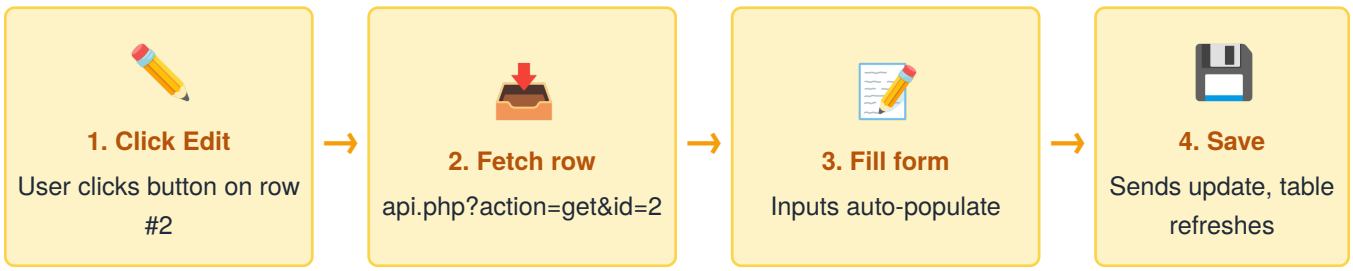
```
form.addEventListener('submit', async (e) => {
    e.preventDefault();

    const isEdit = empId.value !== '';
    const data   = new FormData();
    data.append('action',   isEdit ? 'update' : 'create');
    if (isEdit) data.append('id', empId.value);
    data.append('name',     name.value);
    data.append('email',    email.value);
    data.append('position', position.value);
    data.append('salary',   salary.value);

    await fetch('api.php', { method: 'POST', body: data });

    form.reset();
    empId.value = '';   // reset hidden field too
    loadEmployees();
});
```

10b The Edit Flow Visualised



The hidden ID field — the secret sauce

The trick that makes one form do both Add and Edit is the hidden `empId` field:

- Empty → we are adding a new row → action is `create`.
- Has a number → we are editing an existing row → action is `update` and we send the id.

Test it

1. Click Edit on any row.
2. The form should fill with that row's data.
3. Change the salary and click Save.
4. The table should refresh with the new value — no page reload.

BONUS UX: A "CANCEL EDIT" BUTTON

Add a small button next to Save that runs `form.reset(); empId.value=''` — useful when the user clicks Edit by mistake.

WHY TWO PHP CASES INSTEAD OF ONE?

`get` reads a single row to populate the form. `update` writes the changes back. They have different SQL and different inputs, so they get their own cases. Resist the urge to merge them.

11 DELETE — Remove with Confirmation

The shortest CRUD operation, but the most dangerous.

Step 1 — Add the PHP delete case

api.php — delete case

```
    case 'delete':
        $stmt = $pdo->prepare("DELETE FROM employees WHERE id = ?");
        $stmt->execute([$_POST['id']]);
        echo json_encode(['success' => true]);
        break;
```

Step 2 — JavaScript with confirmation

app.js — deleteEmp function

```
async function deleteEmp(id) {
    if (!confirm('Delete this employee? This cannot be undone.')) return;

    const data = new FormData();
    data.append('action', 'delete');
    data.append('id',     id);

    await fetch('api.php', { method: 'POST', body: data });
    loadEmployees();
}
```

Why the confirm dialog matters

Delete cannot be undone. `confirm()` is a built-in browser popup that returns `true` if the user clicks OK and `false` if they click Cancel. The `if (!...) return` pattern stops the function immediately if they cancel.

Test it

1. Click Delete on any row.
2. Confirm in the popup.
3. The row disappears instantly, no reload.

SOFT DELETE IS USUALLY BETTER

In real apps you rarely truly delete — you mark a row as deleted by adding a `deleted_at TIMESTAMP NULL` column and only show rows where it is NULL. That way you can undo and audit. Worth knowing for your next project.

YOU NOW HAVE A FULL CRUD APP

Create, Read, Update, Delete — all over AJAX, all returning JSON, all without a page reload. This pattern works for any database table you ever build: tasks, products, jobs, students, orders.

12 Live Auto-Refresh — True Live Data

When two people use the app, both should see each other's changes.

Right now, if you open the app in two browser tabs and add an employee in tab A, tab B will not show it until you reload. Let us fix that.

The simplest approach — polling

Tell JavaScript to call `loadEmployees()` every few seconds, automatically.

app.js — add at the bottom

```
// Poll for changes every 5 seconds
setInterval(loadEmployees, 5000);
```

That is it. One line. `setInterval(fn, ms)` runs the function every `ms` milliseconds.

The smarter approach — only refresh if something changed

Polling every 5 seconds is wasteful — most of the time nothing has changed. A common optimisation is to ask the server "what is your last update timestamp?" and only re-render if it is newer than what we last saw.

api.php — add a tiny status case

```
    case 'last_update':
        $stmt = $pdo->query("SELECT MAX(created_at) AS t FROM employees");
        echo json_encode($stmt->fetch());
        break;
```

app.js — smart polling

```
let lastSeen = '';

async function checkForUpdates() {
    const res = await fetch('api.php?action=last_update');
    const r   = await res.json();
    if (r.t !== lastSeen) {
        lastSeen = r.t;
        loadEmployees();
    }
}
setInterval(checkForUpdates, 5000);
```

When to use real-time tech instead

Polling is perfect for dashboards, score tickers, and admin panels. For a chat app or live multiplayer game where you need sub-second updates, look into **WebSockets** (Ratchet for PHP, Socket.IO for Node.js) or **Server-Sent Events**. Save those for after you have mastered AJAX.

TEST THE LIVE FEEL

Open the app in two browser windows side by side. Add an employee in one. Within 5 seconds, it appears in the other. That is real live data.

13 Common Errors & How to Fix Them

Beginners hit these constantly. You are not alone.

What you see	What it means	Fix
Apache will not start in XAMPP	Port 80 is taken (often by Skype or IIS)	Change Apache to port 8080 in httpd.conf, or stop the offending program.
`Access denied for user 'root'`	Wrong DB password	XAMPP default is empty password. Check `db.php`.
JSON error in browser console	PHP printed an error before the JSON	Open `api.php?action=read` directly — read the raw output and fix the error.
"Loading…" never goes away	JavaScript error or PHP returned bad JSON	Open DevTools → Console tab. Read the red error message.
Form submits but page reloads	You forgot `e.preventDefault()`	Add it as the first line inside the submit handler.
Edit fills the form with "undefined"	Typo in input `id` attributes	Make sure HTML `id="name"` matches JS `name.value`.
404 on api.php	File is in the wrong folder	Confirm it is in `C:\xampp\htdocs\live-crud\api.php`.
Special characters show as `?`	Wrong charset	Make sure the PDO DSN includes `charset=utf8mb4` and the table uses `utf8mb4`.
SQL error appears as JSON	SQL syntax mistake	Copy the error text into Google. 99% of beginner SQL errors have a Stack Overflow answer.

The universal debugging routine

1. **Open DevTools (F12)** → Network tab → tick "Fetch/XHR".
2. Trigger the action.
3. Click the failing request → look at the Response tab.
4. If the response is HTML instead of JSON, PHP threw an error. Read it.
5. If the response is empty, check XAMPP is running.

ADD TEMPORARY DEBUGGING

Inside any PHP case, add `error_log(print_r($_POST, true));` — it writes to `C:\xampp\apache\logs\error.log` so you can see exactly what arrived.

14 What to Build Next

You now own a pattern. Reuse it everywhere.

Project ideas using the exact same stack

- **Task manager** — replace the employees table with `tasks(id, title, done, due_date)`. Add a checkbox to mark tasks done.
- **Job board admin** — manage your career postings from a custom panel instead of phpMyAdmin.
- **Comment moderation** — list pending comments, approve or delete each one with a click.
- **Course enrollment tracker** — students register, you mark them paid, the table shows live status.
- **Inventory system** — for a small shop. Add stock-in, stock-out, and a search box that filters as you type.

Skills to add next, in order

1. **Search and pagination** — when the table has 500 rows, you need a search box and Next / Previous buttons. Both are AJAX.
2. **Server-side validation** — never trust the form. Check email format, required fields, and length on the PHP side too.
3. **File uploads** — let users upload a profile photo. Store the filename in MySQL, the file on disk.
4. **Login system** — sessions, password hashing with `password_hash()`, and protecting `api.php` so only logged-in users can call it.
5. **Move to PDO transactions** — when you save data across multiple tables (e.g. an order with items), wrap them in a transaction so they all succeed or all fail.

Deploying to a real host

When you upload to LankaHost or any shared hosting:

1. Create a database in cPanel and note the name, user, and password.
2. Update `db.php` with those values.
3. Upload all files via FTP to `public_html/` (or a subfolder).
4. Run `schema.sql` in the host's phpMyAdmin to create the table.
5. Visit your domain. Done.

A final word

You learned more than CRUD. You learned the **request → JSON → render** pattern that powers nearly every modern web app, from Twitter to Trello. Every new feature you build from now on is a variation on this loop.

BUILT WITH THIS PATTERN

Job boards, dashboards, learning management systems, e-commerce admin panels, CRMs, ticketing systems, content moderation tools, internal company tools.

— End of tutorial —

Published by EGOTECHWORLD (PV00352315) · egotechworld.com

For more PHP, Python, and web development guides, visit our courses page.