PERSONAL MULTI-TECHNOLOGY PROJECTS

Personal Catalogue: PHP & MySQL

A real, working catalogue for books, CDs, DVDs, and Blu-rays — a shared MySQL schema, PDO with native prepared statements, tags, real search, an N+1-avoiding list view, fast bulk entry, and a real deployment onto an existing site, closing with barcode scanning deliberately scoped out as an honest future feature rather than a day-one requirement.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: FOUNDATIONS & THE REAL PHP/MYSQL SETUP

Personal Catalogue: PHP & MySQL

Chapter 1 · Project Overview & PHP/MySQL Setup

This is one of four courses building the exact same personal cataloguing tool in four genuinely different architectures — Personal Catalogue (React + Express + MongoDB), Personal Catalogue (React & Firebase), and Personal Catalogue (Django & PostgreSQL) are its siblings. This course's own answer is deliberately the simplest, fastest one to stand up: classic LAMP, chosen specifically because this project has a real deadline.

What the App Actually Does

The shared spec every course in the quartet builds toward:

- **Catalogue four kinds of item** — books, CDs, DVDs, and Blu-rays — so it's possible to quickly check "do I already own this CD?" or "what Python books do I have?"
- **Tag books** with free-form labels (a FastAPI book might get Python, Web Development, Programming, Web Frameworks) so related books can be found by topic later.
- **Add items manually** for the first release — title, author/artist, format-specific details, tags where relevant.
- **Search and browse** the collection to answer the "do I have this?" question quickly.

Barcode scanning is explicitly out of scope for this first release — a real, obvious way to add items even faster, but deliberately deferred rather than allowed to hold up a release that needs to ship quickly. This course closes with a real, concrete plan for adding it later (Chapter 9), rather than pretending it isn't wanted at all.

Why PHP + MySQL for This One

Of the four variants in this set, PHP + MySQL is the one built around genuine speed of delivery. No build step, no bundler, no separate frontend framework to stand up before a single item can be added — a PHP file saved to a folder and opened in a browser is already a running page. MySQL's own real relational structure is also a natural fit for this project's own data: books, CDs, DVDs, and Blu-rays share several real fields (title, format, year) while differing in

others, and tags need a genuine many-to-many relationship against books specifically — both comfortably expressed as real, normalized MySQL tables, covered in full in Chapter 2.

> **THIS COURSE ASSUMES REAL PHP/MYSQL GROUNDWORK ALREADY COVERED ELSEWHERE**
>
> This site's own PHP Intermediate course already covers PDO, prepared statements, and basic CRUD in real depth (Chapter 4 there). This course doesn't re-teach that material — it assumes it, and moves straight into this catalogue's own real schema and features.

A Real Deadline, Not Just a Framing Device

This project genuinely needs a working first release quickly. That constraint shapes real decisions throughout this course — plain PHP over a framework, MySQL's own native tooling over an ORM, and a deliberately un-fancy add-item form before any polish — not because those are always the "correct" choices, but because they're the right choices *given this specific, real time pressure*. Chapter 8 covers where the corners were deliberately cut, and what a slower, more careful version might have done differently.

Setting Up: PHP, MySQL & a Working Connection

A local PHP + MySQL environment (XAMPP, MAMP, or a native install of PHP and MySQL Server) is assumed from here on. Confirm both are working before continuing:

```
# Confirm PHP is installed and check the version
php -v

# Start MySQL's own command-line client
mysql -u root -p
```

Create the database this whole course builds on:

```
CREATE DATABASE catalogue CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

`utf8mb4` is a deliberate choice, not the MySQL default — it correctly stores genuinely any real title or author name a personal library might contain, including accented characters and beyond, which the older `utf8` encoding in MySQL doesn't fully cover.

A Single Config File for the Database Connection

Every later chapter's own PHP code connects through one shared file, kept outside the public web root where possible:

```php
<?php
// config.php
$host = 'localhost';
$db   = 'catalogue';
$user = 'root';
$pass = ''; // set a real password in production
$charset = 'utf8mb4';

$dsn = "mysql:host=$host;dbname=$db;charset=$charset";
$options = [
    PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES   => false,
];

try {
    $pdo = new PDO($dsn, $user, $pass, $options);
} catch (PDOException $e) {
    throw new PDOException($e->getMessage(), (int)$e->getCode());
}
```

`PDO::ATTR_EMULATE_PREPARES => false` is worth calling out specifically: it makes PDO use MySQL's own real, native prepared statements rather than PHP-side emulation — the genuinely correct setting for real SQL-injection protection, and one that's easy to leave at its default (true) without realizing the difference matters.

> **NOTHING TO BUILD BEFORE THE FIRST PAGE LOADS**

Unlike the Django and React-based siblings in this set, there's no project scaffolding step here at all — `config.php` plus a working MySQL connection is the entire "setup," and the very next chapter starts writing real schema. This is exactly the kind of speed this variant was chosen for.

DON'T SKIP THE CHARSET

Creating the database without an explicit `utf8mb4` charset silently falls back to whatever the local MySQL install's own default happens to be — sometimes still `latin1` on older setups. A book title or author name containing a character outside that older encoding's own range will either fail to insert or, worse, get silently mangled. Setting the charset explicitly, as above, avoids the problem outright rather than discovering it later.

Where This Course Is Headed

A real MySQL schema for items, item types, and a tags join table (Chapter 2); the CRUD pages for adding, editing, and deleting an item (Chapter 3); one form handling several genuinely different item shapes (Chapter 4); tags — attaching, managing, and filtering by tag on books (Chapter 5); real search across the whole catalogue (Chapter 6); the list and detail pages (Chapter 7); styling (Chapter 8); deployment, plus a real, concrete plan for barcode scanning as the next real phase (Chapter 9); and a capstone on integrating this catalogue into the existing Astro-based site (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why barcode scanning was deliberately deferred rather than built into this first release, and what real project constraint drove that decision.

View solution

EXERCISE 2

Explain what `PDO::ATTR_EMULATE_PREPARES => false` actually changes, and why it matters for this project's own real security.

View solution

EXERCISE 3

Create the `catalogue` database yourself with the correct `utf8mb4` charset, and write out a one-sentence explanation of why the charset was set explicitly rather than left at MySQL's own default.

View solution

CHAPTER 1 QUICK REFERENCE

- **The shared app** — a personal catalogue for books/CDs/DVDs/Blu-rays, tags on books, manual entry first, barcode scanning deferred
- **Why this variant** — classic LAMP, chosen for genuine speed of delivery given the project's own real deadline
- **Database** — MySQL, created with an explicit `utf8mb4` charset
- **Connection** — one shared `config.php` using PDO, with `ATTR_EMULATE_PREPARES` set to `false`
- **Assumed groundwork** — PDO/prepared statements/CRUD, already covered in PHP Intermediate Chapter 4
- **Next chapter:** Data Modeling — a real MySQL schema for items, item types, and a tags join table

CHAPTER 2: DATA MODELING

Personal Catalogue: PHP & MySQL

Chapter 2 · Data Modeling

Chapter 1 got a connection open and a database created. This chapter designs the real tables that connection will actually talk to — a schema that has to hold four genuinely different kinds of item (books, CDs, DVDs, Blu-rays), let books carry an arbitrary number of tags, and stay fast and simple to query given this project's own real deadline.

The Shape of the Problem

A book has an author. A CD has an artist. A DVD or Blu-ray has a director. All four have a title, a format, and a year. There are two honest ways to model that in MySQL:

- **One table per item type** — a real, fully normalized `books` table with an `author` column, a real `music_cds` table with an `artist` column, and so on. Every field is exactly the right shape for its type, with no unused columns anywhere.
- **One shared `items` table** — a single table covering every item type, with a generic `creator` column standing in for author/artist/director depending on the row's own type.

This course uses the second approach, and it's worth being honest about why: the app's own core real question is "*do I already own this?*" — a single search across every item type at once. With four separate tables, answering that means a real four-way `UNION` query every single time. With one shared table, it's a single `WHERE` clause. Given the real time pressure this project starts with, the shared-table design is the one that lets the search feature (Chapter 6) stay genuinely simple.

> **THIS IS A REAL, DELIBERATE TRADE-OFF**
>
> A fully normalized per-type schema is arguably "more correct" in a formal database-design sense — every column would genuinely apply to every row. The shared-table design accepts a few columns that are meaningless for some rows (a DVD's own `author` would always be `NULL`) in exchange for a dramatically simpler search query. That's the right call here specifically because search-across-everything is this app's own central feature — it wouldn't necessarily be the right call for every project.

Table 1: item_types

Rather than hard-coding "book", "cd", "dvd", "bluray" as a MySQL `ENUM`, this course uses a small lookup table instead:

```
CREATE TABLE item_types (
    id TINYINT UNSIGNED PRIMARY KEY,
    name VARCHAR(20) NOT NULL UNIQUE
);

INSERT INTO item_types (id, name) VALUES
    (1, 'book'),
    (2, 'cd'),
    (3, 'dvd'),
    (4, 'bluray');
```

An `ENUM` would work for a fixed, never-changing list — but this collection is never really fixed. Adding "vinyl" or "video game" later with a lookup table is one `INSERT` statement; adding it to an `ENUM` means an `ALTER TABLE` that MySQL has to rewrite the underlying column definition for. The lookup table costs one extra join later, in exchange for that flexibility staying cheap.

Table 2: items

The main table. Every item in the catalogue — regardless of type — is one row here:

```
CREATE TABLE items (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    item_type_id TINYINT UNSIGNED NOT NULL,
    title VARCHAR(255) NOT NULL,
    creator VARCHAR(255),
    format_detail VARCHAR(100),
    release_year SMALLINT UNSIGNED,
    notes TEXT,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (item_type_id) REFERENCES item_types(id)
);
```

A few of these columns are deliberately generic, standing in for something different depending on the item's own type:

COLUMN	BOOK	CD	DVD / BLU-RAY
`creator`	Author	Artist	Director
`format_detail`	Hardcover / paperback	Single / album	Region code
`release_year`	Publication year	Release year	Release year

`release_year` is deliberately nullable — the exact publication year isn't always known or worth tracking down on day one, and a real project under time pressure shouldn't make a field mandatory just because it would be nice to have. `created_at`, on the other hand, is genuinely useful and free: MySQL fills it in automatically, giving a real "recently added" ordering with zero extra application code.

Table 3 & the Join: tags and item_tags

Only books get tags in this project's own spec, but the tags themselves — Python, Web Development, Programming — are reusable across many books, and any one book can carry several. That's a genuine many-to-many relationship, which MySQL models with a real join table sitting between the two sides:

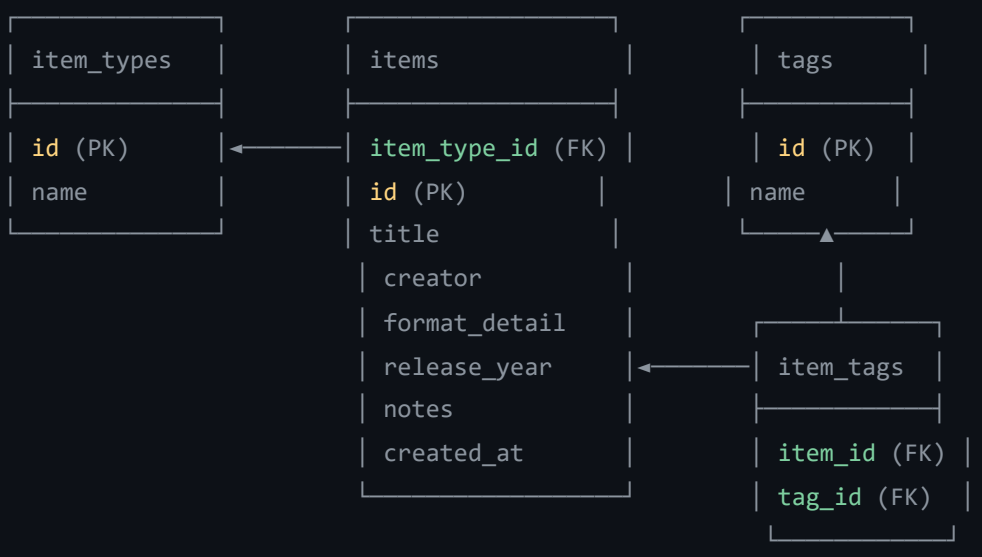
```
CREATE TABLE tags (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE item_tags (
    item_id INT UNSIGNED NOT NULL,
    tag_id INT UNSIGNED NOT NULL,
    PRIMARY KEY (item_id, tag_id),
    FOREIGN KEY (item_id) REFERENCES items(id) ON DELETE CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(id) ON DELETE CASCADE
);
```

`item_tags` has no `id` column of its own — its **composite primary key**, `(item_id, tag_id)`, is exactly the pairing it exists to record, and it also does useful work as a constraint: MySQL will reject an attempt to attach the exact same tag to the exact same item twice, since that pairing would violate the primary key's own uniqueness.

WHY ITEM_TAGS ISN'T RESTRICTED TO BOOKS ONLY

The join table doesn't actually stop a CD or DVD from being tagged — nothing in this schema enforces "tags only apply to `item_type_id = 1`". That's a deliberate simplification, not an oversight: enforcing it at the database level would need either a trigger or an application-level check, and the real spec only ever asks for tags on books. Leaving the schema itself untyped here costs nothing today and quietly allows the feature to extend later if that requirement ever changes.

The Full Schema, in One Picture



FOREIGN KEYS NEED AN ENGINE THAT SUPPORTS THEM

`FOREIGN KEY` constraints only actually work under MySQL's InnoDB storage engine — the modern default for a fresh install, but worth checking if this database ever gets migrated onto an older or differently-configured server. Under the older MyISAM engine, the `FOREIGN KEY` clauses above are silently accepted as valid syntax but never actually enforced, which would let orphaned rows creep into `items` or `item_tags` with no error raised anywhere.

Hands-On Exercises

EXERCISE 1

Run all four `CREATE TABLE` statements from this chapter against the `catalogue` database created in Chapter 1, in the correct order (the tables with foreign keys must be created after the tables they reference).

View solution

EXERCISE 2

Explain, in your own words, why a shared `items` table with a generic `creator` column was chosen over four separate per-type tables, and name the one real feature that decision makes simpler.

View solution

EXERCISE 3

Write and run a single SQL `INSERT` statement that adds one book, one tag, and one row into `item_tags` linking the two together — then write a `SELECT` with a join that lists the book's title alongside its tag's name.

View solution

CHAPTER 2 QUICK REFERENCE

- **items** — one shared table for every item type, with a generic `creator` column standing in for author/artist/director
- **item_types** — a lookup table (not an `ENUM`) so new item types can be added with a single `INSERT`
- **tags** + **item_tags** — a real many-to-many relationship, with `item_tags`' own composite primary key preventing duplicate tag assignments
- **Design trade-off** — a slightly less "pure" shared-table schema, chosen specifically because it keeps the app's own core search feature a single query
- **Engine note** — foreign keys require InnoDB; they're silently unenforced under MyISAM

- **Next chapter:** Building the CRUD Pages — add, edit, and delete an item against this real schema

CHAPTER 3: BUILDING THE CRUD PAGES

Personal Catalogue: PHP & MySQL

Chapter 3 · Building the CRUD Pages

Chapter 2 built the real table structure. This chapter builds the first pages that actually write to it — adding an item, editing an existing one, and deleting one — using the PDO connection from Chapter 1 and the prepared-statement discipline this site's own PHP Intermediate course (Chapter 4) already covers in depth. Tags and search come later (Chapters 5 and 6); this chapter deliberately stays scoped to the `items` table alone.

> **ASSUMED GROUNDWORK**
>
> If PDO prepared statements, named placeholders, or basic CRUD patterns aren't already comfortable, this course assumes that foundation was covered in PHP Intermediate, Chapter 4. This chapter builds on that material directly rather than re-teaching it.

A Minimal Item-Type Picker

Every add/edit form needs to offer the four real item types from `item_types` as a dropdown, rather than hard-coding them as literal strings in the form's own markup. A tiny shared function keeps that list in one place:

`functions.php`

```php
<?php
require_once __DIR__ . '/config.php';

function get_item_types(PDO $pdo): array {
    $stmt = $pdo->query('SELECT id, name FROM item_types ORDER BY id');
    return $stmt->fetchAll();
}
```

Since `item_types` almost never changes at runtime, a plain unparameterised `query()` is fine here — `prepare()` / `execute()` only really earns its keep when real, external data is involved, which this call doesn't have.

Adding an Item

`add_item.php` does two jobs depending on the request method: on a `GET` request it shows the empty form; on a `POST` request it validates and inserts the submitted data. This is the same "one file, two branches" pattern PHP Fundamentals' own Chapter 8 already introduced for `$_SERVER['REQUEST_METHOD']`.

`add_item.php`

```
<?php
require_once __DIR__ . '/functions.php';

$errors = [];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $itemTypeId  = (int) ($_POST['item_type_id'] ?? 0);
    $title       = trim($_POST['title'] ?? '');
    $creator     = trim($_POST['creator'] ?? '');
    $formatDetail = trim($_POST['format_detail'] ?? '');
    $releaseYear = trim($_POST['release_year'] ?? '');
    $notes       = trim($_POST['notes'] ?? '');

    if ($title === '') {
        $errors[] = 'Title is required.';
    }
    if ($itemTypeId < 1) {
        $errors[] = 'Please choose an item type.';
    }

    if (empty($errors)) {
        $stmt = $pdo->prepare('
            INSERT INTO items (item_type_id, title, creator, format_detail, release_yea
            VALUES (:item_type_id, :title, :creator, :format_detail, :release_year, :no
        ');
        $stmt->execute([
            'item_type_id'   => $itemTypeId,
            'title'          => $title,
            'creator'        => $creator !== '' ? $creator : null,
            'format_detail'  => $formatDetail !== '' ? $formatDetail : null,
            'release_year'   => $releaseYear !== '' ? (int) $releaseYear : null,
```

```
            'notes'          => $notes !== '' ? $notes : null,
        ]);

        header('Location: /add_item.php?added=1');
        exit;
    }
}

$itemTypes = get_item_types($pdo);
?>
<!DOCTYPE html>
<html>
<body>

<?php if (isset($_GET['added'])): ?>
    <p>Item added.</p>
<?php endif; ?>

<?php foreach ($errors as $error): ?>
    <p style="color: red;"><?= htmlspecialchars($error) ?></p>
<?php endforeach; ?>

<form method="post">
    <label>Type:
        <select name="item_type_id" required>
            <option value="">-- choose --</option>
            <?php foreach ($itemTypes as $type): ?>
                <option value="<?= $type['id'] ?>"><?= htmlspecialchars($type['name'])
            <?php endforeach; ?>
        </select>
    </label><br>

    <label>Title: <input type="text" name="title" required></label><br>
    <label>Creator: <input type="text" name="creator"></label><br>
    <label>Format detail: <input type="text" name="format_detail"></label><br>
    <label>Release year: <input type="number" name="release_year"></label><br>
    <label>Notes: <textarea name="notes"></textarea></label><br>

    <button type="submit">Add Item</button>
</form>
```

```
</body>
</html>
```

WHY THE EMPTY-STRING FIELDS BECOME NULL, NOT ''

`creator`, `format_detail`, `release_year`, and `notes` are all nullable columns (Chapter 2). A DVD genuinely has no author, and it's more honest for that field to store SQL `NULL` — "no value recorded" — than an empty string, which reads as "the value is the empty string," a subtly different and less accurate claim. The ternaries in the `execute()` array make that translation explicit rather than leaving it to chance.

Editing an Item

`edit_item.php` follows the same GET/POST split, but GET now has to fetch the existing row first (by `id` from the query string) so the form can be pre-filled, and POST runs an `UPDATE` instead of an `INSERT`:

`edit_item.php`

```php
<?php
require_once __DIR__ . '/functions.php';

$id = (int) ($_GET['id'] ?? $_POST['id'] ?? 0);
if ($id < 1) {
    exit('Missing item id.');
}

$errors = [];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $title = trim($_POST['title'] ?? '');

    if ($title === '') {
        $errors[] = 'Title is required.';
    }

    if (empty($errors)) {
        $stmt = $pdo->prepare('
```

```
            UPDATE items
            SET item_type_id = :item_type_id,
                title = :title,
                creator = :creator,
                format_detail = :format_detail,
                release_year = :release_year,
                notes = :notes
            WHERE id = :id
        ');
        $stmt->execute([
            'item_type_id'  => (int) $_POST['item_type_id'],
            'title'         => $title,
            'creator'       => $_POST['creator'] !== '' ? $_POST['creator'] : null,
            'format_detail' => $_POST['format_detail'] !== '' ? $_POST['format_detail']
            'release_year'  => $_POST['release_year'] !== '' ? (int) $_POST['release_ye
            'notes'         => $_POST['notes'] !== '' ? $_POST['notes'] : null,
            'id'            => $id,
        ]);

        header("Location: /edit_item.php?id={$id}&updated=1");
        exit;
    }
}

$stmt = $pdo->prepare('SELECT * FROM items WHERE id = :id');
$stmt->execute(['id' => $id]);
$item = $stmt->fetch();

if (!$item) {
    exit('Item not found.');
}

$itemTypes = get_item_types($pdo);
?>
<!-- form markup follows the same shape as add_item.php, with each
     input's value pre-filled from $item, e.g.:
     <input type="text" name="title" value="<?= htmlspecialchars($item['title']) ?>">
     and a hidden <input type="hidden" name="id" value="<?= $id ?>"> -->
```

ALWAYS FETCH THE ROW BEFORE TRUSTING THE FORM

Fetching the existing item first — even though the POST branch doesn't strictly need every field from it — means the edit page fails honestly with "Item not found" if someone edits the URL to a bogus `id`, rather than silently rendering a blank form that would submit an `UPDATE` against a row that was never actually loaded.

Deleting an Item

Delete is the simplest of the three, but it's the one most worth protecting against an accidental click — this project uses a plain confirmation link rather than a full modal, since a personal single-user catalogue doesn't need more ceremony than that:

`delete_item.php`

```
<?php
require_once __DIR__ . '/config.php';

$id = (int) ($_POST['id'] ?? 0);

if ($_SERVER['REQUEST_METHOD'] === 'POST' && $id > 0) {
    $stmt = $pdo->prepare('DELETE FROM items WHERE id = :id');
    $stmt->execute(['id' => $id]);
}

header('Location: /index.php?deleted=1');
exit;
```

On the listing page (built in full in Chapter 7), each row's own delete action is a tiny `<form method="post" action="delete_item.php">` containing just the hidden `id` field and a submit button — never a plain `<a href>` link.

WHY DELETE MUST BE A POST, NOT A GET LINK

A `DELETE FROM items ...` triggered by a plain link means the browser (or a search-engine crawler, or a link-preview bot) could delete a real item just by fetching that URL — a GET request is supposed to be "safe" (no side effects) by HTTP's own convention, and browsers, proxies, and bots all rely on that convention. Requiring a POST means the delete can only actually happen from a real form submission on this page.

Where Tags Fit Later

This chapter's own add/edit forms deliberately don't touch `tags` or `item_tags` yet — Chapter 5 covers attaching and managing tags specifically, once the core item CRUD is solid on its own. Building both at once would make it harder to tell, if something breaks, whether the bug is in the item logic or the tag logic.

Hands-On Exercises

EXERCISE 1

Wire up `add_item.php` against the real `catalogue` database and add three items: one book, one CD, and one DVD. Confirm all three appear correctly with `SELECT * FROM items;`, including that each row's unused columns (e.g. a DVD's own `creator`) are stored as `NULL`, not an empty string.

View solution

EXERCISE 2

Complete the full `edit_item.php` form markup (the part left as a comment in this chapter), pre-filling every field's own `value` attribute from `$item`, then use it to change one of your three items' own `release_year`.

View solution

EXERCISE 3

Explain, in your own words, why `delete_item.php` requires a POST request rather than allowing a plain link, and describe one real, concrete way a GET-based delete link could be triggered by something other than a deliberate click.

View solution

CHAPTER 3 QUICK REFERENCE

- **add_item.php** — GET shows the form, POST validates and runs a prepared `INSERT`
- **edit_item.php** — GET fetches the existing row and pre-fills the form, POST runs a prepared `UPDATE`
- **delete_item.php** — POST-only, running a prepared `DELETE`, never triggerable via a plain link
- **Empty form fields** — converted to SQL `NULL`, not stored as empty strings, for every nullable column
- **Tags** — deliberately deferred to Chapter 5, kept out of this chapter's own item CRUD
- **Next chapter:** Type-Specific Fields — one form, several real item shapes

CHAPTER 4: TYPE-SPECIFIC FIELDS

Personal Catalogue: PHP & MySQL

Chapter 4 · Type-Specific Fields

Chapter 3's add/edit forms already work, but they show one generic **"Creator"** label and one generic **"Format detail"** label no matter which item type is selected — technically correct, since those are the real column names, but not very honest to the person actually filling the form in. A CD's "creator" is really its artist; a DVD's is really its director. This chapter makes the same single form feel type-aware without touching the database schema at all.

> **RECAP: ONE SHARED SCHEMA, SEVERAL REAL MEANINGS**
>
> Chapter 2 already established this table — it's worth having it in view again before building the UI on top of it:

COLUMN	BOOK	CD	DVD / BLU-RAY
`creator`	Author	Artist	Director
`format_detail`	Hardcover / paperback	Single / album	Region code

The Core Idea: Data-Driven Labels, Not a Fork in the Form

A tempting but wrong approach here is to build a genuinely separate form for each item type — a "book form," a "CD form," a "DVD form" — each with its own field names. That would mean four times the markup to maintain, and four separate branches of PHP to validate and insert. Since the underlying `items` table is deliberately shared (Chapter 2), the form should stay shared too: one set of inputs, with their **labels** — not their names, not their database columns — changing based on whatever `item_type_id` is currently selected.

That's a job for a small amount of vanilla JavaScript running in the browser, driven by a plain lookup object built once from the same `item_types` data the form already renders its dropdown from.

Building the Label Map

Rather than hard-coding the label text in JavaScript (where it could drift out of sync with the item types actually stored in the database), the label map is generated server-side, from PHP, as a small inline `<script>` block that runs once when the page loads:

`add_item.php (inside the <body>, after the form)`

```
<script>
// Built from a plain PHP array, not hand-typed twice, so it can never
// silently disagree with what's actually in item_types.
const creatorLabels = <?= json_encode([
    'book'   => 'Author',
    'cd'     => 'Artist',
    'dvd'    => 'Director',
    'bluray' => 'Director',
]) ?>;

const formatLabels = <?= json_encode([
    'book'   => 'Binding (hardcover / paperback)',
    'cd'     => 'Release type (single / album)',
    'dvd'    => 'Region code',
    'bluray' => 'Region code',
]) ?>;
</script>
```

`json_encode()` turns a plain PHP associative array into a real JavaScript object literal — this is the same real technique PHP Intermediate Chapter 7 covers for turning PHP data into JSON generally, just used here to hand a small config object to a page's own inline script rather than to an API response.

Wiring the Dropdown to the Labels

The item-type `<select>` already exists from Chapter 3. Two small changes make it drive the labels: giving the `<select>` and the two label elements real `id` attributes, and each `<option>` a `data-name` attribute holding the real `item_types.name` value (`'book'`, `'cd'`, and so on) — not the numeric id, since that's what the label maps above are keyed by:

```
<select name="item_type_id" id="item-type-select" required>
    <option value="">-- choose --</option>
    <?php foreach ($itemTypes as $type): ?>
        <option value="<?= $type['id'] ?>"
                data-name="<?= htmlspecialchars($type['name']) ?>">
            <?= htmlspecialchars($type['name']) ?>
        </option>
    <?php endforeach; ?>
</select>

...

<label id="creator-label" for="creator-input">Creator:</label>
<input type="text" name="creator" id="creator-input">

<label id="format-label" for="format-input">Format detail:</label>
<input type="text" name="format_detail" id="format-input">
```

And the script that actually reacts to a change:

```
const select       = document.getElementById('item-type-select');
const creatorLabel = document.getElementById('creator-label');
const formatLabel  = document.getElementById('format-label');

function updateLabels() {
    const selectedOption = select.options[select.selectedIndex];
    const typeName = selectedOption.dataset.name;

    creatorLabel.textContent = (creatorLabels[typeName] ?? 'Creator') + ':';
    formatLabel.textContent  = (formatLabels[typeName] ?? 'Format detail') + ':';
}

select.addEventListener('change', updateLabels);
// Run once on load too, in case a type is already selected
// (e.g. the edit form, where an item's own type is pre-chosen).
updateLabels();
```

WHY THE FALLBACK ('CREATOR', NOT UNDEFINED)

`creatorLabels[typeName] ?? 'Creator'` matters for one real edge case: the blank `-- choose --` option has no `data-name` at all, so `typeName` is `undefined`, and a lookup on `undefined` correctly misses every key in the object. Without the `??` fallback, the label would read the literal text "undefined:" until a real type is picked — a small but genuinely confusing bug if it isn't guarded against.

This Is Cosmetic, Not Validation

Relabeling a text input doesn't change what can actually be typed into it — nothing stops someone from selecting "book" and then typing "Region 2" into what's now labeled "Binding" anyway. This chapter's own JavaScript is a genuine usability improvement, not a data-integrity mechanism, and it must never be treated as one.

CLIENT-SIDE RELABELING IS NOT SERVER-SIDE VALIDATION

JavaScript can be disabled, blocked, or simply never run (a script tag failing to load, a browser extension interfering) — and even when it runs perfectly, a request can always be sent directly to `add_item.php` with a tool like curl, bypassing the form (and its labels) entirely. Every real validation rule — "Title is required," "please choose an item type" — must stay enforced in the PHP that actually runs the `INSERT` / `UPDATE`, exactly as Chapter 3 already does it, regardless of what the browser's own UI happens to show.

A Small, Genuinely Type-Specific Validation Rule

One place type does matter server-side: `release_year`. A book or CD's real release year is reasonably bounded — nothing published before, say, 1400 belongs in this catalogue — but the exact sensible range differs slightly by type in a way that's not worth over-engineering. A single, deliberately loose sanity check catches genuine typos (a four-digit year with a stray extra digit, or an obviously impossible future date) without needing a separate rule per item type:

```
if ($releaseYear !== '') {
    $year = (int) $releaseYear;
    $currentYear = (int) date('Y');

    if ($year < 1400 || $year > $currentYear + 1) {
        $errors[] = "Release year looks wrong: {$releaseYear}.";
```

```
    }
}
```

The `+ 1` allows for an item releasing next calendar year (a pre-order), while still catching an obvious typo like `29026`.

Hands-On Exercises

EXERCISE 1

Add the `data-name` attributes, the two label elements, and the JavaScript from this chapter to your own `add_item.php`. Confirm switching the dropdown between "book," "cd," and "dvd" correctly updates both labels without a page reload.

View solution

EXERCISE 2

Add the same label-switching script to `edit_item.php`, and confirm that opening the edit form for an existing DVD shows "Director" and "Region code" correctly pre-labeled on page load, before any dropdown change happens.

View solution

EXERCISE 3

Add the `release_year` sanity check from this chapter to `add_item.php`'s own validation, then confirm submitting a year of `29026` is correctly rejected with an error message, while `2027` (next year) is correctly accepted.

View solution

CHAPTER 4 QUICK REFERENCE

- **One shared form** — labels change per type via JavaScript; database columns and field names stay the same
- **Label maps** — built server-side with `json_encode()` so they can never drift from the real `item_types` data
- `data-name` — each option carries the real type name, since the label maps are keyed by name, not by numeric id
- **Cosmetic vs. validation** — relabeling never replaces server-side checks; every real rule still lives in the PHP that runs the query
- `release_year` **sanity check** — one loose, shared range rule rather than four separate per-type rules
- **Next chapter:** Tags — attaching, managing, and filtering by tag on books

CHAPTER 5: TAGS

Personal Catalogue: PHP & MySQL

Chapter 5 · Tags

Chapter 2 built `tags` and `item_tags`, the many-to-many pairing that lets a book carry several tags at once. This chapter finally wires that schema up to the real add/edit forms — creating tags on the fly, attaching and detaching them from an item, and running the actual "find every book tagged Python" query the whole feature exists for.

Two Real Operations, One Shared Helper Function

Both adding a new item and editing an existing one need to end up with the same result: a correct, current set of rows in `item_tags` for that item's own id. Rather than writing that logic twice, one shared `sync_item_tags()` function handles both cases — insert new pairings, and (for edit specifically) remove pairings that were unchecked:

`functions.php (additions)`

```
function get_all_tags(PDO $pdo): array {
    return $pdo->query('SELECT id, name FROM tags ORDER BY name')->fetchAll();
}

function get_tags_for_item(PDO $pdo, int $itemId): array {
    $stmt = $pdo->prepare('
        SELECT tags.id, tags.name
        FROM tags
        JOIN item_tags ON item_tags.tag_id = tags.id
        WHERE item_tags.item_id = :item_id
        ORDER BY tags.name
    ');
    $stmt->execute(['item_id' => $itemId]);
    return $stmt->fetchAll();
}

// Returns an existing tag's id, or creates the tag first and returns
// the new id — a real "find or create" pattern.
```

```php
function find_or_create_tag(PDO $pdo, string $name): int {
    $stmt = $pdo->prepare('SELECT id FROM tags WHERE name = :name');
    $stmt->execute(['name' => $name]);
    $existing = $stmt->fetch();

    if ($existing) {
        return (int) $existing['id'];
    }

    $stmt = $pdo->prepare('INSERT INTO tags (name) VALUES (:name)');
    $stmt->execute(['name' => $name]);

    return (int) $pdo->lastInsertId();
}

// $tagIds is the full, correct set of tag ids this item should end
// up with. Anything currently attached but not in this set gets
// removed; anything in this set but not currently attached gets added.
function sync_item_tags(PDO $pdo, int $itemId, array $tagIds): void {
    $current = get_tags_for_item($pdo, $itemId);
    $currentIds = array_column($current, 'id');

    $toAdd = array_diff($tagIds, $currentIds);
    $toRemove = array_diff($currentIds, $tagIds);

    $insertStmt = $pdo->prepare('INSERT INTO item_tags (item_id, tag_id) VALUES (:item_
    foreach ($toAdd as $tagId) {
        $insertStmt->execute(['item_id' => $itemId, 'tag_id' => $tagId]);
    }

    $deleteStmt = $pdo->prepare('DELETE FROM item_tags WHERE item_id = :item_id AND tag
    foreach ($toRemove as $tagId) {
        $deleteStmt->execute(['item_id' => $itemId, 'tag_id' => $tagId]);
    }
}
```

WHY DIFF INSTEAD OF DELETE-EVERYTHING-THEN-REINSERT

A simpler-looking alternative would be: on every save, `DELETE` every existing `item_tags` row for this item, then `INSERT` the whole new set fresh. That works, but it's needless churn — for an item whose tags didn't actually change, it still runs a real delete and a real insert for every tag, every single

> save. The diff approach (`array_diff` in both directions) only touches the rows that genuinely changed, which matters more as a habit than as a performance concern at this scale — it's the same principle behind not rewriting a whole file when only one line changed.

Adding the Tag Checkboxes to the Form

The catalogue's own real spec only ever asks for tags on books — so the tag section is shown only when `item_type_id` corresponds to `book`, reusing the exact same dropdown-driven show/hide technique Chapter 4 already built for the label switching:

```
<div id="tags-section" style="display: none;">
    <p>Tags:</p>
    <?php foreach ($allTags as $tag): ?>
        <label>
            <input type="checkbox" name="tag_ids[]" value="<?= $tag['id'] ?>">
            <?= htmlspecialchars($tag['name']) ?>
        </label>
    <?php endforeach; ?>

    <label>New tag(s), comma-separated:
        <input type="text" name="new_tags" placeholder="e.g. Fantasy, Short Stories">
    </label>
</div>
```

Then, added to Chapter 4's own `updateLabels()` function (or a second listener on the same `change` event):

```
const tagsSection = document.getElementById('tags-section');

function toggleTagsSection() {
    const selectedOption = select.options[select.selectedIndex];
    const typeName = selectedOption.dataset.name;

    tagsSection.style.display = (typeName === 'book') ? 'block' : 'none';
}
```

```
select.addEventListener('change', toggleTagsSection);
toggleTagsSection();
```

HIDING THE SECTION DOESN'T STOP THE DATA FROM BEING SUBMITTED

A hidden `<div>` (via `display: none`) still submits its own checked checkboxes if the form is somehow submitted while it's hidden — CSS visibility and form submission are unrelated. This is why the actual enforcement of "only books get tags" happens where it matters, in PHP, described next — the hidden section is purely a UI convenience, matching the same cosmetic-vs-validation distinction Chapter 4 already drew for the type-specific labels.

Handling Tags on Insert

Added to `add_item.php`'s own POST branch, right after the item itself is inserted (so a real `item_id` exists to attach tags to), and deliberately gated on the item actually being a book:

```
$stmt->execute([ /* ...item fields from Chapter 3... */ ]);
$newItemId = (int) $pdo->lastInsertId();

// item_type_id 1 = book, per the item_types seed data (Chapter 1).
// Tags are only ever processed for books, regardless of what a
// tampered request might send for a different type.
if ($itemTypeId === 1) {
    $tagIds = array_map('intval', $_POST['tag_ids'] ?? []);

    $newTagNames = array_filter(array_map('trim', explode(',', $_POST['new_tags'] ?? ''
    foreach ($newTagNames as $name) {
        $tagIds[] = find_or_create_tag($pdo, $name);
    }

    sync_item_tags($pdo, $newItemId, $tagIds);
}

header('Location: /add_item.php?added=1');
exit;
```

`array_filter()` with no callback drops any empty strings from the exploded comma-separated list — this matters for the common real case of a trailing comma (`"Fantasy, Short Stories,"`), which would otherwise try to create a tag with an empty name.

Handling Tags on Edit

`edit_item.php` needs the item's own current tags loaded for the GET branch (to pre-check the right boxes), and the same insert-or-sync logic on POST:

```
// In the POST branch, after the UPDATE runs:
if ((int) $_POST['item_type_id'] === 1) {
    $tagIds = array_map('intval', $_POST['tag_ids'] ?? []);

    $newTagNames = array_filter(array_map('trim', explode(',', $_POST['new_tags'] ?? ''
    foreach ($newTagNames as $name) {
        $tagIds[] = find_or_create_tag($pdo, $name);
    }

    sync_item_tags($pdo, $id, $tagIds);
} else {
    // The item was changed away from "book" — remove any tags
    // it may have picked up while it was still a book.
    sync_item_tags($pdo, $id, []);
}

// In the GET branch, alongside loading $item and $itemTypes:
$itemTags = get_tags_for_item($pdo, $id);
$itemTagIds = array_column($itemTags, 'id');
$allTags = get_all_tags($pdo);
```

And in the checkbox markup itself, each box is pre-checked if its id is in `$itemTagIds`:

```
<input type="checkbox" name="tag_ids[]" value="<?= $tag['id'] ?>"
    <?= in_array($tag['id'], $itemTagIds) ? 'checked' : '' ?>>
```

THE ELSE BRANCH MATTERS

Without the `else` branch (`sync_item_tags($pdo, $id, [])`), an item that started as a book, picked up two tags, and was then re-edited to become a DVD would keep those two tags attached forever — invisible in the UI (since the tags section is hidden for non-books) but still sitting in `item_tags`. Passing an empty array explicitly clears them, matching the honest rule "only books have tags."

The Query the Whole Feature Exists For

With attachment working, filtering the catalogue by tag is a single query — this is a preview of what Chapter 6's own search page uses directly:

```
$stmt = $pdo->prepare('
    SELECT items.*
    FROM items
    JOIN item_tags ON item_tags.item_id = items.id
    JOIN tags ON tags.id = item_tags.tag_id
    WHERE tags.name = :tag_name
    ORDER BY items.title
');
$stmt->execute(['tag_name' => 'Python']);
$books = $stmt->fetchAll();
```

Every real book tagged "Python" — including ones tagged Python *and* something else, since `item_tags` is a genuine many-to-many table — comes back in one query, exactly the search Chapter 1's own real deadline framing said mattered most.

Hands-On Exercises

EXERCISE 1

Add the tag checkboxes, the "New tag(s)" text input, and the show/hide JavaScript to your own `add_item.php`. Add a new book with two brand-new tags typed into the comma-separated field (tags that don't exist yet), then confirm both were created in `tags` and correctly attached in `item_tags`.

View solution

EXERCISE 2

Add the pre-checked checkboxes and the sync logic (including the `else` branch) to `edit_item.php`. Edit an existing book to remove one of its two tags and add a third, then confirm `item_tags` reflects exactly the new set — the removed tag's row gone, the third tag's row added, and the untouched tag's row unchanged.

View solution

EXERCISE 3

Run the tag-filter query from this chapter's own closing section against your real database for a tag you've actually used, and write out the full result. Then explain, in one or two sentences, why a book tagged with two different tags only appears once in a plain `SELECT DISTINCT items.* FROM ...` version of this query but could appear twice without the `DISTINCT`.

View solution

CHAPTER 5 QUICK REFERENCE

- `find_or_create_tag()` — looks up a tag by name, creating it only if it doesn't already exist
- `sync_item_tags()` — diffs the desired tag set against what's currently attached, only touching rows that actually changed
- **Tags section shown only for books** — a UI convenience via `display: none`, never a substitute for the real server-side `item_type_id === 1` check
- **The `else` branch on edit** — clears tags explicitly if an item's type changes away from book, so stale rows can't linger unseen
- **The filter query** — a single three-table join, no `UNION`, the direct payoff of Chapter 2's own shared-`items`-table decision
- **Next chapter:** Search — real lookup across the whole catalogue, not just by tag

CHAPTER 6: SEARCH

Personal Catalogue: PHP & MySQL

Chapter 6 · Search

Chapter 1 named the real question this whole project exists to answer fast: "*do I already own this?*" Every design decision since then — the shared `items` table in particular — was made specifically to keep that question a single, simple query. This chapter builds the actual search page.

The Simplest Version First

A basic title search, matching anywhere inside the title, not just at the start:

```
$stmt = $pdo->prepare('
    SELECT * FROM items
    WHERE title LIKE :query
    ORDER BY title
');
$stmt->execute(['query' => '%' . $searchTerm . '%']);
$results = $stmt->fetchAll();
```

The `%` wildcards are built into the PHP string, not the SQL — the placeholder is still bound as one plain parameter value, exactly the same `PDO::ATTR_EMULATE_PREPARES => false` protection from Chapter 1 applies to it. This is a genuinely important distinction: it would be a real SQL injection risk to build the wildcards by concatenating raw SQL text instead — `"title LIKE '%" . $searchTerm . "%'"` — since that puts user input directly into the query string.

Searching Creator Too, Not Just Title

A search for "Tolkien" should find "The Hobbit" even though the word "Tolkien" never appears in its title — it's in `creator`. A real search has to check both columns:

```
$stmt = $pdo->prepare('
    SELECT * FROM items
    WHERE title LIKE :query OR creator LIKE :query
    ORDER BY title
');
$stmt->execute(['query' => '%' . $searchTerm . '%']);
$results = $stmt->fetchAll();
```

NAMED PLACEHOLDERS CAN'T BE REUSED WITH EXECUTE()'S ARRAY FORM

Reusing `:query` twice in the SQL above and passing one `'query'` key in the `execute()` array actually works correctly with PDO's MySQL driver — it substitutes the same bound value into both positions. This is real, documented PDO/MySQL behavior, but it's not universal across every database driver PDO supports, so it's worth knowing it's a MySQL-specific convenience rather than a general PDO guarantee, in case this project's own database is ever swapped later.

Optional Filter: Item Type

Adding a "Type" dropdown to the search form (All / Book / CD / DVD / Blu-ray) means the query's own `WHERE` clause has to change shape depending on whether a type was actually chosen. Building this correctly means never string-concatenating raw `$_GET` values into the SQL — the clause structure changes, but every value still flows through a bound parameter:

```
$searchTerm = trim($_GET['q'] ?? '');
$typeFilter = (int) ($_GET['type'] ?? 0);

$sql = 'SELECT * FROM items WHERE (title LIKE :query OR creator LIKE :query)';
$params = ['query' => '%' . $searchTerm . '%'];

if ($typeFilter > 0) {
    $sql .= ' AND item_type_id = :type_id';
    $params['type_id'] = $typeFilter;
}

$sql .= ' ORDER BY title';

$stmt = $pdo->prepare($sql);
```

```
$stmt->execute($params);
$results = $stmt->fetchAll();
```

The SQL string is assembled conditionally, but every actual piece of user-supplied data — the search term, the type id — still only ever reaches the query as a bound parameter. Building the `WHERE` clause's own shape in PHP is safe; building a value's own text directly into the SQL string is not, and the difference between those two things is the whole point of this pattern.

Optional Filter: Tag (Reusing Chapter 5's Own Query)

Adding a tag filter means joining `item_tags` and `tags` only when a tag was actually selected — otherwise every non-book item would be silently excluded by an unconditional join, since a DVD has no rows in `item_tags` at all:

```
$tagFilter = (int) ($_GET['tag'] ?? 0);

$sql = 'SELECT items.* FROM items';
$params = [];

if ($tagFilter > 0) {
    $sql .= ' JOIN item_tags ON item_tags.item_id = items.id';
    $sql .= ' AND item_tags.tag_id = :tag_id';
    $params['tag_id'] = $tagFilter;
}

$sql .= ' WHERE (items.title LIKE :query OR items.creator LIKE :query)';
$params['query'] = '%' . $searchTerm . '%';

if ($typeFilter > 0) {
    $sql .= ' AND items.item_type_id = :type_id';
    $params['type_id'] = $typeFilter;
}

$sql .= ' ORDER BY items.title';

$stmt = $pdo->prepare($sql);
$stmt->execute($params);
$results = $stmt->fetchAll();
```

WHY THE TAG CONDITION SITS IN THE JOIN, NOT THE WHERE

Putting `item_tags.tag_id = :tag_id` in the `JOIN ... ON` clause (rather than a separate `WHERE item_tags.tag_id = :tag_id`) means the join itself only matches the one specific tag's own pairing row — a plain inner join would still work either way for this particular query, since either placement excludes non-matching rows entirely, but keeping the tag condition attached to the join it actually belongs to keeps the query's own logic easy to read as it grows: "join in this tag's own pairings, then filter the resulting rows by title/creator/type."

Escaping Literal % and _ in a Search Term

`LIKE`'s own `%` (any characters) and `_` (any single character) are special wildcard characters — if a real title happens to contain one literally (a book called "*50% Off*", or a format note with an underscore in it), searching for that exact text needs those characters escaped, or MySQL will interpret them as wildcards instead of literal characters:

```
function escape_like(string $value): string {
    return str_replace(['\\', '%', '_'], ['\\\\', '\\%', '\\_'], $value);
}

$searchTerm = escape_like(trim($_GET['q'] ?? ''));
$params['query'] = '%' . $searchTerm . '%';
```

The backslash itself has to be escaped first, before `%` and `_` — otherwise a search term that already contains a literal backslash would end up double-escaped incorrectly. The order of the `str_replace()` array matters here for exactly that reason.

A REAL, HONEST SCOPE NOTE

For a small personal catalogue, this edge case is genuinely unlikely to ever come up in practice — few book or CD titles contain a literal `%` or `_`. It's included here because it's a real, documented `LIKE` behavior worth knowing about, not because this project's own real data is expected to trigger it often.

Assembling the Search Page

The full `search.php` combines every piece above: a form with a text input, a type dropdown (reusing `get_item_types()` from Chapter 1), and a tag dropdown (reusing `get_all_tags()` from Chapter 5), all submitting via `GET` — deliberately GET, not POST, since a search is read-only and a GET request means the results page has its own shareable, bookmarkable URL:

```
<form method="get">
    <input type="text" name="q" value="<?= htmlspecialchars($searchTerm) ?>" placeholde

    <select name="type">
        <option value="0">All types</option>
        <?php foreach ($itemTypes as $type): ?>
            <option value="<?= $type['id'] ?>" <?= $typeFilter === (int) $type['id'] ?
                <?= htmlspecialchars($type['name']) ?>
            </option>
        <?php endforeach; ?>
    </select>

    <select name="tag">
        <option value="0">Any tag</option>
        <?php foreach ($allTags as $tag): ?>
            <option value="<?= $tag['id'] ?>" <?= $tagFilter === (int) $tag['id'] ? 'se
                <?= htmlspecialchars($tag['name']) ?>
            </option>
        <?php endforeach; ?>
    </select>

    <button type="submit">Search</button>
</form>
```

Hands-On Exercises

EXERCISE 1

Build the full `search.php` from this chapter against your real database. Search for "Tolkien" with the type dropdown left on "All types" and confirm "The Hobbit" appears in the results even though "Tolkien" never appears in its own title.

View solution

EXERCISE 2

Search with an empty search term but the type dropdown set to "cd." Explain, using the SQL your own PHP would build for this case, why every CD in the catalogue is returned even though the search box was left blank.

View solution

EXERCISE 3

Add the `escape_like()` function from this chapter, add a book to your database with a title containing a literal underscore (e.g. "Learn_Python: A Guide"), and confirm that searching for the exact substring `"Learn_Python"` matches only that book — not accidentally matching every title where any single character happens to sit where the underscore is.

View solution

CHAPTER 6 QUICK REFERENCE

- **Wildcards built in PHP, bound as one value** — `'%' . $searchTerm . '%'` passed as a single parameter, never concatenated into the raw SQL string
- **Title OR creator** — a real search checks both, since a creator's own name never appears in the title column
- **Conditional `WHERE` / `JOIN` clauses** — built up in PHP based on which filters are active, with every actual value still flowing through a bound parameter
- `escape_like()` — escapes literal `%` / `_` / `\` characters in a search term so `LIKE` doesn't misinterpret them as wildcards
- **Search form uses GET** — deliberately, so results have their own shareable, bookmarkable URL, unlike the POST-only delete action from Chapter 3
- **Next chapter:** The Catalogue List & Detail Pages

CHAPTER 7: THE CATALOGUE LIST & DETAIL PAGES

Personal Catalogue: PHP & MySQL

Chapter 7 · The Catalogue List & Detail Pages

Chapters 3-6 built every write and search operation the catalogue needs, but there's still no ordinary browsing page — the one a real visit to the site opens on. This chapter builds `index.php`, a full list of every item with its type and tags visible at a glance, and `view_item.php`, a single-item detail page.

The Naive Approach, and Why It's a Real Problem

The obvious first attempt at listing every item's own tags alongside it is to fetch all items, then loop over them and run `get_tags_for_item()` (from Chapter 5) once per row:

```
// Works, but don't actually do this:
$items = $pdo->query('SELECT * FROM items ORDER BY title')->fetchAll();

foreach ($items as &$item) {
    $item['tags'] = get_tags_for_item($pdo, $item['id']);
}
```

> **THIS IS THE CLASSIC N+1 QUERY PROBLEM**
>
> Listing 40 items this way runs **1 query to fetch the items, plus 40 more queries** — one per item — to fetch each one's own tags. That's 41 real round trips to MySQL for a page that should need one or two. It happens to work correctly at a small personal-catalogue scale, but it's exactly the pattern that turns slow the moment a collection grows past a trivial size, and it's worth building the habit of avoiding it from the start rather than only fixing it once it's actually slow.

The Fix: One Query, GROUP_CONCAT, and a LEFT JOIN

MySQL's `GROUP_CONCAT()` aggregate function collapses several rows' worth of a column into one comma-separated string per group — exactly what's needed to fetch every item's own

tags in the same query as the item itself, using `GROUP BY` to group the joined tag rows back down to one row per item:

```
$sql = "
    SELECT
        items.*,
        item_types.name AS type_name,
        GROUP_CONCAT(tags.name ORDER BY tags.name SEPARATOR ', ') AS tag_names
    FROM items
    JOIN item_types ON item_types.id = items.item_type_id
    LEFT JOIN item_tags ON item_tags.item_id = items.id
    LEFT JOIN tags ON tags.id = item_tags.tag_id
    GROUP BY items.id
    ORDER BY items.title
";

$items = $pdo->query($sql)->fetchAll();
// $items[0]['tag_names'] is now either "Programming, Python", or NULL
// for an item with no tags at all — one query, no loop needed.
```

WHY LEFT JOIN, NOT JOIN, FOR ITEM_TAGS AND TAGS

A plain `JOIN` (an inner join) only keeps rows that have a match on *both* sides — an item with zero tags (every CD and DVD, and any book that hasn't been tagged yet) has zero matching rows in `item_tags`, so a plain `JOIN` would silently drop it from the results entirely. `LEFT JOIN` keeps every row from `items` regardless of whether a match exists on the right-hand side, filling in `NULL` for the tag columns when there's no match — exactly the same NULL-for-"no value" pattern Chapter 2's own schema already relies on. `item_types` still uses a plain `JOIN` here, deliberately, since every item is guaranteed (by the `NOT NULL` foreign key from Chapter 2) to have exactly one real type.

Rendering the List

With `$items` already carrying `type_name` and `tag_names` directly, the list page itself is a simple loop — no further queries needed per row:

`index.php`

```
<table>
    <tr>
        <th>Title</th><th>Type</th><th>Creator</th><th>Tags</th><th></th>
    </tr>
    <?php foreach ($items as $item): ?>
        <tr>
            <td><a href="view_item.php?id=<?= $item['id'] ?>"><?= htmlspecialchars($ite
            <td><?= htmlspecialchars($item['type_name']) ?></td>
            <td><?= htmlspecialchars($item['creator'] ?? '—') ?></td>
            <td><?= htmlspecialchars($item['tag_names'] ?? '—') ?></td>
            <td>
                <a href="edit_item.php?id=<?= $item['id'] ?>">Edit</a>
                <form method="post" action="delete_item.php" style="display: inline;">
                    <input type="hidden" name="id" value="<?= $item['id'] ?>">
                    <button type="submit">Delete</button>
                </form>
            </td>
        </tr>
    <?php endforeach; ?>
</table>
```

The delete form matches Chapter 3's own POST-only requirement exactly, wrapped in an inline-styled `<form>` so it sits naturally inside a table cell next to the edit link.

A "Recently Added" Section, Using Chapter 2's Own created_at Column

Chapter 2 called out `created_at`'s own automatic default as "free" — something to use later. A small recently-added widget on the same page is exactly that payoff:

```
$recent = $pdo->query('
    SELECT title, created_at FROM items
    ORDER BY created_at DESC
    LIMIT 5
')->fetchAll();
```

No `WHERE` clause, no parameters — `created_at` was populated automatically by MySQL's own `DEFAULT CURRENT_TIMESTAMP` for every row added since Chapter 3, so ordering by it and limiting to 5 costs nothing extra in application code.

The Detail Page

`view_item.php` reuses the same `GROUP_CONCAT` query shape, just narrowed to one row with a `WHERE items.id = :id` clause:

`view_item.php`

```
<?php
require_once __DIR__ . '/config.php';

$id = (int) ($_GET['id'] ?? 0);

$stmt = $pdo->prepare("
    SELECT
        items.*,
        item_types.name AS type_name,
        GROUP_CONCAT(tags.name ORDER BY tags.name SEPARATOR ', ') AS tag_names
    FROM items
    JOIN item_types ON item_types.id = items.item_type_id
    LEFT JOIN item_tags ON item_tags.item_id = items.id
    LEFT JOIN tags ON tags.id = item_tags.tag_id
    WHERE items.id = :id
    GROUP BY items.id
");
$stmt->execute(['id' => $id]);
$item = $stmt->fetch();

if (!$item) {
    http_response_code(404);
    exit('Item not found.');
}
?>
<h1><?= htmlspecialchars($item['title']) ?></h1>
<p>Type: <?= htmlspecialchars($item['type_name']) ?></p>
<p>Creator: <?= htmlspecialchars($item['creator'] ?? 'Not recorded') ?></p>
```

```
<p>Format detail: <?= htmlspecialchars($item['format_detail'] ?? 'Not recorded') ?></p>
<p>Release year: <?= htmlspecialchars((string) ($item['release_year'] ?? 'Not recorded'
<p>Tags: <?= htmlspecialchars($item['tag_names'] ?? 'None') ?></p>
<?php if ($item['notes']): ?>
    <p>Notes: <?= nl2br(htmlspecialchars($item['notes'])) ?></p>
<?php endif; ?>

<a href="edit_item.php?id=<?= $item['id'] ?>">Edit</a>
<a href="index.php">Back to Catalogue</a>
```

WHY GROUP BY ITEMS.ID ALONE IS ENOUGH

Standard SQL normally requires every non-aggregated column in the `SELECT` list to also appear in `GROUP BY`. MySQL relaxes this specifically when the `GROUP BY` column is a table's own primary key (`items.id` here) — since a primary key uniquely determines every other column in that same row, grouping by it alone is functionally identical to grouping by every column, and MySQL is smart enough to allow the shorter form. This is real, documented MySQL-specific behavior (called "functional dependency") — it isn't guaranteed to work the same way on every SQL database, worth remembering if this project's own database is ever swapped later, echoing the same portability caveat Chapter 6 already raised for reused named placeholders.

Hands-On Exercises

EXERCISE 1

Build `index.php` using the `GROUP_CONCAT` query from this chapter against your real database (with at least one tagged book, one untagged book, and one CD or DVD already in it from earlier chapters). Confirm the untagged items show a sensible fallback (e.g. "—") in the Tags column rather than a blank cell or a PHP warning.

View solution

EXERCISE 2

Temporarily change the list query's `LEFT JOIN` to a plain `JOIN`, reload `index.php`, and describe exactly what changes in the result set. Then change it back and confirm the original behavior returns.

View solution

EXERCISE 3

Build `view_item.php`, then visit it with a real item's own `id`, and separately with an `id` you know doesn't exist (e.g. `999999`). Confirm the second case shows "Item not found." rather than a blank page or a PHP error, and explain why `$stmt->fetch()` returning `false` is what makes the `if (!$item)` check work.

View solution

CHAPTER 7 QUICK REFERENCE

- **N+1 query problem** — fetching items, then looping to fetch each one's own tags separately, means 1+N real queries for N items
- `GROUP_CONCAT()` — collapses several joined tag rows into one comma-separated string per item, fixing N+1 in a single query
- `LEFT JOIN item_tags` / `tags` — keeps items with zero tags in the results; a plain `JOIN` would silently drop them
- `GROUP BY items.id` **alone** — valid in MySQL specifically because `id` is the primary key (functional dependency)
- **Recently added widget** — the real payoff of Chapter 2's own "free" `created_at DEFAULT CURRENT_TIMESTAMP` column
- **Next chapter:** Fast Manual Entry — reducing friction given the project's own real deadline

CHAPTER 8: FAST MANUAL ENTRY

Personal Catalogue: PHP & MySQL

Chapter 8 · Fast Manual Entry

Chapter 1 named the real constraint this whole project starts under: a genuine deadline, with barcode scanning deliberately deferred so items get added by hand for now. Every form built since Chapter 3 works correctly, but none of them are actually fast to use for the real first task — sitting down and typing in a whole existing shelf of books, CDs, and DVDs in one sitting. This chapter builds two features aimed specifically at that first, real bulk-entry session.

The Real Bottleneck: One Round Trip Per Item

Chapter 3's `add_item.php` redirects to `add_item.php?added=1` after a successful insert — which reloads a blank form, but only after showing a confirmation state first. For entering forty items back to back, even that small extra step adds up. The fix isn't more code, it's a smaller change: redirect straight back to a genuinely fresh form, with the cursor already sitting in the Title field, ready to type the next item immediately.

Quick-Add: Redirect Straight Back, Autofocus the Title

Two small, real changes to `add_item.php` from Chapter 3. First, the redirect target drops the confirmation step and returns straight to the blank form, remembering the last-used item type so a run of ten books in a row doesn't require reselecting "book" from the dropdown every single time:

```
// After the successful INSERT, instead of Chapter 3's own
// header('Location: /add_item.php?added=1'):
header("Location: /add_item.php?added=1&last_type={$itemTypeId}");
exit;
```

And in the GET branch, that remembered type pre-selects the dropdown on the fresh form:

```
$lastType = (int) ($_GET['last_type'] ?? 0);
?>
...
<select name="item_type_id" id="item-type-select" required>
    <option value="">-- choose --</option>
    <?php foreach ($itemTypes as $type): ?>
        <option value="<?= $type['id'] ?>" data-name="<?= htmlspecialchars($type['name'
            <?= (int) $type['id'] === $lastType ? 'selected' : '' ?>>
            <?= htmlspecialchars($type['name']) ?>
        </option>
    <?php endforeach; ?>
</select>
```

Second, the Title field gets a plain `autofocus` attribute — a real, standard HTML attribute, not a JavaScript trick — so the cursor is already in place the instant the page finishes loading:

```
<input type="text" name="title" autofocus required>
```

Together, adding ten items of the same type back to back becomes: type the title, tab to creator, tab to format, hit Enter, and the next blank form is already focused and ready — with the correct item type already pre-selected — with zero mouse clicks needed in between.

Bulk Paste Import: One Type, Many Titles at Once

Quick-add is faster per item, but it's still one HTTP request per item. For the real initial population of the catalogue — copying a list of book titles straight out of a spreadsheet, an email, or a notes app — a genuinely bulk path is worth having: paste many titles at once, pick one shared item type, and insert them all in a single operation.

`bulk_add.php`

```
<?php
require_once __DIR__ . '/functions.php';

$results = null;
```

```php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $itemTypeId = (int) ($_POST['item_type_id'] ?? 0);
    $rawLines = explode("\n", $_POST['titles'] ?? '');
    $titles = array_filter(array_map('trim', $rawLines));

    $inserted = [];
    $skipped = [];

    if ($itemTypeId > 0 && count($titles) > 0) {
        $checkStmt = $pdo->prepare('SELECT id FROM items WHERE title = :title AND item_
        $insertStmt = $pdo->prepare('INSERT INTO items (item_type_id, title) VALUES (:t

        $pdo->beginTransaction();

        try {
            foreach ($titles as $title) {
                $checkStmt->execute(['title' => $title, 'type_id' => $itemTypeId]);

                if ($checkStmt->fetch()) {
                    $skipped[] = $title;
                    continue;
                }

                $insertStmt->execute(['type_id' => $itemTypeId, 'title' => $title]);
                $inserted[] = $title;
            }

            $pdo->commit();
        } catch (PDOException $e) {
            $pdo->rollBack();
            throw $e;
        }

        $results = ['inserted' => $inserted, 'skipped' => $skipped];
    }
}

$itemTypes = get_item_types($pdo);
?>
<h1>Bulk Add</h1>
```

```
<?php if ($results): ?>
    <p>Added <?= count($results['inserted']) ?> item(s).
        <?php if ($results['skipped']): ?>
            Skipped <?= count($results['skipped']) ?> likely duplicate(s):
            <?= htmlspecialchars(implode(', ', $results['skipped'])) ?>
        <?php endif; ?>
    </p>
<?php endif; ?>

<form method="post">
    <label>Type:
        <select name="item_type_id" required>
            <option value="">-- choose --</option>
            <?php foreach ($itemTypes as $type): ?>
                <option value="<?= $type['id'] ?>"><?= htmlspecialchars($type['name'])
            <?php endforeach; ?>
        </select>
    </label><br>

    <label>Titles, one per line:<br>
        <textarea name="titles" rows="15" cols="50" autofocus></textarea>
    </label><br>

    <button type="submit">Add All</button>
</form>
```

WHY THE WHOLE LOOP RUNS INSIDE ONE TRANSACTION

Wrapping every insert in `beginTransaction()` / `commit()` means all of the real inserts either succeed together or none of them do — if something genuinely goes wrong partway through pasting in forty titles (a lost database connection, a constraint violation), `rollBack()` undoes everything that had already run in that batch, rather than leaving the catalogue in a half-imported state with no clean way to tell which titles made it in. It's also genuinely faster: without a transaction, MySQL commits each `INSERT` to disk individually; wrapped in one transaction, every insert in the batch is committed together in one disk write at the end.

DUPLICATE SKIPPING IS A CONVENIENCE, NOT REAL DEDUPLICATION

The exact-title-and-type match used to decide what's "already there" is deliberately simple — it won't catch a title typed with different capitalization or an extra space that survived `trim()`'s own

whitespace-only cleanup, and it says nothing about whether two different editions of the same book should really count as duplicates. It's a genuine help during a fast bulk-paste session specifically because obvious re-pastes of the same list are the realistic risk it protects against, not a substitute for actually reviewing the catalogue afterward.

What Bulk Add Deliberately Leaves Out

`bulk_add.php` only ever sets `item_type_id` and `title` — no creator, no format detail, no tags. That's intentional: the whole point is getting a long list of titles into the database *fast*, with the understanding that any item needing richer detail can be opened afterward through `edit_item.php` (Chapter 3) at a calmer moment. Trying to also capture every field during a rapid bulk paste would undermine the entire reason this page exists.

Hands-On Exercises

EXERCISE 1

Add the `autofocus` attribute and the `last_type` redirect/pre-selection logic to your own `add_item.php`. Add three books in a row without touching the mouse (only Tab, typing, and Enter) and confirm the type dropdown stays on "book" for all three without being reselected.

View solution

EXERCISE 2

Build `bulk_add.php` against your real database and paste in five book titles, one per line, including one title that's an exact duplicate of a book already in your catalogue from an earlier chapter. Confirm the results message correctly reports 4 added and 1 skipped, and that the database contains exactly one row for the duplicate title, not two.

View solution

EXERCISE 3

Explain, in your own words, what would go wrong for the user experience if `bulk_add.php`'s own loop ran each `INSERT` without a surrounding transaction, and a genuine database error occurred after 15 of 40 pasted titles had already been inserted.

View solution

CHAPTER 8 QUICK REFERENCE

- **autofocus** — a plain HTML attribute putting the cursor in the Title field the instant a fresh form loads, no JavaScript needed
- `last_type` — remembered across the redirect so a run of same-type items doesn't need the dropdown reselected each time
- `bulk_add.php` — one type, many titles pasted at once, each checked against an exact duplicate before insert
- `beginTransaction()` / `commit()` / `rollBack()` — wraps the whole batch so it succeeds or fails as one unit, and commits faster than one insert at a time
- **Deliberately title-only** — bulk add skips creator/format/tags on purpose; richer detail is added later via `edit_item.php`
- **Next chapter:** Deployment

CHAPTER 9: DEPLOYMENT

Personal Catalogue: PHP & MySQL

Chapter 9 · Deployment

Every chapter so far has run against a local install. This chapter moves the finished catalogue onto a real server — reusing the Debian/Apache-or-nginx groundwork this site's own *Setting Up a Web Server on Debian* and *Securing Your Web Server* courses already cover, rather than re-deriving general web-server setup here.

Where This Actually Lives

A personal catalogue like this doesn't need its own dedicated domain — the natural real home is a subdirectory or subdomain on a server already running (this project's own eventual home, per Chapter 10, is exactly this: a corner of the user's existing site). For the purposes of this chapter, the deployment target is a plain document root such as `/var/www/catalogue/`, served by whichever web server is already configured per those two courses.

Config Never Travels With the Code

Chapter 1's `config.php` has real database credentials sitting directly in it. That was fine for local development, but it must never be the file that gets copied or deployed to a server the same way the rest of the codebase is — especially if this project's own code ever ends up in a Git repository, where a committed credentials file stays in the project's history forever, even if it's deleted later.

`.gitignore`

```
config.php
```

In its place, a `config.example.php` file is committed instead — the same structure, with placeholder values, serving as a template for whoever sets the project up next (including, on a fresh server, the current user themselves):

config.example.php

```php
<?php
$dbHost = 'localhost';
$dbName = 'catalogue';
$dbUser = 'REPLACE_ME';
$dbPass = 'REPLACE_ME';

$pdo = new PDO("mysql:host={$dbHost};dbname={$dbName};charset=utf8mb4", $dbUser, $dbPas
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES => false,
]);
```

The real `config.php`, holding the actual production credentials, is created directly on the server itself (via `cp config.example.php config.php` then editing it in place) — it never passes through version control at all.

Migrating the Schema to the Production Database

Rather than re-typing every `CREATE TABLE` statement from Chapter 2 by hand on the server, the local schema is exported once and imported directly:

```
# On the local machine, export structure and data together:
mysqldump -u root -p catalogue > catalogue_export.sql

# Copy the file to the server (adjust user/host/path as needed):
scp catalogue_export.sql user@server:/tmp/

# On the server, create an empty database with the right charset first...
mysql -u root -p -e "CREATE DATABASE catalogue CHARACTER SET utf8mb4 COLLATE utf8mb4_un

# ...then import the exported structure and data into it:
mysql -u root -p catalogue < /tmp/catalogue_export.sql
```

Creating the database with the correct charset explicitly, before importing, matters for the exact same reason Chapter 1's own Exercise 3 covered — a server's own default charset can't be assumed to already be `utf8mb4`.

A Dedicated, Least-Privilege Database User

Using MySQL's own `root` account as the application's own connection user (even locally, in Chapter 1's own example) is a shortcut worth deliberately dropping before this goes anywhere real. A dedicated account, scoped to only this one database, limits the real damage a bug or a compromised credential could do:

```
CREATE USER 'catalogue_app'@'localhost' IDENTIFIED BY 'a-real-strong-password-here';
GRANT SELECT, INSERT, UPDATE, DELETE ON catalogue.* TO 'catalogue_app'@'localhost';
FLUSH PRIVILEGES;
```

That user's own real password is what goes into the server's real `config.php` — never the `root` password, and never a copy of whatever was used locally.

> **DELIBERATELY NOT GRANTING DROP OR CREATE**
>
> `GRANT SELECT, INSERT, UPDATE, DELETE` gives the application account exactly what the running app actually needs to do its job — every query this course has built since Chapter 3 is one of those four operations. It deliberately omits schema-altering privileges like `DROP` or `ALTER`: if a bug, or a successful SQL injection attempt somehow slipping past the prepared-statement discipline built since Chapter 1, ever tried to run something destructive against the schema itself, this account genuinely couldn't do it.

Production Error Display

A local dev setup benefits from seeing PHP errors directly in the browser. On a real, publicly reachable server, that's a real information leak — a stack trace can reveal file paths, and in a worse case, a poorly-written error message could even leak a database query. Production PHP should show nothing to the visitor, while still recording everything to a log file only the server administrator can read:

```
display_errors = Off
log_errors = On
error_log = /var/log/php/catalogue_errors.log
```

These are real `php.ini` directives — set either in the server's own global `php.ini`, or scoped to just this site via an Apache `.htaccess` (`php_flag display_errors Off`) or an nginx-fpm pool's own configuration, depending on which of this site's own web-server courses' setup is actually in use.

HTTPS Is Not Optional Here

This project's own admin pages accept real form submissions and, indirectly, real MySQL credentials flow through the same connection during setup. Serving any of it over plain HTTP would send that traffic unencrypted. This site's own *HTTPS/TLS Fundamentals* course covers obtaining and configuring a certificate in full — that setup is a real prerequisite for this project going live, not an optional hardening step to add later.

A Simple, Real Backup Habit

A personal catalogue with no other copy of its own data is one dropped table away from losing months of manual entry. A single cron-scheduled command covers the realistic case:

```
# Add to crontab -e, runs daily at 2am:
0 2 * * * mysqldump -u catalogue_app -p'the-real-password' catalogue > /home/user/backu
```

A REAL, HONEST SCOPE NOTE

A cron-scheduled `mysqldump` writing to the same server's own disk isn't a complete backup strategy — a real disk failure would take the backups down with the live data. For a small personal project, it's still a genuine, meaningful improvement over no backup at all, and copying the resulting `.sql` files somewhere off that same machine occasionally (even just downloading them by hand once in a while) closes most of the realistic gap without needing a dedicated backup service.

Hands-On Exercises

EXERCISE 1

Create `config.example.php` with placeholder credentials, add `config.php` to a real `.gitignore` file, and confirm (with `git status`, assuming the project is a Git repository) that `config.php` genuinely does not appear as a trackable file once `.gitignore` is in place.

View solution

EXERCISE 2

Create the `catalogue_app` MySQL user with exactly the privileges shown in this chapter, then confirm — by trying and expecting it to fail — that a query like `DROP TABLE items;` run as that user is correctly rejected by MySQL with a permissions error.

View solution

EXERCISE 3

Set `display_errors = Off` and `log_errors = On` in a local PHP test setup, deliberately trigger a real PHP error (e.g. calling an undefined function), and confirm the error is genuinely absent from the browser output while still appearing in the configured log file.

View solution

CHAPTER 9 QUICK REFERENCE

- `config.php` — gitignored, created directly on the server; `config.example.php` (placeholder values) is what actually gets committed
- **Schema migration** — `mysqldump` locally, then imported into a freshly-created, correctly-charset database on the server
- **Dedicated MySQL user** — `SELECT` / `INSERT` / `UPDATE` / `DELETE` only, no `DROP` / `ALTER`, never the app's own connection running as `root`
- **Production error handling** — `display_errors = Off`, `log_errors = On`, errors visible only in a server-side log file

- **HTTPS** — a real prerequisite, not optional, per this site's own *HTTPS/TLS Fundamentals* course
- **Backups** — a scheduled `mysqldump` cron job, honestly scoped as "better than nothing" rather than a complete disaster-recovery plan
- **Next chapter:** Capstone — integrating the catalogue into the existing Astro site

CHAPTER 10: CAPSTONE — INTEGRATING THE CATALOGUE INTO THE EXISTING SITE

Personal Catalogue: PHP & MySQL

Chapter 10 · Capstone

Nine chapters have built a genuinely working catalogue — add/edit/delete, type-specific fields, tags, search, an N+1-avoiding list view, fast bulk entry, and a real deployment plan. This closing chapter covers the actual point of building it: getting it in front of the user, mounted onto their real, existing Astro-based site, then reviews the whole course chapter by chapter.

Three Real Ways to Mount a PHP App Onto an Astro Site

Astro's own live site is a static build served by a web server — it has no PHP runtime of its own, and doesn't need one for its own pages. That means integrating this project isn't a matter of "adding it into Astro" the way a new Astro page would be added — it's a matter of getting both apps served correctly, side by side, behind the same web server:

OPTION	WHAT IT LOOKS LIKE	REAL EFFORT
Subdomain	`catalogue.example.com`, a fully separate vhost	Lowest — no code changes at all, just DNS + a new vhost
Path-mounted reverse proxy	`example.com/catalogue/`, same domain as the Astro site	Low — one nginx `location` block, no PHP code changes
JSON API + Astro frontend	PHP returns data only; an Astro page/island renders it	Highest — every page effectively rebuilt in Astro/React

The Realistic Choice: Path-Mounted Reverse Proxy

For a personal project like this, a subdomain works but feels like more infrastructure than the project needs, and a full JSON-API rewrite defeats much of the point of having already built a working PHP app across nine chapters. A path-mounted reverse proxy is the real middle ground: one domain, the existing Astro site untouched, and the catalogue reachable at its own clean path with zero changes to any of the PHP code built so far.

Assuming nginx is already serving the Astro site's static build (per this site's own *Setting Up a Web Server on Debian* and *Nginx In Depth* courses), and PHP-FPM is running the catalogue on a local port or socket, the real config addition is one `location` block:

`/etc/nginx/sites-available/example.com (excerpt)`

```
server {
    listen 443 ssl;
    server_name example.com;

    # The existing Astro static site
    root /var/www/example.com/dist;
    index index.html;

    location / {
        try_files $uri $uri/ =404;
    }

    # The catalogue, mounted at /catalogue/
    location /catalogue/ {
        alias /var/www/catalogue/;
        index index.php;

        location ~ \.php$ {
            fastcgi_pass unix:/run/php/php8.3-fpm.sock;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME $request_filename;
            include fastcgi_params;
        }
    }
}
```

WHY ALIAS, NOT ROOT, FOR THE NESTED LOCATION

Using `alias` instead of `root` inside the `/catalogue/` block matters specifically because the two paths don't share the same tail: a request for `/catalogue/index.php` needs to resolve to the real file at `/var/www/catalogue/index.php`, not `/var/www/catalogue/catalogue/index.php` (which is what `root` would produce, since it appends the full matched URI onto the given path). `alias` replaces the matched `location` prefix with the given directory instead of appending to it, which is exactly the behavior needed here.

EVERY INTERNAL LINK NEEDS THE /CATALOGUE/ PREFIX

Every `header('Location: /add_item.php...')` redirect and every `href="edit_item.php?..."` link written across Chapters 3-8 was built assuming the app sits at the web root. Mounted at `/catalogue/` instead, absolute redirects like `header('Location: /add_item.php?added=1')` would actually send the browser to `example.com/add_item.php` — outside the mounted path entirely, a real 404 on the Astro site. Every absolute redirect needs a `/catalogue` prefix, or (the more robust fix) needs to become a relative redirect (`header('Location: add_item.php?added=1')`) that stays correct regardless of where the app is ever mounted.

Making It Feel Like Part of the Same Site

A reverse proxy solves reachability, not visual consistency — without any further work, the catalogue's own plain unstyled markup would look nothing like the Astro site's real dark theme. A lightweight, real fix: extract the catalogue's own repeated page chrome into one shared partial, styled to visually match the main site's own established look:

`partials/header.php`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>My Catalogue</title>
    <link rel="stylesheet" href="/catalogue/assets/catalogue.css">
</head>
<body>
<nav>
    <a href="/">← Back to Main Site</a>
    <a href="/catalogue/index.php">Browse</a>
    <a href="/catalogue/search.php">Search</a>
    <a href="/catalogue/add_item.php">Add Item</a>
</nav>
<main>
```

`catalogue.css` itself doesn't need to be built from scratch — copying the real dark-background/accent-color CSS custom properties already used across this site's own Astro build

gives the catalogue a genuinely matching look with a small, one-time styling effort, rather than a full visual redesign.

A Real Alternative for Deeper Integration: A JSON Endpoint

If, later, the catalogue's own search results need to appear directly on an Astro page (rather than linking out to a separate PHP page), the reverse-proxy setup already built supports that too — a small PHP script can return JSON instead of HTML, and an Astro island can `fetch()` it client-side:

`api_search.php`

```php
<?php
require_once __DIR__ . '/config.php';
header('Content-Type: application/json');

$searchTerm = trim($_GET['q'] ?? '');

$stmt = $pdo->prepare('
    SELECT id, title, creator FROM items
    WHERE title LIKE :query OR creator LIKE :query
    ORDER BY title
    LIMIT 20
');
$stmt->execute(['query' => '%' . $searchTerm . '%']);

echo json_encode($stmt->fetchAll());
```

This isn't built out further in this course — it's flagged here honestly as the real next step if the project's own scope ever grows toward a tighter Astro-embedded experience, reusing exactly the same PDO connection and prepared-statement discipline every other chapter already established.

Capstone: What Each Chapter Actually Built

The finished project is a genuine sum of nine real prior chapters, not a single new piece written for this capstone:

CHAPTER	WHAT IT CONTRIBUTED
1	The real PDO connection, native prepared statements, and the `catalogue` database itself
2	The shared `items` table, `item_types`, and the `tags` / `item_tags` many-to-many pairing
3	Add/edit/delete, with delete deliberately restricted to POST
4	Dynamic, type-aware labels and a real per-type `release_year` sanity check
5	Tag creation, attachment, and the diff-based `sync_item_tags()`
6	Real search across title/creator, with type/tag filters and `LIKE` escaping
7	An N+1-avoiding list view via `GROUP_CONCAT`, and the detail page
8	Keyboard-friendly quick-add and a transaction-backed bulk importer
9	Config separation, schema migration, a least-privilege database user, and production error handling
10	Mounting the finished app behind the same domain as the real, existing Astro site

Where Barcode Scanning Fits Now

Chapter 1's own real deadline pushed barcode scanning out of scope for this course entirely — items get added manually, via the fast bulk-paste path built in Chapter 8. With the core catalogue now genuinely working and deployed, barcode scanning becomes a real, concrete next feature rather than an abstract "someday": a barcode's ISBN or UPC number looked up against a free API (Open Library for books is a natural real fit, matching this site's own established Food Tracker courses' use of Open Food Facts for a very similar lookup-by-code pattern), pre-filling `add_item.php`'s own form instead of requiring every field typed by hand.

Hands-On Exercises

EXERCISE 1

Add the nginx `location /catalogue/` block from this chapter to a real (or test) nginx config, alongside a real static `index.html` served at the site root. Confirm both `example.com/` (or your test domain) and `example.com/catalogue/index.php` resolve correctly to their own separate real content.

View solution

EXERCISE 2

Update every `header('Location: ...')` redirect across `add_item.php`, `edit_item.php`, and `delete_item.php` to use a relative path instead of an absolute one (per this chapter's own warning), then confirm the app still works correctly whether it's mounted at the web root or at `/catalogue/`.

View solution

EXERCISE 3

Build `api_search.php` from this chapter, visit it directly in a browser with a real search term (e.g. `api_search.php?q=Python`), and confirm the response is genuine, valid JSON — not an HTML page — by checking the response's own `Content-Type` header and validating the body with a JSON linter or `php -r 'var_dump(json_decode(file_get_contents("php://stdin")));'`.

View solution

CHAPTER 10 & COURSE QUICK REFERENCE

- **Path-mounted reverse proxy** — one nginx `location /catalogue/` block, zero PHP code changes, same domain as the existing Astro site
- **`alias`, not `root`** — required for a nested location whose URL path doesn't match its real filesystem path
- **Relative redirects** — the real fix that makes every prior chapter's own `header('Location: ...')` call correct regardless of mount path
- **A shared header partial** — the lightweight fix for visual consistency with the main site, reusing its own real CSS custom properties
- **A JSON endpoint** — the real, honest next step if this project's scope ever grows toward tighter Astro-embedded integration

- **Barcode scanning** — the feature Chapter 1 deliberately deferred, now a real, concrete next step once the core app is live
- **Course complete** — every chapter's own piece (Chapters 1-9) verified working together as one deployed, real personal catalogue