Mapping HTML Forms to Database Tables

When a user submits information through an HTML form, that data needs to be organized and stored in a database. This process requires careful mapping, where each form field corresponds to a specific column in a database table.

---

1. Anatomy of the HTML Form Field

The crucial link between the HTML form and the database is the `name` **attribute** of the input tag. This attribute provides the identifier (or "key") for the data submitted by the user.

HTML Element	**Key Attribute**	**Purpose**
<input type="text" **name**="user_name">	user_name	Identifies the data submitted for the user's name.
<input type="email" **name**="user_email">	user_email	Identifies the data submitted for the user's email address.
<textarea **name**="bio_text">	bio_text	Identifies the text submitted in the large text area (bio).
<select **name**="country_select">	country_select	Identifies the selection made in the dropdown menu.

---

2. Anatomy of the Database Table

A database stores data in **tables**, which are composed of **columns** (also called **fields**) and **rows** (also called **records**).

Component	**Description**	**Example (for a Profile)**
Table	The collection of related data.	User_Profiles
Column (Field)	Defines a specific piece of information to be stored.	user_name, email_address, join_date

Component	Description	Example (for a Profile)
Data Type	Specifies the type of data that can be stored in the column (e.g., text, number, date).	VARCHAR, INT, DATE

3. The Mapping Process (Generic Profile Example)

Let's use a simple **Generic Profile Form** and map its fields to a database table called User_Profiles.

A. The HTML Form Fields

HTML Field (Example Label)	name Attribute (The Key)	Data Type (Expected)
Full Name	full_name	Text
Email Address	email_address	Text (Unique)
Age	user_age	Number (Integer)
Bio	user_bio	Long Text
Join Date (e.g., hidden field)	join_date	Date

B. The Database Table Schema (User_Profiles)

The columns in the table are created to match the expected data from the form fields, including an **ID column** which is a best practice for a unique identifier.

Column Name	Data Type (e.g., SQL)	Purpose	Mapped HTML name
user_id	INT PRIMARY KEY	**Unique Identifier** for each user (Not from form).	N/A
full_name	VARCHAR(100)	Stores the user's full name.	full_name

Column Name	Data Type (e.g., SQL)	Purpose	Mapped HTML name
email_address	VARCHAR(150)	Stores the email address.	email_address
user_age	INT	Stores the user's age.	user_age
user_bio	TEXT	Stores the longer bio information.	user_bio
join_date	DATE	Stores the date the profile was created.	join_date

C. How the Data Flows

When the user clicks the "Submit" button, the web server/application-side code (like PHP, Python, or Node.js) does the following:

1. It receives the submitted data as key-value pairs (e.g., `full_name` = "Jane Doe", `user_age` = "28").
2. It uses a SQL command (like `INSERT INTO`) to place the values into the correct columns of the `User_Profiles` table.

For example, the server might execute a command that conceptually looks like this:

```
INSERT INTO User_Profiles (full_name, email_address, user_age, user_bio, join_date) VALUES ('Jane Doe', 'jane@example.com', 28, 'Love coding!', '2025-11-12');
```

---

4. Data Integrity: Required Fields and `NOT NULL`

To ensure the database collects essential information, we use validation on the HTML form and structural constraints in the database. These two concepts must align.

A. HTML Required Field

In HTML, the **`required` attribute** is used on an input field to tell the browser that the user *must* enter data before submitting the form.

HTML Element	Purpose
`<input type="text" name="full_name" **required**>`	Prevents the form from submitting if the **Full Name** field is empty.

HTML Element	Purpose
`<input type="email" name="email_address" required>`	Prevents submission if the **Email Address** field is empty.

B. Database NOT NULL Constraint

In the database table definition (the schema), the **NOT NULL constraint** is applied to a column to prevent any row from being created or updated unless that column has a value. If the web application tries to insert a record where a NOT NULL column is missing data, the database will return an error.

C. The Mapping: Enforcing Requirements

The **best practice** is to apply the required attribute to any form field that corresponds to a database column defined as NOT NULL.

Column Name	Data Type (e.g., SQL)	Constraint	Mapped HTML name	HTML Field Attribute
full_name	VARCHAR(100)	**NOT NULL**	full_name	**required**
email_address	VARCHAR(150)	**NOT NULL**	email_address	**required**
user_bio	TEXT	*NULL allowed*	user_bio	*(No required attribute)*

Concept	Purpose	Where it's Applied
required	User Interface (UI) validation; improves user experience.	HTML Form
NOT NULL	Database structural integrity; a fundamental data rule.	Database Table Column

This dual approach provides the highest level of data integrity: the **HTML** helps the user, and the **database constraint** provides a final, secure safety net.

---

5. Using Multiple Database Tables (Normalization)

The goal of a well-designed database is to avoid **redundancy** (repeating data) and maintain data consistency. This is achieved by splitting information into multiple, specialized tables and linking them together using **relationships**. This process is called **Normalization**.

When to Use Multiple Tables

You should use multiple tables whenever a piece of information can have **multiple values** for a single record, or when a value in one table is repeated across many records.

A common example is an **Address** for a user profile: A user might have many addresses (Home, Work, Shipping, Billing). If you tried to store all addresses in the User_Profiles table, you'd have to add many columns (e.g., shipping_address_line1, billing_address_line1), which is inefficient and rigid.

How to Map and Link Tables

When you split the data, you link the tables using a **Foreign Key**.

1. **Primary Key (PK):** A column (like user_id) that uniquely identifies each row in its *own* table.
2. **Foreign Key (FK):** A column in **Table B** that refers back to the Primary Key in **Table A**. This creates the link.

Mapping Example: One User, Many Addresses

Element	**Single Table Approach (Bad)**	**Multiple Table Approach (Good)**
Data Redundancy	User's name/email is repeated for every address.	User details stored only once in User_Profiles.
Flexibility	Limited to the number of address columns you define.	Unlimited addresses can be added for one user.

Table 1: User_Profiles **(The Main Data)**

Column Name	Data Type	Constraint	Primary Key?
user_id	INT	NOT NULL	**PK** (Unique ID for this table)
full_name	VARCHAR(100)	NOT NULL	No
email_address	VARCHAR(150)	NOT NULL	No

Table 2: User_Addresses **(The Repeating Data)**

Column Name	Data Type	Constraint	Foreign Key?	Links to...
address_id	INT	NOT NULL	PK	N/A
user_id_fk	INT	NOT NULL	**FK**	User_Profiles.user_id
street_address	VARCHAR(255)	NOT NULL	No	N/A
city	VARCHAR(100)	NOT NULL	No	N/A

By using the **Foreign Key** (user_id_fk), you can easily find all the addresses that belong to a specific user, linking the two tables efficiently. The HTML form data for the address fields would be processed and inserted into the User_Addresses table, while the main profile data goes to User_Profiles.

E-commerce Order Mapping: One-to-Many Relationship

A single order placed by a customer needs to be split into at least two tables to follow the principle of **normalization**:

1. Orders **Table:** Stores information that is **unique to the entire transaction** (like the date, total amount, and shipping details).
2. Order_Items **Table:** Stores information about **each individual product** included in the order (the items, their quantity, and price at the time of purchase).

A. HTML Form Fields for an Order

The form collects data for the customer and the items they want to buy.

HTML Field (Example Label)	**name Attribute (The Key)**	**Expected Data Type**	**Destination Table**
Customer Name	customer_name	Text	Orders
Shipping Address	shipping_address	Text	Orders
Total Amount	order_total	Decimal/Currency	Orders
Product ID (Hidden field or checkbox)	product_id[]	Number	Order_Items
Quantity (for that product)	quantity[]	Number	Order_Items

Note: The [] *in the name attributes for products indicates that multiple values will be sent for a single submission, which is critical for handling multiple line items.*

B. The Database Table Schema

When the form is submitted, the server-side code must execute **multiple** `INSERT` **statements** to populate both tables, ensuring the records are correctly linked.

Table 1: Orders *(One Record per Transaction)*
This table handles the one-time details of the order.

Column Name	**Data Type (e.g., SQL)**	**Constraint**	**Primary Key?**	**Mapped HTML name**
order_id	INT	NOT NULL	**PK**	N/A (Generated by DB)
customer_name	VARCHAR(100)	NOT NULL	No	customer_name

Column Name	Data Type (e.g., SQL)	Constraint	Primary Key?	Mapped HTML name
shipping_address	VARCHAR(255)	NOT NULL	No	shipping_address
order_date	DATE	NOT NULL	No	N/A (Generated by server)
total_amount	DECIMAL(10, 2)	NOT NULL	No	order_total

Table 2: Order_Items (Many Records per Transaction)
This table holds the details for each product in the order and uses a **Foreign Key** to link back to the main order.

Column Name	Data Type (e.g., SQL)	Constraint	Foreign Key?	Links to...
item_id	INT	NOT NULL	PK	N/A
order_id_fk	INT	NOT NULL	**FK**	Orders.order_id
product_id	INT	NOT NULL	No	product_id[]
quantity	INT	NOT NULL	No	quantity[]
price_at_purchase	DECIMAL(8, 2)	NOT NULL	No	N/A (Fetched from Products table)

C. The Flow of Data

1. **Insert Order:** The code first inserts the customer and shipping data into the **Orders** table.

2. **Get ID:** The database returns the newly created **order_id** (the Primary Key, e.g., 501).
3. **Insert Items:** For *each* product submitted in the form, the code then inserts a separate row into the **Order_Items** table, using that retrieved **order_id** **(501)** as the **Foreign Key** (order_id_fk).
4.

This establishes a **one-to-many relationship**: **one** record in the Orders table can be linked to **many** records in the Order_Items table.