BUT INFORMATIQUE · 2E ANNÉE · R3.01 DÉVELOPPEMENT WEB

TP2
Le site des prix Nobel

PHP & PostgreSQL : les concepts clés des 14 exercices

Correction du TP · Support de projection et de révision



Ex. 1 → 14

Quatre étapes pour construire un site dynamique

1

Lire la base

Ex. 1 → 4

- Schéma, clés primaires et étrangères
- Connexion avec PDO
- `fetch()` / `fetchAll()`
- Paramètres d'URL
- Injection SQL et XSS

`nbpn.php` · `last.php`
`infos.php`

2

Organiser le code

Ex. 5 → 8

- Classe `Model` et singleton
- Gabarits de page (`begin.html`, `end.html`)
- Fonctions partagées
- Validation des entrées

`Model.php` · `home.php`
`last25.php` · `informations.php`

3

Modifier les données

Ex. 9 → 12

- Formulaires, GET et POST
- Ajout : `INSERT`
- Suppression : `DELETE`
- Modification : `UPDATE`

`form_add.php` · `add.php`
`remove.php` · `update.php`

4

Naviguer dans les données

Ex. 13 → 14

- Pagination : `LIMIT` / `OFFSET`
- Recherche multicritère
- Requête SQL construite dynamiquement
- Sessions

`list_all.php` · `search.php`
`form_search.php`

→ **Fil rouge.** Les exercices 1 à 5 sont des pages indépendantes. À partir de l'exercice 6, on construit **un seul site** « Nobel Prizes », enrichi d'une fonctionnalité à chaque exercice (dossiers `exercice6` → `exercice14`).

Architecture

Que se passe-t-il quand on ouvre une page PHP ?



Exécuté côté serveur

Le PHP tourne sur le serveur. Le navigateur ne reçoit que le HTML produit : « Afficher le code source » ne montre jamais le PHP.

HTTP n'a pas de mémoire

Chaque requête repart de zéro : les variables PHP disparaissent à la fin du script. Pour transmettre une information : URL (ex. 4, 8), formulaire (ex. 9 → 12), session (ex. 14).

Avec Docker

`docker compose up -d` démarre **web** (localhost:8080), **db** (PostgreSQL) et **adminer** (localhost:8081) pour explorer la base.

À RETENIR À chaque requête, PHP interroge la base puis fabrique une page HTML qu'il renvoie au navigateur.

La base de données : deux tables liées

nobels

id	SERIAL	PK
year	INT	NOT NULL
category	VARCHAR(128)	NOT NULL FK
name	VARCHAR(128)	NOT NULL
birthdate	INT	
birthplace	VARCHAR(64)	
county	VARCHAR(128)	
motivation	TEXT	

REFERENCES

clé étrangère

categories

category	PK

chemistry · economics
literature · medicine
peace · physics

DANS PSQL

```
\i database_nobel.sql
SELECT count(*) FROM nobels;    -- 839
```

- **Clé primaire (PK)** : identifie chaque ligne de façon unique (`id` , `category`).
- **Clé étrangère (FK)** : `nobels.category` doit exister dans `categories` . Sinon PostgreSQL refuse l'insertion, et PDO lève une exception (ex. 10).
- **SERIAL** : PostgreSQL numérote `id` tout seul. On ne le fournit jamais dans un `INSERT` .
- **NOT NULL** : `year` , `category` et `name` sont obligatoires. Les autres colonnes peuvent valoir `NULL` : il faudra le gérer à l'affichage.
- Le script commence par `DROP TABLE IF EXISTS` : on peut le relancer pour **remettre la base à zéro** après les tests d'ajout et de suppression.

À RETENIR Les contraintes (PK, FK, NOT NULL) protègent les données même si le PHP oublie une vérification.

Se connecter avec PDO

CONFIG.PHP (VERSION SIMPLIFIÉE)

```php
define('DB_DSN', 'pgsql:host=aquabdd;dbname=etudiants');
define('DB_USER', 'numero_etudiant');
define('DB_PASSWORD', 'code_INE');
```

EXERCICE2/NBPN.PHP

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

$bd = new PDO(DB_DSN, DB_USER, DB_PASSWORD);
$bd->setAttribute(PDO::ATTR_ERRMODE,
                  PDO::ERRMODE_EXCEPTION);

$req = $bd->query('SELECT count(*) FROM nobels');
$nb  = $req->fetchColumn();        // 839

echo "<p>La base contient $nb prix Nobel.</p>";
```

- **PDO** (*PHP Data Objects*) : la même API pour PostgreSQL, MySQL, SQLite… Seul le DSN change.
- **DSN** : `pilote:host=serveur;dbname=base`.
- **Mode exception** : une erreur SQL lève une `PDOException` au lieu de passer inaperçue (c'est le mode par défaut depuis PHP 8, mais on l'écrit explicitement).
- `query()` exécute une requête **sans donnée venant de l'utilisateur**.
- `fetchColumn()` lit directement une valeur : idéal pour `count(*)`.
- `config.php` : les identifiants à **un seul endroit**, partagés par tous les exercices.

À RETENIR `new PDO(dsn, login, mot de passe)` + mode exception : les deux premières lignes de tout accès à la base.

Lire les résultats : `fetch()` ou `fetchAll()` ?

LAST.PHP — FETCH()

```php
$req = $bd->query('SELECT * FROM nobels
                   ORDER BY year DESC LIMIT 25');

// une ligne à chaque appel, puis false
while ($ligne = $req->fetch(PDO::FETCH_ASSOC)) {
    echo htmlspecialchars($ligne['name']);
}
```

LAST_FETCHALL.PHP — FETCHALL()

```php
$req = $bd->query('SELECT * FROM nobels
                   ORDER BY year DESC LIMIT 25');

// toutes les lignes d'un coup, dans un tableau
$nobels = $req->fetchAll(PDO::FETCH_ASSOC);
foreach ($nobels as $ligne) {
    echo htmlspecialchars($ligne['name']);
}
```

	fetch()	fetchAll()
Renvoie	une ligne, puis `false`	un tableau de lignes (`[]` si vide)
Mémoire	une ligne à la fois	tout le résultat
Idéal pour	un seul résultat attendu (ex. 4, 8)	renvoyer une liste depuis le Model (ex. 5 →)

Mode	Une ligne ressemble à…
FETCH_ASSOC	`['id' => 19, 'year' => 1903, 'name' => …]`
FETCH_COLUMN	avec `fetchAll` : `['chemistry', 'economics', …]` (ex. 9)
par défaut	`FETCH_BOTH` : chaque valeur en double (nom + numéro)

À RETENIR `fetch()` renvoie `false` quand il n'y a plus de ligne : c'est ce qui arrête la boucle `while`.

Paramètres d'URL et injection SQL

`infos.php?id=19` → `$_GET['id']` vaut `"19"`, **toujours une chaîne**, et l'utilisateur peut y écrire n'importe quoi.

✕ À NE PAS FAIRE : CONCATÉNER

```php
$req = $bd->query('SELECT * FROM nobels WHERE id = '
                  . $_GET['id']);
```

L'attaque

URL saisie : `infos.php?id=0 OR 1=1`

Requête exécutée :

`SELECT * FROM nobels WHERE id = 0 OR 1=1`

La condition est toujours vraie : **toute la table** est renvoyée. Avec d'autres valeurs, on peut lire d'autres tables, voire modifier les données.

✓ À FAIRE : REQUÊTE PRÉPARÉE (INFOS.PHP)

```php
if (!isset($_GET['id'])) {
    echo '<p>Aucun identifiant</p>';
} else {
    $req = $bd->prepare('SELECT * FROM nobels WHERE id = :id');      // ①
    $req->bindValue(':id', (int) $_GET['id'], PDO::PARAM_INT);       // ②
    $req->execute();                                                 // ③
    $ligne = $req->fetch(PDO::FETCH_ASSOC);    // false si aucun prix
}
```

1. `prepare()` : le SQL avec un **marqueur** `:id`
2. `bindValue()` : la valeur, avec son type
3. `execute()` puis `fetch()`

Le SQL est analysé **avant** de recevoir les données : une valeur ne peut jamais devenir du code SQL.

À RETENIR Toute valeur venant de l'utilisateur passe par un marqueur `:nom`, jamais par une concaténation.

Afficher sans danger : `htmlspecialchars()`

Scénario : quelqu'un ajoute (ex. 10) un prix dont le nom est `<script>alert('Piraté !')</script>`

✕ LE NAVIGATEUR EXÉCUTE LE SCRIPT CHEZ CHAQUE VISITEUR

```php
echo '<td>' . $nobel['name'] . '</td>';
```

✓ LE NOM S'AFFICHE COMME DU SIMPLE TEXTE

```php
echo '<td>' . htmlspecialchars($nobel['name']) . '</td>';
```

- Aussi dans les attributs : `value="<?php echo htmlspecialchars($infos['name']); ?>"` (ex. 12).
- Colonne pouvant être `NULL` : `htmlspecialchars((string) $infos['county'])`, car passer `null` est déprécié depuis PHP 8.1.

Caractère	devient
`<` `>`	`<` `>`
`&`	`&`
`"` `'`	`"` `'`

Les deux règles d'or

1. Vers la **base** → requête préparée
2. Vers la **page** → `htmlspecialchars()`

Cette faille s'appelle **XSS** (*Cross-Site Scripting*).

À RETENIR On échappe au moment d'afficher, selon la destination : dans du HTML, c'est `htmlspecialchars()`.

La classe `Model` : un singleton

EXERCICE5/MODEL.PHP

```php
class Model {
    private static $instance = null;   // l'unique objet Model
    private $bd;                       // la connexion PDO

    private function __construct() {   // new Model() interdit dehors
        $this->bd = new PDO(DB_DSN, DB_USER, DB_PASSWORD);
        $this->bd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    public static function get_model() {
        if (self::$instance === null) {
            self::$instance = new Model();   // 1er appel : création
        }
        return self::$instance;              // ensuite : le même objet
    }

    public function get_nb_nobel_prizes() {
        $req = $this->bd->query('SELECT count(*) FROM nobels');
        return (int) $req->fetchColumn();
    }
}
```

- **Singleton** : une classe dont il n'existe qu'**une seule instance**.
- Constructeur `private` : impossible d'écrire `new Model()` hors de la classe.
- `static` : la propriété ou la méthode appartient à la **classe**, pas à un objet. On y accède avec `self::` ou `Model::`.
- `$this->bd` : propriété de l'objet courant.
- Bénéfice : **une seule connexion** à la base par page, même si plusieurs fichiers utilisent le Model.

UTILISATION

```php
$nb = Model::get_model()->get_nb_nobel_prizes();
Model::get_model() === Model::get_model();   // true
```

À RETENIR Singleton = constructeur privé + propriété statique + méthode statique qui renvoie toujours le même objet.

Le Model : le seul fichier qui parle SQL

LES PAGES

- home.php
- last25.php
- informations.php
- form_add.php
- add.php
- remove.php
- update.php
- list_all.php
- search.php

Ex.	Méthode de Model	Requête SQL
5	`get_last()`	`SELECT … ORDER BY year DESC, id LIMIT 25`
5	`get_nb_nobel_prizes()`	`SELECT count(*) FROM nobels`
8	`get_nobel_prize_informations($id)`	`SELECT … WHERE id = :id`
9	`get_categories()`	`SELECT category FROM categories`
10	`add_nobel_prize($infos)`	`INSERT INTO nobels … VALUES …`
11	`remove_nobel_prize($id)`	`DELETE … WHERE id = :id`
12	`updateNobelPrize($infos)`	`UPDATE … SET … WHERE id = :id`
13	`get_nobel_prizes_with_limit(…)`	`… LIMIT :limit OFFSET :offset`
14	`search_nobel_prizes(…)`	`… WHERE` construit selon les filtres

PostgreSQL
nobels
categories

À RETENIR Les pages gèrent les paramètres et l'affichage ; le Model gère les données. C'est un premier pas vers l'architecture MVC.

Structure d'une page et gabarits partagés

EXERCICE7/LAST25.PHP

```php
<?php
require_once "Model.php";
$nobels = Model::get_model()->get_last();    // ① les données en premier

$title = "List of the last 25 Nobel prizes"; // ② lu par begin.html
require "begin.html";                        // ③ haut de page commun
?>
<h1> List of the last 25 Nobel prizes </h1>
<table>
    <?php foreach ($nobels as $nobel) { ?>
        <tr>
            <td><?php echo htmlspecialchars($nobel['name']); ?></td>
            <td><?php echo $nobel['year']; ?></td>
        </tr>
    <?php } ?>
</table>
<?php require "end.html"; /* ④ bas de page commun */ ?>
```

BEGIN.HTML (EXTRAIT)

```php
<title><?php echo $title; ?> – Nobel Prizes</title>
...
<nav> le menu, commun à toutes les pages </nav>
<main id="content">
```

- Un fichier inclus par `require` est **exécuté comme du PHP**, même s'il s'appelle `.html`.
- Il **partage les variables** de la page : `$title` doit être défini avant le `require`.
- `require` : erreur fatale si le fichier manque. `include` : simple avertissement. `require_once` : jamais inclus deux fois (classes, fonctions).
- On alterne PHP et HTML : `<?php foreach (…) { ?>` … HTML … `<?php } ?>`

À RETENIR D'abord la logique (données, vérifications), ensuite l'affichage. Le haut et le bas de page sont écrits une seule fois.

Ex. 8

Valider les entrées : la page d'informations

FUNCTIONS.PHP : FONCTIONS PARTAGÉES PAR PLUSIEURS PAGES

```php
function is_strictly_positive_integer($value) {
    return is_string($value)                  // pas un tableau (?id[]=1)
        && preg_match('/^[0-9]+$/', $value)   // uniquement des chiffres
        && (int) $value > 0;                  // et pas 0
}
```

INFORMATIONS.PHP

```php
if (!isset($_GET['id']) || !is_strictly_positive_integer($_GET['id'])) {
    $message = 'The id must be a strictly positive integer.';   // absent ou invalide
} else {
    $infos = Model::get_model()->get_nobel_prize_informations((int) $_GET['id']);
    if ($infos === false) {
        $message = 'There is no nobel prize with such id.';     // inexistant
    }
}
```

Dans le tableau des prix, chaque nom devient un lien :

```
<a href="informations.php?id=<?php echo $nobel['id']; ?>">
```

`$_GET['id']`	Résultat
`"19"`	✓ Marie Curie, 1903
`"abc"` `"1.5"` `"-3"`	✕ invalide
`"0"`	✕ invalide
`?id[]=1` (tableau)	✕ invalide
absent	✕ invalide
`"99999"`	✓ format · ✕ inexistant

- Trois cas à traiter : **absent**, **invalide**, **valide mais inexistant**.
- `=== false` : comparaison stricte, car `fetch()` renvoie exactement `false`.

À RETENIR Ne jamais faire confiance à l'URL : vérifier la présence, puis le format, puis l'existence en base.

Formulaires : GET ou POST, et champs générés depuis la base

FORM_ADD.PHP (EXTRAIT)

```php
<form action="add.php" method="post">
  <label> Name (required): <input type="text" name="Name"/> </label>
  <label> Year (required): <input type="text" name="Year"/> </label>

  <fieldset>
    <legend> Category (required): </legend>
    <?php foreach (Model::get_model()->get_categories() as $category) { ?>
      <label>
        <input type="radio" name="Category"
               value="<?php echo htmlspecialchars($category); ?>"/>
        <?php echo htmlspecialchars(ucfirst($category)); ?>
      </label>
    <?php } ?>
  </fieldset>

  <textarea name="Motivation"></textarea>
  <input type="submit" value="Add in database"/>
</form>
```

	GET	POST
Données	dans l'URL `?id=19`	dans le corps de la requête
En PHP	`$_GET`	`$_POST`
Usage	lire, chercher, liens partageables	modifier la base

- L'attribut `name` devient la clé : `name="Year"` → `$_POST['Year']`, toujours une chaîne.
- Boutons radio générés depuis la table `categories` : une nouvelle catégorie apparaît sans toucher au HTML.
- `get_categories()` utilise `fetchAll(PDO::FETCH_COLUMN)` → `['chemistry', 'economics', …]`.
- Accessibilité : un `<label>` par champ, `<fieldset>` + `<legend>` pour un groupe.

À RETENIR `name` du champ = clé dans `$_GET` ou `$_POST`. On utilise POST pour tout ce qui modifie la base.

Ajouter un prix : `INSERT`

form_add.php → POST → add.php → check_data() → Model::add_nobel_prize() → INSERT → message

MODEL.PHP

```php
public function add_nobel_prize($infos) {
    $req = $this->bd->prepare('INSERT INTO nobels
            (year, category, name, birthdate,
             birthplace, county, motivation)
        VALUES (:year, :category, :name, :birthdate,
                :birthplace, :county, :motivation)');
    $req->execute(array(
        ':year'     => $infos['year'],
        ':category' => $infos['category'],
        ':name'     => $infos['name'],
        // … idem pour les 4 autres colonnes
    ));
    return $req->rowCount() === 1;   // 1 ligne ajoutée ?
}
```

ADD.PHP

```php
$infos = check_data();   // false si Name, Category ou Year invalide

if ($infos === false) {
    $message = 'Error: name, category and year are required.';
} else {
    try {
        if (Model::get_model()->add_nobel_prize($infos)) {
            $message = 'The nobel prize has been added.';
        }
    } catch (PDOException $e) {
        // ex. catégorie inexistante : clé étrangère violée
        $message = 'Error: the nobel prize has not been added.';
    }
}
```

Vérifier côté serveur : `check_data()` teste Name, Category et Year (entier > 0), applique `trim()` et met `null` dans les champs facultatifs vides.

`execute(array(…))` : une autre façon de lier les valeurs, sans `bindValue()`. Pas de colonne `id` : `SERIAL` s'en charge.

`rowCount()` : nombre de lignes touchées.
`try / catch` : intercepte l'erreur SQL pour afficher un message propre.

À RETENIR Valider les données, exécuter une requête préparée, puis vérifier le résultat (`rowCount()` ou exception).

Supprimer (`DELETE`) et modifier (`UPDATE`)

Supprimer · ex. 11

MODEL.PHP

```php
$req = $this->bd->prepare(
    'DELETE FROM nobels WHERE id = :id');
$req->bindValue(':id', $id, PDO::PARAM_INT);
$req->execute();
```

- Une icône par ligne du tableau : `<a href="remove.php?id=19">`
- `remove.php` : id valide ? → prix existant ? → suppression → message.
- **Sans `WHERE`, `DELETE` vide toute la table.**
- Pour aller plus loin : sur un vrai site, on supprime via un formulaire POST avec confirmation. Un simple lien GET peut être suivi par un robot ou déclenché par une page piégée (CSRF).

Modifier · ex. 12

FORM_UPDATE.PHP?ID=19 : FORMULAIRE PRÉ-REMPLI

```php
<input type="hidden" name="Id" value="<?php echo $infos['id']; ?>"/>

<input type="text" name="Name"
  value="<?php echo htmlspecialchars($infos['name']); ?>"/>

<input type="radio" name="Category" value="physics"
  <?php echo $category === $infos['category'] ? 'checked' : ''; ?>/>

<textarea name="Motivation"><?php echo htmlspecialchars(…); ?></textarea>
```

- **Champ caché** : l'id repart avec le formulaire, sans être affiché.
- Pré-remplir : `value`, `checked`, ou le contenu du `<textarea>`.
- `update.php` réutilise `check_data()`, puis `UPDATE nobels SET … WHERE id = :id`.

À RETENIR `UPDATE` et `DELETE` : toujours un `WHERE id = :id`. On réutilise les vérifications écrites pour l'ajout.

Pagination : `LIMIT` et `OFFSET`

839 prix ÷ 25 par page = **34 pages** (la dernière en contient 14)

page 3 = **OFFSET 50 LIMIT 25** → lignes 51 à 75

```
offset   = (page − 1) × 25
nb_pages = ceil(total / 25)
```

MODEL.PHP

```php
$req = $this->bd->prepare('SELECT * FROM nobels
    ORDER BY year DESC, id LIMIT :limit OFFSET :offset');
$req->bindValue(':limit',  $limit,  PDO::PARAM_INT);
$req->bindValue(':offset', $offset, PDO::PARAM_INT);
```

‹ 1 2 **3** 4 5 6 7 8 9 10 ›

LIST_ALL.PHP?PAGE=3

```php
$page = 1;                    // absente ou invalide → 1
if (isset($_GET['page'])
        && is_strictly_positive_integer($_GET['page'])) {
    $page = (int) $_GET['page'];
}
$offset = ($page - 1) * 25;
$nobels = $model->get_nobel_prizes_with_limit($offset, 25);
require "list_table.php";     // le tableau, partagé
display_pages_links($page, $nb_pages, 'list_all.php');
```

- **ORDER BY stable** (`, id`) : sinon une ligne peut apparaître sur deux pages, ou sur aucune.
- `list_table.php` : le même tableau pour `last25`, `list_all` et `search`. On évite de dupliquer le code.
- `display_pages_links()` : jusqu'à 10 numéros autour de la page courante, avec les flèches précédent / suivant.

À RETENIR `OFFSET = (page − 1) × taille de page`, avec un `ORDER BY` qui donne toujours le même ordre.

Recherche multicritère : un `WHERE` construit à la volée

MODEL.PHP : BUILD_SEARCH_CONDITIONS($FILTERS)

```php
$conditions = array();  $params = array();

if (isset($filters['Name'])) {
    $conditions[] = 'name ILIKE :name';                // insensible à la casse
    $params[':name'] = '%' . $filters['Name'] . '%';   // % = n'importe quoi
}
$signs = array('<=' => '<=', '>=' => '>=', '==' => '=');   // liste blanche
if (isset($filters['Year'], $filters['SignYear'])
        && isset($signs[$filters['SignYear']])) {
    $conditions[] = 'year ' . $signs[$filters['SignYear']] . ' :year';
    $params[':year'] = (int) $filters['Year'];
}
if (!empty($filters['Categories'])) {                  // un marqueur par valeur
    $markers = array();
    foreach (array_values($filters['Categories']) as $i => $category) {
        $markers[] = ':cat' . $i;
        $params[':cat' . $i] = $category;
    }
    $conditions[] = 'category IN (' . implode(', ', $markers) . ')';
}
$where = count($conditions) > 0 ? 'WHERE ' . implode(' AND ', $conditions) : '';
```

EXEMPLE DE RECHERCHE

Name : `curie` · Year `>=` `1900`

Categories : `chemistry`, `physics`

REQUÊTE OBTENUE

```sql
SELECT * FROM nobels
WHERE name ILIKE :name
  AND year >= :year
  AND category IN (:cat0, :cat1)
ORDER BY year DESC, id
LIMIT :limit OFFSET :offset
```

`:name` → `'%curie%'` · `:year` → `1900`
`:cat0` → `'chemistry'` · `:cat1` → `'physics'`

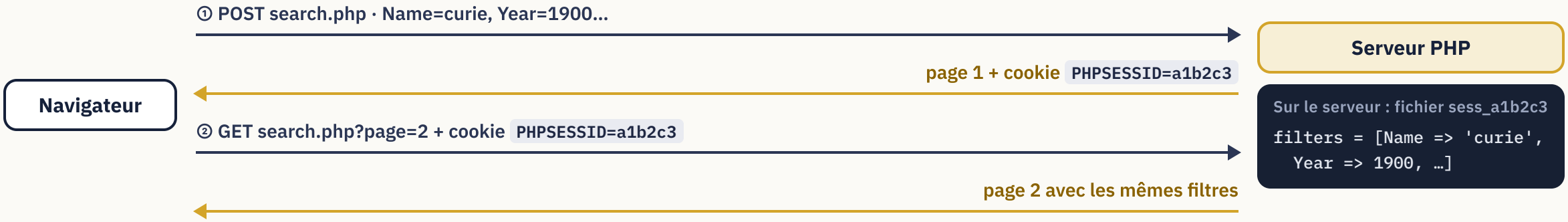
4 résultats : Marie Curie (×2), Pierre Curie, Irène Joliot-Curie.

- Un opérateur ne peut pas être un marqueur : on le choisit dans une **liste blanche**.
- `name="Categories[]"` : `$_POST['Categories']` est un tableau.

À RETENIR La structure du SQL peut varier, mais les valeurs passent toujours par des marqueurs.

Sessions : garder les filtres d'une page à l'autre

Le problème. Le formulaire de recherche est envoyé en POST. Mais le lien « page 2 » est un simple GET (`search.php?page=2`) : `$_POST` est vide et les filtres sont perdus.



SEARCH.PHP

```php
session_start();                    // avant tout affichage !

if (isset($_POST['Search'])) {      // on arrive du formulaire
    $filters = array();             // … rempli à partir de $_POST
    $_SESSION['filters'] = $filters; // on mémorise
}
// on arrive d'un lien « page N » : on relit les filtres
$filters = isset($_SESSION['filters']) ? $_SESSION['filters'] : array();
```

- Les données restent **sur le serveur**. Le navigateur ne garde qu'un identifiant, dans le cookie `PHPSESSID` .
- `session_start()` envoie un en-tête HTTP : il doit être appelé avant le moindre `echo` .
- Autre solution : mettre les filtres dans l'URL des liens (`?page=2&Name=curie`). La recherche devient alors partageable.

À RETENIR HTTP ne se souvient de rien d'une requête à l'autre ; la session, si.

Pour réviser

Huit pièges fréquents

1 ✗ Concaténer `$_GET` ou `$_POST` dans le SQL : injection SQL
✓ `prepare()` + marqueur `:id`

2 ✗ Afficher une donnée telle quelle : faille XSS
✓ `htmlspecialchars()` à chaque affichage

3 ✗ Lire `$_GET['id']` sans vérifier : « Undefined array key »
✓ `isset()` d'abord, puis le format

4 ✗ Utiliser le résultat de `fetch()` sans le tester
✓ `if ($ligne === false)` → message « inexistant »

5 ✗ `DELETE` ou `UPDATE` sans `WHERE` : toute la table
✓ toujours `WHERE id = :id`

6 ✗ Paginer sans ordre stable : lignes en double ou manquantes
✓ `ORDER BY year DESC, id`

7 ✗ Compter sur les contrôles du navigateur (`required`…)
✓ tout revérifier en PHP : une requête se forge facilement

8 ✗ `session_start()` après un `echo` : « headers already sent »
✓ en toute première ligne du script

À RETENIR Les deux règles d'or : requête préparée vers la base, `htmlspecialchars()` vers la page.

Pour réviser

Mémo de révision

PDO

`new PDO(dsn, u, p)`	connexion
`setAttribute(…)`	mode exception
`query($sql)`	requête sans donnée externe
`prepare($sql)`	requête avec marqueurs `:nom`
`bindValue(…)`	lie une valeur et son type
`execute([…])`	exécute (valeurs en tableau)
`fetch()`	1 ligne ou `false`
`fetchAll()`	tableau de lignes
`fetchColumn()`	1 valeur (ex. `count(*)`)
`rowCount()`	nb de lignes modifiées

SQL · CRUD

C	`INSERT INTO … VALUES (…)`	ex. 10
R	`SELECT … WHERE … ORDER BY …`	ex. 2 → 8
U	`UPDATE … SET … WHERE id = :id`	ex. 12
D	`DELETE FROM … WHERE id = :id`	ex. 11

`LIMIT n OFFSET k`	pagination (ex. 13)
`ILIKE '%x%'`	contient x, sans casse
`IN (a, b)`	appartient à une liste
`count(*)`	nombre de lignes

PHP CÔTÉ WEB

`$_GET`	paramètres d'URL
`$_POST`	formulaire POST
`$_SESSION`	mémoire entre pages
`isset()`	la clé existe ?
`htmlspecialchars()`	affichage sûr
`require_once`	inclure une fois
`preg_match()`	tester un format
`(int) $x`	conversion en entier
`self:: / $this->`	classe / objet
`ceil()`	arrondi supérieur

▶ S'entraîner : dans `TP2`, `docker compose up -d` puis `localhost:8080/exercice14/home.php`. Remettre la base à zéro dans psql : `\i /sql/database_nobel.sql`.