PHP

PHP is a **server-side scripting language** used mainly for building **dynamic websites** and **web applications** □□. It runs on the web server to generate HTML (and can also work with databases like **MySQL**) so pages can show user-specific or data-driven content.

- □□ PHP Zero → Expert (with a light WordPress focus) — Table of Contents
- Anki Cards: PHP Core Terminology
- Vibe Coding
    - LinkShorter v2 – Ausführliche Anleitung

□□ PHP Zero → Expert (with a light WordPress focus) — Table of Contents

> Structure: **Chapters** → **Subchapters** (you'll later request like `c2.3`).
> Each chapter builds forward, but key ideas are **revisited gently** so you don't have to constantly jump around. WordPress parallels are sprinkled throughout □□

---

1) Getting Started: PHP in the Real World □□

1.1 What PHP *is* (and isn't), where it runs, and what it's great at
1.2 Setting up your environment
1.2.1 Local stacks (XAMPP/MAMP/Laragon/Docker)
1.2.2 PHP versions, `php.ini`, and extensions
1.3 Your first PHP script: request → response mental model
1.4 How PHP projects are organized (files, includes, entry points)
1.5 WordPress parallel: where PHP "lives" in WordPress (themes, plugins, core)

---

2) PHP Fundamentals: Syntax, Types, and Control Flow □□

2.1 Variables, constants, and basic output
2.2 Types and type juggling (int/float/string/bool/null)
2.3 Strings in depth (interpolation, concatenation, heredoc/nowdoc)

2.4 Arrays (indexed, associative) and common operations
2.5 Control flow: `if/elseif/else`, `switch`, match expressions
2.6 Loops: `for`, `foreach`, `while`, `break/continue`
2.7 Practical mini-exercises (formatting, parsing, simple transforms)
2.8 WordPress parallel: arrays everywhere (hooks, query args, theme data)

---

3) Functions, Scope, and Working with Data □□

3.1 Defining functions, parameters, defaults, and return values
3.2 Scope, globals, static variables, and “why things disappear”
3.3 Passing by value vs by reference
3.4 Useful built-ins (strings, arrays, dates) without memorizing everything
3.5 Namespaces (why they matter even in small projects)
3.6 WordPress parallel: pluggable functions, naming, and avoiding collisions

---

4) HTTP, Forms, and Input Handling □□

4.1 HTTP essentials: methods, headers, status codes
4.2 Superglobals: `$_GET`, `$_POST`, `$_SERVER`, `$_COOKIE`, `$_FILES`
4.3 Forms end-to-end (build, submit, validate, respond)
4.4 Redirects and the PRG pattern (Post/Redirect/Get)
4.5 File uploads safely
4.6 WordPress parallel: admin forms, nonces, request handling conventions

---

5) Defensive PHP: Validation, Sanitization, and Security Basics □□

5.1 Threat model basics (what can go wrong)
5.2 Validation vs sanitization vs escaping (clear separation)

5.3 Output escaping for HTML (and why context matters)
5.4 Password hashing and authentication fundamentals
5.5 Sessions and cookies (secure defaults)
5.6 Common web vulnerabilities: XSS, CSRF, SQLi, SSRF (practical overview)
5.7 WordPress parallel: `sanitize_*`, `esc_*`, nonces, roles/capabilities

---

6) Working with Files, JSON, and Common Formats □□

6.1 Reading/writing files safely
6.2 Paths, directories, and portability
6.3 JSON encode/decode and pitfalls
6.4 CSV basics (import/export)
6.5 Date/time handling (timezone sanity)
6.6 WordPress parallel: media/filesystem API concepts, JSON in REST responses

---

7) Error Handling, Debugging, and Testing Mindset □□

7.1 Errors vs exceptions, and how PHP reports problems
7.2 Try/catch patterns and custom exceptions
7.3 Logging strategies (what to log, what not to log)
7.4 Debugging workflows (Xdebug basics, var_dump discipline)
7.5 Intro to automated testing concepts (unit vs integration)
7.6 WordPress parallel: WP_DEBUG, debug.log, common debugging workflows

---

8) Object-Oriented PHP: From Practical to Pro □□

8.1 Classes/objects, properties, methods
8.2 Constructors, visibility, and encapsulation

8.3 Inheritance vs composition (when to use which)
8.4 Interfaces and abstract classes
8.5 Traits (pros/cons)
8.6 Static usage (when it’s fine, when it’s a trap)
8.7 WordPress parallel: OOP patterns in plugins, service classes, admin pages

---

9) Modern PHP Practices: Autoloading, Composer, and Standards ⚙

9.1 Composer fundamentals (packages, versions, lockfiles)
9.2 Autoloading and PSR-4
9.3 Common PSRs (PSR-12, PSR-3, PSR-4) and why they help
9.4 Dependency injection (practical introduction)
9.5 Configuration patterns (env, config files)
9.6 WordPress parallel: Composer in plugins/themes, when to bundle dependencies

---

10) Databases with PHP: SQL, PDO, and Data Modeling □□

10.1 Relational database basics and schema thinking
10.2 SQL essentials (SELECT/INSERT/UPDATE/DELETE, joins)
10.3 PDO: prepared statements and safe queries
10.4 Transactions and consistency
10.5 Basic modeling: one-to-many, many-to-many
10.6 Performance basics: indexes and query shape
10.7 WordPress parallel: `$wpdb`, custom tables, and when *not* to create them

---

11) Building a Web App: Routing, Controllers, and Views □□

11.1 Simple routing (front controller pattern)
11.2 Organizing “MVC-ish” code without overengineering
11.3 Templating approaches (plain PHP templates done right)
11.4 Handling errors and 404s cleanly
11.5 Pagination, filters, and query parameters
11.6 WordPress parallel: templates, the loop conceptually, template hierarchy mindset

---

12) APIs: Consuming and Serving HTTP Services □□

12.1 Making HTTP requests (cURL / modern clients)
12.2 REST basics and JSON API conventions
12.3 Authentication patterns (API keys, OAuth overview)
12.4 Building a simple JSON API in PHP
12.5 Versioning and backwards compatibility
12.6 WordPress parallel: WP REST API usage and custom endpoints

---

13) Performance, Caching, and Scalability ⚡

13.1 Profiling mindset: find bottlenecks before “optimizing”
13.2 Opcode cache (OPcache) basics
13.3 Caching layers: in-memory, file-based, HTTP caching
13.4 Efficient I/O, streaming, and avoiding large memory spikes
13.5 Async-ish patterns (queues, cron) at a practical level
13.6 WordPress parallel: transients, object cache, page cache, cron behavior

---

14) WordPress Development Track (Sprinkled Knowledge → Structured Practice) □□

14.1 WordPress architecture overview (request lifecycle, hooks)
14.2 Theme fundamentals: templates, enqueueing, child themes
14.3 Plugin fundamentals: headers, structure, activation hooks
14.4 Hooks deep dive: actions vs filters, priorities, args
14.5 Security in WP: capabilities, nonces, escaping
14.6 Custom Post Types & Taxonomies (mental model + practice)
14.7 Meta fields and options (when to use what)
14.8 WP REST API: extend and consume
14.9 Data: `WP_Query`, `$wpdb`, and performance considerations
14.10 Practical patterns: settings pages, admin UI, shortcodes, blocks overview

---

15) Professional Practices: Architecture, Maintenance, and Delivery □□

15.1 Designing for change: boundaries, modules, and refactors
15.2 Documentation that stays useful (READMEs, docblocks)
15.3 Version control workflows (Git) for solo + teams
15.4 CI basics (linting, tests, static analysis)
15.5 Deployment overview (shared hosting, VPS, containers)
15.6 Observability basics: logs, metrics, error reporting
15.7 WordPress parallel: release discipline for plugins/themes and compatibility

---

16) Capstone Projects (Pick One or Do All) □□□

16.1 Pure PHP mini-app: form-heavy CRUD app with auth + admin
16.2 API project: PHP JSON API + a tiny client
16.3 WordPress plugin: production-style plugin (settings, CPT, REST endpoint)
16.4 WordPress theme: custom theme with performance + security best practices
16.5 Hardening & polish: testing, docs, deployment checklist

How to proceed

Send something like `c2.3` and I'll generate the full lesson for *Chapter 2, Subchapter 3* (including explanations, examples, a few exercises, and small WordPress parallels where relevant).

Anki Cards: PHP Core Terminology

Core Terminology □□

1. In web PHP, a *single execution* of code to respond to an HTTP request is often called a {{c1::request}} (or script run).
2. PHP runs via an {{c1::interpreter}} (the PHP engine), not by producing a native binary like C/C++.
3. PHP is {{c1::server-side}}: the browser receives the {{c2::output}} (HTML/JSON), not the {{c3::PHP source}}.
4. Built-in arrays that are always available (e.g., `$_GET`, `$_POST`, `$_SERVER`) are called {{c1::superglobals}}.
5. PHP's automatic conversion between types (e.g., string ↔ int) is {{c1::type juggling}}.
6. `declare(strict_types=1);` enables {{c1::strict typing}} behavior for {{c2::scalar type hints}}.
7. **Validation** asks: {{c1::"Is this allowed?"}}; invalid input is typically {{c2::rejected}}.
8. **Sanitization** asks: {{c1::"Can we make this safe/clean?"}}; input is {{c2::transformed}}.
9. **Escaping** asks: {{c1::"How do I safely output this in a context?"}} (HTML/attr/JS/URL), and is done at {{c2::output time}}.
10. **XSS** happens when unescaped output lets an attacker run {{c1::JavaScript}} in the victim's browser.
11. **CSRF** is a {{c1::forged request}} problem; typical defense is a per-request {{c2::token/nonce}}.
12. **SQL injection** is prevented by using {{c1::prepared statements}} instead of unsafe string concatenation.
13. `require` is {{c1::fatal}} if the file is missing; `include` emits a {{c2::warning}} and continues.
14. Composer-style class loading is {{c1::autoloading}}, commonly via {{c2::PSR-4}}.
15. A {{c1::namespace}} prevents naming collisions by qualifying names like `MyApp\Foo`.
16. {{c1::Dependency injection}} means {{c2::passing dependencies in}} rather than creating them inside the class/function.
17. {{c1::PDO}} is PHP's standard DB interface and supports {{c2::prepared statements}}.
18. In WordPress, a hook is a {{c1::callback point}}: an {{c2::action}} "does something," a {{c3::filter}} "modifies a value."

Daily PHP Constructs (“Commands”) □□

19. `echo` outputs {{c1::strings}} (and can output multiple args separated by commas).
20. `print` is like echo but returns {{c1::1}} (so it’s usable in expressions).
21. `var_dump($x)` shows both {{c1::type}} and {{c2::value}} (great for debugging).
22. `print_r($x, true)` returns the output as a {{c1::string}} when the second argument is {{c2::true}}.
23. `die()` / `exit()` {{c1::stops execution}} immediately (often after a redirect).
24. `include_once` / `require_once` ensure a file is included at most {{c1::once}} per request.

Control Flow (If / Switch / Match / Loops) □□

25. `if (...) {}` runs only when the condition is {{c1::true}}.
26. `switch` typically needs `{{c1::break}}` to avoid fall-through into the next case.
27. `match (...) { ... }` is an {{c1::expression}} that {{c2::returns a value}} (unlike `switch`).
28. `match` uses {{c1::strict comparisons}} (no type juggling like loose `switch` cases can do).
29. `foreach ($arr as $value)` iterates over the array’s {{c1::values}}.
30. `foreach ($arr as $k => $v)` gives both the {{c1::key}} and the {{c2::value}}.
31. `break` exits the {{c1::current loop/switch}}; `continue` skips to the {{c2::next iteration}}.
32. A `do { ... } while (...);` loop runs the body at least {{c1::once}}.

Functions & Organization □□

33. A function can define a default parameter like `function f($x = 123)`, meaning it’s {{c1::optional}} when calling.
34. `return` exits a function and optionally provides a {{c1::value}}.

35. `global $x;` accesses a variable from the {{c1::global scope}} (best used {{c2::sparingly}}).
36. `static $x = 0;` inside a function persists {{c1::between calls}} during the same request.
37. A function with a return type `: int` promises it will return an {{c1::integer}} (or throw).
38. In modern PHP, use `{{c1::strict_types}}` when you want stricter scalar parameter/return behavior.

Error Handling & Exceptions □□

---

39. `try { ... } catch (Throwable $e) { ... }` catches both {{c1::Exception}} and many {{c2::Error}} types.
40. `finally { ... }` runs whether an exception was {{c1::thrown}} or not (good for cleanup).
41. `throw new Exception('msg');` {{c1::raises}} an exception to be handled by a caller.
42. If an exception is not caught, it typically causes a {{c1::fatal error}} and aborts the request.

OOP: Classes, Visibility, Inheritance □□

---

43. `new ClassName()` creates an {{c1::object instance}}.
44. `public` members are accessible {{c1::everywhere}}; `protected` inside {{c2::class + subclasses}}; `private` only inside the {{c3::declaring class}}.
45. `extends` means {{c1::inheritance}}; `implements` means fulfilling an {{c2::interface contract}}.
46. `$this->` accesses the {{c1::current object}} instance members.
47. `self::` refers to the {{c1::current class}}; `parent::` refers to the {{c2::parent class}}.
48. A `trait` is a mechanism for {{c1::code reuse}} across classes (without inheritance).
49. An `abstract` class cannot be {{c1::instantiated}} directly.
50. An `interface` defines {{c1::method signatures}} that implementing classes must provide.

Variables, Types, Operators □□

---

51. PHP variables start with a {{c1::$}} sign.
52. `define('APP_ENV', 'dev')` defines a {{c1::constant}} at runtime; `const` defines a constant at {{c2::compile time}} (and can be used in classes).
53. Scalar types: {{c1::int}}, {{c2::float}}, {{c3::string}}, {{c4::bool}}.
54. `null` represents an {{c1::absence}} of value.
55. `.` is {{c1::string concatenation}} in PHP.
56. `.=` performs concatenation and {{c1::assignment}} in one step.
57. `==` is {{c1::loose comparison}} (type juggling); `===` is {{c2::strict}} (type + value).
58. The “spaceship” operator `<=>` returns {{c1::-1}}, {{c2::0}}, or {{c3::1}} for ordering comparisons.
59. Null coalescing `??` uses the right-hand side only if the left is {{c1::null or undefined}}.
60. Nullsafe `?->` stops and returns {{c1::null}} if the left side is {{c2::null}}.
61. `&&` / `||` are {{c1::short-circuit}} boolean operators.
62. The ternary `cond ? a : b` picks {{c1::a}} when cond is true, else {{c2::b}}.

Strings □□

---

63. In single quotes `'...'`, variables are generally {{c1::not interpolated}}.
64. In double quotes `"..."`, variables like `$name` are {{c1::interpolated}}.
65. `strlen($s)` returns the string length in {{c1::bytes}} (multibyte text may need {{c2::mb_strlen}}).
66. `strpos($haystack, $needle)` returns the position or {{c1::false}} (so use `=== false` checks).
67. `trim($s)` removes whitespace from the {{c1::start and end}} of a string.
68. `explode(',', $s)` converts a string into an {{c1::array}}.
69. `implode(',', $arr)` converts an array into a {{c1::string}}.
70. `sprintf("Hi %s", $name)` returns a formatted {{c1::string}} without echoing it.

Arrays (Workhorse) □□

---

71. `[]` creates an {{c1::array}} literal (indexed or associative).
72. Indexed array example: `$a = [10, 20, 30];` uses numeric {{c1::indexes}}.
73. Associative array example: `['name' => 'Ada']` uses string {{c1::keys}}.
74. `$a[] = 99;` appends to the {{c1::end}} of an indexed array.
75. `count($arr)` returns the number of {{c1::elements}}.
76. `in_array($needle, $haystack, true)` uses strict checking when the third argument is {{c1::true}}.

77. `array_key_exists('k', $arr)` checks for the presence of a {{c1::key}} even if its value is {{c2::null}}.
78. `array_map(fn($x) => ..., $arr)` transforms each element and returns a {{c1::new array}}.
79. `array_filter($arr, $fn)` keeps elements where the callback returns {{c1::true}}.
80. `array_reduce($arr, $fn, $initial)` folds an array into a single {{c1::value}}.
81. `sort($arr)` sorts values and {{c1::reindexes}} numeric keys.
82. `asort($arr)` sorts by value while {{c1::preserving keys}}.
83. `ksort($arr)` sorts by {{c1::key}}.

HTTP & Superglobals □□

84. Query string parameters are read from {{c1::$_GET}}.
85. Form body parameters are commonly read from {{c1::$_POST}}.
86. Request metadata (method, headers info, URI) is found in {{c1::$_SERVER}}.
87. Uploaded file info is in {{c1::$_FILES}} (name/type/tmp_name/error/size).
88. Session data uses {{c1::$_SESSION}} after calling {{c2::session_start()}}.
89. A safe read pattern: `$q = $_GET['q'] ?? '';` avoids an {{c1::undefined index}} notice.
90. `header('Location: /path');` triggers an HTTP {{c1::redirect}}.
91. After sending a Location header, you should call {{c1::exit}} to stop further output.
92. `http_response_code(404);` sets the HTTP status code to {{c1::404}}.

Security Defaults □□

93. For HTML output, `htmlspecialchars($s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8')` prevents {{c1::XSS}} in HTML/text contexts.
94. `ENT_QUOTES` escapes both {{c1::single}} and {{c2::double}} quotes.
95. Passwords should be stored using {{c1::password_hash}} (not md5/sha1).
96. Verify a password with {{c1::password_verify($pw, $hash)}}.
97. SQL safety best practice: use {{c1::prepared statements}} with bound parameters (not string concatenation).
98. CSRF defense: include a per-request {{c1::token}} and verify it on submission.
99. Never trust `$_GET/$_POST` types: always {{c1::validate}} and/or {{c2::cast}} (e.g., `(int)`).
100. Output escaping is {{c1::context-dependent}} (HTML vs attribute vs URL vs JS).

Composer & Autoloading ⚙

01. `composer.json` declares dependencies and {{c1::autoload rules}}.
02. `composer.lock` pins the {{c1::exact versions}} installed.
03. Composer's autoloader entry file is {{c1::vendor/autoload.php}}.
04. In code, `require __DIR__ . '/vendor/autoload.php';` enables {{c1::autoloading}}.
05. PSR-4 maps {{c1::namespaces}} to {{c2::directory paths}}.

PDO (Database) □□

06. In PDO, setting `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION` makes DB errors throw {{c1::exceptions}}.
07. A prepared statement is created with `$pdo->{{c1::prepare}}(...)`.
08. Parameters are provided via `$stmt->{{c1::execute}}(['email' => $email])` (named placeholders).
09. Fetching one row as an associative array can be done with `$stmt->fetch(PDO::{{c1::FETCH_ASSOC}})`.
10. After an INSERT, `$pdo->{{c1::lastInsertId}}()` gets the last generated ID (driver-dependent).

WordPress Parallels □□

11. WordPress actions are registered with {{c1::add_action}}; filters with {{c2::add_filter}}.
12. A filter callback must {{c1::return}} the modified value; an action callback typically {{c2::does not}}.
13. WordPress escaping helpers: `esc_html`, `esc_attr`, {{c1::esc_url}} for URLs.
14. WordPress sanitizers include `sanitize_text_field` and {{c1::sanitize_email}}.
15. WordPress CSRF protection uses {{c1::nonces}} (e.g., `wp_nonce_field`, `check_admin_referer`).
16. `$wpdb->prepare(...)` is the WordPress pattern for {{c1::safe SQL}}.

"I Forget This" Reminders □□

17. Prefer the full opening tag {{c1::<?php}} (avoid short tags).
18. In pure PHP files, it’s common to omit the closing tag `?>` to avoid accidental {{c1::whitespace output}}.
19. In PHP, the string `'0'` is {{c1::falsy}} (so strict comparisons can matter).
20. HTTP headers must be sent before any {{c1::output}} (even whitespace), otherwise you get “headers already sent.”

Extra High-Value Additions (Fits the Topic)

---

21. `error_reporting(E_ALL);` and `ini_set('display_errors', '1');` are useful in {{c1::development}} (but not in production).
22. Prefer `filter_input(INPUT_GET, 'q', FILTER_SANITIZE_SPECIAL_CHARS)` for simple input handling, but still {{c1::validate}} properly.
23. `json_encode($data, JSON_UNESCAPED_UNICODE)` produces {{c1::JSON}} output; set header `Content-Type: application/json`.
24. To read JSON request bodies: `$raw = file_get_contents('php://input'); $data = json_decode($raw, true);` — `true` yields an {{c1::associative array}}.
25. `isset($x)` is false if `$x` is {{c1::not set}} or {{c2::null}}.
26. `empty($x)` treats values like `0`, `'0'`, `[]`, and `null` as {{c1::empty}} (be careful).
27. Use `===` when checking `strpos(...)` results because position {{c1::0}} is a valid match but is {{c2::falsy}}.
28. The directory constant `__DIR__` gives the current file’s {{c1::directory path}} (safer than relative paths).
29. `require` vs `require_once`: `require_once` adds overhead; prefer {{c1::autoloading}} for classes instead of many `*_once`.
30. Use `password_hash` with `PASSWORD_DEFAULT` so the algorithm can {{c1::upgrade over time}}.
31. In prepared statements, never interpolate variables directly into SQL; bind them as {{c1::parameters}}.
32. In PHP 8+, `match` has no fall-through and will throw `UnhandledMatchError` if no case matches and there’s no {{c1::default}}.
33. For arrays, `===` compares both order and types; for “same key/value pairs regardless of order,” use {{c1::ksort}} first.
34. In OOP, prefer constructor injection: `__construct(private Logger $logger)` is promoted property syntax (PHP {{c1::8.0+}}).
35. For safe redirects, validate destination URLs to prevent {{c1::open redirect}} vulnerabilities.

If you tell me whether you’re using **PHP 7.4, 8.0–8.4**, and whether you’re focusing on **WordPress plugin/theme dev** or **general backend**, I can generate a second deck with *scenario-based* clozes (debugging, forms, auth, PDO pitfalls) □□□

Vibe Coding

LinkShorter v2 – Ausführliche Anleitung

1. Überblick & Architektur

LinkShorter ist ein selbstgehosteter URL-Shortener, der vollständig in **PHP** geschrieben ist und eine **SQLite**-Datenbank verwendet – es wird also kein MySQL/MariaDB-Server benötigt. Die gesamte Anwendung besteht aus einer einzigen Einstiegsdatei (index.php), die als **Front-Controller** fungiert: Alle HTTP-Requests werden über die .htaccess per `mod_rewrite` an diese Datei weitergeleitet.

Technische Kernkomponenten

Komponente	Datei	Verantwortung
Routing & Controller	index.php	Einziger Entry-Point, wertet `$_GET['route']` aus
Datenbank	includes/db.php	SQLite-Verbindung via PDO, Schema-Migration
Authentifizierung	includes/auth.php	Session-basiertes Login
Slug-Generierung	includes/slug.php	Zufällige Kurzlinks via SHA-256
Link-Logik	includes/links.php	CRUD, Click-Recording, Batch-Import
OpenGraph	includes/opengraph.php	Metadaten-Extraktion von Ziel-URLs
QR-Code	includes/qrcode.php	Eigene QR-Code-Implementierung (kein externer Service!)

Datenfluss bei einem Kurzlink-Aufruf

```
Browser → Apache (.htaccess Rewrite)
       → index.php?route=mein-slug
```

```
→ getLinkBySlug("mein-slug")
→ Prüfung: aktiv? abgelaufen? Passwort? Crawler?
→ recordClick() → HTTP 302 Redirect → Ziel-URL
```

---

2. Installation & Konfiguration

Voraussetzungen

- PHP ≥ 7.4 mit aktivierten Extensions: `pdo_sqlite`, `gd` (für PNG-QR-Codes), `mbstring`
- Apache mit `mod_rewrite` aktiviert
- Schreibrechte auf das Verzeichnis `data/`

Schritt 1: Dateien hochladen

Laden Sie alle Dateien auf Ihren Webserver hoch. Die Verzeichnisstruktur sollte so aussehen:

```
/
├── index.php
├── config.php
├── .htaccess
├── assets/
│   ├── style.css
│   └── app.js
├── includes/
│   ├── db.php
│   ├── auth.php
│   ├── slug.php
│   ├── links.php
│   ├── opengraph.php
│   └── qrcode.php
├── templates/
│   ├── dashboard.php
│   ├── edit.php
│   ├── login.php
│   ├── stats.php
│   ├── settings.php
```

```
│   ├── password.php
│   ├── og_proxy.php
│   ├── unavailable.php
│   └── 404.php
└── data/           ← wird automatisch erstellt
    ├── linkshorter.db
    ├── qr_cache/
    └── qr_icon.svg
```

Schritt 2: Konfiguration anpassen

Öffnen Sie config.php und passen Sie die Werte an:

```
<?php
define('ADMIN_USERNAME', 'admin');          // Ihr gewünschter Benutzername
define('ADMIN_PASSWORD', 'IhrSicheresPasswort');  // UNBEDINGT ÄNDERN!
define('BASE_URL', 'https://kurz.example.com/');  // Ihre Domain mit abschließendem /
define('DB_PATH', __DIR__ . '/data/linkshorter.db');
define('SITE_TITLE', 'LinkShorter');
define('QR_ICON_PATH', __DIR__ . '/data/qr_icon.svg');
```

Technischer Hintergrund: Die Konstanten werden mit `define()` festgelegt und sind damit **global** in allen inkludierten Dateien verfügbar. ADMIN_PASSWORD wird im Klartext gespeichert – es wird **nicht** gehasht, da es bei jedem Login direkt verglichen wird (in attemptLogin). Dies ist ein bewusster Trade-off für Einfachheit bei einer Single-User-Anwendung.

Schritt 3: Erster Aufruf

Beim ersten Aufruf von `https://kurz.example.com/` erstellt getDB automatisch:

- Das `data/`-Verzeichnis
- Die SQLite-Datenbank mit den Tabellen `links`, `clicks` und `settings`

Die Funktion nutzt **WAL-Modus** (`PRAGMA journal_mode=WAL`), was parallele Lese- und Schreibzugriffe ermöglicht und die Performance bei gleichzeitigen Zugriffen deutlich verbessert.

---

3. Das Admin-Dashboard

Login

Rufen Sie `https://kurz.example.com/` auf. Sie werden zum Login weitergeleitet, da die Route `?page=login` aktiv wird. Die Authentifizierung funktioniert **session-basiert**:

1. attemptLogin vergleicht die Eingaben mit den Konstanten aus config.php
2. Bei Erfolg wird `$_SESSION['logged_in'] = true` gesetzt
3. Alle geschützten Seiten prüfen via requireLogin, ob die Session gültig ist

Dashboard-Oberfläche

Nach dem Login sehen Sie das Dashboard (dashboard.php) mit:

- **Link-Erstellungsformular** (oben)
- **Suchleiste** mit Live-Filterung
- **Link-Tabelle** mit Sortierung nach Slug, URL, Klicks oder Erstellungsdatum

Technischer Hintergrund zur Sortierung: Die Funktion getAllLinks baut die SQL-Query dynamisch zusammen. Erlaubte Spalten sind in einem Whitelist-Array definiert, um **SQL-Injection** zu verhindern:

```
$allowed = ['slug', 'url', 'clicks', 'created_at', 'active'];
if (!in_array($sort, $allowed)) $sort = 'created_at';
```

Die Sortierrichtung wird ebenfalls validiert (`ASC` oder `DESC`), bevor sie in die Query eingebaut wird.

---

4. Links erstellen & verwalten

Einzelnen Link erstellen

1. Geben Sie die **Ziel-URL** ein (Pflichtfeld)
2. Optional: **Custom Slug** – wird automatisch bereinigt (nur `a-zA-Z0-9_-` erlaubt)
3. Optional: **Passwort** – wird mit `password_hash()` als bcrypt-Hash gespeichert
4. Optional: **Ablaufdatum** – als `datetime-local`

5. Optional: **Max Clicks** – nach Erreichen wird der Link deaktiviert

Technischer Hintergrund zur Slug-Generierung: Wenn kein Custom Slug angegeben wird, generiert generateSlug einen 6-Zeichen-Code. Der Algorithmus ist bewusst kryptographisch robust:

```
Eingabe = microtime() + random_bytes(8) + Versuchszähler
          ↓
SHA-256 Hash
          ↓
6 Zeichen aus dem Zeichensatz [a-zA-Z0-9] extrahiert
          ↓
Kollisionsprüfung gegen Datenbank (max 100 Versuche)
```

Die Kombination aus `microtime()` (Zeitstempel mit Mikrosekunden) und `random_bytes()` (kryptographisch sichere Zufallsbytes) macht Kollisionen extrem unwahrscheinlich. Bei 62 möglichen Zeichen pro Position ergibt ein 6-Zeichen-Slug **$62^6 \approx 56{,}8$ Milliarden** mögliche Kombinationen.

Batch-Import

Klicken Sie auf **„Batch Import"** im Dashboard. Es öffnet sich ein Modal-Dialog, in dem Sie eine URL pro Zeile eingeben können. batchCreateLinks verarbeitet jede Zeile einzeln:

1. Trennung nach Zeilenumbrüchen und Trimming
2. URL-Validierung via `filter_var($url, FILTER_VALIDATE_URL)`
3. Automatische Slug-Generierung für jede URL
4. Ergebnisanzeige mit Erfolgs-/Fehlerstatus

Link bearbeiten

Auf der Edit-Seite (edit.php) können Sie alle Eigenschaften eines Links ändern. Besonders interessant ist die **Passwort-Verwaltung** mit drei Optionen:

Auswahl	`password_action`	Verhalten
Behalten / Kein Passwort	`keep`	`password` wird aus `$data` entfernt
Neues setzen	`change`	Neues Passwort wird bcrypt-gehasht
Entfernen	`remove`	`password` wird auf `''` gesetzt → in updateLink wird es zu `null`

Technischer Hintergrund: In updateLink wird die OpenGraph-Daten-Aktualisierung **nur** ausgelöst, wenn sich die URL geändert hat. Das verhindert unnötige HTTP-Requests an die Ziel-URL:

```
$og = ($url !== $link['url']) ? fetchOpenGraph($url) : [
    'og_title' => $link['og_title'],
    'og_description' => $link['og_description'],
    'og_image' => $link['og_image'],
];
```

Link löschen

Das Löschen erfolgt via POST-Request mit einer JavaScript-Bestätigung. Dank `ON DELETE CASCADE` in der Datenbank-Schema-Definition werden zugehörige Click-Datensätze automatisch mitgelöscht.

---

5. Passwortgeschützte Links

Wenn ein Link mit Passwort versehen ist, zeigt der Server die Seite password.php an. Der Ablauf:

```
Besucher → /mein-slug
         → getLinkBySlug() findet Link mit password ≠ null
         → GET-Request: Zeige Passwort-Formular
         → POST-Request mit link_password:
            → password_verify(eingabe, hash) → true:  recordClick + Redirect
                                             → false: Fehlermeldung
```

Wichtig: Das Passwort wird **immer** als bcrypt-Hash gespeichert (`password_hash($password, PASSWORD_DEFAULT)`). Beim Vergleich wird `password_verify()` verwendet, was automatisch den Salt aus dem Hash extrahiert. Das bedeutet: Selbst wenn die Datenbank kompromittiert wird, sind die Link-Passwörter nicht im Klartext einsehbar.

---

6. QR-Codes

LinkShorter enthält eine **vollständig eigene QR-Code-Implementierung** – es werden keine externen APIs oder Libraries benötigt.

Technischer Tiefgang: QR-Code-Generierung

Die Funktion qrEncode implementiert den kompletten QR-Code-Standard:

1. **Versionswahl:** Basierend auf der Datenlänge wird die minimale QR-Version (1–40) bestimmt. Version 1 hat 21×21 Module, Version 40 hat 177×177 Module. Die Kapazitätstabelle `$capacityL` enthält die maximale Byte-Kapazität für **Error Correction Level L** (Low, ~7% Fehlerkorrektur).
2. **Matrix-Aufbau:**
   - placeFinderPattern: Drei 7×7-Erkennungsmuster in den Ecken (oben-links, oben-rechts, unten-links)
   - Timing-Patterns: Abwechselnde Module in Zeile 6 und Spalte 6
   - placeAlignmentPattern: Ab Version 2 werden Ausrichtungsmuster platziert (Positionen aus getAlignmentPatternPositions)
3. **Datenkodierung** (encodeData):
   - Mode Indicator: `0100` (Byte-Modus)
   - Zeichenzähler: 8 Bit (Version 1–9) oder 16 Bit (ab Version 10)
   - Daten als 8-Bit-Bytes
   - Padding mit `0xEC 0x11` (abwechselnd)
   - Reed-Solomon-Fehlerkorrektur via generateECCodewords
4. **Galois-Feld-Arithmetik:** Für die Fehlerkorrektur werden Berechnungen im **GF(2^8)** durchgeführt. Die Funktionen gfMultiply, gfExp und gfLog implementieren die Multiplikation über das irreduzible Polynom `0x11D` ($x^8 + x^4 + x^3 + x^2 + 1$).
5. **Data-Interleaving:** Bei mehreren Blöcken werden die Daten- und EC-Codewords interleaved (verschachtelt), um Burst-Fehler besser zu korrigieren.
6. **Maskierung:** Alle 8 Maskmuster werden getestet. calculatePenalty berechnet die Strafpunkte nach vier Regeln:
   - Fünf oder mehr gleiche Module in einer Reihe
   - 2×2-Blöcke gleicher Module
   - Spezielle Muster (1:1:3:1:1)
   - Verhältnis dunkler zu heller Module nahe 50%

   Das Mask-Pattern mit der **niedrigsten Penalty** wird gewählt.

Ausgabeformate

- **PNG** (getQRCodePNG): Erzeugt via GD-Library (`imagecreatetruecolor`), mit optionalem Center-Icon via ImageMagick (`convert`-Befehl)
- **SVG** (getQRCodeSVG): Reine XML-Generierung, Icon wird als eingebettetes SVG eingefügt

Caching

Beide Formate werden im Verzeichnis `data/qr_cache/` gecacht (24 Stunden TTL). Der Cache-Key ist ein MD5-Hash aus `Daten + Größe + Format`.

Custom Icon

Unter **Settings** können Sie ein SVG-Icon hochladen (admin/action mit action=upload_icon), das im Zentrum aller QR-Codes angezeigt wird. Beim Upload wird der QR-Cache geleert, damit die neuen QR-Codes das Icon enthalten.

---

7. OpenGraph-Proxying

Wenn ein Kurzlink in sozialen Netzwerken geteilt wird, erkennt LinkShorter den Crawler anhand des User-Agents:

```
$isCrawler = preg_match(

'/facebookexternalhit|Twitterbot|LinkedInBot|WhatsApp|Slackbot|TelegramBot|Discordbot|bot|crawler|spider/i',
  $ua
);
```

Statt den Crawler weiterzuleiten, wird og_proxy.php ausgeliefert – eine minimale HTML-Seite mit den **OpenGraph-Meta-Tags** der Ziel-URL. Das bewirkt, dass in der Vorschau (z.B. auf Facebook, Twitter, Slack) das **Bild, der Titel und die Beschreibung der Original-Seite** angezeigt werden, obwohl der geteilte Link eine kurze URL ist.

Technischer Hintergrund: fetchOpenGraph extrahiert beim Erstellen eines Links die Meta-Daten:

1. HTTP-Request an die Ziel-URL mit 10-Sekunden-Timeout
2. HTML-Parsing via `DOMDocument`
3. Extraktion von `og:title`, `og:description`, `og:image`
4. Fallback auf `<title>`-Tag, wenn kein `og:title` vorhanden

Die SSL-Verifikation ist bewusst deaktiviert (`verify_peer => false`), um Probleme mit selbstsignierten Zertifikaten zu vermeiden. In Produktionsumgebungen sollte das ggf. angepasst werden.

Wichtig: Das OpenGraph-Proxying funktioniert auch für passwortgeschützte Links – Crawler erhalten die Vorschau, ohne ein Passwort eingeben zu müssen. Normale Benutzer werden weiterhin nach dem Passwort gefragt.

---

8. Statistiken & Click-Tracking

Was wird erfasst?

Bei jedem erfolgreichen Redirect (auch nach Passworteingabe) ruft recordClick zwei Datenbankoperationen aus:

1. **Inkrement des Klickzählers** in der `links`-Tabelle
2. **Detaillierter Click-Eintrag** in der `clicks`-Tabelle:
   - `ip`: IP-Adresse des Besuchers (`$_SERVER['REMOTE_ADDR']`)
   - `user_agent`: Browser-Kennung
   - `referer`: Woher der Besucher kam
   - `clicked_at`: Zeitstempel (automatisch via SQLite `datetime('now')`)

Stats-Seite

Die Stats-Seite (stats.php) zeigt:

- **Gesamtklicks** als große Zahl
- **Max Clicks** (Limit oder ∞)
- **Status** (aktiv/inaktiv)
- **Letzte 100 Klicks** als Tabelle mit IP, Referer, User-Agent und Zeitstempel

Die Abfrage in getClickStats ist auf 100 Einträge limitiert (`LIMIT 100`), um bei viel-geklickten Links die Performance zu gewährleisten.

Link-Deaktivierung

Ein Link wird automatisch als „nicht verfügbar" angezeigt, wenn:

- `active = 0` (manuell deaktiviert)
- `max_clicks` erreicht wurde (`clicks >= max_clicks`)
- `expires_at` in der Vergangenheit liegt

In allen drei Fällen wird unavailable.php angezeigt.

---

9. Die REST-API

LinkShorter bietet zwei API-Endpunkte, die mit **HTTP Basic Authentication** geschützt sind.

`POST /api/shorten`

Erstellt einen neuen Kurzlink.

Request:

```
POST /api/shorten HTTP/1.1
Authorization: Basic base64(username:password)
Content-Type: application/json

{
  "url": "https://example.com/sehr-lange-url",
  "slug": "custom",           // optional
  "password": "geheim",       // optional
  "expires_at": "2025-12-31T23:59", // optional
  "max_clicks": 100           // optional
}
```

Response (Erfolg):

```
{
  "short_url": "https://kurz.example.com/custom",
  "slug": "custom"
}
```

Response (Fehler):

```
{
  "error": "Slug already exists"
}
```

`GET /api/check`

Prüft die Erreichbarkeit und Authentifizierung einer Instanz. Wird von der Chrome-Extension beim Hinzufügen einer neuen Instanz verwendet.

Response:

```
{
  "status": "ok",
  "title": "LinkShorter"
}
```

Technischer Hintergrund

Die Authentifizierung wird im index.php inline geprüft:

```
$authHeader = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
if (preg_match('/^Basic\s+(.+)$/i', $authHeader, $m)) {
    $decoded = base64_decode($m[1]);
    list($user, $pass) = explode(':', $decoded, 2);
    // Vergleich mit ADMIN_USERNAME und ADMIN_PASSWORD
}
```

Hinweis: Apache kann den `Authorization`-Header manchmal nicht an PHP weiterleiten. In diesem Fall muss in der `.htaccess` folgende Zeile ergänzt werden:

```
SetEnvIf Authorization "(.*)" HTTP_AUTHORIZATION=$1
```

---

10. Die Chrome-Extension

Überblick

Die Chrome-Extension ermöglicht es, die **aktuelle Tab-URL** mit einem Klick zu kürzen, ohne das Dashboard öffnen zu müssen. Sie unterstützt **mehrere LinkShorter-Instanzen**, was nützlich ist, wenn man verschiedene Domains für verschiedene Zwecke nutzt.

Installation

1. Navigieren Sie in Chrome zu `chrome://extensions/`
2. Aktivieren Sie den **Entwicklermodus** (oben rechts)
3. Klicken Sie **„Entpackte Erweiterung laden"**
4. Wählen Sie den Ordner `chrome-extension/`

Dateien der Extension

Datei	Funktion
manifest.json	Extension-Konfiguration (Manifest V3)
popup.html	UI-Struktur
popup.css	Styling
popup.js	Gesamte Logik

Instanz hinzufügen

1. Klicken Sie auf das **⚙-Symbol** (Settings)
2. Geben Sie ein:
   - **Name**: Anzeigename (z.B. „Mein Server")
   - **URL**: Die BASE_URL Ihrer LinkShorter-Installation
   - **Username/Password**: Wie in config.php konfiguriert
3. Klicken Sie **„Add & Verify"**

Technischer Hintergrund: Beim Hinzufügen wird zunächst ein `GET /api/check` ausgeführt, um die Verbindung und Authentifizierung zu testen. Erst wenn `{"status": "ok"}` zurückkommt, wird die Instanz gespeichert. Die Instanzen werden in `chrome.storage.sync` gespeichert, was bedeutet, dass sie über mehrere Chrome-Installationen hinweg **synchronisiert** werden (wenn der Nutzer in Chrome eingeloggt ist).

URL kürzen

1. Navigieren Sie zur gewünschten Webseite
2. Klicken Sie auf das LinkShorter-Icon in der Toolbar
3. Die **aktuelle Tab-URL** wird automatisch eingetragen (via `chrome.tabs.query`)
4. Optional: Wählen Sie eine andere Instanz oder geben Sie einen Custom Slug ein
5. Klicken Sie **„Shorten"**

6. Die verkürzte URL erscheint mit **Copy-Button**

Berechtigungen

Die Extension benötigt nur zwei Berechtigungen (manifest.json):

- `activeTab`: Zugriff auf die URL des aktiven Tabs
- `storage`: Speichern der Instanz-Konfigurationen

Es werden **keine** Host-Permissions benötigt, da `fetch()` in Manifest V3 standardmäßig CORS-Requests durchführen darf (die API-Endpunkte müssen allerdings CORS erlauben oder die Extension muss die Requests direkt an die URL senden).

---

11. Automatische Link-Expiration (Cron)

Einrichtung

Der Endpunkt `/cron` deaktiviert alle abgelaufenen Links. Die Funktion expireLinks führt folgendes SQL aus:

```
UPDATE links SET active = 0
WHERE expires_at IS NOT NULL
  AND expires_at <= datetime('now')
  AND active = 1
```

Cron-Job einrichten

Fügen Sie in Ihrer Crontab folgenden Eintrag hinzu (z.B. alle 5 Minuten):

```
*/5 * * * * curl -s https://kurz.example.com/cron > /dev/null 2>&1
```

Alternativ via PHP-CLI:

```
*/5 * * * * php /var/www/html/index.php route=cron > /dev/null 2>&1
```

Die Antwort ist ein JSON-Objekt: `{"expired": 3}` – die Anzahl der gerade deaktivierten Links.

Hinweis: Der Cron-Endpunkt ist **nicht authentifiziert**. Er führt jedoch nur eine Statusänderung durch (aktiv → inaktiv) und gibt keine sensiblen Daten zurück. Wenn Sie das absichern möchten, können Sie einen API-Key prüfen oder den Zugriff per `.htaccess` einschränken.

Wichtig: Auch ohne Cron werden abgelaufene Links beim Aufrufen als „nicht verfügbar" angezeigt, da die Prüfung `strtotime($link['expires_at']) <= time()` direkt in der Routing-Logik stattfindet. Der Cron-Job sorgt lediglich dafür, dass der `active`-Status in der Datenbank korrekt gesetzt wird (relevant für die Dashboard-Anzeige).

---

12. Sicherheitshinweise

Passwort in config.php

Das Admin-Passwort in config.php steht im **Klartext**. Stellen Sie sicher, dass:

- Die Datei nicht über den Webserver direkt abrufbar ist
- Ein starkes Passwort verwendet wird (nicht das Standard-`hackme123`!)

Datenbankschutz

Die .htaccess blockiert den direkten Zugriff auf `.db`- und `.sqlite`-Dateien:

```
<FilesMatch "\.db$">
    Require all denied
</FilesMatch>
```

XSS-Schutz

Alle Benutzereingaben werden in den Templates mit `htmlspecialchars()` escaped, z.B.:

```
<?= htmlspecialchars($link['slug']) ?>
```

SQL-Injection-Schutz

Alle Datenbankabfragen verwenden **Prepared Statements** mit Paramter-Binding:

```
$stmt = $db->prepare('SELECT * FROM links WHERE slug = ?');
$stmt->execute([$slug]);
```

Die einzige Ausnahme ist die dynamische `ORDER BY`-Klausel in getAllLinks, die aber über ein Whitelist-Array abgesichert ist.

CSRF-Schutz

Aktuell gibt es **keinen CSRF-Token-Schutz**. Da die Anwendung nur einen einzigen Admin-User hat, ist das Risiko begrenzt, aber bei einer Erweiterung sollte ein Token-System implementiert werden.

---

13. Technische Architektur im Detail

Routing-System

Das Routing in index.php funktioniert als **Kaskade von if-Statements**:

```
Eingang: $_GET['route'] (via .htaccess Rewrite)
         ↓
1. Exakte Routen: 'cron', 'api/shorten', 'api/check', 'admin/action'
         ↓
2. QR-Routen: 'qr/png', 'qr/svg', 'qr/png/download', 'qr/svg/download'
         ↓
3. Slug-Auflösung: Beliebiger Pfad → getLinkBySlug()
         ↓
4. Seiten-Routing: $_GET['page'] → login, dashboard, edit, stats, settings
```

Datenbank-Schema

```
links (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  slug TEXT UNIQUE NOT NULL,        -- Der Kurzlink-Code
  url TEXT NOT NULL,                -- Ziel-URL
  password TEXT DEFAULT NULL,       -- bcrypt-Hash oder NULL
  expires_at TEXT DEFAULT NULL,     -- ISO-8601 Ablaufzeit
  max_clicks INTEGER DEFAULT NULL,  -- Klick-Limit
  clicks INTEGER DEFAULT 0,         -- Aktueller Zähler
  active INTEGER DEFAULT 1,         -- 1 = aktiv, 0 = deaktiviert
  og_title TEXT,                    -- OpenGraph-Cache
  og_description TEXT,
  og_image TEXT,
  created_at TEXT,                  -- datetime('now')
  updated_at TEXT
)

clicks (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  link_id INTEGER NOT NULL,         -- FK → links.id
  ip TEXT,
  user_agent TEXT,
  referer TEXT,
  clicked_at TEXT                   -- datetime('now')
  FOREIGN KEY (link_id) REFERENCES links(id) ON DELETE CASCADE
)

settings (
  key TEXT PRIMARY KEY,
  value TEXT                        -- Aktuell nicht aktiv genutzt
)
```

Frontend-JavaScript

app.js ist bewusst minimal und framework-frei:

- **Batch-Modal:** Toggle der CSS-Klasse `active` auf dem Overlay
- **QR-Modal:** Dynamisches Setzen der `src`- und `href`-Attribute basierend auf dem Slug
- **Slug-Preview:** Live-Aktualisierung beim Tippen mit Regex-Filterung
- **Copy-Button:** Via `navigator.clipboard.writeText()` mit visueller Bestätigung

- **Delete-Confirmation:** Nativer `confirm()`-Dialog

Performance-Überlegungen

- **SQLite WAL-Modus:** Ermöglicht gleichzeitige Leser während ein Schreibvorgang läuft
- **QR-Code-Cache:** Verhindert wiederholte Berechnung (24h TTL)
- **OpenGraph-Cache in DB:** Meta-Daten werden nur beim Erstellen/Ändern der URL abgerufen
- **Statische Assets:** CSS und JS werden direkt ausgeliefert (kein Build-Prozess nötig)

Zusammenfassung

LinkShorter v2 ist eine **schlanke, selbstgehostete Lösung** ohne externe Abhängigkeiten. Die bemerkenswerteste technische Leistung ist die vollständige **QR-Code-Implementierung in reinem PHP**, inklusive Reed-Solomon-Fehlerkorrektur und Galois-Feld-Arithmetik. Die Chrome-Extension ergänzt das System um einen **komfortablen Workflow** direkt aus dem Browser heraus, wobei die Multi-Instanz-Unterstützung besonders für Nutzer mit mehreren Domains nützlich ist.