8. PHP в веб-разработке. Микрофреймворки для PHP

Типы веб-приложений по способу разработки

Разработка веб-приложения vs приготовление пиццы

Разработка веб-приложения	Приготовление пиццы
Чистый язык, самописное решение	Муку помололи сами, помидоры вырастили на грядке. Собрали урожай, сделали тесто, приготовили "как рука взяла".
Использование фреймворка	Купили все продукты по отдельности, тесто раскатали сами, приготовили по рецепту
Использование CMS	Купили готовые полуфабрикаты (наборы для приготовления), приготовили по рецепту
Использование конструктора, no-code	Заказали готовую пиццу с доставкой

Роль PHP в веб-разработке

По уровню абстрагирования от языка программирования

- **Чистый язык**, самописное решение.
  Пишут только на PHP
- **Фреймворки** — наборы инструментов, библиотек, шаблонов и готовых решений, которые упрощают и ускоряют разработку веб-приложений.
  Используют разные языки, в том числе PHP
- **CMS** (Content Management System, готовые системы для управления контентом) и **low-code** решения.
  Самые популярные написаны на PHP
- **Конструкторы** и платформы для создания сайтов, не требующие программирования (решения **no-code**).

PHP в CMS, фреймворках и самописных проектах

Различия в подходах к программированию на PHP:

- Структура проекта
- Использование готовых компонентов
- Работа с базами данных
- Безопасность
- Производительность

Конструкторы сайтов (no-code)

No-code



Конструкторы предоставляют полностью готовое решение для создания сайта.

Нужно только выбрать шаблон, настроить его под свои нужды и добавить контент.

Плюсы:

- Быстро и удобно
- Не требуется специальных навыков

Минусы:

- Ограниченная кастомизация
- Подходит только для простых задач

No-code

1. Универсальные конструкторы сайтов

Подходят для создания сайтов-визиток, портфолио, блогов, корпоративных сайтов.

- Wix
  - Простой drag-and-drop редактор, сотни шаблонов.
  - Плюсы: Гибкость, интеграция с приложениями (например, для онлайн-записи).
- Squarespace
  - Красивые шаблоны, акцент на дизайн.
  - Плюсы: Подходит для творческих проектов, блогов, небольших магазинов.
- Tilda
  - Блоковый редактор с анимацией, подходит для лендингов и презентаций.
  - Плюсы: Красивые анимированные шаблоны, интеграция с CRM.
- Webflow
  - Визуальный редактор с возможностями, близкими к коду.
  - Плюсы: Гибкий дизайн, поддержка адаптивности и CMS.

No-code

2. Платформы для интернет-магазинов

Для создания онлайн-магазинов без программирования.

- **Shopify**
  - Лидер в e-commerce: готовые шаблоны, встроенная платежная система.
  - Плюсы: Все включено, поддержка дропшиппинга.
- **Ecwid**
  - Встраивается в существующие сайты (WordPress, Wix).
  - Плюсы: Интеграция с маркетплейсами, бесплатный тариф.
- **Squarespace Commerce**
  - Красивые шаблоны для брендированных магазинов.
  - Плюсы: Встроенная аналитика, поддержка подписок.

No-code

3. Конструкторы лендингов

Для создания одностраничных сайтов (лендингов) и рекламных страниц.

- Carrd
  - Минималистичный конструктор для одностраничных сайтов.
  - Плюсы: Низкая цена, скорость создания.
- Instapage
  - Фокус на конверсионные страницы с A/B-тестированием.
  - Плюсы: Инструменты для маркетинга.
- Leadpages
  - Готовые шаблоны для лидогенерации.
  - Плюсы: Интеграция с email-рассылками.

No-code

4. No-Code для веб-приложений

Для создания сложных приложений с уникальной логикой.

- **Bubble**
  - Платформа для разработки веб-приложений без кода.
  - Плюсы: Можно создать соцсеть, маркетплейс, SaaS.
- **Adalo**
  - Создание мобильных и веб-приложений с базами данных.
  - Плюсы: Визуальный редактор логики.
- **Glide**
  - Превращает Google Sheets в приложения.
  - Плюсы: Быстрое прототипирование.

CMS
(Content Management System)

CMS

CMS (например, WordPress, Joomla, Drupal) — это системы управления контентом, которые предоставляют готовую структуру и дизайн, но требуют настройки и установки дополнительных модулей (плагинов) для расширения функционала.



Плюсы:

- Больше гибкости, чем у конструкторов
- Можно создавать сложные сайты без написания кода с нуля

Минусы:

- Требует времени на настройку и обучение

Классические и headless CMS

1. Классические CMS (WordPress, Drupal, Joomla)

Архитектура :

- **Монолитная** (фронтенд и бэкенд тесно связаны).
- Контент хранится в базе данных, а шаблоны (темы) определяют его отображение.

Как работает :

- Бэкенд :
  - Управляет контентом (база данных).
  - Обрабатывает логику (плагины, модули).
  - Генерирует HTML-страницы на сервере при запросе пользователя.
- Фронтенд :
  - Использует готовые шаблоны (темы) для отображения контента.
  - HTML/CSS/JavaScript генерируются сервером (PHP) и отправляются браузеру.
  - Пользователь видит страницу, которая уже содержит контент и дизайн.

Классические и headless CMS

2. Headless CMS (Strapi, Contentful, Ghost)

Архитектура :

- **Децентрализованная** (бэкенд и фронтенд разделены).
- Контент предоставляется через API (REST/GraphQL), а фронтенд разрабатывается отдельно.

Как работает :

- Бэкенд :
  - Управляет контентом (база данных).
  - Предоставляет данные через API (например, JSON).
  - Нет встроенной логики для генерации HTML.
- Фронтенд :
  - Разрабатывается независимо (React, Vue.js, мобильные приложения).
  - Запрашивает данные через API и отображает их.
  - Контент подтягивается динамически (AJAX, SSR, SSG).

Сценарии использования

- Классические CMS :
  - Блоги, корпоративные сайты, небольшие проекты.
  - Когда нужна скорость запуска и минимальные технические навыки.
- Headless CMS :
  - Многоплатформенные приложения (веб + мобильные + IoT).
  - Проекты с высокими требованиями к производительности и масштабируемости.
  - Использование современных фронтенд-технологий (React, Vue, Svelte).

Статистика по языкам для CMS

1. **PHP** — доминирует в CMS благодаря WordPress, Joomla и Drupal.
2. **JavaScript/Node.js** — активно используется в headless CMS и современных решениях.
3. **Python** — популярен для CMS, ориентированных на гибкость.
4. **C#** — используется в enterprise-проектах.
5. **Java** — применяется в крупных организациях.
6. **Go** — используется для статических генераторов.

Почему другие языки менее популярны для CMS?

- **Сложность развертывания** : Java и C# требуют больше ресурсов и настроек.
- **Ориентация на ниши** : Python и Ruby чаще используются для веб-приложений, а не CMS.
- **Рост PHP-экосистемы** : WordPress и аналоги создали стандарты, которые сложно превзойти.
- **Тренды** : Статические генераторы (Hugo, Jekyll) и headless CMS (Contentful, Strapi) набирают популярность, но часто используются в связке с фронтенд-фреймворками.

Веб-фреймворки

Фреймворки



Инструменты для разработки уникальных веб-приложений.

Предоставляют структуру, базовую функциональность и инструменты, но всю логику, дизайн и функционал нужно создавать самостоятельно.

Плюсы:

- Максимальная свобода и контррль
- Возможность создавать уникальные решения

Минусы:

- Требует глубоких знания программирования
- Длительный процесс разработки

Фреймворки для PHP

1. Laravel
    - **Особенности** : Элегантный синтаксис, ORM (Eloquent), миграции, Blade-шаблоны, встроенная аутентификация.
    - **Сильные стороны** : Активное сообщество, богатая экосистема (Laravel Mix, Nova, Forge).
    - **Сценарии** : Стартапы, SaaS, сайты с высокой нагрузкой.
    - **Примеры** : Laravel используется в стартапах и корпорациях (например, Pfizer, BBC).
2. Symfony
    - **Особенности** : Модульная архитектура, повторяемые компоненты, интеграция с Drupal.
    - **Сильные стороны** : Гибкость, подходит для enterprise-проектов.
    - **Сценарии** : Крупные приложения, микросервисы.

Фреймворки для Python

1. Django
   - **Особенности** : «Батарейки включены» (ORM, админка, auth), MTV-архитектура.
   - **Сильные стороны** : Быстрая разработка, безопасность.
   - **Сценарии** : Социальные сети, контент-платформы (например, Instagram, Pinterest).
2. Flask
   - **Особенности** : Минимализм, гибкость, расширяемость через библиотеки.
   - **Сильные стороны** : Идеален для микросервисов и API.
   - **Сценарии** : Небольшие приложения, REST API, прототипирование.

Фреймворки для JavaScript/Node.js

1. Express.js
   - **Особенности** : Минималистичный, middleware-ориентированный, гибкий.
   - **Сильные стороны** : Широко используется, поддерживает тысячи пакетов npm.
   - **Сценарии** : API, веб-серверы, микросервисы.
2. NestJS
   - **Особенности** : Модульная архитектура, TypeScript, интеграция с GraphQL.
   - **Сильные стороны** : Структурированность, подходит для сложных проектов.
   - **Сценарии** : Enterprise-приложения, масштабируемые системы.
3. Next.js
   - **Особенности** : SSR, статическая генерация, гибридный рендеринг.
   - **Сильные стороны** : SEO-оптимизация, интеграция с React.
   - **Сценарии** : Маркетинговые сайты, динамические веб-приложения.

Фреймворки

Ruby

1. Ruby on Rails
    - **Особенности** : Convention over Configuration, ActiveRecord, миграции.
    - **Сильные стороны** : Быстрая разработка, сообщество (GitHub, Shopify).
    - **Сценарии** : Стартапы, MVP, проекты с быстрым тайм-ту-маркет.

Java

1. Spring Boot
    - **Особенности** : Dependency Injection, Spring Security, Spring Data.
    - **Сильные стороны** : Enterprise-ориентированность, интеграция с Java EE.
    - **Сценарии** : Банковские системы, крупные корпоративные приложения.

Фреймворки

C#

1. ASP.NET Core
    - **Особенности** : Высокая производительность, интеграция с Azure, кроссплатформенность.
    - **Сильные стороны** : Корпоративные приложения, игры (Unity), IoT.
    - **Сценарии** : Enterprise-решения, микросервисы, облачные приложения.

Go (Golang)

1. Gin
    - **Особенности** : Высокая скорость, минималистичность, маршрутизация.
    - **Сильные стороны** : Подходит для высоконагруженных API.
    - **Сценарии** : Микросервисы, REST API.
2. Echo
    - **Особенности** : Легковесность, middleware, поддержка WebSocket.
    - **Сильные стороны** : Простота, быстрый старт.

Фреймворки

Итог: как выбрать?

- Для быстрого старта : Ruby on Rails или Laravel .
- Для enterprise-проектов : Spring Boot или ASP.NET Core .
- Для высоконагруженных API : Express.js , Gin или Echo .
- Для стартапов с фокусом на Python : Django или Flask .
- Для гибридных приложений (SSR + статика) : Next.js .

Каждый фреймворк имеет свои сильные стороны, поэтому выбор зависит от требований проекта, команды и экосистемы.

Чистый PHP

Чистый PHP



Нужно самостоятельно реализовывать даже базовые функции (авторизация, CRUD-операции, защита от SQL-инъекций и XSS-атак и т.п.).

Плюсы:



- Полный контроль над кодом
- Высокая производительность
- Низкая зависимость от внешних обновлений
- Минимальные требования к хостингу

Минусы:

- Сложность разработки
- Риски безопасности
- Сложность понимания и поддержки кода (PHP + HTML)
- Отсутствие стандартизации, сложность интеграции сторонних сервисов

PHP-программист...
Кто это?

PHP в CMS, фреймворках и самописных проектах

Различия в подходах к программированию на PHP:

- Структура проекта
- Использование готовых компонентов
- Работа с базами данных
- Безопасность
- Производительность

PHP в CMS, фреймворках и чистом виде

Структура проекта

- CMS (WordPress) :
  - **Жёсткая структура** : Проект строится вокруг ядра CMS, которое определяет расположение файлов (например, папки `wp-content/themes` , `wp-content/plugins`).
  - **Ограниченная свобода** : Разработчик зависит от архитектуры CMS (например, WordPress использует глобальные объекты `$wpdb` , `WP_Query`).
  - **Пример** : Создание темы или плагина требует следования правилам WordPress (например, использование хуков `add_action()` , `add_filter()`).
- Чистый PHP :
  - **Полная свобода** : Разработчик сам определяет структуру проекта, маршрутизацию, работу с БД.
  - **Пример** : Скрипт для генерации PDF-документов или простой API без использования сторонних систем.
- Фреймворки (Laravel) :
  - **Стандартизированная структура** : MVC-архитектура, папки `app/` , `routes/` , `resources/` .
  - **Пример** : Использование Eloquent ORM для работы с базой данных через модели.

PHP в CMS, фреймворках и чистом виде

Использование готовых компонентов

- CMS (WordPress) :
  - **Готовые решения** : Плагины (WooCommerce, Yoast SEO) и темы позволяют быстро добавить функционал.
  - **Зависимость от экосистемы** : Обновления CMS или плагинов могут сломать кастомный код.
  - **Пример** : Добавление формы обратной связи через плагин Contact Form 7 вместо написания кода с нуля.
- Чистый PHP :
  - **Всё с нуля** : Нужно самостоятельно реализовывать даже базовые функции (авторизация, CRUD-операции).
  - **Пример** : Написание скрипта для регистрации пользователей без использования библиотек.
- Фреймворки (Laravel) :
  - **Модули и пакеты** : Использование готовых компонентов (например, пакеты для авторизации, кеширования).
  - **Пример** : Использование Laravel Breeze для быстрого старта с аутентификацией.

PHP в CMS, фреймворках и чистом виде

Работа с базой данных

- CMS (WordPress) :
  - **Использование глобальных функций** : Например, `Swpdb` для запросов или `WP_Query` для выборки постов.
  - **Стандартные таблицы** : Работа с таблицами, которые уже созданы CMS (например, `wp_posts`, `wp_users`).
  - **Пример** :

```php
$posts = new WP_Query(['category_name' => 'news']);
```

- Чистый PHP :
  - **Ручное управление** : Подключение к БД через `mysqli` или PDO, написание SQL-запросов.

- Фреймворки (Laravel) :
  - **ORM (Eloquent)** : Работа с БД через объекты, миграции, сиды.
  - **Пример** :

```php
$users = User::where('role', 'admin')->get();
```

PHP в CMS, фреймворках и чистом виде

Безопасность

- CMS (WordPress) :
  - **Уязвимости плагинов** : Некачественные плагины могут создать дыры в безопасности.
  - **Автоматизация** : Встроенные функции экранирования (например, `esc_html()` , `wp_kses()`).
  - **Пример** : Использование `nonce` для защиты форм.
- Чистый PHP :
  - **Ручная защита** : Нужно самому обрабатывать SQL-инъекции, XSS-атаки.
- **Фреймворки (Laravel)** :
  - **Встроенная защита** : CSRF-токены, экранирование Blade-шаблонов, защита от SQL-инъекций через Eloquent.
  - **Пример** :

```php
{{ $user->name }} <!-- Автоматическое экранирование -->
```

PHP в CMS, фреймворках и чистом виде

Производительность

- CMS (WordPress) :
  - **Высокая нагрузка** : Много запросов к БД из-за плагинов и тем.
  - **Оптимизация** : Кэширование (плагины W3 Total Cache), CDN.
- Чистый PHP :
  - **Минимальный overhead** : Нет лишних запросов или слоёв абстракции.
- Фреймворки (Laravel) :
  - **Умеренный overhead** : ORM и сервис-провайдеры добавляют нагрузку, но есть инструменты для оптимизации (кэширование маршрутов, конфигов).

Чистый PHP, фреймворки и CMS

Когда что использовать?

- CMS (WordPress) :
  - Блоги, корпоративные сайты, интернет-магазины (с WooCommerce).
  - **Плюсы** : Быстрый старт, огромное количество готовых решений.
  - **Минусы** : Ограниченная кастомизация, зависимость от плагинов.
- Чистый PHP :
  - Простые скрипты, утилиты, проекты с уникальной логикой.
  - **Плюсы** : Полный контроль.
  - **Минусы** : Долгая разработка, высокая стоимость поддержки.
- Фреймворки (Laravel) :
  - Сложные приложения (SaaS, API, соцсети).
  - **Плюсы** : Чистый код, масштабируемость.
  - **Минусы** : Требует изучения структуры и паттернов.

Переход от чистого PHP к фреймворкам

Основные задачи веб-разработки

Задачи веб-разработки

- **Базовые задачи** — фундамент, без которого не обходится ни одно приложение
- **Дополнительные задачи** возникают при росте проекта и требуют интеграции специальных инструментов (ORM, шаблонизатоов, систем аутентификации и т.д.)

Базовые задачи веб-разработки

Эти задачи образуют основу веб-приложения:

- **Маршрутизация** определяет, куда направить запрос.
- **Обработка запросов** работает с данными и бизнес-логикой.
- **Middleware** решает общие задачи (логирование, безопасность), упрощая повторное использование кода.

Маршрутизация (Routing)

Определение, какой код выполнится при обращении к конкретному URL и HTTP-методу (GET, POST и т.д.).

Пример:

- `GET /about` → показать страницу «О нас».
- `POST /login` → обработать данные авторизации.

Динамические маршруты:

- `GET /user/123` → параметр `123` передаётся в обработчик.

Обработка HTTP-запросов

Работа с данными, пришедшими от клиента (через URL, тело запроса, заголовки), и формирование ответа.

Ключевые аспекты:

- **Получение данных:**
  - GET-параметры: `?id=5`.
  - POST-данные (формы, JSON).
- **Формирование ответа:**
  - Статус (200 — успех, 404 — не найдено).
  - Тело ответа (HTML, JSON, текст).

Пример:

php Копироват

```php
// Обработка POST-запроса
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $data = json_decode(file_get_contents('php://input'), true);
    // Логика обработки данных...
}
```

Работа с Middleware

Функции, которые выполняются **до** или **после** основного обработчика запроса.

Используются для повторяющихся задач.

Примеры:

- **Логирование:** запись информации о запросе.
- **Аутентификация:** проверка токена или сессии.
- **CORS:** добавление заголовков для кросс-доменных запросов.

Как работает:

```php
// Middleware для проверки авторизации
function authMiddleware($request, $next) {
    if (!isset($_SESSION['user'])) {
        return "Ошибка: не авторизован";
    }
    return $next($request); // Передача управления дальше
}
```

Дополнительные задачи веб-разработки

Возникают при росте проекта и требуют специальных инструментов:

- Работа с базами данных
- Разграничение прав пользователей
- Формирование ответа на запрос в разных форматах (HTML, JSON, XML, PDF, ...)
- Валидация входящих данных
- Поддержка безопасности, устойчивости к атакам
- Обработка ошибок на сервере
- Работа с очередями задач
- Поддержка тестирования кода
- ...

Дополнительные задачи веб-разработки

1. Работа с базами данных

- **Подключение к БД** (MySQL, PostgreSQL, MongoDB).
- **Выполнение запросов** (SQL или NoSQL).
- **ORM (Object-Relational Mapping)** для удобной работы с данными.

  Пример (Laravel Eloquent):

```php
$user = User::where('email', 'user@example.com')->first();
```

- **Миграции** для управления структурой БД.

Дополнительные задачи веб-разработки

2. Разграничение прав пользователей

- **Аутентификация** (проверка логина/пароля, JWT-токенов).
- **Авторизация** (разрешения для разных ролей: админ, пользователь, гость).

  Пример:

php Копировать

```php
// Middleware для проверки прав
if (!$user->hasRole('admin')) {
    throw new AccessDeniedException();
}
```

- **Управление ролями и разрешениями** (например, через библиотеки типа Laravel Permissions).

Дополнительные задачи веб-разработки

3. Формирование вывода

- **Шаблонизация** (генерация HTML с использованием Twig, Blade, Smarty).

  Пример (Blade):

```html
<h1>{{ $post->title }}</h1>
<p>{{ $post->content }}</p>
```

- **API-ответы** (формирование JSON/XML).
- **Генерация файлов** (PDF, Excel, CSV).
- **Кэширование** для ускорения отдачи данных.

Дополнительные задачи веб-разработки

- **Валидация данных** (проверка корректности ввода).
- **Локализация** (многоязычность интерфейса).
- **Обработка ошибок** (отображение страниц 404, 500).
- **Безопасность** (защита от XSS, CSRF, SQL-инъекций).
- **Работа с очередями задач** (отправка email в фоне).
- **Тестирование** (юнит-тесты, интеграционные тесты).

Веб-разработка на чистом PHP

Что есть в стандартном PHP

1. **HTTP-протокол** :
   - Суперглобальные массивы `$_GET` , `$_POST` , `$_SERVER` , `$_FILES` .
   - Функции вроде `header()` , `setcookie()` .
2. **Шаблонизация** :
   - Встроенный синтаксис `<?php ?>` и `<?= ?>` для смешивания HTML и PHP.
3. **Веб-сервер** :
   - `php -S localhost:8000` для быстрого запуска.
4. **Базовые функции** :
   - Работа с БД (`mysqli` , `PDO`).
   - Обработка форм, сессии, JSON.

Стандартный PHP — плюсы

1. **Простота для новичков**

   Не требуется изучать архитектурные паттерны или фреймворки — можно сразу писать код.
2. **Скорость разработки маленьких проектов**

   Для простых скриптов (например, форма обратной связи) такой подход позволяет быстро получить результат.
3. **Минимум шаблонного кода**

   Нет необходимости разделять код на классы, конфигурировать маршрутизацию или настраивать слои.

Стандартный PHP — минусы

- Сложность поддержки и дублирование кода
- Потенциальные проблемы с безопасностью
- Сложность тестирования кода
- Проблемы масштабирования кода

Недостатки стандартного PHP

1. **Сложность поддержки**

   Код превращается в «макаронный клубок» (spaghetti code), где HTML, SQL и PHP-логика перемешаны.

   Пример: Поиск уязвимости в обработке данных или исправление ошибки в шаблоне требует анализа всего файла.

2. **Дублирование кода**

   Одинаковые блоки (например, подключение к БД, проверка авторизации) копируются в разных файлах.

   Пример: Изменение пароля от БД потребует правок во всех скриптах.

Недостатки стандартного PHP

3. **Проблемы с безопасностью**

   Легко допустить уязвимости (например, SQL-инъекции), когда логика обработки данных смешана с выводом.

   Пример: Запрос `SELECT * FROM users WHERE name='$_GET[user]'` без экранирования.

4. **Сложность тестирования**

   Невозможно протестировать отдельные части кода (например, бизнес-логику) без запуска всего скрипта.

5. **Ограниченная читаемость**

   Разработчикам сложно понять структуру проекта из-за хаотичного смешения компонентов.

Недостатки стандартного PHP

Проблемы масштабирования:

1. **Сложность внесения изменений**

   Добавление новой фичи или исправление бага может привести к поломке других частей приложения.

   Пример: Исправление HTML-формы может случайно нарушить логику обработки данных.

2. **Параллельная работа в команде**

   Несколько разработчиков не могут одновременно работать над разными частями проекта (логикой, интерфейсом, БД), так как всё находится в одном файле.

Недостатки стандартного PHP

Проблемы масштабирования:

3. **Производительность**
   Повторяющиеся операции (например, подключение к БД в каждом файле) увеличивают нагрузку на сервер.
4. **Отсутствие единой точки входа**
   Каждый файл обрабатывает запросы независимо, что усложняет внедрение общих механизмов (логирование, кэширование, аутентификация).

Недостатки стандартного PHP

Смешанный подход

```php
<?php
// Обработка запроса, логика и вывод в одном файле
$username = $_GET['user'];
$conn = mysqli_connect("localhost", "root", "", "db");
$result = mysqli_query($conn, "SELECT * FROM users WHERE name='$username'");
echo "<h1>Привет, " . mysqli_fetch_assoc($result)['name'] . "!</h1>";
?>
```

Фреймворки для решения проблем чистого PHP

Типы фреймворков

- **Микрофреймворки** — это "конструкторы" для минимальной функциональности. Дают свободу, но требуют больше ручной работы.

- **Крупные фреймворки** — "полноценные станки" с готовыми решениями. Ускоряют разработку, но требуют изучения их структуры и инструментов.

Микрофреймворки

Микрофреймворки

Микрофреймворки структурируют код, разделяя обработку запросов, логику и вывод, но без избыточной сложности MVC-фреймворков. Например:

- **Маршрутизация** заменяет хаотичные `if-else` для обработки URL.
- **Middleware** позволяет централизованно обрабатывать аутентификацию, логирование и т.д.
- **PSR-стандарты** (для PHP) обеспечивают совместимость с другими библиотеками.

Микрофреймворки

- Основная идея — предоставить минимальный набор инструментов для создания веб-приложения, сохраняя при этом гибкость и простоту.

- Микрофреймворки фокусируются на базовых задачах:
  - маршрутизация
  - обработка HTTP-запросов
  - работа с middleware

  не навязывая сложную архитектуру или избыточные компоненты.

- Можно самостоятельно выбрать, какие части приложения оставить простыми, а какие структурировать.

Микрофреймворки

Ключевые принципы микрофреймворков:

1. **Минимализм**

   Микрофреймворки включают только необходимые компоненты (например, маршрутизация, обработка запросов/ответов). Всё остальное (ORM, шаблонизаторы, системы аутентификации) подключается отдельно по выбору разработчика.
2. **Гибкость**

   Нет строгих правил архитектуры (например, MVC). Разработчик сам решает, как организовать код: использовать паттерны, классы, функции или смешанный подход.

Микрофреймворки

3. **Простота настройки**

   Минимальная конфигурация. Например, в микрофреймворке **Slim** (PHP) можно создать приложение за несколько строк:

php Копировать

```php
use Slim\Factory\AppFactory;

$app = AppFactory::create();
$app->get('/hello', function ($request, $response) {
    $response->getBody()->write("Привет, мир!");
    return $response;
});
$app->run();
```

4. **Небольшой размер**

   Микрофреймворки имеют маленький объём кода, что упрощает изучение и интеграцию в проект.

Преимущества микрофреймворков

1. **Быстрый старт**

   Идеальны для небольших проектов, API, микросервисов или прототипов. Не нужно тратить время на настройку сложных структур.
2. **Полный контроль**

   Разработчик сам выбирает компоненты (базы данных, шаблонизаторы, библиотеки) вместо того, чтобы подстраиваться под решения, навязанные фреймворком.
3. **Производительность**

   Меньше кода → меньше накладных расходов. Например, **Express.js** работает быстрее, чем «тяжёлые» фреймворки вроде NestJS.
4. **Лёгкость интеграции**

   Микрофреймворки можно использовать в части приложения. Например, добавить **Flask** для обработки API в существующем проекте.

Недостатки микрофреймворков

1. **Отсутствие готовых решений**

   Для реализации сложных функций (аутентификация, миграции БД, очереди задач) придётся подключать дополнительные библиотеки или писать код самостоятельно.
2. **Риск «велосипедизации»**

   Без готовых компонентов разработчики могут потратить время на реализацию того, что уже есть в крупных фреймворках.
3. **Ограниченная поддержка**

   Некоторые микрофреймворки (например, Silex) перестают развиваться, тогда как крупные фреймворки (Laravel, Django) имеют долгосрочную поддержку.

Применимость микрофреймворков

- Для **маленьких проектов** (лендинги, простые API).
- В **микросервисной архитектуре**.
- Для **прототипирования идей**.
- Когда нужен **полный контроль над структурой** приложения.

Фреймворки и стандарты PSR

Стандартизация фреймворков

Для унификации работы с HTTP-сообщениями и клиентами в PHP разработаны стандарты PSR-7 , PSR-17 и PSR-18.

Они решают проблему несовместимости между фреймворками и библиотеками, обеспечивая единые интерфейсы.

- PSR-7 — стандарт для HTTP-сообщений.
- PSR-17 — фабрики для создания PSR-7 объектов.
- PSR-18 — унифицированный HTTP-клиент.

PSR-7: HTTP Message Interfaces

Интерфейсы для работы с HTTP-сообщениями (запросы и ответы) и их компонентами (URI, заголовки, тело).

- `Psr\Http\Message\MessageInterface` — базовый интерфейс для всех сообщений.
- `Psr\Http\Message\RequestInterface` — HTTP-запрос.
- `Psr\Http\Message\ResponseInterface` — HTTP-ответ.
- `Psr\Http\Message\ServerRequestInterface` — расширяет `RequestInterface` для серверных запросов (например, данные из `$_SERVER`).
- `Psr\Http\Message\UriInterface` — URI (URL).
- `Psr\Http\Message\StreamInterface` — поток для тела запроса/ответа.
- `Psr\Http\Message\UploadedFileInterface` — загруженные файлы.

PSR-7: HTTP Message Interfaces

PSR-7 позволяет:

- Использовать универсальные компоненты (например, middleware) в любом фреймворке.
- Передавать запросы/ответы между библиотеками без конвертации.

PSR-17: HTTP Factories

Интерфейсы для фабрик , создающих объекты PSR-7.

- `RequestFactoryInterface` — создает `RequestInterface`.
- `ResponseFactoryInterface` — создает `ResponseInterface`.
- `ServerRequestFactoryInterface` — создает `ServerRequestInterface`.
- `StreamFactoryInterface` — создает `StreamInterface`.
- `UploadedFileFactoryInterface` — создает `UploadedFileInterface`.
- `UriFactoryInterface` — создает `UriInterface`.

PSR-17: HTTP Factories

PSR-17 предоставляет стандартные методы для создания объектов, что позволяет:

- Не зависеть от конкретной реализации (например, использовать Guzzle или Slim для создания запроса через единый интерфейс).
- Упростить внедрение зависимостей (DI) в фреймворках.

```
use Psr\Http\Message\RequestFactoryInterface;

class MyService {
    public function __construct(private RequestFactoryInterface $requestFactory) {}

    public function createRequest(): RequestInterface {
        return $this->requestFactory->createRequest('GET', 'https://example.com');
    }
}
```

PSR-18: HTTP Client

Интерфейс `ClientInterface` для отправки HTTP-запросов.

Метод

- `sendRequest(RequestInterface $request): ResponseInterface`

отправляет запрос и возвращает ответ.

PSR-18: HTTP Client

Интерфейс `ClientInterface` для отправки HTTP-запросов.

Метод

- `sendRequest(RequestInterface $request): ResponseInterface`

отправляет запрос и возвращает ответ.

PSR-18: HTTP Client

PSR-18 позволяет:

- Писать код, независимый от конкретного HTTP-клиента.
- Легко заменять клиент (например, Guzzle на Symfony HttpClient) без изменения логики приложения.

```php
use Psr\Http\Client\ClientInterface;
use Psr\Http\Message\RequestFactoryInterface;

class ApiService {
    public function __construct(
        private ClientInterface $client,
        private RequestFactoryInterface $requestFactory
    ) {}

    public function fetchData(): ResponseInterface {
        $request = $this->requestFactory->createRequest('GET', 'https://api.example.com/d
        return $this->client->sendRequest($request);
    }
}
```

Стандартизация фреймворков

Преимущества стандартизации:

- **Взаимозаменяемость**. Можно комбинировать компоненты из разных библиотек.
- **Снижение зависимости**. Код не привязан к конкретной реализации (например, Guzzle).
- **Упрощение тестирования**. Можно легко подменять HTTP-клиенты и запросы в тестах.

Стандартизация фреймворков

Пример интеграции:

```
// Создаем запрос через PSR-17
$request = (new GuzzleRequestFactory())->createRequest('GET', 'https://api.example.com'

// Отправляем через PSR-18 клиент (например, Symfony HttpClient)
$client = new SymfonyHttpClient();
$response = $client->sendRequest($request);

// Работаем с ответом (PSR-7)
echo $response->getBody();
```

Микрофреймворк Slim

Slim

Создан Джошуа Локхартом (Josh Lockhart) в 2011 году.

Принцип "меньше кода — больше гибкости" (родоначальник фреймворк Sinatra для Ruby)

- Поддержка PSR-7
- Встроенные middleware
- Маршрутизация с поддержкой параметров
- Возможность интеграции с любыми библиотеками PHP

Маршрутизация в Slim

Маршрутизация связывает URL-адреса и HTTP-методы с обработчиками (функциями или классами).

Примеры:

Статический маршрут (GET):

php Копировать

```php
use Slim\Factory\AppFactory;

$app = AppFactory::create();

// Отображение главной страницы
$app->get('/', function ($request, $response) {
    return $response->write("Добро пожаловать на главную!");
});

$app->run();
```

Маршрутизация в Slim

Динамический маршрут (GET):

php Копировать

```php
// Профиль пользователя с ID
$app->get('/user/{id}', function ($request, $response, $args)
    $userId = $args['id']; // Получаем параметр из URL
    return $response->write("Профиль пользователя: $userId");
});
```

Маршрут с HTTP-методом POST:

php Копировать

```php
// Обработка формы входа
$app->post('/login', function ($request, $response) {
    $data = $request->getParsedBody(); // Получаем данные из
    $login = $data['login'] ?? '';
    $password = $data['password'] ?? '';

    // Логика аутентификации...
    return $response->withStatus(200)->write("Вход выполнен!"
});
```

Обработка HTTP-запросов в Slim

Slim предоставляет объекты `$request` и `$response` для работы с данными.

Получение данных из запроса:

php

```php
$app->post('/submit', function ($request, $response) {
    // GET-параметры
    $queryParams = $request->getQueryParams();
    $page = $queryParams['page'] ?? 1;

    // POST-данные (например, форма)
    $postData = $request->getParsedBody();
    $username = $postData['username'] ?? '';

    // JSON-данные
    $jsonData = $request->getBody()->getContents();
    $data = json_decode($jsonData, true);

    return $response->write("Данные получены!");
});
```

Обработка HTTP-запросов в Slim

Формирование ответа:

php Копировать

```php
$app->get('/api/users', function ($request, $response) {
    $users = [
        ['id' => 1, 'name' => 'Алексей'],
        ['id' => 2, 'name' => 'Мария']
    ];

    // Отправка JSON
    return $response
        ->withHeader('Content-Type', 'application/json')
        ->withStatus(200)
        ->write(json_encode($users));
});
```

Работа с middleware в Slim

Middleware — это функции, которые выполняются до или после основного обработчика маршрута.

Пример 1: Логирование запросов

php Копировать

```php
// Middleware для логирования
$loggingMiddleware = function ($request, $handler) {
    $startTime = microtime(true);
    $response = $handler->handle($request);
    $endTime = microtime(true);
    $time = $endTime - $startTime;

    $uri = $request->getUri();
    $method = $request->getMethod();
    echo "[$method] $uri — Время выполнения: $time сек\n";

    return $response;
};

// Добавляем middleware ко всем маршрутам
$app->add($loggingMiddleware);
```

Работа с middleware в Slim

Пример 2: Проверка авторизации

php Копировать

```php
// Middleware для авторизации
$authMiddleware = function ($request, $handler) {
    $token = $request->getHeaderLine('Authorization');

    if (!$token || $token !== 'secret_token') {
        return $response
            ->withStatus(401)
            ->write('Доступ запрещен!');
    }

    // Если токен валиден — передаем управление дальше
    return $handler->handle($request);
};

// Применяем middleware к конкретному маршруту
$app->get('/admin', function ($request, $response) {
    return $response->write("Добро пожаловать в админку!");
})->add($authMiddleware);
```

Работа с middleware в Slim

Как это работает вместе:

1. **Запрос приходит в приложение.**
2. **Middleware обрабатывает запрос** (логирование, проверка токена).
3. **Маршрутизатор определяет обработчик** на основе URL и HTTP-метода.
4. **Обработчик выполняет логику** (работа с БД, формирование ответа).
5. **Middleware может модифицировать ответ** (например, добавить заголовки).
6. **Ответ отправляется клиенту.**