

Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.M71-мм

# Эволюция PHP-фреймворков как отражение развития экосистемы веб-разработки (2005–2025)

*Альшаеб Басель*

Отчёт по учебной практике

Преподаватель:  
Басков Антон Андреевич

Санкт-Петербург  
2025

# Оглавление

<b>1. Введение</b>	<b>3</b>
<b>2. Предпосылки: фрагментированная экосистема PHP до 2015 года и необходимость стандартизации</b>	<b>5</b>
2.1. Отсутствие единых интерфейсов и высокая фрагментация	5
2.2. Проблема совместимости и отсутствие переиспользуемых компонентов . . . . .	6
2.3. Отсутствие стандартизированной модели HTTP . . . . .	6
2.4. Отсутствие единых правил автозагрузки и зависимостей . . . . .	7
2.5. Проблемы тестируемости и внедрения зависимостей . . . . .	8
<b>3. Развитие стандартов и архитектурных решений в экосистеме PHP (2015–2025)</b>	<b>10</b>
3.1. Composer как основа модернизации экосистемы (2015–2025)	10
3.2. PSR-7 и стандартизация HTTP-модели (2015) . . . . .	12
3.3. PSR-11 и унификация DI-контейнеров (2017) . . . . .	14
3.4. PSR-15 и middleware-архитектура (2017–2018) . . . . .	15
3.5. Влияние PHP 7 и PHP 8 на архитектуру фреймворков . .	16
3.6. Symfony: эволюция компонентной модели . . . . .	17
3.7. Laravel: эволюция DX и влияние стандартов . . . . .	17
<b>4. Отвергнутые и отложенные изменения</b>	<b>19</b>
4.1. Несостоявшийся PSR для маршрутизации (Router PSR)	19
4.2. Попытка создать PSR для ORM и абстракции работы с базами данных . . . . .	21
4.3. PSR-14 (Event Dispatcher): стандарт, который останется частичным . . . . .	22
4.4. Попытка создания PSR для валидаторов и форм . . . . .	22
4.5. Споры вокруг PSR-18 (HTTP Client) . . . . .	23
4.6. Предложение о расширении PSR-11 (унифицированная конфигурация контейнера) . . . . .	24



# 1. Введение

PHP традиционно играет важную роль в динамичном развитии веб-разработки в течение почти трех десятилетий. Согласно актуальной статистике W3Techs[61] по состоянию на 1 ноября 2025 года, PHP используется в 72.9% всех веб-сайтов с известным серверным языком.

PHP[20] стал основой для многих систем управления контентом и прикладных веб-решений благодаря низкому порогу входа и широкой доступности хостинга.

Тем не менее, устойчивость языка и его экосистемы на протяжении столь длительного периода в значительной степени связана с развитием общих архитектурных практик и стандартов, которые определили индустрию PHP[20]-разработки.

В начале 2000-х годов появились первые PHP[20]-фреймворки. CakePHP (2005)[2], Symfony (2005)[40] и CodeIgniter (2006)[5] поставили основы MVC-подхода и организовали разработку веб-приложений без стандартов.

Тем не менее, по мере роста сложности веб-приложений стало очевидно, что архитектура 2000-х годов сталкивалась с важными ограничениями, такими как непрозрачная архитектура, несовместимость компонентов, отсутствие унифицированных интерфейсов и низкая testируемость.

В течение следующих десяти лет (2005–2015) произошли значительные изменения в PHP[20]-стеке. Это было связано с разработкой PHP-FIG [34] и ряда стандартов PSR [27], появлением Composer [47], унификацией HTTP-модели[33], распространением принципов DI [28] и архитектурой middleware [30].

Тем не менее, между 2015 и 2025 годами произошли наиболее значительные структурные изменения в PHP[20]-фреймворке. Эти изменения радикально изменили архитектурные решения, принципы проектирования и практики масштабирования.

Эти изменения включали выход PHP 7[35], массовую типизацию и строгую модель ошибок, переход Symfony на компонентный подход,

стримительный рост Laravel, падение Zend Framework и его перерождение в Laminas.

Цель этой работы состоит в том, чтобы проанализировать основные архитектурные и технологические изменения в PHP[20]-фреймворках за 2015–2025 годы, определить причины этих изменений, оценить эффективность принятых решений и определить, какие идеи были отвергнуты или изменились в ходе обсуждений. В анализе используются материалы рассылок PHP-FIG[34], предложения RFC Internals PHP, официальные публикации Symfony[40] и Laravel[?], дискуссии GitHub о стандартах Composer и PSR, а также выступления разработчиков на конференциях, таких как SymfonyCon[42] и Laracon[19].

Таким образом, в статье рассматривается не только развитие определенных фреймворков, но и более широкий процесс создания профессиональной экосистемы PHP, в которой технологические решения являются основной движущей силой прогресса.

## **2. Предпосылки: фрагментированная экосистема PHP до 2015 года и необходимость стандартизации**

К 2015 году промышленная разработка PHP[20] началась с противоречий.

С другой стороны, благодаря своей простоте развёртывания и широкой совместимости PHP[20] продолжал доминировать в веб-разработке.

Напротив, экосистема испытывала значительные архитектурные ограничения из-за отсутствия единых правил и несовместимости основных компонентов фреймворка.

Эти ограничения замедлили язык и фреймворки.

### **2.1. Отсутствие единых интерфейсов и высокая фрагментация**

До того, как в экосистеме PHP появилась PHP-FIG[34], не существовало общепризнанных соглашений по:

- Система каталогов.
- Автоматическая загрузка классов.
- Форматы запросов и ответов HTTP.
- Интерфейсы контейнеров, которые зависят от зависимостей.
- Методы middleware.

Каждый фреймворк, включая Symfony[40], Zend Framework[64], CakePHP[2] и CodeIgniter[5], разработал свои собственные решения, которые часто не пересекались.

Что привело к эффекту «изолированных островов» в PHP-мире, заключалось в том, что библиотеки и инструменты не могли быть повторно использованы между проектами.

Например:

- Symfony[40] использовал собственный автозагрузчик и собственные соглашения об именовании классов.
- Zend Framework[64] применял иной подход к структуре каталогов.
- Многие библиотеки требовали ручного подключения файлов и не имели механизма декларирования зависимостей.

Это создавало тесную связность между фреймворком и компонентами. Интеграции часто строились на неформализованных паттернах, а меняющиеся детали реализации ломали совместимость.

## **2.2. Проблема совместимости и отсутствие переиспользуемых компонентов**

Ключевой проблемой периода до 2012–2014 годов была невозможность использовать одну и ту же библиотеку в разных фреймворках.

Например, HTTP-клиенты, логгеры и шаблонизаторы были жестко «привязаны» к конкретному фреймворку, а попытки переносить решения приводили к конфликтам стилей, соглашений и точек расширения.

Symfony[40] первым попытался решить эту проблему, выделив «компонентный подход[43]» (начиная с Symfony 2[41]), но без стандартизации на уровне индустрии это был лишь частичный шаг.

Разработчики других фреймворков могли использовать компоненты Symfony, но не было гарантий совместимости.

## **2.3. Отсутствие стандартизированной модели HTTP**

Одним из главных технологических ограничений было отсутствие стандартизированного представления:

- HTTP-запроса.
- HTTP-ответа.

- Атрибутов, заголовков, стримов.

Каждый фреймворк определял собственные классы:

- Symfony HttpFoundation [56].
- Zend  
Http [65] (Request / Response).
- Slim  
Http [48].
- Собственные реализации в CakePHP [1].

Это делало невозможным:

- Взаимозаменяемые middleware.
- Стандартные фильтры и обработчики.
- Переносимость кода между фреймворками.
- Унифицированные HTTP-клиенты.

Проблема была настолько глубокой, что в рассылке FIG обсуждение PSR-7[33] длилось почти 3 года. Это ещё один признак того, насколько фундаментальной была задача стандартизации.

## 2.4. Отсутствие единых правил автозагрузки и зависимостей

До релиза Composer [47] (2012) и PSR-4 [32] (2013–2014):

- Большинство библиотек подключались вручную через `require` или `include`.
- Не было декларативного управления зависимостями.

- Поставка кода происходила через ZIP-архивы или PEAR (который плохо поддерживался и был неинтуитивен).

*Autoloading* был одной из самых болезненных частей PHP-разработки:

- Фреймворки требовали строгих соглашений по структуре каталогов.
- Библиотеки не могли легко объявлять свои зависимости.
- Конфликты версий были обычным делом.

Composer [47] радикально изменил эту ситуацию, но его широкое принятие началось только после 2014–2015 — и именно этот период стал отправной точкой изменений, анализируемых далее.

## 2.5. Проблемы тестируемости и внедрения зависимостей

До принятия PSR-11 [28] (Container Interface) каждый фреймворк реализовывал собственный DI-контейнер, и ни один из них не был совместим с другим:

- Symfony DependencyInjection [52].
- Laravel Container [17].
- Zend  
ServiceManager [66].
- Pimple [39].

Отсутствие общего интерфейса:

- Усложняло создание многоразовых пакетов.
- Делало невозможным перенос middleware-компонентов.

- Тормозило развитие архитектур, основанных на инверсии управления.

Архитектура DI<sup>1</sup> была одной из ключевых болевых точек, которую индустрия смогла решить только в последние 10 лет (2015–2025), в ходе стандартизации PSR-11 [28].

---

<sup>1</sup>**DI — Dependency Injection.** Внедрение зависимостей: объект получает свои зависимости извне, а не создаёт их самостоятельно.

### **3. Развитие стандартов и архитектурных решений в экосистеме PHP (2015–2025)**

Период 2015–2025 годов стал для PHP-фреймворков временем глубокой технологической перестройки.

На протяжении предыдущего десятилетия (2005–2015) были заложены фундаментальные идеи: создание PHP-FIG [34], появление Composer [47], формирование первых PSR-стандартов [27], PSR-4 [32], PSR-7 [33], PSR-11 [28], PSR-15 [30].

Однако именно после выхода PHP 7 [35] и широкого принятия PSR-7 [33], PSR-11 [28], PSR-15 [30] начался качественный переход к современной архитектуре веб-приложений.

В этом разделе анализируются ключевые архитектурные изменения, причины их появления и влияние на экосистему.

#### **3.1. Composer как основа модернизации экосистемы (2015–2025)**

Хотя Composer [47] был выпущен в 2012 году, его массовое принятие произошло в период 2015–2017 годов.

Именно в это время большинство фреймворков и библиотек окончательно перешли на декларативное управление зависимостями.

##### **3.1.1. Причины изменения:**

До Composer [47] управление зависимостями основывалось на:

- Ручном подключении файлов через `require` или `include`.
- Распространении пакетов через ZIP-архивы или PEAR.
- Отсутствии разрешения конфликтов версий.

Это приводило к «зависимостному адду» [62] и мешало развитию фреймворков.

### **3.1.2. Принятое решение**

Composer [47] ввёл:

- файл `composer.json` [3] в качестве декларации зависимостей.
- Автозагрузчик, основанный на PSR-4 [32].
- Семантическое версионирование [44].
- Центральный репозиторий Packagist [37].

Философия Composer [47] повлияла на всю экосистему: фреймворки стали не «монолитами», а наборами компонентов, которые можно выбирать, комбинировать и обновлять независимо.

### **3.1.3. Результаты**

- Исчезла жёсткая привязка библиотек к конкретным фреймворкам.
- Появилось огромное количество независимых пакетов.
- Разработчики получили прозрачное управление зависимостями.

Composer [47] стал фактическим стандартом и сделал возможным внедрение последующих PSR.

### **3.1.4. PSR-4 и единая модель автозагрузки**

PSR-4 [32] (2014) стал ключевым переходом от хаотичных соглашений к единообразной системе.

Но именно в период 2015–2020, когда фреймворки начали массово переписывать внутренние структуры под PSR-4 [32].

### **3.1.5. Причины**

До PSR-4 [32]:

- Каждый фреймворк имел собственные правила размещения классов.
- Не было единообразных пространств имён.
- Библиотеки не могли интегрироваться между собой.

### **3.1.6. Принятое решение**

PSR-4 [32] определил:

- Строгие правила сопоставления пространств имён с каталогами.
- Автоматическую загрузку классов без ручных `include` или `require`.

### **3.1.7. Результаты**

- Библиотеки стали взаимозаменяемыми.
- Фреймворки уменьшили собственный «клей» и стали ориентироваться на компоненты.
- Composer [47] получил техническую основу для генерации стандартизированного автозагрузчика.

PSR-4 [32] стал первым шагом к «общему грамматическому строю» всей экосистемы.

## **3.2. PSR-7 и стандартизация HTTP-модели (2015)**

PSR-7 [33] (HTTP Message Interface).

Это одно из самых значимых изменений за всё существование PHP-фреймворков.

### 3.2.1. Причины

До PSR-7:

- Каждый фреймворк представлял HTTP-запрос/ответ по-своему.
- Невозможно было переносить `middleware`.
- Невозможно было использовать один и тот же HTTP-клиент или роутер между фреймворками.
- Не существовало общего интерфейса потоков `streams`, `cookies`, `headers`.

### 3.2.2. Принятое решение

PSR-7 создал единый интерфейс для:

- `Request`.
- `Response`.
- `Stream`.
- `UploadedFile`.
- `URI`.

Это впервые позволило библиотекам работать независимо от фреймворка.

### 3.2.3. Результаты

- Появление Slim 3 [49], Zend Diactoros [63], Guzzle PSR-7 clients [7].
- Появление кросс-фреймворковых `middleware`.
- Symfony добавил PSR-7-бриджи [58].
- Laravel адаптировал совместимость на уровне интеграций [15].

PSR-7 [33] стал фундаментом для PSR-15 [30] и новой культуры `middleware` в PHP.

### 3.3. PSR-11 и унификация DI-контейнеров (2017)

Ещё одной ключевой проблемой PHP-фреймворков была несовместимость контейнеров зависимостей.

#### 3.3.1. Причины

До PSR-11:

- Каждый фреймворк имел собственный DI.
- Пакеты не могли объявлять зависимости абстрактно.
- Невозможно было использовать библиотеку, требующую объект из контейнера другого фреймворка.

Контейнеры Laravel, Symfony, Zend имели разные API.

#### 3.3.2. Принятое решение

PSR-11 ввёл два интерфейса:

- ContainerInterface .
- NotFoundExceptionInterface .

Разработчики библиотек получили способ запрашивать зависимости без привязки к конкретному фреймворку.

#### 3.3.3. Результаты

- DI стал общим архитектурным механизмом, а не «особенностью» конкретного фреймворка.
- Middleware -компоненты стали переносимыми.
- Снизилась фрагментация библиотек.

PSR-11 [28] стал фундаментом для PSR-15 [30] и новой культуры middleware в PHP.

## 3.4. PSR-15 и middleware-архитектура (2017–2018)

После PSR-7 [33] стало возможно стандартизировать поведение обработки запросов. Это позволило создать цепочки `middleware`.

### 3.4.1. Причины

Лишь некоторые фреймворки (Slim [50], Zend Expressive [11]) имели развитую `middleware`-модель. Symfony использовал HttpKernel, Laravel использовал фильтры и `middleware`, но их интерфейсы были несовместимы.

### 3.4.2. Принятое решение

PSR-15 [30] определил:

- `MiddlewareInterface` .
- `RequestHandlerInterface` .

Это дало PHP ту же модель, что и:

- Rack (Ruby).
- WSGI (Python).
- Connect (Node.js).

### 3.4.3. Результаты

- Expressive / Mezzio (Zend → Laminas) стал первым PSR-15-first фреймворком [11].
- Slim 4 полностью перешёл на PSR-15 [50].
- Symfony HttpKernel получил адаптеры [57].
- Laravel сохранил собственный контракт, но стал совместим через адаптеры [15].

Middleware стала центральной архитектурной единицей PHP-приложений.

### 3.5. Влияние PHP 7 и PHP 8 на архитектуру фреймворков

#### 3.5.1. PHP 7 (2015)

- Увеличение производительности в 2–3 раза.
- Строгая модель ошибок (движение от предупреждений к исключениям).
- Scalar type hints, return types.

#### 3.5.2. PHP 8 (2020)

- Union types.
- Attributes.
- Match.
- JIT.
- Улучшенная типобезопасность.

#### 3.5.3. Результаты для фреймворков

- Symfony 3/4 [52] переписали DI-контейнер под строгую типизацию.
- Laravel [12] стал массово переходить к типизированным сигнатурам.
- Появилась культура строгих DTO, Value Objects, Immutable объектов.
- Фреймворки сократили магию и усилили контрактность API.

Типизация стала центральным архитектурным трендом 2020-х годов.

## 3.6. Symfony: эволюция компонентной модели

Symfony стал главным драйвером стандартизации.

### 3.6.1. Основные изменения

- Переход от «полного фреймворка» к компонентам [43] (HttpFoundation, EventDispatcher, Console, Routing).
- Адаптация архитектуры под PSR-7/PSR-11 [58, 28].
- Внедрение autowiring и автоконфигурации.
- Появление Symfony Flex как современного пакета приложений.

### 3.6.2. Результаты

Большинство фреймворков (включая Laravel [12]) стали использовать компоненты Symfony либо идеологию Symfony:

- HttpKernel → архитектурный эталон.
- EventDispatcher → основа для гибких систем расширения.

Symfony стал «стандартной библиотекой» для PHP вне ядра языка.

## 3.7. Laravel: эволюция DX и влияние стандартов

Laravel [12] (2011) стал главным популяризатором:

- Единообразных API.
- Конвенций.
- Быстрой разработки (DX-first).

В период 2015–2025:

### **3.7.1. Основные изменения**

- Переход на PSR-4, Composer и частично PSR-11.
- Внедрение middleware-модели совместимой с PSR-15.
- Адаптация к PHP 7/8 (тиปизация, атрибуты).
- Появление Horizon, Octane, Sail, Pint — инфраструктурных компонентов.
- Более строгая структура маршрутизации и DI.

### **3.7.2. Результаты**

Laravel [12] стал «массовым воплощением» стандартизированной экосистемы PHP:

- Он интегрирует PSR-совместимые библиотеки.
- Служит входной точкой для новых разработчиков.
- Влияет на индустриальные практики (DX, миграции, Eloquent как ORM-эталон).

## 4. Отвергнутые и отложенные изменения

Как и в случае с другими зрелыми технологиями, процесс стандартизации PHP-фреймворков сопровождался многочисленными предложениями, не вошедшими в финальные версии PSR-стандартов или отложенными на неопределённый срок.

Эти обсуждения позволяют понять, какие архитектурные решения были сочтены слишком узкими, слишком рискованными или концептуально несовместимыми с направлением развития экосистемы.

Ниже рассмотрены наиболее значимые инициативы, не приведшие к формированию стандарта.

### 4.1. Несостоявшийся PSR для маршрутизации (Router PSR)

Одним из регулярно поднимаемых предложений в рассылках PHP-FIG [25] начиная с 2015 года было создание стандарта для роутинга HTTP-запросов.

#### 4.1.1. Причины появления инициативы

Фреймворки использовали разные архитектуры маршрутов:

- Laravel применяет декларативную синтаксическую модель (fluent API) [16].
- Symfony использует аннотации, YAML и PHP-конфигурации [59].
- Slim и Mezzio строят маршрутизацию вокруг middleware [51].
- FastRoute — чисто функциональная библиотека без привязки к фреймворку [38].

Отсутствие общего интерфейса приводило к невозможности создать:

- Единый набор middleware для маршрутизации.

- Универсальные инструменты тестирования маршрутов.
- Переносимые роутинговые DSL.

#### 4.1.2. Причины отказа

В обсуждениях FIG (2016–2018) было выявлено несколько проблем:

##### Различие моделей маршрутизации

Одни фреймворки используют controller-based архитектуру (Laravel [16], Symfony [59]), другие (например Slim [51], Mezzio) используют middleware-based архитектуру. Приведение этих моделей к единому интерфейсу оказалось практически невозможным.

##### Слишком высокий уровень абстракции

Любой интерфейс становился либо:

- Слишком низкоуровневым (и не решал задачи).
- Или слишком высоким.

##### Стандарт рисковал закрепить устаревший подход

FIG избегает «Навязывать» архитектурные решения, чтобы не мешать инновациям.

#### Итог

Инициатива была закрыта как слишком сложная и недостаточно универсальная. Разработчики договорились, что роутинг останется частью каждого фреймворка, а интеграции будут строиться через PSR-7 [33] и PSR-15 [30].

## **4.2. Попытка создать PSR для ORM и абстракции работы с базами данных**

Регулярно обсуждалось создание стандарта для доступа к данным аналога JDBC для Java. Попытки разработать единый интерфейс для ORM поднимались в рассылке PHP-FIG с 2014 по 2020 год [22].

### **Мотивация**

- Множество несовместимых ORM: Doctrine ORM [4], Eloquent ORM [13], Propel [45], RedBeanPHP [6].
- Попытки создать стандартный QueryBuilder или EntityManager обсуждались с 2014 по 2020 год.

### **Причины отказа**

- Слишком разные философии данных.
  - Doctrine — ориентирована на DDD и Unit of Work.
  - Eloquent — ActiveRecord и stateful-модель.
  - Propel — XML-генерация моделей.
  - RedBean — динамические схемы.
- Разработчики ORM не готовы к унификации.  
Doctrine имеет строгую архитектуру, Laravel — гибкую, Eloquent использует «магические» свойства.
- Большая часть индустрии предпочитает свободу.  
FIG не хотел повторить опыт Java EE, где стандарты замедляли эволюцию ORM.

### **Итог**

PSR для ORM был признан нежизнеспособным.

## 4.3. PSR-14 (Event Dispatcher): стандарт, который останется частичным

PSR-14 [29] был принят в 2019 году, но его разработка сопровождалась огромным количеством противоречий в рассылках PHP-FIG [23].

### Проблемы

- Фреймворки используют разные event models.
  - Symfony: синхронный диспетчер событий, на основе объектных слушателей [53].
  - Laravel: разделение событий и слушателей + очередь + broadcast [14].
  - Zend Framework: агрегаторы событий [9].

Найти общую модель оказалось крайне трудно.

- Глубокие различия в семантике слушателей.

Например, прекращение обработки событий отсутствует во многих системах.

- Слишком узкий охват стандарта.

PSR-14 определяет слишком абстрактный интерфейс:

```
interface EventDispatcherInterface {  
    public function dispatch(object
```

### Итог

PSR-14 принят, но в экосистеме остаётся «вторичным» стандартом. Существенная часть сообщества его игнорирует.

## 4.4. Попытка создания PSR для валидаторов и форм

Это обсуждение велось эпизодически с 2016 по 2021 годы [26].

## Мотивация

- Symfony Form + Validator имеют сложную, но зрелую модель [54, 60].
- Laravel Validation — декларативная модель со строковыми правилами [18].
- Respect/Validation — функциональная библиотека [46].

Наличие множества несовместимых подходов порождало предложение создать переносимый стандарт.

## Причины отказа

- Слишком разный уровень абстракции.
- Слишком разный DSL.

Сравните:

- 'required|min:6' (Laravel),
- new Length(['min'=>6]) (Symfony),
- v::stringType()->length(6) (Respect).

- Нежелание ограничивать инновации.

Фреймворки активно экспериментируют со схемами данных.

## Итог

PSR для валидаторов и форм был признан нежизнеспособным.

## 4.5. Споры вокруг PSR-18 (HTTP Client)

PSR-18 [31] был принят в 2019 году, но с серьёзными дискуссиями [24].

## Причины споров

- Разные модели ошибок (исключения vs error responses).
- Различия в реализации Guzzle [8], HTTPPlug [36], Symfony HttpClient [55].
- Споры о синхронности vs асинхронности.

Некоторые разработчики хотели:

- Асинхронный интерфейс (в стиле Promise).
- Unified Streaming API.
- Поддержку cancellation.

FIG решил ограничиться минимальным синхронным интерфейсом, что вызвало критику, но позволило стандартизировать общие ожидания.

## Итог

Стандарт принят как «минимальный необходимый», остальные аспекты deliberately не стандартизированы.

## 4.6. Предложение о расширении PSR-11 (унифицированная конфигурация контейнера)

Иногда обсуждался стандарт для [21]:

- Регистрации сервисов.
- Определения параметров.
- Описания factories.

Это сделало бы DI-контейнеры полностью совместимыми.

## **Причины отказа**

- Разные модели конфигурации контейнера.
  - Symfony использует YAML/PHP/XML [52].
  - Laravel использует bindings и closures [17].
  - Laminas: Массивы-конфигурации [10].

Невозможно привести к общему знаменателю.

- Опасение закрепления устаревших подходов Стандартизация могла заморозить развитие DI.

## **Итог**

PSR-11 [28] остался минималистичным.

# Список литературы

- [1] CakePHP Team. CakePHP Request and Response Objects.— URL: <https://book.cakephp.org/2/en/controllers/request-response.html> (дата обращения: 2025-12-07).
- [2] CakePHP Team. CakePHP 1.0 Release.— 2006.— May.— URL: <https://bakery.cakephp.org/articles/view/cakephp-1-0-released> (дата обращения: 2025-12-07). Archived announcement of CakePHP 1.0 release. Development started in 2005.
- [3] Composer Team. The composer.json Schema.— 2012.— URL: <https://getcomposer.org/doc/04-schema.md> (дата обращения: 2025-12-07). Официальная документация по схеме composer.json.
- [4] Doctrine Project. Doctrine ORM.— URL: <https://www.doctrine-project.org/projects/orm.html> (дата обращения: 2025-02-01).
- [5] EllisLab. CodeIgniter 1.0 Release.— URL: <https://codeigniter.com/> (дата обращения: 2024-11-01). Initial release of CodeIgniter framework. Archived information available.
- [6] Gabor de Mooij. RedBeanPHP.— URL: <https://redbeanphp.com/> (дата обращения: 2025-02-01).
- [7] Guzzle Team. Guzzle 6: PSR-7 Compatible HTTP Client.— URL: <https://docs.guzzlephp.org/en/stable/> (дата обращения: 2025-02-01).
- [8] Guzzle Team. Guzzle HTTP Client.— URL: <https://docs.guzzlephp.org/en/stable/> (дата обращения: 2025-02-01).
- [9] Laminas Project. Laminas EventManager.— URL: <https://docs.laminas.dev/laminas-eventmanager/> (дата обращения: 2025-02-01).

- [10] Laminas Project. Laminas ServiceManager Configuration. — URL: <https://docs.laminas.dev/laminas-servicemanager/v4/configuring-the-service-manager/> (дата обращения: 2025-02-01).
- [11] Laminas Project. Mezzio (formerly Zend Expressive) — PSR-15 Middleware Framework. — URL: <https://docs.mezzio.dev/mezzio/> (дата обращения: 2025-02-01).
- [12] Laravel Contributors. Laravel Documentation. — URL: <https://laravel.com/docs> (дата обращения: 2024-11-01). Official Laravel documentation, including version history.
- [13] Laravel Team. Eloquent ORM. — URL: <https://laravel.com/docs/eloquent> (дата обращения: 2025-02-01).
- [14] Laravel Team. Laravel Events and Listeners. — URL: <https://laravel.com/docs/events> (дата обращения: 2025-02-01).
- [15] Laravel Team. Laravel PSR-7 Compatibility via Symfony Bridge. — URL: <https://laravel.com/docs/5.5/requests#psr7-requests> (дата обращения: 2025-02-01).
- [16] Laravel Team. Laravel Routing. — URL: <https://laravel.com/docs/routing> (дата обращения: 2025-02-01).
- [17] Laravel Team. Laravel Service Container. — URL: <https://laravel.com/docs/master/container> (дата обращения: 2025-12-07). Official documentation for the Laravel Service Container.
- [18] Laravel Team. Laravel Validation. — URL: <https://laravel.com/docs/validation> (дата обращения: 2025-02-01).
- [19] Otwell Taylor. Laracon. — URL: <https://laracon.net> (дата обращения: 2025-12-07).
- [20] PHP Documentation Group. PHP. — URL: <https://www.php.net/manual/en/> (дата обращения: 2025-12-07). Version 8.4.

- [21] PHP-FIG. PHP-FIG Mailing List: Discussions About Expanding PSR-11. — URL: <https://groups.google.com/g/php-fig/search?q=PSR-11+configuration> (дата обращения: 2025-02-01).
- [22] PHP-FIG. PHP-FIG Mailing List: ORM Standardization Discussions. — URL: <https://groups.google.com/g/php-fig/search?q=ORM> (дата обращения: 2025-02-01).
- [23] PHP-FIG. PHP-FIG Mailing List: PSR-14 Event Dispatcher Discussions. — URL: <https://groups.google.com/g/php-fig/search?q=PSR-14> (дата обращения: 2025-02-01).
- [24] PHP-FIG. PHP-FIG Mailing List: PSR-18 Discussions. — URL: <https://groups.google.com/g/php-fig/search?q=PSR-18> (дата обращения: 2025-02-01).
- [25] PHP-FIG. PHP-FIG Mailing List: Routing PSR Discussions. — URL: <https://groups.google.com/g/php-fig/search?q=routing> (дата обращения: 2025-02-01).
- [26] PHP-FIG. PHP-FIG Mailing List: Validator and Form Standard Discussions. — URL: <https://groups.google.com/g/php-fig/search?q=validator> (дата обращения: 2025-02-01).
- [27] PHP-FIG. PSR-0: Autoloading Standard. — URL: <https://www.php-fig.org/psr/psr-0/> (дата обращения: 2024-11-01).
- [28] PHP-FIG. PSR-11: Container Interface. — URL: <https://www.php-fig.org/psr/psr-11/> (дата обращения: 2025-12-07).
- [29] PHP-FIG. PSR-14: Event Dispatcher. — URL: <https://www.php-fig.org/psr/psr-14/> (дата обращения: 2025-02-01).
- [30] PHP-FIG. PSR-15: HTTP Server Request Handlers. — URL: <https://www.php-fig.org/psr/psr-15/> (дата обращения: 2025-12-07).
- [31] PHP-FIG. PSR-18: HTTP Client. — URL: <https://www.php-fig.org/psr/psr-18/> (дата обращения: 2025-02-01).

- [32] PHP-FIG. PSR-4: Autoloading Standard. — URL: <https://www.php-fig.org/psr/psr-4/> (дата обращения: 2024-11-01). Improved autoloading standard that replaced PSR-0.
- [33] PHP-FIG. PSR-7: HTTP Message Interface. — URL: <https://www.php-fig.org/psr/psr-7/> (дата обращения: 2025-12-07).
- [34] PHP Framework Interop Group. PHP-FIG Charter and Bylaws. — URL: <https://www.php-fig.org/bylaws/> (дата обращения: 2025-12-07). Original charter document. Updated versions exist.
- [35] PHP Group. PHP 7.0.0 Release Announcement. — URL: [https://www.php.net/releases/7\\_0\\_0.php](https://www.php.net/releases/7_0_0.php) (дата обращения: 2024-11-01).
- [36] PHP HTTP Group. HTTPPlug: HTTP Client Abstraction. — URL: <https://httpplug.io/> (дата обращения: 2025-02-01).
- [37] Packagist Team. Packagist: The PHP Package Repository. — 2011. — URL: <https://packagist.org/> (дата обращения: 2025-12-07). Основной репозиторий пакетов для Composer.
- [38] Popov Nikita. FastRoute: Fast Request Router for PHP. — URL: <https://github.com/nikic/FastRoute> (дата обращения: 2025-02-01).
- [39] Potencier Fabien. Pimple: A Simple Service Container for PHP. — URL: <https://github.com/silexphp/Pimple> (дата обращения: 2025-12-07).
- [40] Potencier Fabien. Symfony 1.0 Release Announcement. — URL: <https://symfony.com/blog/symfony-1-0-released> (дата обращения: 2024-11-01). Official announcement of Symfony 1.0. The project started in 2005.
- [41] Potencier Fabien. Symfony 2.0 Release Announcement. — URL: <https://symfony.com/releases/2.0> (дата обращения: 2025-12-07). Official announcement of Symfony 2.0. The project started in 2007.

- [42] Potencier Fabien. Symfony 6: The Future of PHP Frameworks // SymfonyCon. — URL: <https://live.symfony.com> (дата обращения: 2025-12-07).
- [43] Potencier Fabien. Symfony2 Components: Standalone Libraries. — URL: <https://symfony.com/blog/symfony2-components-as-standalone-packages> (дата обращения: 2025-12-07). Introduction of Symfony's component-based architecture.
- [44] Preston-Werner Tom. Semantic Versioning 2.0.0. — 2010. — URL: <https://semver.org/> (дата обращения: 2025-12-07). Официальная спецификация семантического версионирования.
- [45] Propel Team. Propel ORM. — URL: <https://propelorm.org/> (дата обращения: 2025-02-01).
- [46] Respect Project. Respect. — URL: <https://respect-validation.readthedocs.io/> (дата обращения: 2025-02-01).
- [47] Seldaek Jordi B., Contributors. Composer: Dependency Manager for PHP. — URL: <https://getcomposer.org/> (дата обращения: 2025-12-07). Initial release announcement and ongoing documentation.
- [48] Slim Framework Team. Slim Framework 2.x HTTP Request Object (Slim). — URL: <https://github.com/slimphp/Slim-Documentation/blob/master/docs/objects/request.md> (дата обращения: 2025-12-07).
- [49] Slim Framework Team. Slim Framework 3.0 Release. — URL: <https://www.slimframework.com/docs/v3/> (дата обращения: 2025-02-01).
- [50] Slim Framework Team. Slim Framework 4 — PSR-15 Middleware Architecture. — URL: <https://www.slimframework.com/docs/v4/> (дата обращения: 2025-02-01).
- [51] Slim Framework Team. Slim Routing. — URL: <https://www.slimframework.com/docs/v4/objects/routing.html> (дата обращения: 2025-02-01).

- [52] Symfony Project. Symfony DependencyInjection Component.— URL: [https://symfony.com/doc/current/components/dependency\\_injection.html](https://symfony.com/doc/current/components/dependency_injection.html) (дата обращения: 2025-12-07). Official documentation for the Symfony DependencyInjection component.
- [53] Symfony Project. Symfony EventDispatcher Component.— URL: [https://symfony.com/doc/current/components/event\\_dispatcher.html](https://symfony.com/doc/current/components/event_dispatcher.html) (дата обращения: 2025-02-01).
- [54] Symfony Project. Symfony Form Component.— URL: <https://symfony.com/doc/current/forms.html> (дата обращения: 2025-02-01).
- [55] Symfony Project. Symfony HttpClient Component.— URL: [https://symfony.com/doc/current/http\\_client.html](https://symfony.com/doc/current/http_client.html) (дата обращения: 2025-02-01).
- [56] Symfony Project. Symfony HttpFoundation Component.— URL: [https://symfony.com/doc/current/components/http\\_foundation.html](https://symfony.com/doc/current/components/http_foundation.html) (дата обращения: 2025-12-07).
- [57] Symfony Project. Symfony PSR-15 HTTP Kernel Adapter.— URL: <https://github.com/symfony/psr-http-message-bridge> (дата обращения: 2025-02-01).
- [58] Symfony Project. Symfony PSR-7 Bridge.— URL: <https://github.com/symfony/psr-http-message-bridge> (дата обращения: 2025-02-01).
- [59] Symfony Project. Symfony Routing Component.— URL: <https://symfony.com/doc/current/routing.html> (дата обращения: 2025-02-01).
- [60] Symfony Project. Symfony Validator Component.— URL: <https://symfony.com/doc/current/validation.html> (дата обращения: 2025-02-01).

- [61] Usage statistics of PHP for websites, W3Techs. — URL: <https://w3techs.com/technologies/details/pl-php> (дата обращения: 2025-11-01). Data for websites with a known server-side programming language. Percentage as of 1 November 2025.
- [62] Wikipedia contributors. Dependency hell. — 2004. — URL: [https://en.wikipedia.org/wiki/Dependency\\_hell](https://en.wikipedia.org/wiki/Dependency_hell) (дата обращения: 2025-12-07). Термин впервые задокументирован в 2004 году.
- [63] Zend Technologies. Zend Diactoros: PSR-7 HTTP Message Implementations. — URL: <https://docs.laminas.dev/laminas-diactoros/> (дата обращения: 2025-02-01).
- [64] Zend Technologies. Zend Framework: Open Source PHP Framework. — URL: <https://framework.zend.com/> (дата обращения: 2025-12-07).
- [65] Zend Technologies. ZendComponent (Zend Framework 1.x). — URL: <https://framework.zend.com/manual/1.12/en zend.http.html> (дата обращения: 2025-12-07).
- [66] Zend Technologies. ZendComponent. — URL: <https://packagist.org/packages/zendframework/zend-servicemanager#2.0.3> (дата обращения: 2025-12-07).