Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.М71-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

1. Введение

PHP играет важную роль в веб-разработке на протяжении почти трёх десятилетий, оставаясь одним из наиболее распространённых серверных языков.

Согласно статистике W3Techs по состоянию на 1 ноября 2025 года, PHP используется в 72.9% веб-сайтов, для которых известен серверный язык [61].

Широкая доступность хостинга и относительно низкий порог входа сделали PHP [20] основой для большого числа систем управления контентом и прикладных веб-решений.

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)[39] 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 тестируемость.

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 [21], появлением Composer [47], унификацией HTTP-модели[25], распространением принципов DI [30] и архитектурой middleware [31].

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, стремительный 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[39] и Laravel[**?**], дискуссии GitHub о стандартах Composer и PSR, а также выступления разработчиков на конференциях, таких как SymfonyCon[43] 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[39], Zend Framework[63], 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[39] использовал собственный автозагрузчик и собственные соглашения об именовании классов.
- Zend Framework[63] 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[39] первым попытался решить эту проблему, выделив «компонентный подход[42]» (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 [52].
- Zend
  Http [64] (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[25] 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 [23] (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 [30] (Container Interface) каждый фреймворк реализовывал собственный DI-контейнер, и ни один из них не был совместим с другим:

- Symfony DependencyInjection [56].
- Laravel Container [15].
- Zend
  ServiceManager [65].
- Pimple [40].

Отсутствие общего интерфейса:

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

- Тормозило развитие архитектур, основанных на инверсии управления.

Архитектура DI[1] была одной из ключевых болевых точек, которую индустрия смогла решить только в последние 10 лет (2015–2025), в ходе стандартизации PSR-11 [30].

[1]**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 [21], PSR-4 [23], PSR-7 [25], PSR-11 [30], PSR-15 [31].

Однако именно после выхода PHP 7 [35] и широкого принятия PSR-7 [25], PSR-11 [30], PSR-15 [31] 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 аду» [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 [23].
- 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 [23] (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 [23].

3.1.5. Причины

До PSR-4 [23]:

- 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 [23] 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 [23] стал первым шагом к «общему грамматическому строю» всей экосистемы.

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

PSR-7 [25] (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 [66], Guzzle PSR-7 clients [8].
- Появление кросс-фреймворковых `middleware`.
- Symfony добавил PSR-7-бриджи [59].
- Laravel адаптировал совместимость на уровне интеграций [18].

PSR-7 [25] стал фундаментом для PSR-15 [31] 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 [30] стал фундаментом для PSR-15 [31] и новой культуры `middleware` в PHP.

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

После PSR-7 [25] 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 [31] 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 [58].
- Laravel сохранил собственный контракт, но стал совместим через адаптеры [18].

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 [56] 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 [42] (HttpFoundation, EventDispatcher, Console, Routing).
- Адаптация архитектуры под PSR-7/PSR-11 [59, 30].
- 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 (типизация, 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 [24] 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) [14].
- Symfony использует аннотации, YAML и PHP-конфигурации [57].
- 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 [14], Symfony [57]), 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 [25] и PSR-15 [31].

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 [32] был принят в 2019 году, но его разработка сопровождалась огромным количеством противоречий в рассылках PHP-FIG [26].

Проблемы

- Фреймворки используют разные event models.
  - Symfony: синхронный диспетчер событий, на основе объектных слушателей [55].
  - Laravel: разделение событий и слушателей + очередь + broadcast [17].
  - 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 сообществ его игнорирует.

4.4. Попытка создания PSR для валидаторов и форм

Это обсуждение велось эпизодически с 2016 по 2021 годы [27].

Мотивация

- Symfony Form + Validator имеют сложную, но зрелую модель [53, 54].
- Laravel Validation — декларативная модель со строковыми правилами [16].
- 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 [33] был принят в 2019 году, но с серьёзными дискуссиями [29].

Причины споров

- Разные модели ошибок (исключения vs error responses).
- Различия в реализации Guzzle [7], HTTPlug [36], Symfony HttpClient [60].
- 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 [28]:

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

Это сделало бы DI-контейнеры полностью совместимыми.

Причины отказа

- Разные модели конфигурации контейнера.
  - Symfony использует YAML/PHP/XML [56].
  - Laravel использует bindings и closures [15].
  - Laminas: Массивы-конфигурации [10].

  Невозможно привести к общему знаменателю.

- Опасение закрепления устаревших подходов Стандартизация могла заморозить развитие DI.

Итог

PSR-11 [30] остался минималистичным.

5. Заключение

За период 2015–2025 годов экосистема PHP-фреймворков претерпела глубокую трансформацию, в результате которой веб-разработка на PHP фактически перешла от фрагментированного набора несовместимых архитектур к единому технологическому пространству, основанному на стандартах и переносимых компонентах.

Ключевую роль в этом процессе сыграли стандарты PHP-FIG (PSR-4, PSR-7, PSR-11, PSR-15), Composer и переход языка к строгой модели типизации в версиях PHP 7 и 8.

Эти изменения оказали системное влияние как на внутренние архитектуры фреймворков, так и на методологию разработки приложений.

Принятие PSR-4 и Composer стало отправной точкой для унификации структуры проектов, что позволило разрушить жёсткие границы между экосистемами разных фреймворков.

Стандартизация HTTP-модели (PSR-7) и middleware-контрактов (PSR-15) сформировала общий слой interoperability, дав толчок развитию кросс-фреймворковых библиотек и middleware-стека.

PSR-11 обеспечил совместимость контейнеров зависимостей, благодаря чему переносимость сервисов и компонентов значительно увеличилась.

Совокупно эти стандарты сформировали основу современной PHP-инфраструктуры, в которой логика приложения теперь отделена от конкретного фреймворка.

Наряду со стандартами FIG значительное влияние оказали изменения в самом языке: строгая типизация, исключительная модель ошибок, увеличение производительности PHP 7, а также атрибуты и JIT-компиляция в PHP 8.

Эти изменения стимулировали фреймворки к переработке внутренних механизмов и улучшению архитектурных практик.

Symfony в этот период окончательно утвердился как компонентный фреймворк, определяющий технические ориентиры для всей PHP-индустрии; Laravel, напротив, стал укреплять свою позицию как высо-

коуровневый фреймворк, фокусирующийся на удобстве разработки и интеграции с современными DevOps-практиками.

Современные версии обоих фреймворков демонстрируют высокую степень согласованности с PSR-стандартами и активно используют возможности нового PHP.

Особое значение имеют технологии и стандарты, которые не были приняты.

Несостоявшийся PSR для роутинга, отсутствие стандарта для ORM, частичная применимость PSR-14 — всё это демонстрирует, что стандартизация не может и не должна охватывать все аспекты PHP-экосистемы.

Слишком разнообразные архитектуры и различные философии разработки делают некоторые стандарты непрактичными.

В этом смысле отказ от стандартизации отдельных областей оказался не менее ценным, чем успешные инициативы: он позволил индустрии сохранить гибкость и конкурентное разнообразие.

В целом, архитектурная эволюция PHP-фреймворков в 2015–2025 годах может быть охарактеризована как движение к стандартизации на ключевых уровнях абстракции при сохранении свободы в реализации высокоуровневых концепций.

Этот период стал временем консолидации и зрелости: фреймворки перестали быть самостоятельными островами и стали частями единой экосистемы, в которой интерфейсы важнее реализаций, а архитектура — важнее конкретных технологий.

PHP, часто считавшийся устаревающим, за это десятилетие подтвердил свою жизнеспособность, адаптировавшись к современным требованиям производительности, типобезопасности и модульности.

Эта трансформация стала возможной благодаря открытому процессу разработки, публичным обсуждениям в PHP-FIG и активной роли сообществ Symfony и Laravel.

Таким образом, изменения 2015–2025 годов можно считать одним из самых успешных этапов в истории PHP: язык и его фреймворки не только сохранили позиции, но и стали примером того, как открытая

стандартизация и согласованные архитектурные решения способны преобразовать зрелую технологию в соответствующую требованиям нового времени.

Список литературы

[1] CakePHP Team. CakePHP Request and Response Objects. — 2005. — URL: https://book.cakephp.org/2/en/controllers/request-response.html (дата обращения: 2025-11-17).

[2] CakePHP Team. CakePHP 1.0 Release. — 2006. — May. — URL: https://bakery.cakephp.org/articles/view/cakephp-1-0-released (дата обращения: 2025-11-15). 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: 2025-11-26). Официальная документация по схеме composer.json.

[4] Doctrine Project. Doctrine ORM. — 2006. — URL: https://www.doctrine-project.org/projects/orm.html (дата обращения: 2025-12-07).

[5] EllisLab. CodeIgniter 1.0 Release. — 2006. — URL: https://codeigniter.com/ (дата обращения: 2024-11-15). Initial release of CodeIgniter framework. Archived information available.

[6] Gabor de Mooij. RedBeanPHP. — 2009. — URL: https://redbeanphp.com/ (дата обращения: 2025-12-07).

[7] Guzzle Team. Guzzle HTTP Client. — 2011. — URL: https://docs.guzzlephp.org/en/stable/ (дата обращения: 2025-12-07).

[8] Guzzle Team. Guzzle 6: PSR-7 Compatible HTTP Client. — 2015. — URL: https://docs.guzzlephp.org/en/stable/ (дата обращения: 2025-11-26).

[9] Laminas Project. Laminas EventManager. — 2012. — URL: https://docs.laminas.dev/laminas-eventmanager/ (дата обращения: 2025-12-05).

[10] Laminas Project. Laminas ServiceManager Configuration. — 2012. — URL: https://docs.laminas.dev/laminas-servicemanager/v4/configuring-the-service-manager/ (дата обращения: 2025-12-05).

[11] Laminas Project. Mezzio (formerly Zend Expressive) — PSR-15 Middleware Framework. — 2015. — URL: https://docs.mezzio.dev/mezzio/ (дата обращения: 2025-11-26).

[12] Laravel Contributors. Laravel Documentation. — 2024. — URL: https://laravel.com/docs (дата обращения: 2024-11-26). Official Laravel documentation, including version history.

[13] Laravel Team. Eloquent ORM. — 2011. — URL: https://laravel.com/docs/eloquent (дата обращения: 2025-12-05).

[14] Laravel Team. Laravel Routing. — 2011. — URL: https://laravel.com/docs/routing (дата обращения: 2025-02-01).

[15] Laravel Team. Laravel Service Container. — 2013. — URL: https://laravel.com/docs/master/container (дата обращения: 2025-11-17). Official documentation for the Laravel Service Container.

[16] Laravel Team. Laravel Validation. — 2013. — URL: https://laravel.com/docs/validation (дата обращения: 2025-12-05).

[17] Laravel Team. Laravel Events and Listeners. — 2014. — URL: https://laravel.com/docs/events (дата обращения: 2025-12-05).

[18] Laravel Team. Laravel PSR-7 Compatibility via Symfony Bridge. — 2017. — URL: https://laravel.com/docs/5.5/requests#psr7-requests (дата обращения: 2025-11-27).

[19] Otwell Taylor. Laracon. — URL: https://laracon.net (дата обращения: 2025-12-07).

[20] PHP Documentation Group. PHP. — 2025. — URL: https://www.php.net/manual/en/ (дата обращения: 2025-11-15). Version 8.4.

[21] PHP-FIG. PSR-0: Autoloading Standard. — 2009. — URL: https://www.php-fig.org/psr/psr-0/ (дата обращения: 2024-11-26).

[22] PHP-FIG. PHP-FIG Mailing List: ORM Standardization Discussions. — 2014. — URL: https://groups.google.com/g/php-fig/search?q=ORM (дата обращения: 2025-12-01).

[23] PHP-FIG. PSR-4: Autoloading Standard. — 2014. — URL: https://www.php-fig.org/psr/psr-4/ (дата обращения: 2024-11-26). Improved autoloading standard that replaced PSR-0.

[24] PHP-FIG. PHP-FIG Mailing List: Routing PSR Discussions. — 2015. — URL: https://groups.google.com/g/php-fig/search?q=routing (дата обращения: 2025-12-01).

[25] PHP-FIG. PSR-7: HTTP Message Interface. — 2015. — URL: https://www.php-fig.org/psr/psr-7/ (дата обращения: 2025-11-26).

[26] PHP-FIG. PHP-FIG Mailing List: PSR-14 Event Dispatcher Discussions. — 2016. — URL: https://groups.google.com/g/php-fig/search?q=PSR-14 (дата обращения: 2025-12-01).

[27] PHP-FIG. PHP-FIG Mailing List: Validator and Form Standard Discussions. — 2016. — URL: https://groups.google.com/g/php-fig/search?q=validator (дата обращения: 2025-12-01).

[28] PHP-FIG. PHP-FIG Mailing List: Discussions About Expanding PSR-11. — 2017. — URL: https://groups.google.com/g/php-fig/search?q=PSR-11+configuration (дата обращения: 2025-12-01).

[29] PHP-FIG. PHP-FIG Mailing List: PSR-18 Discussions. — 2017. — URL: https://groups.google.com/g/php-fig/search?q=PSR-18 (дата обращения: 2025-12-01).

[30] PHP-FIG. PSR-11: Container Interface. — 2017. — URL: https://www.php-fig.org/psr/psr-11/ (дата обращения: 2025-11-26).

[31] PHP-FIG. PSR-15: HTTP Server Request Handlers. — 2018. — URL: https://www.php-fig.org/psr/psr-15/ (дата обращения: 2025-11-26).

[32] PHP-FIG. PSR-14: Event Dispatcher. — 2019. — URL: https://www.php-fig.org/psr/psr-14/ (дата обращения: 2025-11-26).

[33] PHP-FIG. PSR-18: HTTP Client. — 2019. — URL: https://www.php-fig.org/psr/psr-18/ (дата обращения: 2025-11-26).

[34] PHP Framework Interop Group. PHP-FIG Charter and Bylaws. — 2009. — URL: https://www.php-fig.org/bylaws/ (дата обращения: 2025-11-26). Original charter document. Updated versions exist.

[35] PHP Group. PHP 7.0.0 Release Announcement. — 2015. — URL: https://www.php.net/releases/7_0_0.php (дата обращения: 2024-11-20).

[36] PHP HTTP Group. HTTPlug: HTTP Client Abstraction. — 2016. — URL: https://httplug.io/ (дата обращения: 2025-12-07).

[37] Packagist Team. Packagist: The PHP Package Repository. — 2011. — URL: https://packagist.org/ (дата обращения: 2025-11-18). Основной репозиторий пакетов для Composer.

[38] Popov Nikita. FastRoute: Fast Request Router for PHP. — 2012. — URL: https://github.com/nikic/FastRoute (дата обращения: 2025-12-02).

[39] Potencier Fabien. Symfony 1.0 Release Announcement. — 2007. — URL: https://symfony.com/blog/symfony-1-0-released (дата обращения: 2024-11-15). Official announcement of Symfony 1.0. The project started in 2005.

[40] Potencier Fabien. Pimple: A Simple Service Container for PHP. — 2010. — URL: https://github.com/silexphp/Pimple (дата обращения: 2025-12-07).

[41] Potencier Fabien. Symfony 2.0 Release Announcement. — 2011. — URL: https://symfony.com/releases/2.0 (дата обращения: 2025-11-15). Official announcement of Symfony 2.0. The project started in 2007.

[42] Potencier Fabien. Symfony2 Components: Standalone Libraries. — 2012. — URL: https://symfony.com/blog/symfony2-components-as-standalone-packages (дата обращения: 2025-11-22). Introduction of Symfony's component-based architecture.

[43] Potencier Fabien. Symfony 6: The Future of PHP Frameworks // SymfonyCon. — 2025. — URL: https://live.symfony.com (дата обращения: 2025-11-15).

[44] Preston-Werner Tom. Semantic Versioning 2.0.0. — 2010. — URL: https://semver.org/ (дата обращения: 2025-11-18). Официальная спецификация семантического версионирования.

[45] Propel Team. Propel ORM. — 2005. — URL: https://propelorm.org/ (дата обращения: 2025-12-07).

[46] Respect Project. Respect. — 2010. — URL: https://respect-validation.readthedocs.io/ (дата обращения: 2025-12-07).

[47] Seldaek Jordi B., Contributors. Composer: Dependency Manager for PHP. — URL: https://getcomposer.org/ (дата обращения: 2025-11-26). Initial release announcement and ongoing documentation.

[48] Slim Framework Team. Slim Framework 2.x HTTP Request Object (Slim). — 2010. — URL: https://github.com/slimphp/Slim-Documentation/blob/master/docs/objects/request.md (дата обращения: 2025-11-15).

[49] Slim Framework Team. Slim Framework 3.0 Release. — 2015. — URL: https://www.slimframework.com/docs/v3/ (дата обращения: 2025-11-22).

[50] Slim Framework Team. Slim Framework 4 — PSR-15 Middleware Architecture. — 2019. — URL: https://www.slimframework.com/docs/v4/ (дата обращения: 2025-11-26).

[51] Slim Framework Team. Slim Routing. — 2019. — URL: https://www.slimframework.com/docs/v4/objects/routing.html (дата обращения: 2025-12-07).

[52] Symfony Project. Symfony HttpFoundation Component. — 2008. — URL: https://symfony.com/doc/current/components/http_foundation.html (дата обращения: 2025-11-18).

[53] Symfony Project. Symfony Form Component. — 2009. — URL: https://symfony.com/doc/current/forms.html (дата обращения: 2025-12-05).

[54] Symfony Project. Symfony Validator Component. — 2009. — URL: https://symfony.com/doc/current/validation.html (дата обращения: 2025-12-05).

[55] Symfony Project. Symfony EventDispatcher Component. — 2010. — URL: https://symfony.com/doc/current/components/event_dispatcher.html (дата обращения: 2025-12-05).

[56] Symfony Project. Symfony DependencyInjection Component. — 2011. — URL: https://symfony.com/doc/current/components/dependency_injection.html (дата обращения: 2025-11-22). Official documentation for the Symfony DependencyInjection component.

[57] Symfony Project. Symfony Routing Component. — 2011. — URL: https://symfony.com/doc/current/routing.html (дата обращения: 2025-12-05).

[58] Symfony Project. Symfony PSR-15 HTTP Kernel Adapter. — 2016. — URL: https://github.com/symfony/psr-http-message-bridge (дата обращения: 2025-12-01).

[59] Symfony Project. Symfony PSR-7 Bridge. — 2016. — URL: https://github.com/symfony/psr-http-message-bridge (дата обращения: 2025-11-26).

[60] Symfony Project. Symfony HttpClient Component. — 2019. — URL: https://symfony.com/doc/current/http_client.html (дата обращения: 2025-12-05).

[61] Usage statistics of PHP for websites, W3Techs. — URL: https://w3techs.com/technologies/details/pl-php (дата обращения: 2025-11-15). 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 (дата обращения: 2025-11-18). Термин впервые задокументирован в 2004 году.

[63] Zend Technologies. Zend Framework: Open Source PHP Framework. — 2006. — URL: https://framework.zend.com/ (дата обращения: 2025-11-15).

[64] Zend Technologies. ZendComponent (Zend Framework 1.x). — 2006. — URL: https://framework.zend.com/manual/1.12/en/zend.http.html (дата обращения: 2025-11-16).

[65] Zend Technologies. ZendComponent. — 2012. — URL: https://packagist.org/packages/zendframework/zend-servicemanager#2.0.3 (дата обращения: 2025-11-16).

[66] Zend Technologies. Zend Diactoros: PSR-7 HTTP Message Implementations. — 2015. — URL: https://docs.laminas.dev/laminas-diactoros/ (дата обращения: 2025-11-22).