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

PHP[?] 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[?]-разработки.

В начале 2000-х годов появились первые PHP[?]-фреймворки. CakePHP (2005)[?], Symfony (2005)[?] и CodeIgniter (2006)[?] 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[?]-стеке. Это было связано с разработкой PHP-FIG [?] и ряда стандартов PSR [?], появлением Composer [?], унификацией HTTP-модели[?], распространением принципов DI [?] и архитектурой middleware [?].

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

Эти изменения включали выход PHP 7[?], 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[?]-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[?], предложения RFC Internals PHP, официальные публикации Symfony[?] и Laravel[?], дискуссии GitHub о стандартах Composer и PSR, а также выступления разработчиков на конференциях, таких как SymfonyCon[?] и Laracon[?].

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[?] началась с противоречий.

С другой стороны, благодаря своей простоте развёртывания и широкой совместимости PHP[?] 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[?], 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[?], Zend Framework[?], CakePHP[?] и CodeIgniter[?], 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[?] использовал собственный автозагрузчик и собственные соглашения об именовании классов.
- Zend Framework[?] 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[?] первым попытался решить эту проблему, выделив «компонентный подход[?]» (начиная с Symfony 2[?]), 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 [?],
- Zend
  Http [?] (Request / Response).
- Slim
  Http [?].
- Собственные реализации в CakePHP [?].

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[?] 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 [?] (2012) и PSR-4 [?] (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 [**?**] 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 [**?**] (Container Interface) каждый фреймворк реализовывал собственный DI-контейнер, и ни один из них не был совместим с другим:

- Symfony DependencyInjection [**?**].
- Laravel Container [**?**].
- Zend
  ServiceManager [**?**].
- Pimple [**?**].

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 [**?**].

[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 [?], появление Composer [?], формирование первых PSR-стандартов [?], PSR-4 [?], PSR-7 [?], PSR-11 [?], PSR-15 [?].

Однако именно после выхода PHP 7 [?] и широкого принятия PSR-7 [?], PSR-11 [?], PSR-15 [?] 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 [?] 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 [?] 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» [?] и мешало развитию фреймворков.

3.1.2. Принятое решение

Composer [?] ввёл:

- файл `composer.json` [?] в качестве декларации зависимостей.
- автозагрузчик, основанный на PSR-4 [?].
- семантическое версионирование [?].
- центральный репозиторий `Packagist` [?].

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

3.1.4. PSR-4 и единая модель автозагрузки

PSR-4 [?] (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 [?].

3.1.5. Причины

До PSR-4 [?]:

- 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 [?] 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 [?] получил техническую основу для генерации стандартизированного автозагрузчика.

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

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

PSR-7 [?] (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 [**?**], Zend Diactoros [**?**], Guzzle PSR-7 clients [**?**].
- Появление кросс-фреймворковых `middleware`.
- Symfony добавил PSR-7-бриджи [**?**].
- Laravel адаптировал совместимость на уровне интеграций [**?**].

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

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

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

3.4.1. Причины

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

3.4.2. Принятое решение

PSR-15 [?] 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 [?].
- Slim 4 полностью перешёл на PSR-15 [?].
- Symfony HttpKernel получил адаптеры [?].
- Laravel сохранил собственный контракт, но стал совместим через адаптеры [?].

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 [**?**] переписали DI-контейнер под строгую типизацию.
- Laravel [**?**] 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 [?] (HttpFoundation, EventDispatcher, Console, Routing).
- адаптация архитектуры под PSR-7/PSR-11 [?, ?].
- внедрение autowiring и автоконфигурации.
- появление Symfony Flex как современного пакета приложений.

3.6.2. Результаты

Большинство фреймворков (включая Laravel [?]) 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 [?] (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 [**?**] 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 [?] 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) [?].
- Symfony использует аннотации, YAML и PHP-конфигурации [?].
- Slim и Mezzio строят маршрутизацию вокруг middleware [?].
- FastRoute — чисто функциональная библиотека без привязки к фреймворку [?].

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 [**?**], Symfony [**?**]), другие (например Slim [**?**], 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).
- либо 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 [**?**] и PSR-15 [**?**].

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 [**?**].

Мотивация

- Множество несовместимых ORM: Doctrine ORM [**?**], Eloquent ORM [**?**], Propel [**?**], RedBeanPHP [**?**].
- 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.
- Бóльшая 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 [?] был принят в 2019 году, но его разработка сопровождалась огромным количеством противоречий в рассылках PHP-FIG [?].

Проблемы

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

  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.

Мотивация

- Symfony Form + Validator имеют сложную, но зрелую модель.
- Laravel Validation — декларативная модель со строковыми правилами.
- Respect/Validation — функциональная библиотека.

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.