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 с определённым языком серверной части [63].

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

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

В начале 2000-х годов в ответ на растущую сложность веб-приложений начали формироваться первые полнофункциональные PHP-фреймворки.

CakePHP (2005) [2], Symfony (2005) [40] и CodeIgniter (2006) [5] заложили основы MVC-подхода и упорядочили разработку веб-приложений в условиях отсутствия единых стандартов.

Однако по мере усложнения проектов стало очевидно, что решения того периода сталкиваются с рядом системных ограничений, включая непрозрачную архитектуру, несовместимость компонентов, отсутствие унифицированных интерфейсов и недостаточную тестируемость.

В течение следующего десятилетия (2005–2015) PHP [20]-стек претерпел значительные изменения. Это было связано с разработкой PHP-FIG [34] и ряда стандартов PSR [21], появлением Composer [48], 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-фреймворках. Эти изменения радикально изменили архитектурные решения, принципы проектирова-

ния и практики масштабирования.

Эти преобразования охватывали выход PHP 7 [35], внедрение массовой типизации и переход к строгой модели обработки ошибок, реорганизацию Symfony [40] в рамках компонентной архитектуры, стремительный развитие Laravel, а также упадок Zend [64] Framework и его последующую трансформацию в проект Laminas.

Цель настоящей работы — провести комплексный анализ ключевых архитектурных и технологических изменений, произошедших в PHP-фреймворках в период с 2015 по 2025 годы. В рамках исследования предполагается выявить причины указанных изменений, оценить эффективность принятых техническиатурных решений, а также определить, какие концепции и подходы были отвергнуты или модифицированы в ходе профессиональных дискуссий.

Для достижения поставленных целей используются следующие источники: материалы рассылок PHP-FIG [34], предложения RFC Internals PHP, официальные публикации Symfony [40] и Laravel, дискуссии GitHub о стандартах Composer и PSR, а также доклады разработчиков, представленные на профильных конференциях — в частности, SymfonyCon [44] и Laracon [19].

Таким образом, в статье рассматривается не только эволюция отдельных фреймворков, но и более широкий процесс формирования профессиональной PHP-экосистемы, в которой технологические инновации выступают в качестве ключевого фактора развития.

2. Особенности развития PHP-экосистемы до стандартизации (до 2015 года)

К 2015 году промышленная разработка на PHP [20] характеризовалась двойственным положением. С одной стороны, язык сохранял широкое распространение и технологическую зрелость как платформа для веб-приложений, обусловленную простотой развёртывания, высокой доступностью хостинговых решений и устойчивой обратной совместимостью. С другой стороны, PHP-экосистема страдала от существенных архитектурных ограничений, вызванных отсутствием унифицированных стандартов проектирования и низкой совместимостью между ключевыми компонентами. Это приводило к увеличению совокупной стоимости сопровождения проектов, а также замедляло темпы развития как сторонних библиотек, так и фреймворков.

2.1. Отсутствие единых интерфейсов

До появления PHP-FIG [34] в экосистеме PHP отсутствовали общепринятые технические соглашения по ключевым аспектам разработки, включая:

- организацию структуры каталогов;
- механизмы автоматической загрузки классов;
- унифицированное представление HTTP-запросов и ответов;
- интерфейсы контейнеров внедрения зависимостей (DI);
- стандартные контракты для middleware и модель обработки HTTP-запросов.

В результате каждый из ведущих фреймворков, включая Symfony [40], Zend [64], CakePHP [2] и CodeIgniter [5], реализовывал собственные, зачастую несовместимые подходы к решению этих задач. Это привело к эффекту фрагментации экосистемы PHP, при котором фреймворки

и библиотеки развивались как изолированные подсистемы с ограниченной совместимостью.

Например, Symfony [40] применял собственный автозагрузчик и строгие соглашения об именовании, тогда как Zend предлагал иную структуру каталогов. Многие сторонние библиотеки того времени подключались вручную и не содержали формализованных метаданных о зависимостях, что исключало автоматизированное управление ими.

Такая ситуация приводила к высокой степени связанности между фреймворками и их компонентами. Интеграционные решения часто основывались на неформализованных соглашениях и паттернах, а изменения в деталях реализации нарушали обратную совместимость.

2.2. Проблема совместимости и отсутствие компонентов повторного использования

Ключевая проблема данного периода заключалась в невозможности использовать одну и ту же библиотеку в разных фреймворках. Такие распространённые компоненты, как HTTP-клиенты, инструменты логгирования и шаблонизаторы, были тесно интегрированы в архитектуру конкретных фреймворков и зависели от их внутренних соглашений, стилей кодирования и точек расширения. Попытки переноса подобных решений между проектами зачастую приводили к конфликтам, обусловленным различиями в API, структуре кода и механизмах интеграции.

Symfony [40] стал одним из первых фреймворков, предпринявших системную попытку преодолеть эту фрагментацию за счёт внедрения компонентного подхода [43] (начиная с Symfony 2 [42]), однако в отсутствие общепринятых отраслевых стандартов такой подход оставался локальным решением, которое не обеспечивало гарантированной совместимости и стабильности интеграций.

2.3. Отсутствие стандартизированной модели HTTP

Одним из ключевых технологических ограничений PHP-экосистемы до появления общих стандартов являлось отсутствие унифицированного представления основных HTTP-компонентов: запроса, ответа, а также их атрибутов, заголовков и потоков данных.

Каждый из ведущих фреймворков реализовывал собственные, несовместимые между собой классы для работы с HTTP-слоем:

- HttpFoundation [54] в Symfony;
- Http [65] (Request / Response) в Zend;
- Http [49] в Slim;
- Собственные реализации в CakePHP [1].

Такая фрагментация затрудняла разработку взаимозаменяемых middleware-компонентов, снижала переносимость кода между проектами и препятствовала созданию единых решений для HTTP-клиентов, фильтров и других инструментов обработки запросов.

Глубина проблемы подтверждается продолжительностью обсуждения стандарта PSR-7 [25] в рабочих группах PHP-FIG: дискуссии по его формулировке велись почти три года. Это свидетельствует о фундаментальном характере задачи стандартизации HTTP-интерфейсов для всей экосистемы.

2.4. Отсутствие единых правил для автозагрузки и управления зависимостями

До появления Composer [48] и принятия стандарта PSR-4 [23] (2013–2014 гг.) подключение сторонних библиотек в PHP-проектах, как правило, осуществлялось вручную с использованием конструкций

`require/include`. Декларативное управление зависимостями фактически отсутствовало, а распространение кода происходило преимущественно через архивы в формате ZIP или посредством системы PEAR.

Отсутствие унифицированных правил автозагрузки классов обусловливало жёсткую привязку к структуре каталогов и способствовало частым конфликтам версий библиотек.

Ситуация принципиально изменилась с появлением Composer [48], который ввёл централизованный и декларативный механизм управления зависимостями и автозагрузкой. Однако его широкое внедрение в промышленную разработку началось лишь в 2014–2015 гг. Этот период стал отправной точкой для изменений, анализируемых в последующих разделах.

2.5. Проблемы тестируемости и внедрения зависимостей

До принятия стандарта PSR-11 [30] (Container Interface) каждый из ведущих PHP-фреймворков реализовывал собственный контейнер внедрения зависимостей (DI-контейнер), причём эти реализации не были совместимы между собой. К числу наиболее распространённых решений относились:

- Symfony Dependency Injection [59];
- Laravel Container [15];
- Zend Service Manager [66];
- Pimple [41].

Отсутствие унифицированного интерфейса контейнера создавало ряд системных ограничений:

- затрудняло разработку многократно используемых пакетов, не привязанных к конкретному фреймворку;

- делало невозможным перенос middleware-компонентов и других сервисов между проектами;
- замедляло развитие архитектур, основанных на принципах внедрения зависимостей и их управления.

Таким образом, архитектура внедрения зависимостей представляла собой одну из ключевых проблем экосистемы PHP, решение которой стало возможным лишь в последние 10 лет (2015–2025) в ходе стандартизации — в частности, с принятием PSR-11 [30].

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

Период 2015–2025 годов стал для PHP-фреймворков этапом глубокой технологической трансформации.

В предшествующее десятилетие (2005–2015) были сформулированы и заложены фундаментальные концепции, определившие дальнейшее развитие экосистемы: создание PHP-FIG [34], появление менеджера зависимостей Composer [48], а также разработка PSR, первых стандартов PHP [21], включая PSR-4 [23], PSR-7 [25], PSR-11 [30], PSR-15 [31].

Однако качественный переход к современной архитектуре веб-приложений начался лишь после выхода PHP 7 [35] и широкого внедрения ключевых стандартов — PSR-7 [25], PSR-11 [30] и PSR-15 [31].

В данном разделе рассматриваются ключевые архитектурные изменения, произошедшие в указанный период, анализируются предпосылки их возникновения и оценивается их влияние на эволюцию PHP-экосистемы.

3.1. Composer как фундамент модернизации экосистемы

Менеджер зависимостей Composer [48] был впервые выпущен в 2012 году, однако его широкое внедрение в PHP-экосистеме произошло в 2015–2017 гг. Именно в этот период подавляющее большинство фреймворков и сторонних библиотек перешли на декларативную модель управления зависимостями.

Как отмечалось выше, до внедрения Composer управление зависимостями оставалось фрагментированным и неформализованным, что создавало ряд проблем при сопровождении крупных проектов и осложняло архитектурную эволюцию фреймворков.

Composer предложил унифицированное решение, включавшее декларативное описание зависимостей с помощью `composer.json` [3], стандартизированный механизм автозагрузки классов на основе PSR-4 [23],

поддержку семантического версионирования (SemVer) [45] и централизованный репозиторий пакетов Packagist [37].

Благодаря этим инновациям архитектурная парадигма PHP-фреймворков трансформировалась: они перестали восприниматься как монолитные системы и стали рассматриваться как композиции слабосвязанных, автономных компонентов, пригодных к обособленному выбору, комбинированию и обновлению.

К началу 2020-х годов Composer фактически закрепился в качестве инфраструктурного стандарта PHP-экосистемы. Это привело к снижению привязки библиотек к конкретным фреймворкам, ускорению обмена компонентами между проектами и формированию устойчивой культуры разработки фреймворк-независимых пакетов, ориентированных на повторное использование.

3.2. Развитие PSR и их роль в архитектурной трансформации PHP-экосистемы

Стандарты PSR, разрабатываемые в рамках PHP-FIG [34], стали центральным инструментом преодоления архитектурной фрагментации PHP-экосистемы в период 2015–2025 годов. В отличие от ранних инициатив по унификации, ориентированных на внутреннюю архитектуру отдельных фреймворков, PSR-стандарты были направлены на установление минимальных, но строго определённых контрактов, обеспечивающих совместимость между фреймворками без навязывания конкретных реализаций.

Эта стратегия позволила сформировать общий архитектурный «язык», на котором могли взаимодействовать разнородные компоненты, независимо от их происхождения. Каждый из ключевых стандартов решал узкую, но критически важную проблему совместимости, при этом внося вклад в общую картину модульной и переносимой архитектуры.

В следующих подразделах рассматриваются наиболее влиятельные PSR-стандарты, сыгравшие решающую роль в трансформации PHP-экосистемы: их предпосылки, ключевые положения и архитектурные

последствия.

3.2.1. PSR-4 и единая модель автозагрузки (2015)

Стандарт PSR-4 [23] стал решающим этапом в переходе от фрагментированных, фреймворк-специфичных соглашений к единой и предсказуемой модели автозагрузки классов в PHP-экосистеме.

Его практическое значение особенно возросло в период 2015–2020 годов, когда основные фреймворки начали систематически приводить структуру каталогов и организацию пространств имён в соответствие с PSR-4 [23]. Это, в свою очередь, предоставило менеджеру зависимостей Composer устойчивую основу для генерации стандартизированного автозагрузчика, совместимого с любым соответствующим пакетом.

До принятия PSR-4 [23] каждый крупный фреймворк использовал собственные, несовместимые между собой правила сопоставления имён классов с путями к файлам. Такая фрагментация затрудняла интеграцию сторонних библиотек, увеличивала сложность сопровождения и негативно сказывалась на масштабируемости проектов.

В отличие от прежней ситуации, когда каждая платформа определяла собственные правила, PSR-4 [23] формализовал два ключевых принципа: во-первых, строгое соответствие между пространствами имён и иерархией каталогов; во-вторых, возможность автоматической загрузки классов без необходимости использования конструкций `include` или `require`.

В результате значительно повысилась переносимость библиотек между проектами, а архитектура фреймворков стала более компонентной: сократилась доля проприетарного инфраструктурного кода, а уровень взаимной совместимости в экосистеме в целом существенно возрос.

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

Стандарт PSR-7 [25] (HTTP Message Interface) стал одним из ключевых факторов в эволюции архитектуры PHP-экосистемы.

До его принятия HTTP-запросы и ответы в каждом фреймворке ре-

ализовывались с использованием собственных, несовместимых между собой моделей. Это препятствовало переносу middleware-компонентов и делало практически невозможной разработку взаимозаменяемых HTTP-инструментов — таких как клиенты, маршрутизаторы или фильтры — между различными стеками фреймворков. Кроме того, отсутствовали унифицированные интерфейсы для потоков данных, заголовков и других элементов HTTP-сообщений.

PSR-7 [25] устранил эту фрагментацию, предложив стандартизированную модель HTTP-сообщений и чётко определённый набор интерфейсов для основных сущностей: `Request`, `Response`, `Stream`, `UploadedFile` и `URI`.

Практическим следствием стандарта стало развитие middleware-компонентов и переносимых HTTP-библиотек. В экосистеме утвердились нативные PSR-7-совместимые реализации, такие как Slim 3 [50] и Zend Diactoros [67]. Крупные фреймворки также адаптировались к новой реальности: Symfony внедрил специализированные PSR-7-бриджи [61], а Laravel обеспечил совместимость через интеграционные слои [18].

Таким образом, PSR-7 [25] не только решил проблему фрагментации HTTP-интерфейсов, но и заложил архитектурную основу для последующей стандартизации — в частности, для разработки контрактов `middleware` в PSR-15 [31].

3.2.3. PSR-11 и унификация DI-контейнеров (2017)

Ещё одной существенной проблемой PHP-экосистемы до стандартизации являлась несовместимость контейнеров внедрения зависимостей: такие фреймворки, как Laravel [11], Symfony [59] и Zend Framework, реализовывали собственные, взаимно несовместимые API, что не позволяло сторонним библиотекам запрашивать зависимости абстрактно и переносимо.

Принятие стандарта PSR-11 [30] устранило данный барьер, определив спецификацию контейнера в виде двух интерфейсов: `ContainerInterface` и `NotFoundExceptionInterface`.

Благодаря этому сторонние компоненты получили возможность вза-

имодействовать с DI-контейнером без привязки к конкретному фреймворку. В результате внедрение зависимостей трансформировалось из внутреннего механизма отдельных фреймворков в общую экосистемную практику. Это способствовало снижению фрагментации пакетов и упростило разработку переносимых компонентов, совместимость которых обеспечивается исключительно соответствием требованиям PSR-11 [30].

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

После принятия стандарта PSR-7 [25], унифицировавшего представление HTTP-сообщений, стала возможной дальнейшая стандартизация самой модели обработки запросов через цепочки middleware.

До появления PSR-15 [31] фреймворки применяли несовместимые подходы к организации middleware-слоя: так, Symfony использовал компонент HttpKernel, Laravel — собственные фильтры и middleware-механизмы, тогда как последовательная middleware-архитектура, основанная на цепочках обработчиков, была характерна лишь для отдельных стеков, таких как Slim [51] и Zend Expressive (впоследствии Mezzio) [10].

Стандарт PSR-15 [31] формализовал два базовых интерфейса: `MiddlewareInterface` и `RequestHandlerInterface`.

Это приблизило PHP-экосистему к устоявшимся в других языках моделям обработки запросов через middleware-цепочки — таким как Rack (Ruby), WSGI (Python) и Connect/Express (Node.js).

Практическое значение PSR-15 [31] заключалось в ускорении разработки переносимых middleware-компонентов и обеспечении единой основы для middleware-ориентированных фреймворков, включая Mezzio [10] и Slim 4 [51]. Фреймворки с иной архитектурой, такие как Symfony и Laravel, сохранили свои внутренние модели, но обеспечили совместимость с PSR-15 посредством адаптеров и мостов [61, 18].

В результате `middleware` превратился в одну из ключевых архитектурных единиц проектирования современных PHP-приложений.

3.3. Влияние PHP 7 и PHP 8 на архитектуру фреймворков

Эволюция языка PHP в версиях 7 [35] (2015) и 8 [39] (2020) оказала фундаментальное воздействие на архитектурные основы современных PHP-фреймворков, инициировав переход от динамической, слабо типизированной модели к более строгой и предсказуемой парадигме проектирования. Ключевым сдвигом стало не просто ускорение выполнения кода или расширение возможностей, а изменение самой культуры взаимодействия компонентов: язык стал требовать от разработчиков — и, соответственно, от фреймворков — большей формальной точности, что напрямую отразилось на проектировании API, управлении зависимостями и организации внутренней логики приложений.

С выходом PHP 7 [35] архитектура PHP-разработки начала кардинально меняться. Переход от нефатальных ошибок и предупреждений к строгой модели на основе исключений, а также появление скалярной типизации и возвращаемых типов позволили оформлять контракты между компонентами на уровне самого кода, а не в виде комментариев. Это побудило фреймворки отказываться от неявных соглашений и динамических зависимостей, которые ранее служили источником ошибок и снижали сопровождаемость. Интерфейсы стали проектироваться как строго типизированные контракты, в которых сигнатуры методов выражали не просто реализацию, а архитектурную спецификацию.

Особенно заметным стало влияние этих изменений на контейнеры внедрения зависимостей. Возможность управлять зависимости на основе типов, поддерживаемых автоматическим связыванием и строгими сигнатурами, позволила сократить объём конфигурационного кода и повысить прозрачность архитектуры.

Релиз PHP 8 [39] усилил и закрепил эти тенденции, добавив важные механизмы, которые ещё больше повысили декларативность и явность архитектуры. Введение объединений (Union Types) устранило необходимость в избыточной проверке типов внутри методов и позволило точно описывать допустимые варианты входных и выходных данных. Атри-

буты позволили перенести метаданные, ранее размещавшиеся в конфигурационных файлах или PHPDoc-аннотациях, непосредственно в тело кода, сделав маршрутизацию, сериализацию, валидацию и другие аспекты логики явными и локализованными. А выражение match, являясь типобезопасной и компактной альтернативой switch, также способствовало написанию более корректного и лаконичного кода.

В результате таких изменений в экосистеме фреймворков усилилось применение строгих архитектурных паттернов: получили широкое распространение Объекты передачи данных (DTO), Объекты-Значения (Value Objects) и неизменяемые структуры данных. Эти подходы, ранее считавшиеся излишне формальными для PHP-среды, стали практически применимыми благодаря возможностям типизации и предсказуемости поведения кода.

Крупнейшие фреймворки, такие как Symfony и Laravel [11], адаптировались к этим изменениям, сохраняя свои особенности, но двигаясь в едином направлении. Symfony сосредоточился на максимальной интеграции с возможностями языка, усилив типобезопасность компонентов и оптимизировав контейнер зависимостей [59], тогда как Laravel [11], оставаясь ориентированным на удобство, постепенно сокращал использование неявной «магии» и усиливал контрактность своих API. В обоих случаях развитие самого языка стало не внешним обстоятельством, а внутренним драйвером архитектурной трансформации.

Таким образом, влияние PHP 7 [35] и PHP 8 [39] на фреймворки проявилось не в виде изолированных языковых нововведений, а как формирование новой дисциплины проектирования, ориентированной на явные контракты, модульность и формальную корректность. Эти изменения не только повысили надёжность и сопровождаемость PHP-приложений, но и сблизили экосистему с архитектурными практиками, принятыми в других современных серверных платформах, тем самым закрепив переход PHP от «скриптового» языка к стабильной среде для построения сложных, масштабируемых систем.

3.4. Влияние Symfony на развитие компонентной модели в PHP

Symfony оказал определяющее влияние на архитектурную трансформацию PHP-экосистемы, выступив не только в роли полноценного веб-фреймворка, но и в качестве поставщика универсальных инфраструктурных компонентов.

Этот переход от монолитной модели «полного фреймворка» к совокупности автономных, слабосвязанных компонентов [43] — таких как HttpFoundation [54], EventDispatcher, Console [55] и Routing [60] — стал ответом на ключевые проблемы PHP-экосистемы. Компоненты разрабатывались как независимые библиотеки с чётко определёнными интерфейсами и минимальными внешними зависимостями, что обеспечило их применимость вне контекста самого Symfony. Подобный подход соответствовал инициативам PHP-FIG, способствуя активной интеграции Symfony с отраслевыми стандартами, в частности с PSR-7 и PSR-11 [61, 30], и тем самым укрепляя совместимость на уровне фундаментальных абстракций.

Дальнейшее развитие архитектуры фреймворка сопровождалось внедрением механизмов автосвязывания (autowiring) [53] и автоконфигурации (auto-configuration), ставших возможными благодаря усилению типобезопасности и развитию рефлексии в PHP 7 и PHP 8. Эти механизмы позволили значительно сократить объём явной конфигурации, перенося информацию о структуре зависимостей непосредственно в сигнатуры классов. В результате архитектура приложений приблизилась к декларативной модели, в которой связи между компонентами становятся явными, предсказуемыми и верифицируемыми на этапе статического анализа, что снижает нагрузку при разработке и сопровождении масштабных систем.

Кульминацией компонентной культуры Symfony стало появление Symfony Flex — инфраструктурного решения, переосмыслившего процесс сборки приложения как динамическую композицию пакетов и конфигураций. Эта модель устранила необходимость в фиксированной

структуре фреймворка и позволила разработчикам гибко конструировать приложения под конкретные задачи, используя только необходимые компоненты.

Таким образом, Symfony сформировал ядро инфраструктурной конвергенции в PHP-экосистеме, вне зависимости от языкового ядра, а его компонентная модель легла в основу современных подходов к разработке расширяемых, переносимых и стандартизированных систем.

3.5. Laravel как проводник DX-практик в PHP-экосистеме

Laravel [12] в период 2015–2025 годов утвердился как ведущий фреймворк, ориентированный на максимальное удобство разработки и единообразие интерфейсов. В отличие от Symfony, Laravel [11] изначально был нацелен на снижение порога входа и унификацию практик прикладной разработки, что обусловило его широкую популярность среди как начинающих, так и опытных разработчиков.

Архитектурная траектория Laravel в указанный период характеризуется постепенной интеграцией стандартов PHP-FIG при одновременном сохранении высокоуровневых, локальных абстракций. Ключевым этапом стало принятие Composer [48] в качестве менеджера зависимостей и внедрение PSR-4 [23] для автозагрузки классов, что позволило интегрировать Laravel [11] в общую экосистему PHP-пакетов и отказаться от проприетарных механизмов загрузки. Поддержка PSR-11 [30] была реализована частично: собственный контейнер сохранил уникальный API, однако получил совместимость на уровне интерфейсов, обеспечив возможность подключения сторонних библиотек без радикальной перестройки внутренней архитектуры.

Значительным шагом в направлении конвергенции с общеотраслевыми архитектурными практиками стало внедрение middleware-модели, совместимой с PSR-15 [31]. Несмотря на сохранение собственного контракта для middleware, Laravel [11] предоставил механизмы адаптации PSR-совместимых компонентов, что позволило использовать унифици-

рованные подходы к обработке HTTP-запросов без нарушения обратной совместимости.

Эволюция самого языка PHP, в особенности в версиях 7 [35] и 8 [39], оказала прямое влияние на внутреннюю структуру фреймворка. При этом Laravel [11] интегрировал новые языковые возможности таким образом, чтобы они не снижали читаемость и выразительность кода, сохранив баланс между архитектурной дисциплиной и удобством использования.

Особую роль в расширении функциональных границ фреймворка сыграло появление специализированных инфраструктурных инструментов — Horizon, Octane, Sail и Pint. Эти компоненты вывели Laravel за пределы традиционного HTTP-слоя, превратив его в полноценную платформу, охватывающую весь жизненный цикл приложения — от локальной разработки и тестирования до эксплуатации в высоконагруженных средах.

В совокупности эти изменения закрепили за Laravel роль центрального элемента стандартизированной PHP-экосистемы. Таким образом, Laravel выступил не только как пассивный реципиент архитектурных трансформаций, но и как активный проводник их в прикладную разработку, оказывая существенное влияние на индустриальные практики и образовательные траектории в PHP-сообществе.

4. Отклоненные и отложенные инициативы

Как и в случае с другими устоявшимися программными экосистемами, процесс стандартизации PHP-фреймворков сопровождался не только успешными инициативами, но и значительным числом предложений, которые не были приняты или были сознательно отложены. Анализ этих инициатив представляет особый интерес, поскольку позволяет выявить границы применимости стандартизации и понять, какие архитектурные решения были признаны избыточными, рискованными либо концептуально несоответствующими направлению развития экосистемы.

В данном разделе рассматриваются наиболее показательные инициативы, не приведшие к формированию стандарта. Их анализ позволяет дополнить картину развития PHP-экосистемы и продемонстрировать, что зрелость технологии проявляется не только в способности к унификации, но и в умении своевременно отказаться от чрезмерной стандартизации.

4.1. Нереализованный стандарт маршрутизации (Router PSR)

Одной из наиболее обсуждаемых, но так и не реализованных инициатив в рамках PHP-FIG [34] стало предложение по стандартизации интерфейса маршрутизации HTTP-запросов — Router PSR [24]. С 2015 года эта тема неоднократно выносилась на обсуждение в рассылках FIG, что свидетельствует как о её практической актуальности, так и о принципиальных трудностях формализации данного архитектурного уровня.

Потребность в стандарте обусловливалась значительной фрагментацией подходов к маршрутизации в экосистеме PHP. Например, Laravel применял декларативную синтаксическую модель (fluent API) [14], тогда как Symfony использовал аннотации, YAML и PHP-конфигурации [60]. В то же время Slim и Mezzio применяли подход, в котором маршру-

тизация реализовывалась посредством *middleware* [52]. Отдельную нишу занимали минималистичные библиотеки вроде FastRoute [38], реализующие чисто функциональный подход без какой-либо фреймворк-специфической логики. Отсутствие общего интерфейса исключало возможность создания переносимых компонентов — будь то универсальные middleware, инструменты для тестирования маршрутов или декларативные DSL.

Однако в ходе обсуждений в 2016–2018 годах стало очевидно, что различия между этими моделями носят не поверхностный, а концептуальный характер. Controller-based (Laravel [14], Symfony [60]) и middleware-based (Slim [52], Mezzio) подходы предполагают разные точки интеграции и жизненные циклы обработки запроса. Попытки выработать единый интерфейс неизменно сталкивались с дилеммой: либо предложить слишком абстрактный контракт, не обеспечивающий практической пользы, либо зафиксировать архитектурные предпочтения одной из моделей, тем самым ограничив свободу проектирования фреймворков.

Ключевым фактором отказа от стандартизации стало также стратегическое стремление PHP-FIG избегать регулирования высокоуровневых архитектурных решений. В условиях активной эволюции подходов к проектированию веб-приложений подобное ограничение было сочтено контрпродуктивным.

В итоге инициатива Router PSR была закрыта как недостаточно универсальная и потенциально ограничивающая. Сообщество пришло к консенсусу: маршрутизация остаётся внутренним делом каждого фреймворка, а взаимодействие между разнородными архитектурами обеспечивается на более низком уровне — через уже устоявшиеся стандарты PSR-7 [25] и PSR-15 [31].

4.2. Попытка стандартизации ORM и абстракции баз данных

В экосистеме PHP на протяжении нескольких лет обсуждалась возможность создания стандарта для унифицированного доступа к данным, функционально сопоставимого с JDBC в экосистеме Java. В период с 2014 по 2020 год в рабочих группах и рассылках PHP-FIG неоднократно поднимался вопрос о разработке общего интерфейса для объектно-реляционных отображений (ORM) [22].

На момент начала обсуждений в PHP-сообществе уже существовало несколько зрелых и широко используемых ORM-решений, включая Doctrine ORM [4], Eloquent ORM [13], Propel [46] и RedBeanPHP [6]. Однако эти библиотеки реализовывали принципиально разные архитектурные парадигмы. Doctrine ORM развивалась в русле концепций Domain-Driven Design и паттерна Unit of Work, предполагая строгую типизацию доменных моделей и явное управление жизненным циклом объектов. Eloquent ORM, напротив, следовал классическому паттерну Active Record, интегрируя логику доступа к данным непосредственно в сущности и активно используя динамические методы. Propel делала ставку на генерацию кода на основе декларативных XML-схем, обеспечивая строгую синхронизацию модели с базой данных, тогда как RedBeanPHP предлагала полностью динамический подход, допускающий эволюцию схемы данных в процессе выполнения приложения.

Эти различия оказались не синтаксическими, а концептуальными: каждая модель предполагала особый способ проектирования домена, управления состоянием и взаимодействия с реляционным хранилищем. Дополнительным аргументом против стандартизации стало стремление избежать повторения опыта Java EE, где преждевременная канонизация persistence-слоя привела к усложнению экосистемы.

В итоге инициатива по созданию PSR для ORM была признана нежизнеспособной. Отказ от стандартизации в этой области следует рассматривать не как неудачу, а как осознанный архитектурный выбор: сообщество предпочло сохранить разнообразие культур работы с данны-

ми, предоставив разработчикам свободу выбора инструмента, наиболее соответствующего задачам проекта и принятой архитектурной дисциплине.

4.3. Ограниченная применимость стандарта PSR-14 (Event Dispatcher)

Стандарт PSR-14 [32], посвящённый диспетчеризации событий, был официально утверждён PHP-FIG в 2019 году, однако его разработка сопровождалась продолжительными и во многом противоречивыми дискуссиями в рассылках сообщества [26].

Ключевая сложность стандартизации заключалась в фундаментальных различиях архитектурных моделей, применяемых в ведущих PHP-фреймворках. Symfony использует синхронную модель диспетчеризации [58], в которой события представляют собой объекты, а обработчики (слушатели) вызываются последовательно. Laravel, напротив, разделяет понятия события и слушателя более декларативно, активно интегрируя механизмы асинхронной обработки через очереди и поддержку broadcast-событий [17]. В Zend Framework применяется модель агрегаторов событий [8], допускающая сложную композицию обработчиков и гибкое управление их приоритетами.

Эти различия затрагивают не только сигнатуры API, но и семантические аспекты обработки: порядок выполнения слушателей, возможность прерывания цепочки, управление состоянием события, а также взаимодействие с асинхронной инфраструктурой. Учёт всех этих факторов в едином стандарте оказался невозможным без наложения архитектурных ограничений. В результате PSR-14 [32] был сформулирован как крайне минималистичный контракт, определяющий лишь метод `dispatch($event)`, без каких-либо предписаний относительно жизненного цикла события, порядка обработки или механизма подписки.

Несмотря на формальное принятие, практическое внедрение PSR-14 [32] в экосистему остаётся ограниченным. Большинство фреймворков продолжают использовать собственные, более выразительные системы

событий, а совместимость с PSR-14, где она реализована, зачастую носит формальный или адаптерный характер. Таким образом, PSR-14 служит наглядной иллюстрацией границ применимости стандартизации: в областях, где архитектурные различия носят не технический, а концептуальный характер, даже формальный консенсус не обеспечивает реальной совместимости.

4.4. Попытка стандартизации валидации и форм

Идея стандартизации механизмов валидации данных и обработки форм обсуждалась в PHP-сообществе эпизодически в период с 2016 по 2021 годы [27]. Поводом для дискуссий стало сосуществование нескольких зрелых, но архитектурно несовместимых решений, активно применяемых в ведущих фреймворках и библиотеках.

Так, Symfony реализует формально строгую и высоко расширяемую модель [56, 57], основанную на объектном представлении форм и валидационных ограничений, где правила встроены в аннотации или конфигурационные объекты. Laravel, напротив, использует декларативный подход [16], в котором валидация задаётся правилами и тесно интегрирована с HTTP-запросами и процессом обработки входных данных. Отдельную нишу занимает библиотека Respect/Validation [47], предлагающая функциональный стиль с возможностью композиции правил через текучие интерфейсы (fluent interface).

Эти различия выходят далеко за рамки синтаксиса API. Они затрагивают уровень абстракции, природу используемых DSL, стратегии интеграции с бизнес-логикой и даже философские взгляды на роль валидации в приложении. Попытка выработать единый стандарт неизбежно привела бы либо к чрезмерному упрощению, лишающему решения выразительности, либо к канонизации одной из парадигм в ущерб другим. Более того, сообщество PHP традиционно рассматривает валидацию как область, тесно связанную с эволюцией моделей данных, пользовательских интерфейсов и практик проектирования, и потому требующую пространства для экспериментов.

В результате инициатива по созданию отдельного PSR для валидации и форм была отклонена. Этот отказ от стандартизации подчёркивает важное методологическое различие: в то время как инфраструктурные уровни (например, HTTP-сообщения или контейнеры зависимостей) выигрывают от унификации, высокоуровневые механизмы, напрямую связанные с бизнес-логикой и пользовательским опытом, требуют архитектурной гибкости и разнообразия подходов.

4.5. PSR-18 и границы стандартизации HTTP-клиентов

Стандарт PSR-18 [33], посвящённый унификации HTTP-клиентов, был утверждён PHP-FIG в 2019 году, однако, как и предыдущие инициативы, его разработка сопровождалась продолжительными дискуссиями в рассылках сообщества [29].

Одной из ключевых точек напряжения стала модель обработки ошибок. Часть участников настаивала на использовании исключений как основного механизма сигнализации о сбоях, в то время как другие выступали за подход, при котором клиент всегда возвращает объект ответа, а ошибки обрабатываются как часть бизнес-логики.

Дополнительную сложность вносило разнообразие архитектурных подходов в существующих библиотеках. Guzzle [7], будучи наиболее распространённым решением, исторически развивался как синхронный клиент с опциональной поддержкой асинхронности через промисы. HTTPlug [36] позиционировался как абстрактный адаптерный слой, позволяющий переключаться между различными реализациями без привязки к конкретному клиенту. Symfony HttpClient [62], в свою очередь, проектировался как глубоко интегрированный компонент экосистемы Symfony с акцентом на расширяемость, поддержку потоковой передачи и тонкий контроль над транспортным уровнем. Попытки выработать единый интерфейс, способный органично включать все эти модели, неизбежно приводили к коллизии требований.

В ходе обсуждений активно рассматривались предложения по вклю-

чению асинхронных интерфейсов в стиле Promise, унифицированного API для стриминга и механизмов отмены запросов. Однако расширение стандарта на эти функциональные области значительно повысило бы его сложность и, как следствие, снизило бы вероятность его принятия и практического внедрения ключевыми реализациями.

В итоге PHP-FIG принял решение ограничиться минималистичным синхронным интерфейсом, определяющим лишь базовый метод sendRequest (RequestInterface). Несмотря на критику со стороны части сообщества, такой подход позволил зафиксировать минимально необходимый общий знаменатель и обеспечить базовую совместимость между различными HTTP-клиентами.

Таким образом, PSR-18 [33] стал примером осознанного архитектурного компромисса: стандартизация была проведена строго на уровне, достаточном для совместимости, но без попытки унифицировать сложные, концептуально разнородные аспекты клиентского сетевого взаимодействия. Это решение подчёркивает эволюцию стратегии PHP-FIG — от амбициозных унификаций к прагматичным, минимально достаточным контрактам.

4.6. Инициатива унификации конфигурации DI-контейнеров (расширение PSR-11)

После утверждения PSR-11 [28], определившего минимальный интерфейс контейнера внедрения зависимостей, в сообществе PHP-FIG неоднократно поднимался вопрос о его расширении. В частности, обсуждалась возможность стандартизации механизмов регистрации сервисов, описания фабрик и задания конфигурационных параметров — шаг, который, в теории, мог бы обеспечить полную взаимозаменяемость DI-контейнеров.

Основной мотивацией инициативы выступало стремление повысить переносимость конфигурации и снизить привязку библиотек к конкретным фреймворкам. Однако уже на ранних этапах обсуждения выявили фундаментальные расхождения в подходах к конфигурированию DI-

контейнеров, характерных для ведущих PHP-фреймворков.

Так, Symfony [59] использует декларативную модель, в которой конфигурация задаётся через YAML, XML или PHP-файлы и транслируется в строго типизированный контейнер с предсказуемым поведением. Laravel, напротив, опирается на динамическую регистрацию сервисов через замыкания (closures) и связывание (bindings) [15], делая ставку на выразительность, гибкость и минимизацию объёма конфигурационного кода. Laminas реализует подход [9], основанный на ассоциативных массивах, подчёркивая явность структуры и простоту композиции конфигураций.

Попытка унифицировать такие модели в рамках единого стандарта неизбежно привела бы либо к переусложнённому и трудно реализуемому интерфейсу, либо к канонизации одного из подходов — с соответствующим ограничением архитектурной автономии остальных решений.

Помимо этого, стандартизация механизмов конфигурирования DI-контейнеров могла бы подавить инновационное развитие в области управления зависимостями и затруднить внедрение новых паттернов.

В итоге было принято сознательное решение сохранить PSR-11 в его минималистичной форме. Этот выбор отражает общую стратегию PHP-FIG: стандартизировать лишь те уровни абстракции, где достигается максимальная совместимость при минимальном вмешательстве в архитектурную философию конкретных фреймворков.

5. Заключение

В период с 2015 по 2025 год экосистема PHP-фреймворков претерпела глубокую архитектурную трансформацию, в результате которой веб-разработка на PHP перешла от фрагментированного ландшафта несовместимых решений к единому технологическому пространству, основанному на стандартах, переносимых компонентах и общей инфраструктуре.

Этот переход стал возможным благодаря синергии трёх ключевых факторов: деятельности PHP-FIG, широкому внедрению системы управления зависимостями Composer и эволюции самого языка в версиях PHP 7 и 8. Стандарты PSR-4 [23], PSR-7 [25], PSR-11 [30] и PSR-15 [31] сформировали ядро новой архитектурной базы. В совокупности они обеспечили унификацию структуры проектов, создали общий слой функциональной совместимости для обработки HTTP-запросов и позволили компонентам, библиотекам и сервисам свободно перемещаться между фреймворками. Composer, в свою очередь, стал технической основой этой экосистемы, устранив искусственные барьеры между проектами и сделав повторное использование кода нормой, а не исключением.

В то же время развитие самого языка — введение строгой типизации, переход к исключительной модели обработки ошибок, появление нативных атрибутов, а также повышение производительности в PHP 7 и PHP 8 — оказало системное влияние на архитектурные практики. Фреймворки были вынуждены пересмотреть внутренние механизмы, сократить использование динамической «магии» и усилить контрактность публичных API. Это, в свою очередь, повысило предсказуемость, тестируемость и сопровождаемость кода.

В этом контексте Symfony утвердился как компонентная платформа, задающая технические ориентиры для всей экосистемы: его независимые компоненты стали стандартом для построения инфраструктурных слоёв. Laravel, сохраняя фокус на удобстве разработки и высокоуровневой выразительности, активно адаптировал PSR-стандарты и современные возможности PHP, одновременно расширяя своё влияние за

пределы HTTP-слоя за счет определенного набора инструментов. Оба фреймворка, несмотря на различия в философии, продемонстрировали высокую степень согласованности с современными архитектурными нормами.

Не менее показательными оказались и нереализованные инициативы. Отказ от стандартизации маршрутизации, отсутствие общего интерфейса для ORM, ограниченное применение PSR-14 [32] и сознательное сохранение PSR-11 в минималистичной форме свидетельствуют о зрелости сообщества: оно осознанно избегает навязывания унификации в тех областях, где архитектурное разнообразие является источником инноваций.

Таким образом, развитие PHP-фреймворков в 2015–2025 годах можно описать как стремление к стандартизации на базовом уровне при сохранении гибкости в проектировании прикладных решений. За это время фреймворки перестали быть замкнутыми системами и стали частями единой экосистемы, где важнее не то, как реализован компонент, а то, как он взаимодействует с другими — через чёткие и общие интерфейсы.

Несмотря на прежние опасения, что PHP устаревает, за последнее десятилетие язык не только сохранил свою популярность, но и значительно обновился. Открытая разработка, обсуждения в группе PHP-FIG и вклад сообществ Symfony и Laravel помогли экосистеме соответствовать современным требованиям: высокой производительности, строгой типизации, модульной структуре и интеграции с инструментами разработки и развёртывания.

В этом смысле 2015–2025 годы стали одним из самых важных этапов в истории PHP — временем, когда зрелая технология не просто выжила, а успешно трансформировалась под новые задачи.

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

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

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

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

[12] Laravel Team. Laravel — The PHP Framework for Web Artisans. — URL: https://laravel.com (дата обращения: 2025-02-01).

[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).

[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). Стандарт автозагрузки классов.

[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.

[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] Popov Nikita, PHP Core Team. PHP 8.0: Union Types, Named Arguments, Attributes, and more. — 2020. — URL: https://www.php.net/releases/8.0/en.php (дата обращения: 2024-11-20).

[40] Potencier Fabien. Symfony 1.0 Release Announcement. — 2007. — URL: https://symfony.com/blog/symfony-1-0-released (дата обращения: 2024-11-15). Официальная страница релиза Symfony 1.0.

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

[42] Potencier Fabien. Symfony 2.0 Release Announcement. — 2011. — URL: https://symfony.com/releases/2.0 (дата обращения: 2025-11-15). Официальная страница релиза Symfony 2.0.

[43] 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.

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

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

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

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

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

[49] 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).

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

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

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

[53] Symfony Project. New in Symfony 3.3: Service Autowiring. — URL: https://symfony.com/blog/new-in-symfony-3-3-service-autowiring (дата обращения: 2025-02-01).

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

[55] Symfony Project. Symfony Console Component. — 2009. — URL: https://symfony.com/doc/current/components/console.html (дата обращения: 2025-02-01).

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

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

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

[59] 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.

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

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

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

[63] 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.

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

[65] 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).

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

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