# Самостоятельная работа

# Сообщение и мастер-класс по эволюции стандартов PSR

## PHP-FIG и PSR

PHP-FIG (PHP Framework Interop Group) — организованная в 2009 году группа разработчиков, основная идея которой находить способы совместной работы, выделяя общие концепции в разработке проектов на PHP.

Участники PHP-FIG

ReactPHP, Composer, Laminas Project (переименованный Zend Framework), Yii framework, CakePHP, Slim, Joomla, Magento, Drupal, phpBB, phpDocumentor и другие.

PSR (PHP Standards Recommendations) — описывает общие концепции, которые уже были проверены и отработаны. Вероятно при создании PSR, группа PHP-FIG вдохновлялась Java Community Process, а первый стандарт был принят в 2010 году.

[Список PSR стандартов](https://www.php-fig.org/psr/) расширяется новыми, а сами стандарты делятся на категории:

Автозагрузка, Интерфейсы, HTTP и Стиль кодирования,

каждому из которых присваивается определенный статус:

Принят, Устаревший, Черновик и Заброшенный.

Далее мы рассмотрим принятые PSR стандарты по категориям

## Автозагрузка

* [PSR-0](https://www.php-fig.org/psr/psr-0/) — Autoloading Standard - Устарел  
  После релиза пространства имен в 2009 году, в 2010 году был опубликован первый стандарт, который стал революцией в решении проблем автозагрузки классов и стал первым шагом на пути объединения фреймворков — наличие общей структуры директорий.
* [PSR-4](https://www.php-fig.org/psr/psr-4/) — Autoloading Standard  
  Прогресс не стоит на месте и в конце 2013 года PHP-FIG публикуют новый стандарт автозагрузки классов. Он может использоваться в дополнение к PSR-0, а также любой другой спецификации автозагрузки. Стандарт также описывает, где размещать файлы, которые будут автоматически загружаться в соответствии со спецификацией. Данный стандарт решает некоторые проблемы/ограничения PSR-0 и используется по умолчанию в Composer.

## Интерфейсы

[PSR-3](https://www.php-fig.org/psr/psr-3/): Logger Interface  
Основная цель данного интерфейса – простым и универсальным способом стандартизировать реализацию логирования. К данному интерфейсу прилагается спецификация, которая описывает в каких случаях какой из методов рекомендуется использовать.  
Если Ваш проект нуждается в расширенном функционале, МОЖНО расширять данный интерфейс, но СЛЕДУЕТ сохранять совместимость с описанными в данном стандарте правилами. Это позволит сторонним библиотекам, применяемых при разработке приложения, использовать централизованную систему логирования.  
На сегодняшний день нет необходимости самостоятельно реализовывать данный интерфейс (разве что в целях обучения), так-как существует отличное решение [Monolog](https://github.com/Seldaek/monolog) для реализации логирования, которое используется во многих проектах.

[PSR-6](https://www.php-fig.org/psr/psr-6/): Caching Interface  
Кэширование широко используется для повышения производительности любого проекта.  
Кэширование также является одной из наиболее распространенных функций многих CMS, фреймворков и библиотек.   
Это привело к ситуации, когда многие библиотеки реализуют свои собственные системы кэширования с различными уровнями функциональности. Эти различия заставляют разработчиков изучать несколько систем, которые могут предоставлять или не предоставлять необходимую им функциональность. Кроме того, разработчики кеширующих библиотек сами сталкиваются с выбором между поддержкой только ограниченного числа платформ или созданием большого количества классов адаптеров.

[PSR-11](https://www.php-fig.org/psr/psr-11/): Container Interface  
Основная цель стандартизировать, как фреймворки и библиотеки будут использовать (DIC) контейнер для доступа к объектам и параметрам. Для этого был описан ContainerInterface.  
Спецификация PSR-11 не описывает то, как необходимо регистрировать зависимости в проекте, однако дает четкую рекомендацию как делать не нужно:  
Пользователи НЕ ДОЛЖНЫ передавать контейнер в объект, чтобы объект мог получить свои собственные зависимости. Это означает, что контейнер используется в качестве Service Locator, который обычно не рекомендуется использовать.  
Отсюда, возникает простой вопрос: «Как это вообще работает»?  
На самом деле все просто, на помощь приходит паттерн Factory, который возьмет на себя задачу создания объекта. А вот сам класс фабрики уже может принимать ContainerInterface и передавать в создаваемый объект необходимые зависимости.   
Данный подход использует middleware framework [Mezzio](https://docs.mezzio.dev/) (это бывший Zend Expressive), что позволяет соблюдать принципы SOLID и получить дополнительную гибкость при создании объектов.

[PSR-13](https://www.php-fig.org/psr/psr-13/): Hypermedia Links  
Не самый популярный стандарт, который предоставляет несколько интерфейсов, чтобы унифицировать общий формат hypermedia ссылок.   
В качестве примера, можно рассмотреть использование hypermedia ссылок в контексте HTML и в различных форматах API. При этом, если контекст использования ссылок в HTML понятен, то с API поможет разобраться статья "[Hypermedia — то без чего Ваше API не совсем REST](https://habr.com/ru/company/aligntechnology/blog/281206/)".  
Примеров использования данного стандарта не много: [Symfony Web Link](https://github.com/symfony/web-link) и [Html Model](https://github.com/Crell/HtmlModel).

[PSR-14](https://www.php-fig.org/psr/psr-14/): Event Dispatcher  
Целью этого PSR является создание общего механизма для диспетчеризации событий, чтобы библиотеки и компоненты могли свободно использоваться в различных приложениях и средах. Для этого предоставляется несколько интерфейсов  
Диспетчеризация событий — это распространенный и хорошо протестированный механизм, позволяющий разработчикам легко и последовательно расширять логику приложения. Детально данный стандарт хорошо описывает статья "[PSR-14 — главное событие в PHP](https://habr.com/ru/company/oleg-bunin/blog/450812/)".  
Чтобы попробовать в действии, предлагаю взглянуть на реализацию [Symfony Event Dispatcher](https://github.com/symfony/event-dispatcher), [YiiSoft Event Dispatcher](https://github.com/yiisoft/event-dispatcher) и [другие](https://github.com/search?l=PHP&o=desc&q=psr-14&s=stars&type=Repositories).

[PSR-16](https://www.php-fig.org/psr/psr-16/): Simple Cache  
Обратите внимание на PSR-6, это действительно «мощная» спецификация для реализации системы кеширования, однако в большинстве проектов такая реализация может оказаться избыточной.   
Поэтому был принят PSR-16. Этот более простой подход направлен на создание стандартизированного оптимизированного интерфейса для общих случаев.

## HTTP

Пожалуй, одной из самых сложных задач, которая нередко возникает является переиспользование кода между различными проектами. Если хорошо абстрагированные участки бизнес логики, некоторые компоненты и модули перенести возможно (с минимальными затратами), то с переносом более высокого уровня фреймворков (например контроллеров) возникают сложности.

Группа PHP-FIG пытается исправить данную проблему и предоставляет стандарты абстракции над HTTP.

[PSR-7](https://www.php-fig.org/psr/psr-7/): HTTP Message Interfaces  
Цель данного стандарта, предоставить общий набор интерфейсов для фреймворков, чтобы последние могли использовать одинаковые абстракции над Request и Response объектами. Это позволит разработчикам писать переиспользуемый, независимый от фреймворка код. Спецификация данного стандарта достаточно объемна

[PSR-15](https://www.php-fig.org/psr/psr-15/): HTTP Handlers  
Спецификация данного стандарта описывает интерфейсы для обработчиков HTTP-запросов и компонентов промежуточного программного обеспечения HTTP-сервера.  
Если не вдаваться во все тонкости, то по сути это возможность писать некие абстрактные контроллеры для последующего переиспользования между различными проектами.   
Middleware framework [Mezzio](https://github.com/mezzio/mezzio) (бывший Zend Expressie) отлично демонстрирует примеры реализации PSR-15.

[PSR-17](https://www.php-fig.org/psr/psr-17/): HTTP Factories  
PSR-17 описывает общий стандарт для фабрик, которые создают HTTP-объекты, совместимые с PSR-7.  
PSR-7 не содержит рекомендации о том, как создавать HTTP-объекты. Это может приводить к трудностям при необходимости их создания внутри компонентов, которые не привязаны к конкретной реализации PSR-7.  
Интерфейсы, описанные в этой спецификации, описывают методы, с помощью которых можно создавать PSR-7 объекты. Посмотреть пример использования PSR-17 можно в простой [реализации PSR-7](https://github.com/Nyholm/psr7).

[PSR-18](https://www.php-fig.org/psr/psr-18/): HTTP Client  
PSR-18 описывает общие интерфейсы для отправки PSR-7 HTTP-запросов и получения HTTP-ответов.   
Это может сделать библиотеки более пригодными для повторного использования, так как уменьшает количество зависимостей и снижает вероятность конфликтов версий.  
Также в спецификации указано, что HTTP-клиенты могут быть заменены согласно принципу подстановки Лисков. Это означает, что все клиенты ДОЛЖНЫ вести себя одинаково при отправке запроса.

## Стиль кодирования

* [PSR-1](https://www.php-fig.org/psr/psr-1/): Basic Coding Standard
* [PSR-2](https://www.php-fig.org/psr/psr-2/): Coding Style Guide Устарел
* [PSR-12](https://www.php-fig.org/psr/psr-12/): Extended Coding Style Guide

Описанные выше спецификации достаточно объемные, поэтому мы рассмотрим только базовые из PSR-1:

* Использование только тэгов <?php и <?=
* Только UTF-8 без BOM для php кода
* Не стоит мешать разный функционал в одном файле (1 файл = 1 класс)
* Пространство имен и классы должны следовать [~~PSR-0~~, PSR-4]
* Классы объявляются в `StudlyCase`
* Константы объявляются в ТАКОМ\_ВИДЕ
* Методы объявляются в `camelCase`