



Онлайн-образование

Проверить, идет ли запись!



Меня хорошо видно && слышно?

Ставьте +, если все хорошо

Напишите в чат, если есть проблемы

Composer



Дмитрий Кириллов

Технический директор

1С-Старт

@esteps_kirillov

Правила вебинара



Активно участвуем



Задаем вопрос в чат или голосом



Off-topic обсуждаем в Slack #php-2022-01 или #general



Вопросы вижу в чате, могу ответить не сразу

Маршрут вебинара

01. ООП в PHP



02. Основы Composer



03. Разработка пакетов



Цели вебинара | После занятия вы сможете

1

Более осознанно работать
с пространствами имён в PHP

2

Разрабатывать приложения
с использованием Composer

3

Создавать и публиковать
собственные пакеты

Введение

Вопросы на засыпку:

Типичные вопросы с собеседований:

- чем отличается composer install от composer update?
- чем отличается composer require от composer require --dev?
- нужно ли хранить composer.lock в git?

01. ООП в PHP

Современные тренды

Эти тренды пришли из Java:

- всё приложение состоит из классов*
- классы лежат в файлах (один класс — один файл)
- связанные между собой классы — это **пакеты**
(один пакет — один каталог)

Особое искусство — грамотно организовать эти классы и пакеты.

* а также интерфейсов и трейтов

Знакомьтесь: стажёр Вася

Васю только позавчера взяли на работу в Blizzard Activision – и попросили написать пару классов из World of Warcraft на PHP!



Разбиваем приложение на классы

Проблема:

- у Васи есть класс
- он хочет создать в нём объект другого класса
- Вася пишет вот такой код:

```
$leeroy = new Paladin();
```

Вопрос:

- почему у Васи ничего не работает?

Подключение PHP-файлов

Решение "в лоб":

- мы разделили наше приложение на несколько файлов
- чтобы работать с классами из других файлов,
нам нужно их сначала **подключить**
- для этого традиционно используются `include` / `require`

Вопрос:

- с какими проблемами мы рано или поздно столкнёмся?

Автозагрузка

В PHP есть специальный механизм автозагрузки:

- PHP встречает в коде **неизвестное** ему имя класса
- он понимает, что этот класс нужно откуда-то подключить
- для этого мы в самом начале нашего кода
должны "рассказать" PHP о том, в каких каталогах
искать эти классы

Автозагрузка

Простейший скрипт автозагрузки:

```
spl_autoload_register(  
    static fn($class) => include "{$class}.php"  
) ;
```

Структурируем приложение

Проблема:

- Вася прочитал книжку про DDD и решил структурировать свои классы
- в итоге у Васи получились два разных класса с одинаковым именем Request

Вопрос:

- с какими проблемами может столкнуться Вася?

Пространство имён (Namespace)

Проблема:

- у нас есть одноимённые классы
- аналогия — одноимённые файлы в разных каталогах
- если **одновременно** подключить эти классы,
возникнет конфликт

Пространство имён (Namespace)

Решение:

- в PHP для каждого файла можно объявить **пространство имён**
- это что-то вроде имени каталога, в котором лежит файл
- пространство имён обычно привязывают к имени приложения + структуре каталогов

Пространство имён (Namespace)

Структура пространства имён

```
namespace Otus\TestApp\Infrastructure\Http;
```

Имя поставщика (vendor)

Otus

Имя пакета/приложения

TestApp

Вложенные пространства имён

Infrastructure\Http

Пространство имён (Namespace)

Привязка к структуре каталогов

```
namespace Otus\TestApp\Infrastructure\Http;
```

Otus\TestApp

~/otus-test-app/src

Infrastructure\Http

~/otus-test-app/src/Infrastructure/Http

Более подробная информация – в стандарте **PSR-4**.

<https://www.php-fig.org/psr/psr-4/>

Пространство имён (Namespace)

Почему это удобно:

- каждый класс лежит в своём каталоге
- пространства имён соответствуют этим каталогам
- у каждого класса получается свой уникальный "адрес", из которого сразу понятно, к чему он относится
- исчезает проблема одноимённых классов

Пространство имён (Namespace)

Как указать пространство имён:

- добавляем в начало файла namespace
- дальше пишем код как обычно

```
<?php  
  
namespace Otus\TestApp\Infrastructure\Http;  
  
class Request  
{  
}
```

Пространство имён (Namespace)

Что получается в итоге:

- мы "телеортируем" класс в "параллельную вселенную"
- классы, которые живут в одной "вселенной",
могут обращаться друг к другу по имени
- все остальные должны указывать **абсолютное имя** класса

```
$request = new 0tus\TestApp\Domain\Model\Request();
```

Как обращаться к классам

Вариант 1. Использовать абсолютное имя

```
$request = new Otus\TestApp\Domain\Model\Request();
```

Плюсы:

- мы чётко видим, с каким классом работаем

Минусы:

- такой код тяжело читать и сопровождать

Как обращаться к классам

Вариант 2. Использовать синонимы

```
<?php  
  
namespace Otus\TestApp\Infrastructure\Ampq;  
  
use Otus\TestApp\Domain\Model\Request;  
  
...  
  
$request = new Request();
```

Как обращаться к классам

Особенности синонимов:

- это **не подключение** файлов
- это **не автозагрузка**
- мы просто говорим PHP, что если ему встретится имя Request — то на самом деле это Otus\TestApp...

Структура типичного класса

Включаем режим
строгой типизации

Указываем
пространство имён

Импортируем
нужные синонимы

Пишем тело класса

```
<?php
declare(strict_types=1);

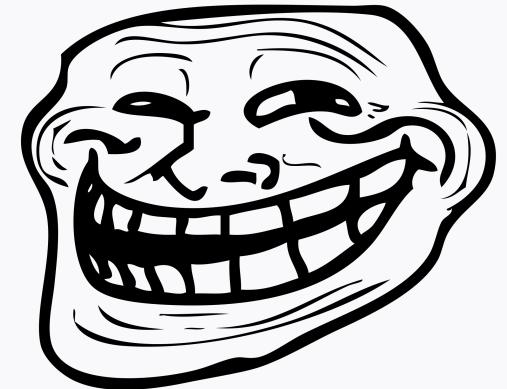
namespace Otus\TestApp\Infrastructure\Http;

use Otus\TestApp\Application\UseCase\RequestUseCase;
use Otus\TestApp\Domain\Model\Request;

class RequestController {
    ...
}
```

Мнение «иксперда»

*Все эти ваши файлы с классами –
это лишний код! Я лучше напишу
всё в одном файле на 2,000 строк,
и никому не придётся шастать
по каталогам!*



02. Основы Composer

Новая задача для стажёра Васи

Васе рассказали про несколько крутых библиотек и попросили подключить их к проекту с помощью Composer



Предпосылки

Проблемы при разработке серьёзных приложений:

- нужно писать довольно сложные автозагрузчики
- пока непонятно, как подключать **чужие библиотеки**
(ведь у них свои пространства имён, каталоги...)
- и совсем непонятно, что делать, если библиотекам
для работы нужны **трети библиотеки**, причём
конкретных версий

Что такое Composer

Это пакетный менеджер:

- мы подключаем Composer к нашему приложению
- затем мы говорим ему: "нам нужны вот такие библиотеки"
- он сам разбирается со всеми нужными зависимостями и конфликтами версий
- после этого он скачивает нужные **пакеты**
- и, наконец, автоматически генерирует код для автозагрузки (как библиотек, так и наших классов)

Установка Composer

Ключевые моменты:

- следуем инструкциям с официального сайта
- рекомендуется сделать файл Composer **исполняемым** и перенести его в один из каталогов PATH
- после этого можно перейти в каталог своего приложения и выполнять оттуда любые команды Composer

Практика: разработка приложения

- 1 Наша задача: создать новое веб-приложение и добавить в него нужные пакеты
- 2 Для этого потребуется разобраться с Composer
- 3 Критерии оценки: работоспособность и осмысленность кода



Тайминг: 10–15 минут

Шпаргалка по Composer

Инициализация приложения:

```
composer init
```

- в **названии (name)** рекомендуется использовать префикс пространства имён (otus/test-app)
- **тип (type)** — project
- остальные поля можно не заполнять
- после выполнения этой команды будет создан composer.json

Шпаргалка по Composer

Добавление пакетов:

```
composer require danielstjules/stringy
```

- имя пакета берём из документации к пакету
- Composer сам выберет подходящую версию
- по умолчанию пакет устанавливается в каталог vendor
- после установки Composer добавит классы пакета в автозагрузку и обновит composer.json / composer.lock

Шпаргалка по Composer

Добавление пакетов, нужных только во время разработки:

```
composer require --dev phpunit/phpunit
```

- эти пакеты будут установлены только локально
- во время деплоя приложения на прод Composer их проигнорирует

Шпаргалка по Composer

Каталог vendor:

- создаётся автоматически
- в момент установки пакета Composer копирует в него исходный код пакета
- у каждого проекта — отдельная копия этого каталога
- этот каталог принято добавлять в .gitignore

Шпаргалка по Composer

Файл composer.json:

- создаётся автоматически; дорабатывается вручную
- содержит информацию о **желаемых** версиях установленных пакетов (обычно – диапазоны)
- в нём также указываются настройки автозагрузки, необходимые расширения PHP и т.д.

Шпаргалка по Composer

Файл `composer.lock`:

- создаётся автоматически (и его лучше не редактировать)
- содержит информацию о **конкретных** версиях установленных пакетов
- этот файл принято класть под контроль версий, чтобы у всех разработчиков был **идентичный** набор пакетов

Шпаргалка по Composer

Настройка автозагрузки:

```
"autoload": {  
    "psr-4": {  
        "Otus\\Composer\\": "src/"  
    }  
}
```

- создаём каталог, в котором будут лежать наши классы
- этот каталог будет привязан к префиксу пространства имён
- вложенные каталоги должны совпадать с вложенностью пространства имён (**включая регистр!**)

Шпаргалка по Composer

Установка всего проекта / недостающих пакетов:

```
composer install
```

- эта команда обычно выполняется сразу после загрузки проекта (или его обновления) из git
- она устанавливает только те зависимости, которых в данный момент не хватает локально
- для её работы очень желателен файл composer.lock

Основы Composer

Шпаргалка по Composer

Установка проекта на прод:

```
composer install --no-dev
```

- Composer установит только те пакеты, которые указаны в разделе require

Шпаргалка по Composer

Обновление пакетов до новых версий:

```
composer update
```

- на выбор новой версии влияют ограничения в composer.json
- рекомендуется обновлять пакеты **поштучно**
- после успешного обновления Composer внесёт правки в composer.lock

Основы Composer

Шпаргалка по Composer

Ключевые отличия install от update:

- **install** **доустанавливает** пакеты,
чьи версии жёстко зафиксированы в composer.lock
- **update** **переустанавливает** пакеты, обновляя их версии
и обновляя composer.lock

Типичный набор команд для совместной работы над проектом:

```
git pull  
composer install
```

Основы Composer

Версии пакетов

Основы SEMVER

```
v1.2.15 // {major}.{minor}.{patch}
```

Major Новая версия несовместима с предыдущей

Minor Добавлен функционал, не ломающий совместимость

Patch Мелкие, незначительные правки

Основы Composer

Версии пакетов

Управление версиями в composer.json

1.2.0

Строго 1.2.0

1.2.*

Любой патч для версии 1.2

~1.2

1.2, 1.3, 1.4... (растёт только последняя цифра)

~1.2.5

1.2.5, 1.2.6... (растёт только последняя цифра)

^1.2.5

Минимум 1.2.5, но строго 1.x (не 2.x!)

>=1.0 <3.0

Минимум 1.0, но строго меньше 3.0

Как добавить пакет в приложение

Основные этапы:

- подключаем новый пакет:

```
composer require danielstjules/stringy
```

- у нас изменятся composer.json и composer.lock
- коммитим их в git
- нашим коллегам потребуется выполнить:

```
git pull  
composer install
```

Проверка знаний

Типичные вопросы с собеседований:

- чем отличается composer install от composer update?
- чем отличается composer require от composer require --dev?
- нужно ли хранить composer.lock в git?

03. Разработка пакетов

Разработка пакетов

Новая задача для стажёра Васи

Васю попросили написать
свою собственную библиотеку
для работы со строками,
чтобы другие разработчики
тоже могли ей пользоваться



Разработка пакетов

Вопросы на засыпку:

Когда есть смысл создать свой пакет?

- переиспользование кода в разных проектах
- библиотека для работы с публичным API

Практика: разработка пакета

- 1 Наша задача: создать свой собственный пакет для Composer
- 2 Этот пакет потом нужно подключить к существующему приложению
- 3 Критерии оценки: работоспособность и осмысленность кода



Тайминг: 10–15 минут

Как разработать свой пакет

Основные этапы:

- инициализируем пакет в Composer
- настраиваем PSR-4
- устанавливаем зависимости
- пишем код, тесты и документацию
- добавляем тэг с номером версии
- публикуем пакет

После публикации пакет можно подключать в других проектах.

Разработка пакетов

Как разработать свой пакет

Отличия от разработки приложения:

- при инициализации указываем **type = library**
- добавляем composer.lock в .gitignore
(не публикуем его в git)

Разработка пакетов

Документация

Основные разделы README.md:

- краткое описание пакета
- требования
- установка
- использование (с примерами)
- дополнительная информация (если требуется)

Публикация новой версии

Основные шаги:

- подготовить описание релиза
- создать тэг с новой версией (например, v1.2.2)
- опубликовать тэг на GitHub
- если вы используете Packagist под учёткой GitHub,
он автоматически подтянет новую версию

Разработка пакетов

Публикация новой версии

Тэги в git:

- привязываются к коммитам
- обычно создаются в основной ветке
- говорят о том, что начиная с данного коммита
у нас вышла новая версия

Разработка пакетов

Публикация новой версии

Пример добавления нового тэга:

```
git tag -a v1.0.1 -m "Bugfix"  
git push origin v1.0.1
```

Где публиковать свой пакет?

Open-source пакеты:

- публикуем на GitHub (публичный репозиторий)
- дублируем на Packagist.org

Коммерческие пакеты:

- публикуем на GitHub (приватный репозиторий)
- при необходимости дублируем на Packagist.com
- есть другие варианты размещения

Разработка пакетов

Проверка знаний

Вопросы:

- в каких случаях требуется хранить пакеты в приватном репозитории?
- придумайте ситуацию, когда нам потребуется изменить major-версию пакета (например, 2.0)?
- почему мы не храним composer.lock в git для пакетов?

Цели вебинара | Проверка достижения целей

1

Более осознанно работать
с пространствами имён в PHP

2

Разрабатывать приложения
с использованием Composer

3

Создавать и публиковать
собственные пакеты

Домашнее задание

- 1 Разработать свой пакет и опубликовать его на GitHub / packagist.org
- 2 Дополнительно прислать composer.json с примером подключения пакета к проекту
- 3 Детали в ЛК



Срок: желательно сдать до 01.03

Следующий вебинар

Тема: PHP WebServers



Четверг, 17 февраля, в 20:00



Ссылка на вебинар будет в ЛК за 15 минут



Материалы к занятию
в ЛК – можно
изучать



Обязательный
материал обозначен
красной лентой



Заполните, пожалуйста,
опрос о занятии по ссылке в чате

Спасибо за внимание!
Приходите на следующие вебинары



Дмитрий Кириллов

Технический директор

1С-Старт

@esteps_kirillov