**OOP(Object Oriented Programming)**

* Advance php features like design patterns,
  + OOPS,
  + Namespace,
  + Traits
* Object Oriented Programming
  + It is the core of any framework and if trainee is clear with OOPs they can easily jump or switch to any new framework and technology.
* Refence links
  + <https://www.youtube.com/watch?v=7GWwjn0bs5g&list=PLJ49JaW9k5klgfZ22BlgZiElIz3cz08ta&index=7>
* The fundamentals of OOPs
  + Traits
    - Traits is a special class introduce in php7 to overcome limitation of multiple inheritance
    - https://www.php.net/manual/en/language.oop5.traits.php
  + Namespace
    - Namespace is useful to understand the directory structure of any framework
    - https://www.sitepoint.com/php-53-namespaces-basics/
  + Standards
    - Standards will teach the trainee how to write the code?
    - <https://www.php-g.org/psr/>
    - Basic Coding Standard
    - Logger Interface
    - Autoloading Standard,
* **Composer**
  + Download composer.phar file and install one php library using php composer.phar
  + <https://getcomposer.org/>
  + Autoload that installed php libraray by doing practial.
  + Autoload understanding : <https://www.brainbell.com/php/auto-loading.html>
* **PHP design patterns**
  + Design patterns are necessary to understand core of most of the frameworks have code written using various design patterns. To understand the basics of design patter go through following links
  + <https://phptherightway.com/pages/Design-Patterns.html>
* **Reference**:
  + PHP design patterns
    - https://www.ibm.com/developerworks/library/os-php-designptrns <http://www.phptherightway.com/pages/DesignPatterns>.
    - html OOPS Basics
    - <https://www.tutorialspoint.com/php/php_object_oriented.htm>
    - Basics of Namespaces
    - <https://www.sitepoint.com/php-53-namespaces-basics/>
* **Type Hint**
  + With Type hinting we can specify the expected data type (arrays, objects, interface, etc.) for an argument in a function declaration. This practice can be most advantageous because it results in better code organization and improved error messages.
  + When we would like to force a function to get only arguments of the type array, we can put the keyword array in front of the argument name,
    - function functionName (array $argumentName)
    - {
    - //code
    - }
  + Type hinting can also be used to force a function to get an argument of type Object
    - class Car {
    - protected $driver;
    - // The constructor can only get Driver objects as arguments.
    - public function \_\_construct(Driver $driver)
    - {
    - $this -> driver = $driver;
    - }
    - }
    - class Driver {}

    - $driver1 = new Driver();
    - $car1 = new Car($driver1);
* **Namespace**
  + Namespace is useful to understand the directory structure of any framework
  + As the size of your PHP code library increases, the more likely you will accidentally reuse a function or class name that has been declared before
  + Namespaced code is defined using a single namespace keyword at the top of your PHP file
  + it must be the first command (with the exception of declare) and no non-PHP code or white-space can precede the command
  + <?php
  + // define this code in the MyProject namespace
  + namespace MyProject;
  + It is not possible to nest namespaces or define two or more namespaces for the same code block
  + However, you can define different namespaced code in the same file, e.g.
  + <?php
  + namespace MyProject1;
  + // PHP code for the MyProject1 namespace
  + namespace MyProject2;
  + **Sub-namespaces**
    - PHP allows you to define a hierarchy of namespace names so libraries can be sub-divided. Sub-namespaces are separated using a backslash (\) character
    - MyProject\SubName
    - MyProject\Database\MySQL
    - Example
      * In a file named lib1.php, we will define a constant, function, and class with the Html namespace:
      * namespace Html;
      * class Table {
      * public $title = "";
      * public $numRows = 0;
      * public function message() {
      * echo "<p>Table '{$this->title}' has {$this->numRows} rows.</p>";
      * }
      * }
      * $table = new Table();
      * $table->title = "My table";
      * $table->numRows = 5;
  + **Importing, Aliases, and Name Resolution**
    - Namespace Importing
      * Namespaces can be imported with the use operator,
      * use App\Lib1;
      * echo \App\Lib1\MYCONST . "\n";
      * echo \App\Lib1\MyFunction() . "\n";
      * echo \App\Lib1\MyClass::WhoAmI() . "\n";
    - Namespace Aliases
      * Namespace aliases are perhaps the most useful construct. Aliases allow us to reference long namespaces using a shorter name..
      * use App\Lib1 as L;
      * use App\Lib2\MyClass as Obj;
      * The first use statement defines AppLib1 as ‘L’. Any qualified names using ‘L’ will be translated to ‘AppLib1’ at compile-time
    - PHP Name Resolution Rules
      * PHP identifier names are resolved using the following namespace rules
      * Calls to fully-qualified functions, classes or constants are resolved at compile-time.
      * Unqualified and qualified names are translated according to the import rules
      * inside a namespace, all qualified names not already translated according to import rules have the current namespace prepended
      * Unqualified class names are translated according to current import rules and the full name is substituted for short imported name
      * Unqualified function calls within a namespace are resolved at run-time
      * Calls to unqualified or qualified class names are resolved at run-time.
  + **The namespace Keyword**
    - The namespace keyword can be used to explicitly reference an item within the current namespace or a sub-namespace. It is the namespace equivalent of self within classes:
    - Autoloading Namespaced Classes
      * One of the best time-saving features of PHP 5 is autoloading.
      * function \_\_autoload($class\_name) {
      * require\_once("classes/$class\_name.php");
      * }
      * the fully-qualified namespace and class name is passed to the \_\_autoload function
* **Traits**
  + Traits is a special class introduce in php7 to overcome limitation of multiple inheritance. It is widely used by us, please study it carefully
  + PHP implements a way to reuse code called Traits.
  + Traits are a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies
  + The semantics of the combination of Traits and classes is defined in a way which reduces complexity, and avoids the typical problems associated with multiple inheritance and Mixins
  + A Trait is similar to a class,
    - trait ezcReflectionReturnInfo {
    - function getReturnType() { /\*1\*/ }
    - function getReturnDescription() { /\*2\*/ }
    - }
    - class ezcReflectionMethod extends ReflectionMethod {
    - **use ezcReflectionReturnInfo;**
    - /\* ... \*/
    - }
  + Traits are declared with the trait keyword:
  + trait message1 {
  + public function msg1() {
  + echo "OOP is fun! ";
  + }
  + }
  + Sometimes you may want to inherit from more than one class. Since this is not possible in PHP and other single inheritance languages like it, we have something called traits
  + Multiple Traits
    - Multiple Traits can be inserted into a class by listing them in the use statement, separated by commas.
    - use Hello, World;
    - as classes can make use of traits, so can other traits. By using one or more traits in a trait definition, it can be composed partially or entirely of the members defined in those other traits.
    - trait Hello {
    - public function sayHello() {
    - echo 'Hello ';
    - }
    - }
    - trait World {
    - public function sayWorld() {
    - echo 'World!';
    - }
    - }
    - trait HelloWorld {
    - use Hello, World;
    - }
    - class MyHelloWorld {
    - use HelloWorld;
    - }
  + Abstract Trait Members
    - Traits support the use of abstract methods in order to impose requirements upon the exhibiting class. Public, protected, and private methods are supported
    - trait Hello {
    - public function sayHelloWorld() {
    - echo 'Hello'.$this->getWorld();
    - }
    - abstract public function getWorld();
    - }
  + Static Trait members
    - Traits can define static variables, static methods and static properties.
    - trait StaticExample {
    - public static function doSomething() {
    - return 'Doing something';
    - }
    - }m
    - Example::doSomething();
  + **Constants**
    - Traits can, as of PHP 8.2.0, also define constants.
    - trait ConstantsTrait {
    - public const FLAG\_MUTABLE = 1;
    - final public const FLAG\_IMMUTABLE = 5;
    - }
* **Standards**
  + Standards will teach the trainee how to write the code?
    - Basic Coding Standard
      * Files MUST use only <?php and <?= tags.
      * Files MUST use only UTF-8 without BOM for PHP code
      * Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side-effects (e.g. generate output, change
      * Namespaces and classes MUST follow an "autoloading" PSR
      * Class names MUST be declared in StudlyCaps
      * Class constants MUST be declared in all upper case with underscore separators.
      * Method names MUST be declared in camelCase
      * PHP Tags
        + PHP code MUST use the long <?php ?> tags or the short-echo <?= ?> tags; it MUST NOT use the other tag variations.
      * Character Encoding
        + PHP code MUST use only UTF-8 without BOM.
      * Side Effects
        + The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., merely from including the file
        + Side effects" include but are not limited to: generating output, explicit use of require or include
        + ini\_set('error\_reporting', E\_ALL);
      * Namespace and Class Names
        + Namespaces and classes MUST follow an "autoloading" PSR:
        + This means each class is in a file by itself, and is in a namespace of at least one level: a top-level vendor name.
        + Class names MUST be declared in StudlyCap
      * Class Constants, Properties, and Methods
        + The term "class" refers to all classes, interfaces, and traits.
      * Constants
        + Class constants MUST be declared in all upper case with underscore separators
      * Properties
        + This guide intentionally avoids any recommendation regarding the use of $StudlyCaps, $camelCase, or $under\_score property names.
        + Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.
      * Methods
        + Method names also declare with realted to functionality
        + Method names MUST be declared in camelCase().
    - Logger Interface
      * This document describes a common interface for logging libraries.
      * The main goal is to allow libraries to receive a Psr\Log\LoggerInterface object and write logs to it in a simple and universal way. Frameworks and CMSs that have custom needs MAY extend the interface for their own purpose, but SHOULD remain compatible with this document.
      * The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described
      * The word implementor in this document is to be interpreted as someone implementing the LoggerInterface in a log-related library or framework.
      * Every method accepts a string as the message, or an object with a \_\_toString() method. Implementors MAY have special handling for the passed objects. If that is not the case, implementors MUST cast it to a string
      * Placeholder names MUST correspond to keys in the context array.
      * Placeholder names MUST be delimited with a single opening brace { and a single closing brace }. There MUST NOT be any whitespace between the delimiters and the placeholder name.
      * Context
        + Every method accepts an array as context data. This is meant to hold any extraneous information that does not fit well in a string. The array can contain anything. Implementors MUST ensure they treat context data with as much lenience as possible. A given value in the context MUST NOT throw an exception nor raise any php error, warning or notice
        + Implementors MUST still verify that the 'exception' key is actually an Exception before using it as such, as it MAY contain anything.
      * Helper classes and interfaces
        + The Psr\Log\AbstractLogger class lets you implement the LoggerInterface very easily by extending it and implementing the generic log method.
        + Similarly, using the Psr\Log\LoggerTrait only requires you to implement the generic log method.
        + The Psr\Log\NullLogger is provided together with the interface. It MAY be used by users of the interface to provide a fall-back "black hole" implementation if no logger is given to them.
        + The Psr\Log\LoggerAwareInterface only contains a setLogger(LoggerInterface $logger) method and can be used by frameworks to auto-wire arbitrary instances with a logger.
        + The Psr\Log\LoggerAwareTrait trait can be used to implement the equivalent interface easily in any class. It gives you access to $this->logger
      * Psr\Log\LoggerInterface
        + public function emergency($message, array $context = array());
        + public function alert($message, array $context = array());
      * Psr\Log\LoggerAwareInterface
        + setLogger
      * Psr\Log\LogLevel
      * + class LogLevel
        + {
        + const EMERGENCY = 'emergency';
        + const ALERT = 'alert';
        + const CRITICAL = 'critical';
        + const ERROR = 'error';
        + const WARNING = 'warning';
        + const NOTICE = 'notice';
        + const INFO = 'info';
        + const DEBUG = 'debug';
        + }
    - **Autoloading Standard**
      * The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described,
      * This PSR describes a specification for autoloading classes from file paths. It is fully interoperable, and can be used in addition to any other autoloading specification, including PSR-0.
      * The term "class" refers to classes, interfaces, traits, and other similar structures.
      * A fully qualified class name has the following form:
        + \<NamespaceName>(\<SubNamespaceNames>)\*\<ClassName>
        + The fully qualified class name MUST have a top-level namespace name, also known as a "vendor namespace".
        + The fully qualified class name MAY have one or more sub-namespace names.
        + The fully qualified class name MUST have a terminating class name.
        + Underscores have no special meaning in any portion of the fully qualified class name.
        + Alphabetic characters in the fully qualified class name MAY be any combination of lower case and upper case.
        + All class names MUST be referenced in a case-sensitive fashion.
      * When loading a file that corresponds to a fully qualified class name
        + A contiguous series of one or more leading namespace and sub-namespace names, not including the leading namespace separator, in the fully qualified class name (a "namespace prefix") corresponds to at least one "base directory".
        + The contiguous sub-namespace names after the "namespace prefix" correspond to a subdirectory within a "base directory", in which the namespace separators represent directory separators. The subdirectory name MUST match the case of the sub-namespace names.
        + The terminating class name corresponds to a file name ending in .php. The file name MUST match the case of the terminating class name.
      * Autoloader implementations MUST NOT throw exceptions, MUST NOT raise errors of any level, and SHOULD NOT return a value.
      * The spl\_autoload\_register() function registers any number of autoloaders, enabling for classes and interfaces to be automatically loaded if they are currently not defined.
      * Any class-like construct may be autoloaded the same way. That includes classes, interfaces, traits, and enumerations
      * spl\_autoload\_register(function ($class\_name) {
      * include $class\_name . '.php';
      * });
      * Autoloading is the process of automatically loading PHP classes without explicitly loading them with the require(), require\_once(), include(), or include\_once() functions.
      * Each class must be defined in a separate file
      * Name your class files the same as your classes
      * An autoloader is a function that takes a class name as an argument and then includes the file that contains the corresponding class
      * As of PHP 7.2.0 the \_\_autoload() function has been deprecated and removed since PHP 8.0.0. Now it is recommended to use the spl\_autoload\_register for that purpose instead.
      * You can see all classes loaded without using the include or require keywords. Since PHP 5.3, you can use spl\_autoload\_register() with namespaces, which means that you can organize your project and autoload your PHP classes without the require or include keyword.
* **Composer**
  + Download composer.phar file and install one php library using php composer.phar
  + Autoload that installed php libraray by doing practial.
  + Autoload understanding
    - <https://www.brainbell.com/php/auto-loading.html>
  + **Basic usage**
    - composer.json: Project setup
      * To start using Composer in your project, all you need is a composer.json file
      * This file describes the dependencies of your project and may contain other metadata as well.
      * It typically should go in the top-most directory of your project/VCS repository.
      * **The require key**
        + The first thing you specify in composer.json is the require key. You are telling Composer which packages your project depends on.
        + Example  
          { "require": {

"monolog/monolog": "2.0.\*"

}

}

* + - * + As you can see, require takes an object that maps package names (e.g. monolog/monolog) to version constraints (e.g. 1.0.\*).
      * **Package names**
        + The package name consists of a vendor name and the project's name
        + Often these will be identical - the vendor name only exists to prevent naming clashes
        + For example, it would allow two different people to create a library named json. One might be named igorw/json while the other might be seldaek/json.
      * **Package version constraints**
        + version constraint 2.0.\*
        + This means any version in the 2.0 development branch, or any version that is greater than or equal to 2.0 and less than 2.1 (>=2.0 <2.1)
      * **Installing dependencies**
        + To initially install the defined dependencies for your project, you should run the update command
        + php composer.phar update
        + resolves all dependencies listed in your composer.json file and writes all of the packages and their exact versions to the composer.lock file, locking the project to those specific versions.
        + You should commit the composer.lock file to your project repo so that all people working on the project are locked to the same versions of dependencies
        + It then implicitly runs the install command. This will download the dependencies' files into the vendor directory in your project.
      * **Commit your composer.lock file to version control**
        + Committing this file to version control is important because it will cause anyone who sets up the project to use the exact same versions of the dependencies that you are using**.**
      * **Installing from composer.lock**
        + If there is already a composer.lock file in the project folder, it means either you ran the update command before, or someone else on the project ran the update command and committed the composer.lock file to the project
        + Either way, running install when a composer.lock file is present resolves and installs all dependencies that you listed in composer.json, but Composer uses the exact versions listed in composer.lock to ensure that the package versions are consistent for everyone working on your project
        + after fetching new changes from your VCS repository it is recommended to run a Composer install to make sure the vendor directory is up in sync with your composer.lock file.
        + **php composer.phar install**
      * **Updating dependencies to their latest versions**
        + As mentioned above, the composer.lock file prevents you from automatically getting the latest versions of your dependencies. To update to the latest versions, use the update command
        + php composer.phar update monolog/monolog [...]
      * **Packagist**
        + Packagist.org is the main Composer repository
        + A Composer repository is basically a package source: a place where you can get packages from.
        + Packagist aims to be the central repository that everybody uses
        + If you go to the Packagist.org website, you can browse and search for packages
        + Any open source project using Composer is recommended to publish their packages on Packagist. A library does not need to be on Packagist to be used by Composer, but it enables discovery and adoption by other developers more quickly
      * **Platform packages**
        + Composer has platform packages,which are virtual packages for things that are installed on the system but are not actually installable by Composer.
        + hhvm represents the version of the HHVM runtime and allows you to apply a constraint, e.g., ^2.3.
        + ext-<name> allows you to require PHP extensions (includes core extensions).
        + lib-<name> allows constraints to be made on versions of libraries used by PHP.
      * **Autoloading**
        + For libraries that specify autoload information, Composer generates a vendor/autoload.php file. You can include this file and start using the classes that those libraries provide without any extra work
        + require \_\_DIR\_\_ . '/vendor/autoload.php';
        + $log = new Monolog\Logger('name');
        + You can even add your own code to the autoloader by adding an autoload field to composer.json.
        + Example

{

"autoload": {

"psr-4": {"Acme\\": "src/"}

}

}

* + - * + Composer will register a PSR-4 autoloader for the Acme namespace.
        + After adding the autoload field, you have to re-run this command:

php composer.phar dump-autoload

* + - * + This command will re-generate the vendor/autoload.php file. See the dump-autoload section for more information.
        + Composer provides its own autoloader. If you don't want to use that one, you can include vendor/composer/autoload\_\*.php files, which return associative arrays allowing you to configure your own autoloader.
  + **Libraries**
    - will tell you how to make your library installable through Composer.
    - As soon as you have a composer.json in a directory, that directory is a package
    - When you add a require to a project, you are making a package that depends on other packages
    - In order to make that package installable you need to give it a name. You do this by adding the name property in composer.json

{

"name": "acme/hello-world",

"require": {

"monolog/monolog": "1.0.\*"

}

}

* + - In this case the project name is acme/hello-world, where acme is the vendor name.
    - Library Versioning
      * you will be maintaining your library using some sort of version control system like git, svn, hg or fossil

{

"version": "1.0.0"

}

* + - **Lock file**
    - For your library you may commit the composer.lock file if you want to. This can help your team to always test against the same dependency versions
    - If you do not want to commit the lock file, and you are using git, add it to the .gitignore.
    - **Publishing to a VCS**
      * The name is not needed in this case, since we don't want to publish the blog as a library. It is added here to clarify which composer.json is being described.

{

"name": "acme/blog",

"repositories": [

{

"type": "vcs",

"url": "https://github.com/username/hello-world"

}

],

"require": {

"acme/hello-world": "dev-master"

}

}

* + - **Publishing to packagist** 
      * Alright, so now you can publish packages. But specifying the VCS repository every time is cumbersome
      * The other thing that you may have noticed is that we did not specify a package repository for monolog/monolog. How did that work? The answer is Packagist.
      * Packagist is the main package repository for Composer, and it is enabled by default. Anything that is published on Packagist is available automatically through Composer.
  + **Command Line Interface**
    - You've already learned how to use the command-line interface to do some things
    - To get help from the command-line, call composer or composer list to see the complete list of commands,then --help combined with any of those can give you more information.
    - **Bash Completions**
      * To install bash completions you can run **composer completion bash > completion.bash.**
      * This will create a completion.bash file in the current directory.
      * Then execute source completion.bash to enable it in the current terminal session**.**
      * Move and rename the completion.bash file to /etc/bash\_completion.d/composer to make it load automatically in new terminals.
      * **Global Options**
      * **options are available with every command:**
        + --verbose (-v): Increase verbosity of messages.
        + --help (-h): Display help information.
        + --quiet (-q): Do not output any message.
        + --no-interaction (-n): Do not ask any interactive question.
        + --no-plugins: Disables plugins.
        + --no-scripts: Skips execution of scripts defined in composer.json.
        + --no-cache: Disables the use of the cache directory.
        + --working-dir (-d): If specified, use the given directory as working directory.
        + --profile: Display timing and memory usage information
        + --ansi: Force ANSI output.
        + --no-ansi: Disable ANSI output.
        + --version (-V): Display this application version
        + Process Exit Codes

0 ok

1 Generic/unknown error code

2 Dependency solving error code

* + - **Init**
      * we looked at how to create a composer.json by hand. There is also an init command available to do this.
      * php composer.phar init (composer init)
      * Options
        + --name: Name of the package.
        + --description: Description of the package.
        + --author: Author name of the package
        + --type: Type of package.
        + --homepage: Homepage of the package.
        + --require: Package to require with a version constraint. Should be in format foo/bar:1.0.0.
        + --require-dev: Development requirements, see --require.
        + --stability (-s): Value for the minimum-stability f--license (-l): License of package.
        + --repository: Provide one (or more) custom repositories
        + --autoload (-a): Add a PSR-4 autoload mapping to the composer.json. Automatically maps your package's namespace to the provided directory.
    - **install / i**
      * The install command reads the composer.json file from the current directory, resolves the dependencies, and installs them into vendor.
      * **php composer.phar install**
      * If there is a composer.lock file in the current directory, it will use the exact versions from there instead of resolving them. This ensures that everyone using the library will get the same versions of the dependencies
      * If there is no composer.lock file, Composer will create one after dependency resolution.
      * Options
        + --prefer-install: There are two ways of downloading a package: source and dist. Composer uses dist by default. If you pass --prefer-install=source (or --prefer-source) Composer will install from source if there is one. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly
        + --dry-run: If you want to run through an installation without actually installing a package, you can use --dry-run
        + --download-only: Download only, do not install packages.
        + --dev: Install packages listed in require-dev
        + --no-dev: Skip installing packages listed in require-dev
        + --no-autoloader: Skips autoloader generation.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + --audit: Run an audit after installation is complete.
        + --audit-format: Audit output format. Must be "table", "plain", "json", or "summary" (default).
        + -classmap-authoritative (-a)

Autoload classes from the classmap only. Implicitly enables --optimize-autoloader

* + - * + --apcu-autoloader: Use APCu to cache found/not-found classes.
        + --ignore-platform-reqs:

--ignore-platform-req (ignore a specific platform requirement(php, hhvm, lib-\* and ext-\*) and force the installation even if the local machine does not fulfill it.

* + - **update / u / upgrade**
      * In order to get the latest versions of the dependencies and to update the composer.lock file, you should use the update command.
      * This command is also aliased as upgrade as it does the same as upgrade does if you are thinking of apt-get or similar package managers
      * php composer.phar update (composer update)
      * This will resolve all dependencies of the project and write the exact versions into composer.lock.
      * If you only want to update a few packages and not all, you can list them as such
      * **php composer.phar update vendor/package vendor/package2**
      * You can also use wildcards to update a bunch of packages at once:
        + **php composer.phar update "vendor/\*"**
      * If you want to downgrade a package to a specific version without changing your composer.json you can use --with and provide a custom version constraint:
        + **php composer.phar update --with vendor/package:2.0.1**
      * **Options**
        + --prefer-install: There are two ways of downloading a package: source and dist. Composer uses dist by default. If you pass --prefer-install=source (or --prefer-source) Composer will install from source if there is one. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly
        + --dry-run: Simulate the command without actually doing anything.
        + --dev: Install packages listed in require-dev (this is the default behavior).
        + --no-dev: Skip installing packages listed in require-dev.
        + --no-install: Does not run the install step after updating the composer.lock file.
        + --no-audit: Does not run the audit steps after updating the composer.lock file.
        + --audit-format: Audit output format. Must be "table", "plain", "json", or "summary" (default).
        + --lock: Only updates the lock file hash to suppress warning about the lock file being out of date.
        + --with: Temporary version constraint to add
        + --no-autoloader: Skips autoloader generation.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + --with-dependencies (-w): Update also dependencies of packages in the argument list,
        + --with-all-dependencies (-W):
        + --ignore-platform-reqs: ignore all platform requirements
        + --ignore-platform-req: ignore a specific platform requirement(php, hhvm, lib-\* and ext-\*)
        + --prefer-stable: Prefer stable versions of dependencies
        + --prefer-lowest: Prefer lowest versions of dependencies.
        + --interactive: Interactive interface with autocompletion to select the packages to update.
        + --root-reqs: Restricts the update to your first degree dependencies.
    - **require / r**
      * The require command adds new packages to the composer.json file from the current directory. If no file exists one will be created on the fly.
      * **php composer.phar require**
      * After adding/changing the requirements, the modified requirements will be installed or updated.
      * If you do not want to choose requirements interactively, you can pass them to the command.
      * php composer.phar require "vendor/package:2.\*" vendor/package2:dev-master
      * If you do not specify a package, Composer will prompt you to search for a package, and given results, provide a list of matches to require.
      * **Options**
        + --dev: Add packages to require-dev.
        + --dry-run: Simulate the command without actually doing anything.
        + --prefer-install: There are two ways of downloading a package: source and dist.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + --no-update: Disables the automatic update of the dependencies (implies --no-install).
        + --no-install: Does not run the install step after updating the composer.lock file.
        + --no-audit: Does not run the audit steps after updating the composer.lock file
        + --audit-format: Audit output format. Must be "table", "plain", "json", or "summary" (default).
        + --update-no-dev: Run the dependency update with the --no-dev
        + --prefer-stable: Prefer stable versions of dependencies.
        + --prefer-lowest: Prefer lowest versions of dependencies.
        + --sort-packages: Keep packages sorted in composer.json.
        + --apcu-autoloader: Use APCu to cache found/not-found classes.
        + -apcu-autoloader-prefix: Use a custom prefix for the APCu autoloader cache. Implicitly enables --apcu-autoloader.
    - **Remove**
      * The remove command removes packages from the composer.json file from the current directory.
      * **php composer.phar remove vendor/package vendor/package2**
      * After removing the requirements, the modified requirements will be uninstalled.
      * Options
        + --dev: Remove packages from require-dev.
        + --dry-run: Simulate the command without actually doing anything.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + --no-update: Disables the automatic update of the dependencies
        + --no-install: Does not run the install step after updating the composer.lock file.
        + --no-audit: Does not run the audit steps after installation is complete. Also see
    - **Bump**
      * The bump command increases the lower limit of your composer.json requirements to the currently installed versions
      * This helps to ensure your dependencies do not accidentally get downgraded due to some other conflict, and can slightly improve dependency resolution performance as it limits the amount of package versions Composer has to look at.
      * **Options**
        + --dev-only: Only bump requirements in "require-dev".
        + --no-dev-only: Only bump requirements in "require".
        + --dry-run: Outputs the packages to bump, but will not execute anything
    - **Reinstall**
      * The reinstall command looks up installed packages by name, uninstalls them and reinstalls them.
      * This lets you do a clean install of a package if you messed with its files, or if you wish to change the installation type using --prefer-install
      * **Php composer.phar reinstall acme/foo acme/bar**
      * You can specify more than one package name to reinstall, or use a wildcard to select several packages at once:
      * **php composer.phar reinstall "acme/\*"**
      * **Options**
        + --prefer-install: There are two ways of downloading a package: source and dist. Composer uses dist by default. If you pass --prefer-install=source (or --prefer-source) Composer will install from source if there is one. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly
        + --no-autoloader: Skips autoloader generation.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + -optimize-autoloader (-o): Convert PSR-0/4 autoloading to classmap to get a faster autoloader. This is recommended especially for production, but can take a bit of time to run so it is currently not done by default.
        + --ignore-platform-reqs:
        + --ignore-platform-req:
    - **check-platform-reqs**
      * The check-platform-reqs command checks that your PHP and extensions versions match the platform requirements of the installed packages.
      * This can be used to verify that a production server has all the extensions needed to run a project after installing
      * **Options**
        + --lock: Checks requirements only from the lock file, not from installed packages.
        + --no-dev: Disables checking of require-dev packages requirements.
        + --format (-f): Format of the output: text (default) or json
    - **global** 
      * The global command allows you to run other commands like install, remove, require or update as if you were running them from the COMPOSER\_HOME directory.
      * The global command allows you to run other commands like install, remove, require or update as if you were running them from the COMPOSER\_HOME directory.
      * This can be used to install CLI utilities globally**.**
      * **php composer.phar global require friendsofphp/php-cs-fixer**
      * Now the php-cs-fixer binary is available globally. Make sure your global vendor binaries directory is in your $PATH environment variable, you can get its location with the following command :
      * php composer.phar global config bin-dir –absolute
      * If you wish to update the binary later on you can run a global update:
      * **php composer.phar global update**
    - **Search**
      * The search command allows you to search through the current project's package repositories. Usually this will be packagist. You pass it the terms you want to search for.
      * php composer.phar search monolog
      * You can also search for more than one term by passing multiple arguments.
      * **Options**
        + --only-name (-N): Search only in package names.
        + --only-vendor (-O): Search only for vendor / organization names, returns only "vendor" as a result.
        + --type (-t): Search for a specific package type.
        + --format (-f): Lets you pick between text (default) or json output format. Note that in the json, only the name and description keys are guaranteed to be present.
    - **show / info**
      * To list all of the available packages, you can use the show command.
      * **php composer.phar show**
      * **To filter the list you can pass a package mask using wildcards**
        + **php composer.phar show "monolog/\*"**
      * **Options**
        + --all: List all packages available in all your repositories
        + --installed (-i): List the packages that are installed
        + --locked: List the locked packages from composer.lock.
        + --platform (-p): List only platform packages
        + --available (-a): List available packages only.
        + --self (-s): List the root package info.
        + -name-only (-N): List package names only.
        + --path (-P): List package paths.
        + --tree (-t): List your dependencies as a tree. If you pass a package name it will show the dependency tree for that package.
        + --latest (-l): List all installed packages including their latest version.
        + --outdated (-o): Implies --latest, but this lists only packages that have a newer version available
        + --ignore: Ignore specified package(s). Use it with the --outdated option if you don't want to be informed about new versions of some packages
        + --no-dev: Filters dev dependencies from the package list.
        + --major-only (-M): Use with --latest or --outdated. Only shows packages that have major SemVer-compatible updates.
        + --minor-only (-m): Use with --latest or --outdated. Only shows packages that have minor SemVer-compatible updates.
        + --patch-only: Use with --latest or --outdated. Only shows packages that have patch-level SemVer-compatible updates.
        + --direct (-D): Restricts the list of packages to your direct dependencies.
        + --strict: Return a non-zero exit code when there are outdated packages.
    - **Outdated**
      * The outdated command shows a list of installed packages that have updates available, including their current and latest versions. This is basically an alias for composer show -lo
      * **The color coding is as such:**
        + **green (=): Dependency is in the latest version and is up to date**
        + **yellow (~): Dependency has a new version available that includes backwards compatibility breaks according to semver, so upgrade when you can but it may involve work.**
        + **red (!): Dependency has a new version that is semver-compatible and you should upgrade it.**
      * **Options**
        + --all (-a): Show all packages, not just outdated (alias for composer show --latest).
        + --direct (-D): Restricts the list of packages to your direct dependencies.
        + --strict: Returns non-zero exit code if any package is outdated.
        + --ignore: Ignore specified package(s). Use it if you don't want to be informed about new versions of some packages
        + --major-only (-M): Only shows packages that have major SemVer-compatible updates.
        + --minor-only (-m): Only shows packages that have minor SemVer-compatible updates.
        + --patch-only (-p): Only shows packages that have patch-level SemVer-compatible updates.
        + --format (-f): Lets you pick between text (default) or json output format.
        + --no-dev: Do not show outdated dev dependencies.
        + --locked: Shows updates for packages from the lock file, regardless of what is currently in vendor dir.
        + --ignore-platform-reqs: ignore all platform requirements (php, hhvm, lib-\* and ext-\*) and force the installation even if the local machine does not fulfill these.
        + --ignore-platform-req: ignore a specific platform requirement(php, hhvm, lib-\* and ext-\*) and force the installation even if the local machine does not fulfill it. Multiple requirements can be ignored via wildcard.
    - **browse / home**
      * The browse (aliased to home) opens a package's repository URL or homepage in your browser.
      * **Options**
        + --homepage (-H): Open the homepage instead of the repository URL.
        + --show (-s): Only show the homepage or repository URL.
    - **Suggests**
      * Lists all packages suggested by the currently installed set of packages. You can optionally pass one or multiple package names in the format of vendor/package to limit output to suggestions made by those packages only
      * Use the --by-package (default) or --by-suggestion flags to group the output by the package offering the suggestions or the suggested packages respectively.
      * If you only want a list of suggested package names, use --list.
      * Option
        + --by-package: Groups output by suggesting package
        + --by-suggestion: Groups output by suggested package.
        + --all: Show suggestions from all dependencies, including transitive ones (by default only direct dependencies' suggestions are shown).
        + --list: Show only list of suggested package names.
        + --no-dev: Excludes suggestions from require-dev packages.
    - **Fund**
      * Discover how to help fund the maintenance of your dependencies. This lists all funding links from the installed dependencies. Use --format=json to get machine-readable output.
      * Options
        + --format (-f): Lets you pick between text (default) or json output format.
    - **depends / why**
      * The depends command tells you which other packages depend on a certain package. As with installation require-dev relationships are only considered for the root package
      * php composer.phar depends doctrine/lexer

output

doctrine/annotations 1.13.3 requires doctrine/lexer (1.\*)

doctrine/common 2.13.3 requires doctrine/lexer (^1.0)

* + - * Options
        + --recursive (-r): Recursively resolves up to the root package
        + --tree (-t): Prints the results as a nested tree, implies -r.
      * **php composer.phar depends psr/log -t**
    - **prohibits / why-no**
      * The prohibits command tells you which packages are blocking a given package from being installed. Specify a version constraint to verify whether upgrades can be performed in your project, and if not why not.
      * **php composer.phar prohibits symfony/symfony 3.1**
      * **Options**
        + --recursive (-r): Recursively resolves up to the root package.
        + --tree (-t): Prints the results as a nested tree, implies -r.
    - **Validate**
      * You should always run the validate command before you commit your composer.json file, and before you tag a release. It will check if your composer.json is valid.
      * **php composer.phar validate**
      * **Options**
        + --no-check-all: Do not emit a warning if requirements in composer.json use unbound or overly strict version constraints.
        + --no-check-lock: Do not emit an error if composer.lock exists and is not up to date.
        + --no-check-publish: Do not emit an error if composer.json is unsuitable for publishing as a package on Packagist but is otherwise valid
        + --with-dependencies: Also validate the composer.json of all installed dependencies.
        + --strict: Return a non-zero exit code for warnings as well as errors.
    - **Status**
      * If you often need to modify the code of your dependencies and they are installed from source, the status command allows you to check if you have local changes in any of them.
      * **php composer.phar status**
      * With the --verbose option you get some more information about what was changed:
      * **php composer.phar status -v**
    - **self-update / selfupdate**
      * To update Composer itself to the latest version, run the self-update command. It will replace your composer.phar with the latest version.
      * **php composer.phar self-update**
      * If you would like to instead update to a specific release specify it:
      * **php composer.phar self-update 2.4.0-RC1**
      * **Options**
        + --rollback (-r): Rollback to the last version you had installed.
        + --clean-backups: Delete old backups during an update. This makes the current version of Composer the only backup available after the update.
        + --no-progress: Do not output download progress.
        + --update-keys: Prompt user for a key update.
        + --stable: Force an update to the stable channel.
        + --preview: Force an update to the preview channel.
        + --snapshot: Force an update to the snapshot channel.
        + --1: Force an update to the stable channel, but only use 1.x versions
        + --2: Force an update to the stable channel, but only use 2.x versions
        + --set-channel-only: Only store the channel as the default one and then exit
    - **Config**
      * the config command allows you to edit Composer config settings and repositories in either the local composer.json file or the global config.json file.
      * Additionally it lets you edit most properties in the local composer.json
    - **Usage**
      * config [options] [setting-key] [setting-value1] ... [setting-valueN]
      * setting-key is a configuration option name and setting-value1 is a configuration value. For settings that can take an array of values (like github-protocols), multiple setting-value arguments are allowed
      * You can also edit the values of the following properties:
      * description, homepage, keywords, license, minimum-stability, name, prefer-stable, type and version
      * **Options** 
        + --global (-g): Operate on the global config file located at $COMPOSER\_HOME/config.json by default. Without this option, this command affects the local composer.json file or a file specified by --file.
        + --editor (-e): Open the local composer.json file using in a text editor as defined by the EDITOR env variable. With the --global option, this opens the global config file.
        + --auth (-a): Affect auth config file (only used for --editor).
        + --unset: Remove the configuration element named by setting-key.
        + --list (-l): Show the list of current config variables. With the --global option this lists the global configuration only.
        + --file="..." (-f): Operate on a specific file instead of composer.json. Note that this cannot be used in conjunction with the --global option.
        + --absolute: Returns absolute paths when fetching \*-dir config values instead of relative
        + --json: JSON decode the setting value, to be used with extra.\* keys
        + --merge: Merge the setting value with the current value, to be used with extra.\* keys in combination with --json.
        + --append: When adding a repository, append it (lowest priority) to the existing ones instead of prepending it (highest priority).
        + --source: Display where the config value is loaded from
    - **Modifying Repositories**
      * In addition to modifying the config section, the config command also supports making changes to the repositories section by using it the following way:
      * **php composer.phar config repositories.foo vcs** [**https://github.com/foo/bar**](https://github.com/foo/bar)
    - **Modifying Extra Values**
      * In addition to modifying the config section, the config command also supports making changes to the extra section by using it the following way:
      * **php composer.phar config extra.foo.bar value**
    - **create-project**
      * You can use Composer to create new projects from an existing package
      * You can deploy application packages.
      * You can check out any package and start developing on patches
      * Projects with multiple developers can use this feature to bootstrap the initial application for development.
      * To create a new project using Composer you can use the create-project command. Pass it a package name, and the directory to create the project in. You can also provide a version as a third argument, otherwise the latest version is used.
      * php composer.phar create-project doctrine/orm path "2.2.\*"
      * Options
        + --stability (-s): Minimum stability of package. Defaults to stable
        + --prefer-install: There are two ways of downloading a package: source and dist. Composer uses dist by default. If you pass --prefer-install=source (or --prefer-source) Composer will install from source if there is one. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly.
        + --repository: Provide a custom repository to search for the package, which will be used instead of packagist.
        + --add-repository: Add the custom repository in the composer.json. If a lock file is present, it will be deleted and an update will be run instead of an install.
        + --dev: Install packages listed in require-dev.
        + --no-dev: Disables installation of require-dev packages.
        + --no-scripts: Disables the execution of the scripts defined in the root package.
        + --no-progress: Removes the progress display that can mess with some terminals or scripts which don't handle backspace characters.
        + --no-secure-http: Disable the secure-http config option temporarily while installing the root package. Use at your own risk. Using this flag is a bad idea.
        + --remove-vcs: Force-remove the VCS metadata without prompting.
        + --no-install: Disables installation of the vendors.
        + --ask: Ask the user to provide a target directory for the new project.
    - **dump-autoload / dumpautoload**
      * If you need to update the autoloader because of new classes in a classmap package for example, you can use dump-autoload to do that without having to go through an install or update.
      * it can dump an optimized autoloader that converts PSR-0/4 packages into classmap ones for performance reasons
      * large applications with many classes, the autoloader can take up a substantial portion of every request's time. Using classmaps for everything is less convenient in development, but using this option you can still use PSR-0/4 for convenience and classmaps for performance.
      * **Options**
        + --optimize (-o): Convert PSR-0/4 autoloading to classmap to get a faster autoloader.
        + --classmap-authoritative (-a): Autoload classes from the classmap only. Implicitly enables --optimize.
        + --apcu: Use APCu to cache found/not-found classes.
        + --apcu-prefix: Use a custom prefix for the APCu autoloader cache. Implicitly enables --apcu.
        + --no-dev: Disables autoload-dev rules. Composer will by default infer this automatically according to the last install or update --no-dev state.
        + --strict-psr: Return a failed status code (1) if PSR-4 or PSR-0 mapping errors are present. Requires --optimize to work.
    - **clear-cache / clearcache / cc**
      * Deletes all content from Composer's cache directories.
      * Options
      * --gc: Only run garbage collection, not a full cache clear
    - **Licenses**
      * Lists the name, version and license of every package installed. Use --format=json to get machine-readable output.
      * Options
        + -format: Format of the output: text, json or summary (default: "text")
        + --no-dev: Remove dev dependencies from the output
    - **run-script / run**
      * --timeout: Set the script timeout in seconds, or 0 for no timeout.
      * --dev: Sets the dev mode.
      * --no-dev: Disable dev mode.
      * --list (-l): List user defined scripts.
    - Exec
      * Executes a vendored binary/script. You can execute any command and this will ensure that the Composer bin-dir is pushed on your PATH before the command runs.
    - **Diagnose**
      * If you think you found a bug, or something is behaving strangely, you might want to run the diagnose command to perform automated checks for many common problems.
      * php composer.phar diagnose
    - **archive**
      * This command is used to generate a zip/tar archive for a given package in a given version. It can also be used to archive your entire project without excluded/ignored files.
      * php composer.phar archive vendor/package 2.0.21 --format=zip
      * Options
        + --format (-f): Format of the resulting archive: tar, tar.gz, tar.bz2 or zip (default: "tar").
        + --dir: Write the archive to this directory (default: ".")
        + --file: Write the archive with the given file name.
    - **Audit**
      * This command is used to audit the packages you have installed for possible security issues. It checks for and lists security vulnerability advisories according to the Packagist.org api.
      * The audit command returns the amount of vulnerabilities found. 0 if successful, and up to 255 otherwise.
      * php composer.phar audit
      * Options
        + --no-dev: Disables auditing of require-dev packages.
        + --format (-f): Audit output format. Must be "table" (default), "plain", "json", or "summary".
        + --locked: Audit packages from the lock file, regardless of what is currently in vendor dir.
    - **Help**
      * To get more information about a certain command, you can use help.
      * php composer.phar help install
    - **Environment variables**
      * You can set a number of environment variables that override certain settings. Whenever possible it is recommended to specify these settings in the config section of composer.json instead.
      * COMPOSER
        + By setting the COMPOSER env variable it is possible to set the filename of composer.json to something else
        + COMPOSER=composer-other.json php composer.phar install
        + The generated lock file will use the same name: composer-other.lock in this example.
      * COMPOSER\_ALLOW\_SUPERUSER
        + If set to 1, this env disables the warning about running commands as root/super user. It also disables automatic clearing of sudo sessions, so you should really only set this if you use Composer as a super user at all times like in docker containers.
      * COMPOSER\_ALLOW\_XDEBUG
        + If set to 1, this env allows running Composer when the Xdebug extension is enabled, without restarting PHP without it.
      * COMPOSER\_AUTH
        + The COMPOSER\_AUTH var allows you to set up authentication as an environment variable
      * COMPOSER\_BIN\_DIR
        + By setting this option you can change the bin (Vendor Binaries) directory to something other than vendor/bin
      * COMPOSER\_CACHE\_DIR
        + the COMPOSER\_CACHE\_DIR var allows you to change the Composer cache directory, which is also configurable via the cache-dir option.
      * COMPOSER\_CAFILE
        + By setting this environmental value, you can set a path to a certificate bundle file to be used during SSL/TLS peer verification.
      * COMPOSER\_DISABLE\_XDEBUG\_WARN
      * COMPOSER\_DISCARD\_CHANGES
      * COMPOSER\_HOME
        + The COMPOSER\_HOME var allows you to change the Composer home directory. This is a hidden, global (per-user on the machine) directory that is shared between all projects
        + Use composer config --global home to see the location of the home directory.
      * COMPOSER\_HOME/config.json
      * COMPOSER\_HTACCESS\_PROTECT
        + Defaults to 1. If set to 0, Composer will not create .htaccess files in the Composer home, cache, and data directories.
      * COMPOSER\_MEMORY\_LIMIT
        + If set, the value is used as php's memory\_limit.
      * COMPOSER\_MIRROR\_PATH\_REPOS
        + If set to 1, this env changes the default path repository strategy to mirror instead of symlink. As it is the default strategy being set it can still be overwritten by repository options.
      * COMPOSER\_NO\_INTERACTION
      * COMPOSER\_PROCESS\_TIMEOUT
      * COMPOSER\_ROOT\_VERSION
      * COMPOSER\_VENDOR\_DIR
      * COMPOSER\_RUNTIME\_ENV
      * HTTP\_PROXY
        + If you are using Composer from behind an HTTP proxy, you can use the standard http\_proxy or HTTP\_PROXY env vars.
      * COMPOSER\_MAX\_PARALLEL\_HTTP
        + Set to an integer to configure how many files can be downloaded in parallel. This defaults to 12 and must be between 1 and 50. If your proxy has issues with concurrency maybe you want to lower this. Increasing it should generally not result in performance gains.
      * HTTP\_PROXY\_REQUEST\_FULLURI
        + If you use a proxy, but it does not support the request\_fulluri flag, then you should set this env var to false or 0 to prevent Composer from setting the request\_fulluri option
      * HTTPS\_PROXY\_REQUEST\_FULLURI
      * COMPOSER\_SELF\_UPDATE\_TARGET
        + If set, makes the self-update command write the new Composer phar file into that path instead of overwriting itself. Useful for updating Composer on a read-only filesystem.
      * no\_proxy or NO\_PROXY
      * COMPOSER\_DISABLE\_NETWORK
        + If set to 1, disables network access (best effort). This can be used for debugging or to run Composer on a plane or a starship with poor connectivity.
      * COMPOSER\_DEBUG\_EVENTS
      * COMPOSER\_NO\_AUDIT
      * COMPOSER\_NO\_DEV
      * COMPOSER\_PREFER\_STABLE
      * COMPOSER\_PREFER\_LOWEST
      * COMPOSER\_IGNORE\_PLATFORM\_REQ or COMPOSER\_IGNORE\_PLATFORM\_REQS
        + COMPOSER\_IGNORE\_PLATFORM\_REQS set to 1, it is the equivalent of passing the --ignore-platform-reqs argument. Otherwise, specifying a comma separated list in COMPOSER\_IGNORE\_PLATFORM\_REQ will ignore those specific requirements
  + **The composer.json schema** 
    - explain all of the fields available in composer.json.
    - JSON schema
      * We have a JSON schema that documents the format and can also be used to validate your composer.json. In fact, it is used by the validate command.
    - Root Package
      * The root package is the package defined by the composer.json at the root of your project. It is the main composer.json that defines your project requirements.
    - Properties
      * Name
        + The name of the package. It consists of vendor name and project name, separated by /
        + monolog/monolog or igorw/event-source
        + The name must be lowercase and consist of words separated by -, . or \_. The complete name should match ^[a-z0-9]([\_.-]?[a-z0-9]+)\*/[a-z0-9](([\_.]?|-{0,2})[a-z0-9]+)\*$.
        + The name property is required for published packages
      * Description
        + A short description of the package. Usually this is one line long.
        + Required for published packages
      * Version
        + The version of the package. In most cases this is not required and should be omitted
        + This must follow the format of X.Y.Z or vX.Y.Z with an optional suffix of -dev, -patch (-p), -alpha (-a), -beta (-b) or -RC.
        + 1.0.0
        + 1.0.2
        + 1.0.0-dev
        + 1.0.0-alpha3
        + Optional if the package repository can infer the version from somewhere, such as the VCS tag name in the VCS repository. In that case it is also recommended to omit
        + Packagist uses VCS repositories, so the statement above is very much true for Packagist as well. Specifying the version yourself will most likely end up creating problems at some point due to human error.
      * Type
        + The type of the package. It defaults to library.
        + Package types are used for custom installation logic
        + If you have a package that needs some special logic, you can define a custom type
        + This could be a symfony-bundle, a wordpress-plugin or a typo3-cms-extension.
        + Composer supports four types:

Library

This is the default. It will copy the files to vendor.

Project

This denotes a project rather than a library.

Metapackage

An empty package that contains requirements and will trigger their installation, but contains no files and will not write anything to the filesystem. As such, it does not require a dist or source key to be installable.

composer-plugin

package of type composer-plugin may provide an installer for other packages that have a custom type.

Only use a custom type if you need custom logic during installation. It is recommended to omit this field and have it default to library.

* + - * Keywords
        + An array of keywords that the package is related to. These can be used for searching and filtering
        + Logging
        + Events
        + Database
        + Redis
        + Templating
        + Some special keywords trigger composer require without the --dev option to prompt users if they would like to add these packages to require-dev instead of require. These are: dev, testing, static analysis.
      * Homepage
        + A URL to the website of the project.
        + Optional
      * Time
        + Release date of the version.
        + Must be in YYYY-MM-DD or YYYY-MM-DD HH:MM:SS format
      * License
        + The license of the package. This can be either a string or an array of strings.
        + The recommended notation for the most common licenses is (alphabetical):
        + Apache-2.0
        + BSD-2-Clause
        + BSD-3-Clause
        + BSD-4-Clause
        + GPL-2.0-only / GPL-2.0-or-later
        + LGPL-2.1-only / LGPL-2.1-or-later
        + MIT
        + {
        + "license": "MIT"
        + } {
        + "license": [
        + "LGPL-2.1-only",
        + "GPL-3.0-or-later"
        + ]
        + }
        + "license": "(LGPL-2.1-only or GPL-3.0-or-later)"
        + Similarly, when multiple licenses need to be applied ("conjunctive license"), they should be separated with
        + "and" and enclosed in parentheses.
      * Authors
        + The authors of the package. This is an array of objects.
        + Each author object can have following properties:
        + name: The author's name. Usually their real name
        + email: The author's email address.
        + homepage: URL to the author's website.
        + role: The author's role in the project (e.g. developer or translator)
    - support
      * Various information to get support about the project.
      * Support information includes the following:
      * email: Email address for support.
      * issues: URL to the issue tracker.
      * forum: URL to the forum.
      * irc: IRC channel for support, as irc://server/channel.
      * source: URL to browse or download the sources.
      * docs: URL to the documentation.
      * rss: URL to the RSS feed.
      * chat: URL to the chat channel
    - funding
      * A list of URLs to provide funding to the package authors for maintenance and development of new functionality.
      * type: The type of funding, or the platform through which funding can be provided, e.g. patreon, opencollective, tidelift or github.
      * url: URL to a website with details, and a way to fund the package.
    - Package links
      * All of the following take an object which maps package names to versions of the package via version constraints
    - Require
      * Map of packages required by this package. The package will not be installed unless those requirements can be met.
    - require-dev
      * Map of packages required for developing this package, or running tests, etc. The dev requirements of the root package are installed by default.
    - Conflict
      * Map of packages that conflict with this version of this package. They will not be allowed to be installed together with your package.
      * Note that when specifying ranges like <1.0 >=1.1 in a conflict link, this will state a conflict with all versions that are less than 1.0 and equal or newer than 1.1 at the same time
    - Replace
      * Map of packages that are replaced by this package. This allows you to fork a package, publish it under a different name with its own version numbers,
      * while packages requiring the original package continue to work with your fork because it replaces the original package.
      * This is also useful for packages that contain sub-packages, for example the main symfony/symfony package contains all the Symfony Components which are also available as individual packages. If you require the main package it will automatically fulfill any requirement of one of the individual components, since it replaces them.
    - Provide
      * Map of packages that are provided by this package. This is mostly useful for implementations of common interfaces. A package could depend on some virtual package e.g. psr/logger-implementation, any library that implements this logger interface would list it in provide
    - Suggest
      * Suggested packages that can enhance or work well with this package. These are informational and are displayed after the package is installed, to give your users a hint that they could add more packages, even though they are not strictly required.
      * suggest{}
    - autoload
      * Autoload mapping for a PHP autoloader
      * PSR-4 and PSR-0 autoloading, classmap generation and files includes are supported.
      * PSR-4
        + Under the psr-4 key you define a mapping from namespaces to paths, relative to the package root. When autoloading a class like Foo\\Bar\\Baz a namespace prefix Foo\\ pointing to a directory src/ means that the autoloader will look for a file named src/Bar/Baz.php and include it if present.
        + Namespace prefixes must end in \\ to avoid conflicts between similar prefixes
        + "autoload": {
        + "psr-4": {
        + "Monolog\\": "src/",
        + "Vendor\\Namespace\\": ""
        + }
        + }
      * PSR-0
        + Under the psr-0 key you define a mapping from namespaces to paths, relative to the package root. Note that this also supports the PEAR-style non-namespaced convention.
        + "psr-0": { "UniqueGlobalClass": "" }
        + If you want to have a fallback directory where any namespace can be, you can use an empty prefix like
        + "psr-0": { "": "src/" }
    - Classmap
      * The classmap references are all combined, during install/update, into a single key => value array which may be found in the generated file vendor/composer/autoload\_classmap.php
      * autoload": {
      * "classmap": ["src/", "lib/", "Something.php"]
      * }
    - Files
      * If you want to require certain files explicitly on every request then you can use the files autoloading mechanism. This is useful if your package includes PHP functions that cannot be autoloaded by PHP
      * "autoload": {
      * "files": ["src/MyLibrary/functions.php"]
      * }
      * Files autoload rules are included whenever vendor/autoload.php is included, right after the autoloader is registered. The order of inclusion depends on package dependencies so that if package A depends on B, files in package B will be included first to ensure package B is fully initialized and ready to be used when files from package A are included.
    - Exclude files from classmaps
      * If you want to exclude some files or folders from the classmap you can use the exclude-from-classmap property. This might be useful to exclude test classes in your live environment, for example, as those will be skipped from the classmap even when building an optimized autoloader.
      * "autoload": {
      * "exclude-from-classmap": ["/Tests/", "/test/", "/tests/"]
      * }
    - Optimizing the autoloader
      * The autoloader can have quite a substantial impact on your request time (50-100ms per request in large frameworks using a lot of classes).
    - autoload-dev
      * This section allows defining autoload rules for development purposes.
      * Classes needed to run the test suite should not be included in the main autoload rules to avoid polluting the autoloader in production and when other people use your package as a dependency.
      * autoload-dev": {
      * "psr-4": { "MyLibrary\\Tests\\": "tests/" }
      * }
    - include-path
      * This is only present to support legacy projects, and all new code should preferably use autoloading. As such it is a deprecated practice, but the feature itself will not likely disappear from Composer
      * "include-path": ["lib/"]
    - target-dir
      * Defines the installation target.
      * In case the package root is below the namespace declaration you cannot autoload properly. target-dir solves this problem.
      * autoload": {
      * "psr-0": { "Symfony\\Component\\Yaml\\": "" }
      * },
      * "target-dir": "Symfony/Component/Yaml"
    - minimum-stability
      * This defines the default behavior for filtering packages by stability. This defaults to stable, so if you rely on a dev package, you should specify it in your file to avoid surprises.
    - prefer-stable
      * When this is enabled, Composer will prefer more stable packages over unstable ones when finding compatible stable packages is possible. If you require a dev version or only alphas are available for a package, those will still be selected granted that the minimum-stability allows for it
      * "prefer-stable": true to enable.
    - Repositories
      * Custom package repositories to use.
      * By default Composer only uses the packagist repository. By specifying repositories you can get packages from elsewhere.
      * Repositories are not resolved recursively. You can only add them to your main composer.json. Repository declarations of dependencies' composer.jsons are ignored
      * The following repository types are supported:
      * Composer
        + Composer repository is a packages.json file served via the network (HTTP, FTP, SSH), that contains a list of composer.json objects with additional dist and/or source information. The packages.json file is loaded using a PHP stream. You can set extra options on that stream using the options parameter.
      * Vcs
        + The version control system repository can fetch packages from git, svn, fossil and hg repositories.
      * package
        + If you depend on a project that does not have any support for Composer whatsoever you can define the package inline using a package repository. You basically inline the composer.json object.
    - Config
      * A set of configuration options. It is only used for projects. See Config for a description of each individual option.
    - Scripts
      * Composer allows you to hook into various parts of the installation process through the use of scripts.
    - Scripts
      * Composer allows you to hook into various parts of the installation process through the use of scripts.
    - Extra
      * Arbitrary extra data for consumption by scripts.
    - Bin
      * A set of files that should be treated as binaries and made available into the bin-dir
    - Archive
      * A set of options for creating package archives
      * name: Allows configuring base name for archive. By default (if not configured, and --file is not passed as command-line argument), preg\_replace('#[^a-z0-9-\_]#i', '-', name) is used.
    - Abandoned
      * Indicates whether this package has been abandoned.
      * It can be boolean or a package name/URL pointing to a recommended alternative.
      * "abandoned": true
      * Then composer show -s will give you versions : \* dev-latest-testing.
  + **Repositories**
    - will explain the concept of packages and repositories, what kinds of repositories are available, and how they work
    - Package
      * Composer is a dependency manager. It installs packages locally. A package is essentially a directory containing something. In this case it is PHP code, but in theory it could be anything. And it contains a package description which has a name and a version
      * the name and the version, there is useful metadata
      * Dist: The dist is a packaged version of the package data. Usually a released version, usually a stable release
      * Source: The source is used for development. This will usually originate from a source code repository, such as git. You can fetch this when you want to modify the downloaded package.
    - Repository
      * A repository is a package source. It's a list of packages/versions. Composer will look in all your repositories to find the packages your project requires.
      * By default, only the Packagist.org repository is registered in Composer. You can add more repositories to your project by declaring them in composer.json.
      * Repositories are only available to the root package and the repositories defined in your dependencies will not be loaded
    - Types
      * Composer
        + The main repository type is the composer repository
        + It uses a single packages.json file that contains all of the package metadata
        + repositories": [
        + {
        + "type": "composer",
        + "url": "https://example.org"
        + }
      * Packages
        + The only required field is packages
        + packages": {
        + "vendor/package-name": {
        + "dev-master": { @composer.json },
        + "1.0.x-dev": { @composer.json },
        + "0.0.1": { @composer.json },
        + "1.0.0": { @composer.json }
        + }
        + The @composer.json marker would be the contents of the composer.json from that package version including as a minimum
      * notify-batch
        + The notify-batch field allows you to specify a URL that will be called every time a user installs a package
        + the URL can be either an absolute path (that will use the same domain as the repository), or a fully qualified URL.
        + "notify-batch": "/downloads/"
      * **metadata-url, available-packages and available-package-patterns**
        + The metadata-url field allows you to provide a URL template to serve all packages which are in the repository. It must contain the placeholder %package%.
        + "metadata-url": "/p2/%package%.json"
        + if your repository only has a small number of packages, and you want to avoid the 404-requests, you can also specify an "available-packages"
        + Alternatively you can specify an "available-package-patterns" key which is an array of package name patterns (with \* matching any string, e.g. vendor/\* would make Composer look up every matching package name in this repository).
      * providers-api
        + The providers-api field allows you to provide a URL template to serve all packages which provide a given package name, but not the package which has that name. It must contain the placeholder %package%.
        + "providers-api": "https://packagist.org/providers/%package%.json",
      * List
        + The list field allows you to return the names of packages which match a given field (or all names if no filter is present). It should accept an optional ?filter=xx query param, which can contain \* as wildcards matching any substring.
        + "packageNames": [
        + "a/b",
        + "c/d"
        + ]
    - provider-includes and providers-url
      * The provider-includes field allows you to list a set of files that list package names provided by this repository. The hash should be a sha256 of the files in this case.
      * The providers-url describes how provider files are found on the server. It is an absolute path from the repository root. It must contain the placeholders %package% and %hash%.
    - cURL or stream options
      * The repository is accessed either using cURL (Composer 2 with ext-curl enabled) or PHP streams. You can set extra options using the options parameter
        + {
        + "type": "composer",
        + "url": "https://example.org",
        + "options": {
        + "http": {
        + "timeout": 60
        + }
        + }
        + }
    - **VCS**
      * VCS stands for version control system. This includes versioning systems like git, svn, fossil or hg. Composer has a repository type for installing packages from these systems
      * Loading a package from a VCS repository
        + There are a few use cases for this. The most common one is maintaining your own fork of a third party library. If you are using a certain library for your project, and you decide to change something in the library, you will want your project to use the patched version. If the library is on GitHub (this is the case most of the time), you can fork it there and push your changes to your fork. After that you update the project's composer.json. All you have to do is add your fork as a repository and update the version constraint to point to your custom branch. In composer.json only, you should prefix your custom branch name with "dev-" (without making it part of the actual branch name). For version constraint naming conventions see Libraries for more information
        + Note that you should not rename the package unless you really intend to fork it in the long term, and completely move away from the original package.
        + Also note that the override will not work if you change the name property in your forked repository's composer.json file as this needs to match the original for the override to work
      * Using private repositories
        + Exactly the same solution allows you to work with your private repositories at GitHub and Bitbucket:
        + "repositories": [
        + {
        + "type": "vcs",
        + "url": "git@bitbucket.org:vendor/my-private-repo.git"
        + }
        + "require": {
        + "vendor/my-private-repo": "dev-master"
        + }
      * Git alternatives
        + Git is not the only version control system supported by the VCS repository
        + The VCS driver to be used is detected automatically based on the URL. However, should you need to specify one for whatever reason, you can use bitbucket, github, gitlab, perforce, fossil, git, svn or hg as the repository type instead of vcs.
        + If you set the no-api key to true on a github repository it will clone the repository as it would with any other git repository instead of using the GitHub API
        + To let Composer choose which driver to use the repository type needs to be defined as "vcs"
        + If you already used a private repository, this means Composer should have cloned it in cache. If you want to install the same package with drivers, remember to launch the command composer clearcache followed by the command composer update to update Composer cache and install the package from dist.
        + VCS driver git-bitbucket is deprecated in favor of bitbucket
    - **Package**
      * you can define the package yourself by using a package repository.
      * Basically, you define the same information that is included in the composer repository's packages.json, but only for a single package. Again, the minimum required fields are name, version, and either of dist or source.
        + {
        + "repositories": [
        + {
        + "type": "package",
        + "package": {
        + "name": "smarty/smarty",
        + "version": "3.1.7",
        + "dist": {
        + "url": "https://www.smarty.net/files/Smarty-3.1.7.zip",
        + "type": "zip"
        + },
        + "source": {
        + "url": "http://smarty-php.googlecode.com/svn/",
        + "type": "svn",
        + "reference": "tags/Smarty\_3\_1\_7/distribution/"
        + },
        + "autoload": {
        + "classmap": ["libs/"]
        + }
        + }
        + }
        + ],
        + "require": {
        + "smarty/smarty": "3.1.\*"
        + }
        + }
        + }
    - Composer will not update the package unless you change the version field.
    - Composer will not update the commit references, so if you use master as reference you will have to delete the package to force an update, and will have to deal with an unstable lock file.
    - Hosting your own
    - While you will probably want to put your packages on packagist most of the time, there are some use cases for hosting your own repository.
    - Private company packages: If you are part of a company that uses Composer for their packages internally, you might want to keep those packages private.
    - Separate ecosystem: If you have a project which has its own ecosystem, and the packages aren't really reusable by the greater PHP community, you might want to keep them separate to packagist. An example of this would be WordPress plugins.bitbucket
    - Private Packagist
      * Private Packagist is a hosted or self-hosted application providing private package hosting as well as mirroring of GitHub, Packagist.org and other package repositories
    - Satis
      * Satis is a static composer repository generator. It is a bit like an ultra- lightweight, static file-based version of packagist.
    - Path
      * In addition to the artifact repository, you can use the path one,
      * which allows you to depend on a local directory,
        + "repositories": [
        + {
        + "type": "path",
        + "url": "../../packages/my-package"
        + }
        + ],
        + "require": {
        + "my/package": "\*"
        + }
      * You can configure the way the package's dist reference (which appears in the composer.lock file) is built.
      * none - reference will be always null.
      * config - reference is built based on a hash of the package's composer.json and repo config
      * ­­­
    - **Disabling Packagist.org**
      * You can disable the default Packagist.org repository by adding this to your composer.json:
        + {
        + "repositories": [
        + {
        + "packagist.org": false
        + }
        + ]
        + }
  + **Config** 
    - will describe the config section of the composer.json
    - process-timeout
      * The timeout in seconds for process executions, defaults to 300 (5mins). The duration processes like git clones can run before Composer assumes they died out. You may need to make this higher if you have a slow connection or huge vendors.
      * To disable the process timeout on a custom command under scripts, a static helper is available:
      * {
      * "scripts": {
      * "test": [
      * "Composer\\Config::disableProcessTimeout",
      * "phpunit"
      * ]
      * }
      * }
    - allow-plugins
      * As of Composer 2.2.0, the allow-plugins option adds a layer of security allowing you to restrict which Composer plugins are able to execute code during a Composer run.
      * When a new plugin is first activated, which is not yet listed in the config option, Composer will print a warning. If you run Composer interactively it will prompt you to decide if you want to execute the plugin or not.
      * Use this setting to allow only packages you trust to execute code. Set it to an object with package name patterns as keys. The values are true to allow and false to disallow while suppressing further warnings and prompts.
      * {
      * "config": {
      * "allow-plugins": {
      * "third-party/required-plugin": true,
      * "my-organization/\*": true,
      * "unnecessary/plugin": false
      * }
      * }
      * }
    - use-include-path
      * Defaults to false. If true, the Composer autoloader will also look for classes in the PHP include path
    - preferred-install
      * Defaults to dist and can be any of source, dist or auto
      * "preferred-install": {
      * "my-organization/stable-package": "dist",
      * "my-organization/\*": "source",
      * "partner-organization/\*": "auto",
      * "\*": "dist"
      * }
      * source means Composer will install packages from their source if there is one. This is typically a git clone or equivalent checkout of the version control system the package uses. This is useful if you want to make a bugfix to a project and get a local git clone of the dependency directly.
      * auto is the legacy behavior where Composer uses source automatically for dev versions, and dist otherwise
      * dist (the default as of Composer 2.1) means Composer installs from dist, where possible. This is typically a zip file download, which is faster than cloning the entire repository
    - use-parent-dir
      * When running Composer in a directory where there is no composer.json, if there is one present in a directory above Composer will by default ask you whether you want to use that directory's composer.json instead.
      * If you always want to answer yes to this prompt, you can set this config value to true. To never be prompted, set it to false. The default is "prompt".
      * php composer.phar config --global use-parent-dir true to set it.
    - store-auths
      * What to do after prompting for authentication, one of: true (always store), false (do not store) and "prompt" (ask every time), defaults to "prompt".
    - github-protocols
      * Defaults to ["https", "ssh", "git"]. A list of protocols to use when cloning from github.com, in priority order. By default git is present but only if secure-http is disabled, as the git protocol is not encrypted
    - github-oauth
      * A list of domain names and oauth keys. For example using {"github.com": "oauthtoken"} as the value of this option will use oauthtoken to access private repositories on github and to circumvent the low IP-based rate limiting of their API.
    - gitlab-domains
      * Defaults to ["gitlab.com"]. A list of domains of GitLab servers. This is used if you use the gitlab repository type.
    - gitlab-oauth
      * A list of domain names and oauth keys. For example using {"gitlab.com": "oauthtoken"} as the value of this option will use oauthtoken to access private repositories on gitlab. Please note: If the package is not hosted at gitlab.com the domain names must be also specified with the gitlab-domains option.
    - gitlab-token
      * A list of domain names and private tokens. Private token can be either simple string, or array with username and token.
      * {"gitlab.com": "privatetoken"}
      * {"gitlab.com": {"username": "gitlabuser", "token": "privatetoken"}}
    - gitlab-protocol
      * A protocol to force use of when creating a repository URL for the source value of the package metadata. One of git or http. (https is treated as a synonym for http.)
    - disable-tls
      * Defaults to false. If set to true all HTTPS URLs will be tried with HTTP instead and no network level encryption is performed
    - secure-http
      * Defaults to true. If set to true only HTTPS URLs are allowed to be downloaded via Composer. If you really absolutely need HTTP access to something then you can disable it, but using Let's Encrypt to get a free SSL certificate is generally a better alternative.
    - bitbucket-oauth
      * {"bitbucket.org": {"consumer-key": "myKey", "consumer-secret": "mySecret"}}
    - cafile
      * Location of Certificate Authority file on local filesystem
      * In PHP 5.6+ you should rather set this via openssl.cafile in php.ini, although PHP 5.6+ should be able to detect your system CA file automatically
    - Capath
      * If cafile is not specified or if the certificate is not found there, the directory pointed to by capath is searched for a suitable certificate. capath must be a correctly hashed certificate directory.
    - http-basic
      * A list of domain names and tokens to authenticate against them
      * {"example.org": "foo"}
      * Authorization: Bearer foo header
    - Platform
      * Lets you fake platform packages (PHP and extensions) so that you can emulate a production env or define your target platform in the config.
    - vendor-dir
      * Defaults to vendor. You can install dependencies into a different directory if you want to. $HOME and ~ will be replaced by your home directory's path in vendor-dir and all \*-dir options below.
    - bin-dir
      * Defaults to vendor/bin. If a project includes binaries, they will be symlinked into this directory.
    - data-dir
      * Defaults to C:\Users\<user>\AppData\Roaming\Composer on Windows

,

* + - cache-dir
      * Defaults to C:\Users\<user>\AppData\Local\Composer on Windows
    - cache-files-dir
      * Defaults to $cache-dir/files. Stores the zip archives of packages.
    - cache-repo-dir
      * Defaults to $cache-dir/repo. Stores repository metadata for the composer type and the VCS repos of type svn, fossil, github and bitbucket
    - cache-vcs-dir
      * Defaults to $cache-dir/vcs
    - cache-files-ttl
      * Defaults to 15552000 (6 months). Composer caches all dist (zip, tar, ...) packages that it downloads.
    - cache-files-maxsize
      * Defaults to 300MiB. Composer caches all dist (zip, tar, ...) packages that it downloads
    - Cache-read-only
      * Defaults to false. Whether to use the Composer cache in read-only mode.
    - bin-compat
      * ­­­be installed. If it is auto then Composer only installs .
    - prepend-autoloade
      * Defaults to true. If false, the Composer autoloader will not be prepended to existing autoloaders
    - autoloader-suffix
      * Defaults to null. Non-empty string to be used as a suffix for the generated Composer autoloader. When null a random one will be generated.
    - optimize-autoloader
      * Defaults to false. If true, always optimize when dumping the autoloader.
    - sort-packages
      * Defaults to false. If true, the require command keeps packages sorted by name
    - classmap-authoritative
      * Defaults to false. If true, the Composer autoloader will only load classes from the classmap. Implies optimize-autoloader
    - apcu-autoloader
      * Defaults to false. If true, the Composer autoloader will check for APCu and use it to cache found/not-found classes when the extension is enabled.
    - use-github-api
      * Defaults to true. Similar to the no-api key on a specific repository, setting use-github-api to false will define the global behavior for all GitHub repositories to clone the repository as it would with any other git repository instead of using the GitHub API.
    - htaccess-protect
      * Defaults to true. If set to false, Composer will not create .htaccess files in the Composer home, cache, and data directories
    - Lock
      * Defaults to true. If set to false, Composer will not create a composer.lock file and will ignore it if one is present.
    - platform-check
    - Defaults to php-only which only checks the PHP version. Set to true to also check the presence of extension. If set to false, Composer will not create and require a platform\_check.php file as part of the autoloader bootstrap.
    - secure-svn-domains
      * Defaults to []. Lists domains which should be trusted/marked as using a secure Subversion/SVN transport.
  + **Runtime Composer utilities**
    - While Composer is mostly used around your project to install its dependencies, there are a few things which are made available to you at runtime.
    - If you need to rely on some of these in a specific version, you can require the composer-runtime-api package
    - Autoload
    - Installed versions
      * composer-runtime-api 2.0 introduced a new Composer\InstalledVersions class which offers a few static methods to inspect which versions are currently installed
    - Knowing whether package X (or virtual package) is present
      * Composer\InstalledVersions::isInstalled('vendor/package');
      * Composer\InstalledVersions::isInstalled('vendor/package', false); // returns true if vendor/package is in require, false
    - Knowing whether package X is installed in version Y
      * Note: To use this, your package must require "composer/semver": "^3.0"
      * Composer\InstalledVersions::satisfies(new VersionParser, 'vendor/package', '2.0.\*');
    - Knowing the version of package X
      * This will return null if the package name you ask for is not itself installed but merely provided or replaced by another package. We therefore recommend using satisfies() in library code at least. In application code you have a bit more control and it is less important.
      * Composer\InstalledVersions::getVersion('vendor/package');
      * returns a normalized version (e.g. 1.2.3.0) if vendor/package is installed
      * Composer\InstalledVersions::getPrettyVersion('vendor/package');
      * returns the original version (e.g. v1.2.3) if vendor/package is installed,
      * Composer\InstalledVersions::getReference('vendor/package');
      * returns the package dist or source reference
      * Knowing a package's own installed version
      * If you are only interested in getting a package's own version, e.g. in the source of acme/foo you want to know which version acme/foo is currently running to display that to the user, then it is acceptable to use getVersion/getPrettyVersion/getReference.
      * Knowing the path in which a package is installed
        + The getInstallPath method to retrieve a package's absolute install path
        + The path, while absolute, may contain ../ or symlinks. It is not guaranteed to be equivalent to a realpath() so you should run a realpath on it if that matters to you.
        + Composer\InstalledVersions::getInstallPath('vendor/package');
        + returns an absolute path to the package installation location if vendor/package is installed
      * Knowing which packages of a given type are installed
        + The getInstalledPackagesByType method accepts a package type (e.g. foo-plugin) and lists the packages of that type which are installed. You can then use the methods above to retrieve more information about each package if needed.
        + Composer\InstalledVersions::getInstalledPackagesByType('foo-plugin');
      * Platform check
        + To avoid an unexpected white page of death with some obscure PHP extension warning in production, you can run composer check-platform-reqs
        + The default value is php-only which only checks the PHP version
      * Autoloader path in binaries
        + $\_composer\_autoload\_path global variable set when running binaries installed with Composer.
      * Binary (bin-dir) path in binaries
        + composer-runtime-api 2.2.2 introduced a new $\_composer\_bin\_dir global variable set when running binaries installed with Composer. Read more about this on the vendor binaries docs.
* Autoload understanding
  + What is autoloading
    - Autoloading is the process of automatically loading PHP classes without explicitly loading them with the require(), require\_once(), include(), or include\_once() functions.
    - To autoload classes, you must follow two rules
    - Each class must be defined in a separate file
    - Name your class files the same as your classes. The class Views would be placed in Views.php, a class called Users would be stored in Users.php and so
  + What is autoloader
    - An autoloader is a function that takes a class name as an argument and then includes the file that contains the corresponding class.
      * function my\_autoloader ( $class ) {
      * $path = $\_SERVER['DOCUMENT\_ROOT'] . '/classes/';
      * require $path . $class .'.php';
      * }
    - To implement an autoloader, PHP provides two functions
      * \_\_autoload(), not recommended, deprecated as of PHP 7.2 and removed as of PHP 8.0.
      * spl\_autoload\_register(), recommended.
  + Why use an autoloader
    - How often have you seen code like this at the top of your PHP file
    - require 'file\_1.php';
    - require 'file\_2.php';
    - All too often, right? The require(), require\_once(), include(), and include\_once() functions load an external PHP file into the current script, and they work wonderfully if you have only a few PHP scripts.
    - However, what if you need to include a hundred PHP scripts? The require() and include() functions do not scale well, and this is why PHP autoloaders are important. An autoloader is a strategy for finding a PHP class, interface, or trait and loading it into the PHP interpreter on-demand at run-time, without explicitly including files.
      * \_\_autoload() magic function
      * \_\_autoload(), a magic function, is automatically invoked by the PHP when we instantiate a class that has not already been loaded.
      * As of PHP 7.2.0 the \_\_autoload() function has been deprecated and removed since PHP 8.0.0. Now it is recommended to use the spl\_autoload\_register for that purpose instead.
      * spl\_autoload\_register()
        + spl\_autoload\_register( function($class) {
        + $path = $\_SERVER['DOCUMENT\_ROOT'] . '/classes/';
        + require\_once $path . $class .'.php';
        + });
  + Registering multiple autoloader functions for each namespace
    - The spl\_autoload\_register() allows to create an autoload chain, a series of functions that can be called to try and load a class or interface:
      * spl\_autoload\_register (function ($class) {
      * // for BrainBell namespace
      * require 'path/to/brainbell/classname.php';
      * });
      * spl\_autoload\_register(function ($class) {
      * // for Views namespace
      * require 'path/to/views/classname.php';
      * });
  + Autoloading classes based on namespace structure
    - PHP can autoload classes without needing an autoloader function if the directory structure containing the classes matches the namespaces of the classes.
    - spl\_autoload\_register();
      * $home = new classes\home();
      * # Loads /www/classes/home.php
      * $home = new views\home();
      * # Loads /www/views/home.php
* PHP design patterns
  + Design patterns are necessary to understand core of most of the frameworks have code written using various design patterns. To understant the basics of design patter go thourgh following links