COMP 3015

Week 7: Dependency Management, PHPUnit

Week 7 Topics

- Dependency Management with Composer: **https://getcomposer.org/**
  - Example: **https://github.com/sebastianbergmann/phpunit**
  - Semantic versioning: **https://semver.org/**
  - Security considerations
    - Third party code that we use can contain vulnerabilities

- Testing with PHPUnit

- Lab: testing and removing hardcoded credentials from our applications
  - PHPUnit and **DotEnv**

Getting started with Composer: installing

Navigate to: **https://getcomposer.org/download/** and follow the download instructions.

Getting started with Composer: initialization

```
~/Work/BCIT/COMP3015/Week7/lab-5-solution $ composer init

  Welcome to the Composer config generator


This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [cfenn/lab-5-solution]:
Description []:
Author [Christian Fenn <christianhfenn@gmail.com>, n to skip]:
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Add PSR-4 autoload mapping? Maps namespace "Cfenn\Lab5Solution" to the entered relative path. [src/, n to skip]: n

{
    "name": "cfenn/lab-5-solution",
    "type": "project",
    "authors": [
        {
            "name": "Christian Fenn",
            "email": "christianhfenn@gmail.com"
        }
    ],
    "require": {}
}

Do you confirm generation [yes]? yes
```

Getting started with Composer

After running **composer init** we'll have a **composer.json** file that looks similar to:

composer.json

```
{
    "name": "cfenn/lab-5-solution",
    "type": "project",
    "authors": [
        {
            "name": "Christian Fenn",
            "email": "christian_fenn@bcit.ca"
        }
    ],
    "require": {}
}
```

Getting started with Composer: require packages

Composer can be used to install packages from a repository such as Packagist

Example: we need to write some tests for our application.

PHPUnit is on Packagist and can be installed using Composer:

```
~/Work/BCIT/COMP3015/Week7/lab-5-solution $ composer require --dev phpunit/phpunit ^9.5
./composer.json has been updated
Running composer update phpunit/phpunit
Loading composer repositories with package information
Updating dependencies
Lock file operations: 0 installs, 0 updates, 11 removals
```

Getting started with Composer: require packages

(continuing from the last slide)

The composer.json file updates to require the phpunit package.

A **vendor/** folder will also be created that contains the code for your required dependencies.

Composer: require-dev

- Some dependencies are only needed in development and/or testing environments.

- These dependencies should be installed using the --dev flag (e.g. phpunit, previous slide)

- In a production environment a deployment script would run
  - **composer install --no-dev** # ignores dependencies in require-dev

Getting started with Composer: require packages

Continuing from the previous slide, PHPUnit ^9.5 is in our composer.json file.

This means we're requesting PHPUnit has: 9.5.0 <= version < 10.0.0

See: **https://getcomposer.org/doc/articles/versions.md#caret-version-range-**

Semantic Versioning: https://semver.org/

MAJOR.MINOR.PATCH

MAJOR: API changes, incompatible with previous versions

MINOR: functionality added, backwards compatible

PATCH: backwards compatible bug fix

Packages don’t have to follow this, but many do. It makes dependency management easier.

Getting started with Composer. The lock file

composer.lock is automatically generated by composer.

It should be committed to version control.

It guarantees that all developers running **$ composer install** will get the exact same versions of dependencies.

- Important because versions are specified as ranges in composer.json
  - Example: https://getcomposer.org/doc/articles/versions.md#caret-version-range-

Getting started with Composer: Additional Commands

$ composer init # create a composer.json file

$ composer require # add a package to the composer.json file and install it

$ composer install # read the composer.json file and install pkgs to the

→ if there's a lock file, use exact versions specified

$ composer update # upgrades packages to the newest allowed versions

$ composer audit # check for security vulnerabilities

$ composer remove # removes a package from composer.json and vendor/

Check docs for more: **https://getcomposer.org/doc/03-cli.md**

Importing required dependencies: autoloading

https://getcomposer.org/doc/01-basic-usage.md#autoloading

- Autoloading allows us to automatically load and use PHP classes without explicitly using **require/require_once/include/include_once**

Composer generates a **vendor/autoload.php** file that can be used to include packages.

How does this work though? See: **spl_autoload_register_example.zip**

Autoloading: takeaway

- When PHP encounters a class that is unknown, there is a mechanism to resolve the dependency that we, as developers, have control over.

- We don't need to explicitly import/require each class (of our own, or from dependencies).
  - We'll see more of this when we learn about **PSR-4 Autoloading**

- Rarely will you use the **spl_autoload_register** function
  - Often we use a library or framework that provides an abstraction

Dependency Security Vulnerabilities

- Dependencies that your application requires can contain security vulnerabilities.
  - Example: https://en.wikipedia.org/wiki/Log4Shell

- As a developer consuming these packages, you need to know how to check for security vulnerabilities and fix them
  - From a consumer point of view, fixes are done by upgrading to a patched version of the package

Dependency Security Vulnerabilities (cont.)

- Check for vulnerabilities: **$ composer audit**
- Fixing vulnerable dependencies as a consumer of those dependencies: search for an updated version of the package and install it in your application.

- If you found a vulnerability in a package your application is using and then later found a patched release of the package, you could install the patched version using:

  $ composer update <package>

  ...or **$ composer require <package>:<version>**

Testing with PHPUnit

Lab example.

Testing tips:

- Test one method at a time
- One test should not be dependent on another test
  - Dependent tests can make it challenging to isolate problems when failures occur
- Check return values, make assertions based on them
- Use **setUp** and **tearDown** methods to get the pre-test and post-test state that you need (see: https://docs.phpunit.de/en/11.2/fixtures.html)

Review

Dependency Management:

- Where does composer search for packages (by default)?
- When is **spl_autoload_register** called?
- What is semantic versioning?
- How can we check for security vulnerabilities in 3rd party packages?
- If a package we're using has a security vulnerability, what should we do?
- Why should the composer.lock file be used, what does it do?

Lab 5 is on D2L