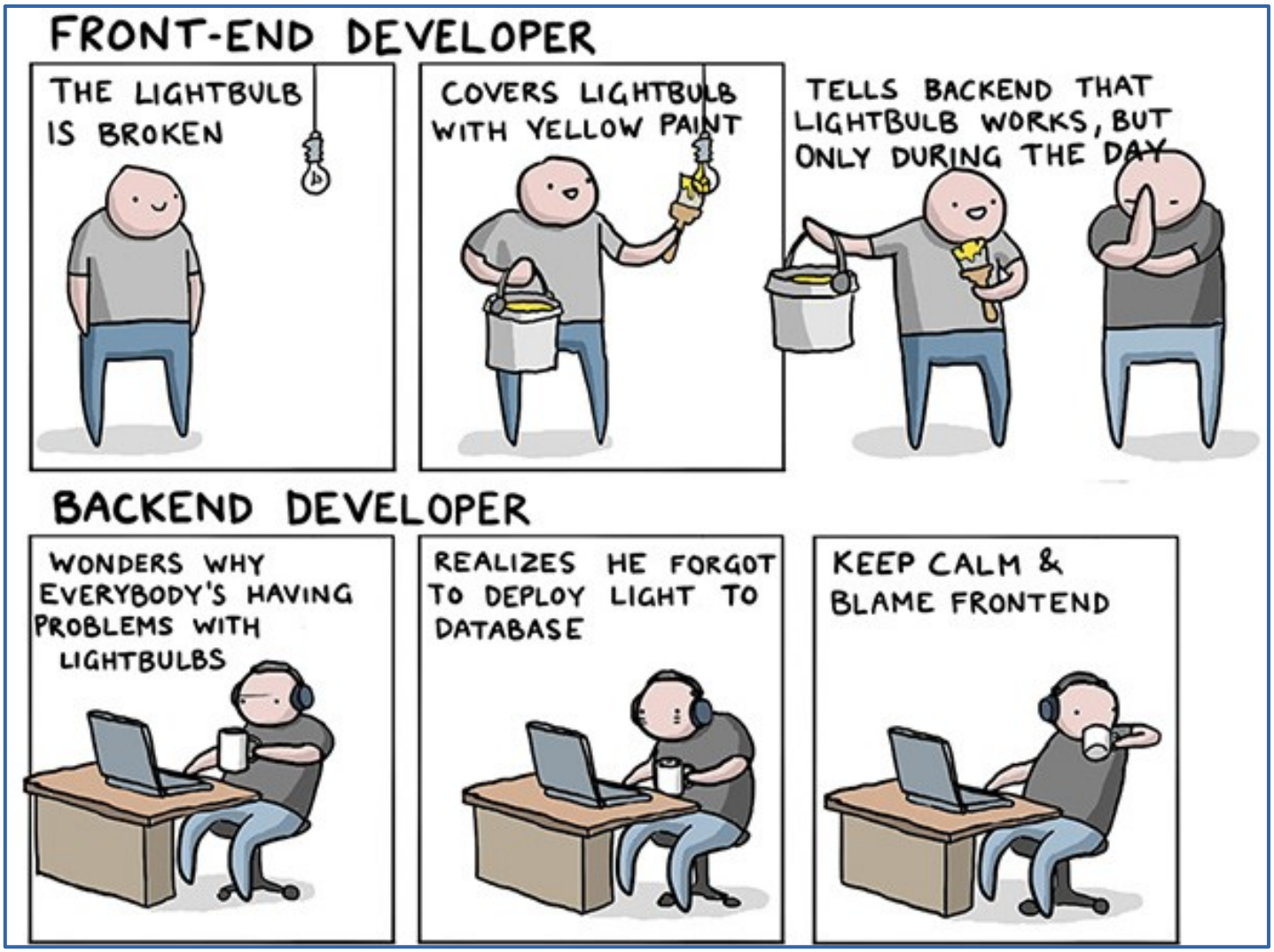
UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT
CHAPTER 7 : MANAGING APPLICATION DEPENDENCIES, ASSETS AND DATA MIGRATION

LOO YIM LING
ylloo@utar.edu.my

Dependencies, Assets and Data Migration

1) Managing application dependencies.
2) Managing assets.
3) Data migration.

Front End... Back End...



Information available on https://www.globalnerdy.com/wp-content/uploads/2016/12/it-explained-with-broken-lightbulb-excerpt.jpg

Managing Application Dependencies

1) Recall that we have used Composer to kick start a Laravel project by installing Laravel using the command line at the beginning of this course.
2) We have also used Composer to install additional packages that we require when developing our Laravel app, such as Laravel UI: Authentication and Authorization, Vue.js.
3) By using Composer to install Laravel and the additional packages, we are using a dependency manager to manage the dependencies in our application.

Dependency Management

1) Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.
2) Various other package/dependency management software are available for other programming/scripting languages. e.g. use NPM to manage application dependencies for JavaScript or Pip for Python.

Dependency Management

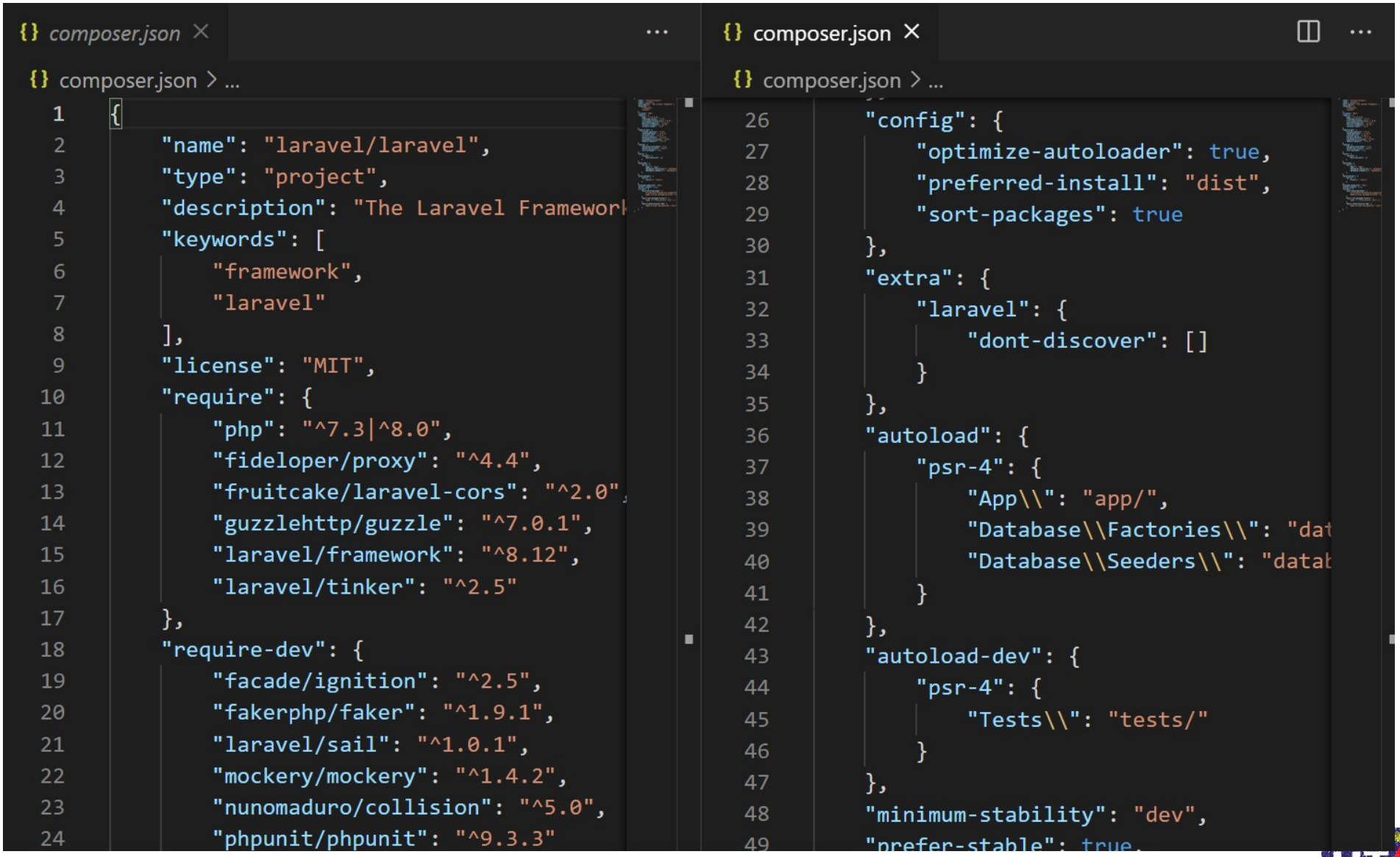
1) The idea behind the use of a dependency manager is as follows:

- Suppose that:
  - You have a project that depends on a number of libraries.
  - Some of those libraries depend on other libraries.
- Using Composer:
  - Enables you to declare the libraries that the project depend on.
  - Finds out which versions of which packages can and need to be installed, and installs them (meaning it downloads them into your project).

Composer

1) 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.
2) Laravel ships with a `composer.json` file with all the required dependencies for a standard Laravel installation already declared.

Composer



composer.json

```
{
    "name": "laravel/laravel",
    "type": "project",
    "description": "The Laravel Framework
    "keywords": [
        "framework",
        "laravel"
    ],
    "license": "MIT",
    "require": {
        "php": "^7.3|^8.0",
        "fideloper/proxy": "^4.4",
        "fruitcake/laravel-cors": "^2.0",
        "guzzlehttp/guzzle": "^7.0.1",
        "laravel/framework": "^8.12",
        "laravel/tinker": "^2.5"
    },
    "require-dev": {
        "facade/ignition": "^2.5",
        "fakerphp/faker": "^1.9.1",
        "laravel/sail": "^1.0.1",
        "mockery/mockery": "^1.4.2",
        "nunomaduro/collision": "^5.0",
        "phpunit/phpunit": "^9.3.3"
    "config": {
        "optimize-autoloader": true,
        "preferred-install": "dist",
        "sort-packages": true
    },
    "extra": {
        "laravel": {
            "dont-discover": []
        }
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/",
            "Database\\Factories\\": "dat
            "Database\\Seeders\\": "datab
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Tests\\": "tests/"
        }
    },
    "minimum-stability": "dev",
    "prefer-stable": true,
```

The “require” key

1) The first (and often only) thing to specify in `composer.json` is the require key.
2) This is simply telling Composer which packages the project depends on.
3) `require` takes an object that maps package names (e.g. `laravel/framework`) to version constraints (e.g. `8.12`)
4) Composer uses this information to search for the right set of files in package repositories using repositories key, or in *Packagist*, the default package repository.

Package Names

1) The package name consists of a vendor name and the project's name.
2) Often these will be identical - the vendor name just exists to prevent naming clashes.
3) e.g., it would allow two different people to create a library named `json`. One might be named `igloo/json` while the other might be `idime/json`.

Package Version Constraints

1) In the above example, one of the packages we are requesting is the `laravel/framework` package with the version constraint `^8.12`
2) This means any version that is greater than (>) version 8.12 can be required for installation.
3) Version constraints can vary in accordance of the need for the web application project.

Package Version Constraints

1) Followings are declaration of version constraints that could be used to specify for packages to be required:
   - Exact version constraint : 8.12
   - Version range (> OR <) : ^8.0 == greater than 8.0
   - Version range (|) : ^7.3|^8.0 == greater than 7.3 OR greater than 8.0
   - Version range (*) : 1.3.* == greater than or equals to 1.3 until lesser than 1.4.0

Installing Dependencies

1) To install the defined dependencies for the project, run `install` command.
2) After this comman one of the followings happens:
   - Installing without `composer.lock`
   - Installing with `composer.lock`

```
composer install
```

Installing Dependencies : without composer.lock

1) If you have never run the command before and there is also no `composer.lock` file present, Composer simply resolves all dependencies listed in your `composer.json` file and downloads the latest version of their files into the vendor directory.
2) When Composer has finished installing, it writes all of the packages and the exact versions of them that it downloaded to the `composer.lock` file, locking the project to those specific versions.

Installing Dependencies : with composer.lock

1) If there is already a composer.lock file as well as `composer.json` file when composer install is executed, it means the installation had been done before this.
2) 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.
3) This is by design, it ensures the project does not break because of unexpected changes in dependencies.

Updating Dependencies to their Latest Versions

1) The `composer.lock` file prevents you from automatically getting the latest versions of your dependencies.
2) To update to the latest versions, use the `update` command.
3) This will fetch the latest matching versions (according to your `composer.json` file) and update the `composer.lock` file with newer versions.

```
composer update
```

NPM

1) NPM is the dependency manager for JavaScript, just like Composer is the dependency manager for PHP.
2) Modern web apps are never complete without any client-side scripting, hence, the use of NPM to manage JavaScript dependencies in our project is required.
3) NPM install command will create the `node_modules` directory in current directory (if one doesn't exist yet) and will download the package to that directory.

```
node -v  //first install node
npm -v  //then install npm
npm install <package_name>
```

NPM

package.json

```
{
    "private": true,
    "scripts": {
        "dev": "npm run development",
        "development": "mix",
        "watch": "mix watch",
        "watch-poll": "mix watch -- --watch-options-poll=1000",
        "hot": "mix watch --hot",
        "prod": "npm run production",
        "production": "mix --production"
    },
    "devDependencies": {
        "axios": "^0.21",
        "bootstrap": "^4.0.0",
        "jquery": "^3.2",
        "laravel-mix": "^6.0.6",
        "lodash": "^4.17.19",
        "popper.js": "^1.12",
        "postcss": "^8.1.14",
        "resolve-url-loader": "^2.3.1",
        "sass": "^1.20.1",
        "sass-loader": "^8.0.0",
        "vue": "^2.5.17",
```

Updating Packages

1) It's a good practice to periodically update the packages your application depends on.
2) Then, if the original developers have improved their code, your code will be improved as well.

```
npm update
```

Managing Assets: JavaScript & CSS Scaffolding

1) While Laravel does not dictate which JavaScript or CSS pre-processors to use, it does provide a basic starting point using Bootstrap and Vue that will be helpful for many applications.
2) Laravel uses `NPM` to install both of these frontend packages
3) CSS: `Laravel Mix` compiling SASS
4) Javascript: `Vue`

```
npm run dev
```

Managing Assets: Using React.js

1) If prefer to use React to build JavaScript front end application, Laravel makes it a cinch to swap the Vue scaffolding with React scaffolding
2) A single Artisan CLI command will remove the Vue scaffolding and replace it with React scaffolding including example components.
3) On any fresh Laravel application, use react command in Artisan CLI:

```
php artisan ui react
php artisan ui react --auth
```

Compiling Assets using Laravel Mix

1) Laravel Mix provides a fluent API for defining Webpack build steps for your Laravel application using several common CSS and Javascript pre-processors.
2) Through simple method chaining, one can fluently define asset pipeline:

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css',
'public/css');
```

Running Laravel Mix

1) Mix is a configuration layer on top of webpack, so to run Mix tasks, one only need to execute one of the NPM scripts that are included in the default Laravel `package.json` file.
2) When one run the `dev` or `prod` scripts, all of application's CSS and JavaScript assets will be compiled and placed in application's public directory:

```
// Run all Mix tasks...
npm run dev
// Run all Mix tasks and minify output...
npm run prod
```

Watching Assets For Changes

1) The `npm run watch` command will continue running in terminal and watch all relevant CSS and JavaScript files for changes.
2) Webpack will automatically recompile assets when it detects a change to one of these files.
3) Webpack may not be able to detect file changes in certain local development environments. If this is the case, consider using the `watch-poll` command.

```
npm run watch
npm run watch-poll
```

Data Migration

1) We have used the migration tool in Laravel to create our database with the required tables, as well as to automatically insert default rows in some tables.
2) Besides that, we can also use the migration tool to write the required script to migrate data from an existing database to the new one, if required.

END OF LECTURE 08