Dependency injection container

Dependency Inversion Principle (DIP) suggests we create modular low-coupling code with the help of extracting clear abstraction subsystems.

For example, if you want to simplify a big class you can split it into many chunks of routine code and extract every chunk into a new simple separated class.

The principle says that your low-level chunks should implement an all-sufficient and clear abstraction, and high-level code should work only with this abstraction and not low-level implementation.

When we split a big multitask class into small specialized classes, we face the issue of creating dependent objects and injecting them into each other.

If we could create one instance before:

$service = new MyGiantSuperService();

And after splitting we will create or get all dependent items and build our service:

$service = new MyService(

new Repository(new PDO('dsn', 'username', 'password')), new Session(),

new Mailer(new SmtpMailerTransport('username', 'password', host')), new Cache(new FileSystem('/tmp/cache')),

);

Dependency injection container is a factory that allows us to not care about building our objects. In Yii2 we can configure a container only once and use it for retrieving our service like this:

$service = Yii::$container->get('app\services\MyService')

We can also use this:

$service = Yii::createObject('app\services\MyService')

Or we ask the container to inject it as a dependency in the constructor of an other service:

use app\services\MyService; class OtherService {

public function construct(MyService $myService) { ... }

}

When we will get the OtherService instance:

$otherService = Yii::createObject('app\services\OtherService')

In all cases the container will resolve all dependencies and inject dependent objects in each other.

In the recipe we create shopping cart with storage subsystem and inject the cart automatically into controller.