Dependency injection is a fancy phrase that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

****

Note: There is no need to bind classes into the container if they do not depend on any interfaces. The container does not need to be instructed on how to build these objects, since it can automatically resolve these objects using reflection.

****

You may use the bindIf method to register a container binding only if a binding has not already been registered for the given type:

```
$this->app->bindIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

****

Binding A Singleton

The singleton method binds a class or interface into the container that should only be resolved one time. Once a singleton binding is resolved, the same object instance will be returned on subsequent calls into the container:

```
use App\Services\Transistor;

use App\Services\PodcastParser;

use Illuminate\Contracts\Foundation\Application;


$this->app->singleton(Transistor::class, function
(Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

****

You may use the **singletonIf** method to register a singleton container binding only if a binding has not already been registered for the given type:

```
$this->app->singletonIf(Transistor::class, function
(Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

****

Binding Scoped Singletons

The scoped method binds a class or interface into the container that should only be resolved one time within a given Laravel request / job lifecycle. While this method is similar to the singleton method, instances registered using the scoped method will be flushed whenever the Laravel application starts a new "lifecycle", such as when a Laravel Octane worker processes a new request or when a Laravel queue worker processes a new job:

```
use App\Services\Transistor;

use App\Services\PodcastParser;

use Illuminate\Contracts\Foundation\Application;


$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

****

Binding Instances

You may also bind an existing object instance into the container using the instance method. The given instance will always be returned on subsequent calls into the container:

use App\Services\Transistor;

use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);

****

Binding Interfaces To Implementations

A very powerful feature of the service container is its ability to bind an interface to a given implementation. For example, let's assume we have an **EventPusher** interface and a **RedisEventPusher** implementation. Once we have coded our **RedisEventPusher** implementation of this interface, we can register it with the service container like so:

use App\Contracts\EventPusher;

use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);

This statement tells the container that it should inject the **RedisEventPusher** when a class needs an implementation of EventPusher. Now we can type-hint the **EventPusher** interface in the constructor of a class that is resolved by the container. Remember, controllers, event listeners, middleware, and various other types of classes within Laravel applications are always resolved using the container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 */
public function __construct(
    protected EventPusher $pusher
) {}
```

****

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, two controllers may depend on different implementations of the **Illuminate\Contracts\Filesystem\Filesystem** contract. Laravel provides a simple, fluent interface for defining this behavior:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;
$this->app->when(PhotoController::class)
        ->needs(Filesystem::class)
        ->give(function () {
            return Storage::disk('local');
        });
$this->app->when([VideoController::class, UploadController::class])
        ->needs(Filesystem::class)
        ->give(function () {
            return Storage::disk('s3');
        });
```

****

Binding Primitives

Sometimes you may have a class that receives some injected classes, but also needs an injected primitive value such as an integer. You may easily use contextual binding to inject any value your class may need:

```
use App\Http\Controllers\UserController;

$this->app->when(UserController::class)
        ->needs('$variableName')
        ->give($value);
```

****

Sometimes a class may depend on an array of tagged instances. Using the giveTagged method, you may easily inject all of the container bindings with that tag:

```
$this->app->when(ReportAggregator::class)
    ->needs('$reports')
    ->giveTagged('reports');
```

****

If you need to inject a value from one of your application's configuration files, you may use the giveConfig method:

```
$this->app->when(ReportAggregator::class)
    ->needs('$timezone')
    ->giveConfig('app.timezone');
```

****

Binding Typed Variadics

Occasionally, you may have a class that receives an array of typed objects using a variadic constructor argument:

```
<?php

use App\Models\Filter;

use App\Services\Logger;

class Firewall

{

    protected $filters;

    public function __construct(

        protected Logger $logger,

        Filter ...$filters,

    ) {

        $this->filters = $filters;

    }

}
```

Using contextual binding, you may resolve this dependency by providing the give method with a closure that returns an array of resolved Filter instances:

```
$this->app->when(Firewall::class)
        ->needs(Filter::class)
        ->give(function (Application $app) {
            return [
                $app->make(NullFilter::class),
                $app->make(ProfanityFilter::class),
                $app->make(TooLongFilter::class),
            ];
        });
```

****

For convenience, you may also just provide an array of class names to be resolved by the container
whenever Firewall needs Filter instances:

```
$this->app->when(Firewall::class)
        ->needs(Filter::class)
        ->give([
            NullFilter::class,
            ProfanityFilter::class,
            TooLongFilter::class,
        ]);
```

****

Variadic Tag Dependencies

Sometimes a class may have a variadic dependency that is type-hinted as a given class **(Report ...$reports).** Using the needs and giveTagged methods, you may easily inject all of the container bindings with that tag for the given dependency:

$this->app->when(ReportAggregator::class)

->needs(Report::class)

->giveTagged('reports');

****

Tagging

Occasionally, you may need to resolve all of a certain "category" of binding. For example, perhaps you are building a **report analyzer** that receives an array of many different Report interface implementations. After registering the **Report implementations**, you can assign them a tag using the **tag method**:

```
$this->app->bind(CpuReport::class, function () {

});

$this->app->bind(MemoryReport::class, function () {

});

$this->app->tag([CpuReport::class, MemoryReport::class],
'reports');
```

****

Once the services have been tagged, you may easily resolve them all via the container's tagged method:

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

****

Extending Bindings

The extend method allows the modification of resolved services. For example, when a service is resolved, you may run additional code to decorate or configure the service. The extend method accepts two arguments, the service class you're extending and a closure that should return the modified service. The closure receives the service being resolved and the container instance:

```
$this->app->extend(Service::class, function (Service $service, Application $app) {
    return new DecoratedService($service);
});
```

****