Service Container

Introduction

The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. 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.

Let's look at a simple example:

```php
<?php

namespace App\Http\Controllers;

use App\Services\AppleMusic;
use Illuminate\View\View;

class PodcastController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected AppleMusic $apple,
    ) {}

    /**
     * Show information about the given podcast.
     */
    public function show(string $id): View
    {
        return view('podcasts.show', [
            'podcast' => $this->apple->findPodcast($id)
        ]);
    }
}
```

In this example, the `PodcastController` needs to retrieve podcasts from a data source such as Apple Music. So, we will **inject** a service that is able to retrieve podcasts. Since the service is injected, we are able to easily "mock", or create a dummy implementation of the `AppleMusic` service when testing our application.

A deep understanding of the Laravel service container is essential to building a powerful, large application, as well as for contributing to the Laravel core itself.

Zero Configuration Resolution

If a class has no dependencies or only depends on other concrete classes (not interfaces), the container does not need to be instructed on how to resolve that class. For example, you may place the following code in your `routes/web.php` file:

```
class Service
{
    // ...
}

Route::get('/', function (Service $service) {
    die($service::class);
});
```

In this example, hitting your application's `/` route will automatically resolve the `Service` class and inject it into your route's handler. This is game changing. It means you can develop your application and take advantage of dependency injection without worrying about bloated configuration files.

Thankfully, many of the classes you will be writing when building a Laravel application automatically receive their dependencies via the container, including controllers, event listeners, middleware, and more. Additionally, you may type-hint dependencies in the `handle` method of queued jobs. Once you taste the power of automatic and zero configuration dependency injection it feels impossible to develop without it.

When to Utilize the Container

Thanks to zero configuration resolution, you will often type-hint dependencies on routes, controllers, event listeners, and elsewhere without ever manually interacting with the container. For example, you might type-hint the `Illuminate\Http\Request` object on your route definition so that you can easily access the current request. Even though we never have to interact with the container to write this code, it is managing the injection of these dependencies behind the scenes:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

In many cases, thanks to automatic dependency injection and facades, you can build Laravel applications without **ever** manually binding or resolving anything from the container. **So, when would you ever manually interact with the container?** Let's examine two situations.

First, if you write a class that implements an interface and you wish to type-hint that interface on a route or class constructor, you must tell the container how to resolve that interface. Secondly, if you are writing a Laravel package that you plan to share with other Laravel developers, you may need to bind your package's services into the container.

Binding

Binding Basics

Simple Bindings

Almost all of your service container bindings will be registered within service providers, so most of these examples will demonstrate using the container in that context.

Within a service provider, you always have access to the container via the `$this->app` property. We can register a binding using the `bind` method, passing the class or interface name that we wish to register along with a closure that returns an instance of the class:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

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

Note that we receive the container itself as an argument to the resolver. We can then use the container to resolve sub-dependencies of the object we are building.

As mentioned, you will typically be interacting with the container within service providers; however, if you would like to interact with the container outside of a service provider, you may do so via the `App` facade:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

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));
});
```

For convenience, you may omit providing the class or interface name that you wish to register as a separate argument and instead allow Laravel to infer the type from the return type of the closure you provide to the `bind` method:

```
App::bind(function (Application $app): Transistor {
    return new Transistor($app->make(PodcastParser::class));
});
```

> [!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.

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));
});
```

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

```
$this->app->scopedIf(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');
    });
```

Contextual Attributes

Since contextual binding is often used to inject implementations of drivers or configuration values, Laravel offers a variety of contextual binding attributes that allow to inject these types of values without manually defining the contextual bindings in your service providers.

For example, the `Storage` attribute may be used to inject a specific storage disk:

```
namespace App\Http\Controllers;

use Illuminate\Container\Attributes\Storage;
use Illuminate\Contracts\Filesystem\Filesystem;

class PhotoController extends Controller
{
    public function __construct(
        #[Storage('local')] protected Filesystem $filesystem
    )
    {
        // ...
    }
}
```

In addition to the `Storage` attribute, Laravel offers `Auth`, `Cache`, `Config`, `DB`, `Log`, `RouteParameter`, and `Tag` attributes:

```php
<?php

namespace App\Http\Controllers;

use App\Models\Photo;
use Illuminate\Container\Attributes\Auth;
use Illuminate\Container\Attributes\Cache;
use Illuminate\Container\Attributes\Config;
use Illuminate\Container\Attributes\DB;
use Illuminate\Container\Attributes\Log;
use Illuminate\Container\Attributes\RouteParameter;
use Illuminate\Container\Attributes\Tag;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Database\Connection;
use Psr\Log\LoggerInterface;

class PhotoController extends Controller
{
    public function __construct(
        #[Auth('web')] protected Guard $auth,
        #[Cache('redis')] protected Repository $cache,
        #[Config('app.timezone')] protected string $timezone,
        #[DB('mysql')] protected Connection $connection,
        #[Log('daily')] protected LoggerInterface $log,
        #[RouteParameter('photo')] protected Photo $photo,
        #[Tag('reports')] protected iterable $reports,
    )
    {
        // ...
    }
}
```

Furthermore, Laravel provides a `CurrentUser` attribute for injecting the currently authenticated user into a given route or class:

```php
use App\Models\User;
use Illuminate\Container\Attributes\CurrentUser;

Route::get('/user', function (#[CurrentUser] User $user) {
    return $user;
})->middleware('auth');
```

Defining Custom Attributes

You can create your own contextual attributes by implementing the `Illuminate\Contracts\Container\ContextualAttribute` contract. The container will call your attribute's `resolve` method, which should resolve the value that should be injected into the class utilizing the attribute. In the example below, we will re-implement Laravel's built-in `Config` attribute:

```
namespace App\Attributes;

use Attribute;
use Illuminate\Contracts\Container\Container;
use Illuminate\Contracts\Container\ContextualAttribute;

#[Attribute(Attribute::TARGET_PARAMETER)]
class Config implements ContextualAttribute
{
    /**
     * Create a new attribute instance.
     */
    public function __construct(public string $key, public mixed $default = null)
    {
    }

    /**
     * Resolve the configuration value.
     *
     * @param  self  $attribute
     * @param  \Illuminate\Contracts\Container\Container  $container
     * @return mixed
     */
    public static function resolve(self $attribute, Container $container)
    {
        return $container->make('config')->get($attribute->key, $attribute->default);
    }
}
```

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
{
    /**
     * The filter instances.
     *
     * @var array
     */
    protected $filters;

    /**
     * Create a new class instance.
     */
    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);
});
```

Resolving

The `make` Method

You may use the `make` method to resolve a class instance from the container. The `make` method accepts the name of the class or interface you wish to resolve:

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

If some of your class's dependencies are not resolvable via the container, you may inject them by passing them as an associative array into the `makeWith` method. For example, we may manually pass the `$id` constructor argument required by the `Transistor` service:

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

The `bound` method may be used to determine if a class or interface has been explicitly bound in the container:

```
if ($this->app->bound(Transistor::class)) {
    // ...
}
```

If you are outside of a service provider in a location of your code that does not have access to the `$app` variable, you may use the `App` facade or the `app` helper to resolve a class instance from the container:

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);

$transistor = app(Transistor::class);
```

If you would like to have the Laravel container instance itself injected into a class that is being resolved by the container, you may type-hint the `Illuminate\Container\Container` class on your class's constructor:

```
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 */
public function __construct(
    protected Container $container,
) {}
```

Automatic Injection

Alternatively, and importantly, you may type-hint the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, middleware, and more. Additionally, you may type-hint dependencies in the `handle` method of queued jobs. In practice, this is how most of your objects should be resolved by the container.

For example, you may type-hint a service defined by your application in a controller's constructor. The service will automatically be resolved and injected into the class:

```
<?php

namespace App\Http\Controllers;

use App\Services\AppleMusic;

class PodcastController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected AppleMusic $apple,
    ) {}

    /**
     * Show information about the given podcast.
     */
    public function show(string $id): Podcast
    {
        return $this->apple->findPodcast($id);
    }
}
```

Method Invocation and Injection

Sometimes you may wish to invoke a method on an object instance while allowing the container to automatically inject that method's dependencies. For example, given the following class:

```
<?php

namespace App;

use App\Services\AppleMusic;

class PodcastStats
{
    /**
     * Generate a new podcast stats report.
     */
    public function generate(AppleMusic $apple): array
    {
        return [
            // ...
        ];
    }
}
```

You may invoke the `generate` method via the container like so:

```
use App\PodcastStats;
use Illuminate\Support\Facades\App;

$stats = App::call([new PodcastStats, 'generate']);
```

The `call` method accepts any PHP callable. The container's `call` method may even be used to invoke a closure while automatically injecting its dependencies:

```
use App\Services\AppleMusic;
use Illuminate\Support\Facades\App;

$result = App::call(function (AppleMusic $apple) {
    // ...
});
```

Container Events

The service container fires an event each time it resolves an object. You may listen to this event using the `resolving` method:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor,
Application $app) {
    // Called when container resolves objects of type "Transistor"...
});

$this->app->resolving(function (mixed $object, Application $app) {
    // Called when container resolves object of any type...
});
```

As you can see, the object being resolved will be passed to the callback, allowing you to set any additional properties on the object before it is given to its consumer.

Rebinding

The `rebinding` method allows you to listen for when a service is re-bound to the container, meaning it is registered again or overridden after its initial binding. This can be useful when you need to update dependencies or modify behavior each time a specific binding is updated:

```
use App\Contracts\PodcastPublisher;
use App\Services\SpotifyPublisher;
use App\Services\TransistorPublisher;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(PodcastPublisher::class, SpotifyPublisher::class);

$this->app->rebinding(
    PodcastPublisher::class,
    function (Application $app, PodcastPublisher $newInstance) {
        //
    },
);

// New binding will trigger rebinding closure...
$this->app->bind(PodcastPublisher::class, TransistorPublisher::class);
```

PSR-11

Laravel's service container implements the PSR-11 interface. Therefore, you may type-hint the PSR-11 container interface to obtain an instance of the Laravel container:

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    // ...
});
```

An exception is thrown if the given identifier can't be resolved. The exception will be an instance of `Psr\Container\NotFoundExceptionInterface` if the identifier was never bound. If the identifier was bound but was unable to be resolved, an instance of `Psr\Container\ContainerExceptionInterface` will be thrown.