Laravel Cheat Sheet

Contents

IoC Usage

In general, where these methods take closures, the closure will act as they should. For example, `App::singleton()` 's closure is immediately invoked to get the instance to be used every time make is called.

Quick intro:

- `bind($abstract, $concrete, $shared)`

  Adds `$abstract` as a key to the container, with `$concrete` being the concrete class to instantiate in its place. Mainly used for providing a concrete implementation for an interface.

- `share($closure)`

  Given a closure (only), makes it act as if it was shared (instance/singleton style), and returns it. Technically equivalent to `App::bind($key, $closure, true)` but goes about it a different way. Mainly used in service providers to add a fully resolvable service to the IoC container.

- `bindShared($abstract, $closure)`

  A shortcut that was introduced in 4.1 that caters to a common pattern. Essentially helps those who want to bind a shared instance in the container. See below for example.

- `singleton($abstract, $concrete)`

  Simply an alias to `bind` with the `$shared` argument set to true. Mainly used for providing a concrete implementation for an interface, but one that should only have one instance (database connection, etc.).

- `extend($abstract, $closure)`

  Allows you to wrap an existing binding with another closure.

- `instance($abstract, $instance)`

  Kinda bypasses the IoC bindings and passes the `$instance` directly into `$app->instances`. Good for mocking, as it'll tell Laravel which instance to give to a class that asks for an abstract.

- `alias($abstract, $alias)`

  Allows you to alias a fully-qualified type to a shorter version for later usage in `make`.

- `make($abstract[, $parameters = array()])`

  Retrieves the binding from the IoC container and returns an instance of the class that is required.

- `build($concrete[, $parameters = array()])`

  While this is public, it appears that the general idea is not to call this. It is called internally by `make` to resolve a standard class's dependencies.

- array access

  Upon getting, internally `App::make` is called on the key passed in.

  Upon setting, internally `App::bind` is called, and the value passed in is converted to a closure if it is not already one.

bind

`App::bind($abstract, $concrete, $shared)`

Given a class abstract pathspec (i.e. the fully-qualified path to an interface), when `make` is called on that pathspec, return the given concrete implementation (if `$concrete` is a string, it's a fully-qualified path to a concrete class to be instantiated in `$abstract`'s place, if a closure, then execute and return the return value of the closure.

If `$shared` is `false` (default), the closure is executed every time `make` is called. If `$shared` is `true`, the closure is executed once (lazily on first call to `make`) and its return value will be cached to return for subsequent `make` calls.

```
<?php interface BlahInterface {} class ConcreteBlah
implements BlahInterface {}
App::bind('BlahInterface', 'ConcreteBlah'); class
SomeController { public __construct(BlahInterface
$blah) { // $blah is an instance of ConcreteBlah } }
```

singleton

`App::singleton($abstract, $concrete)`

In effect this adds `$abstract` as a key to the container with the value of `$concrete` and because `$shared` is set to true, the first time it's instantiated the object goes into `$app->instances` so that the next time `make` is called, the instance is used rather than being re-instantiated.

If `$concrete` is a closure, it is evaluated lazily when the first instance is requested with `make` and then stored in `$app->instances` for future reference.

share

`App::share($closure)`

Wraps the passed closure in a new closure that, the first time it runs, runs the closure and stores the result in a temporary variable. Every time it's called after that, it just returns the result.

Share doesn't store anything in the IoC container, and just returns a closure. Make sure you use it with another IoC feature (most people go for `$app['my.feature'] = $app->share(/* closure */)`, but see below for new usage).

bindShared

`App::bindShared($abstract, $closure)`

`bindShared` is a convenience method for the common pattern of `$app['my.feature'] = $app->share(/* closure */)`. The new method is `$app->bindShared('my.feature', /* closure */)`. It's a little shorter and arguably makes more sense when read, leading to more maintainable code.

extend

`App::extend($abstract, $closure)`

Extend a binding with an outer wrapping closure. When the binding is resolved using `make`, the original closure is executed and then its result (the instance) is passed into the closure that was passed in here.

Presumably this could be used for overriding dependency injection or setting additional dependencies using `$model->setInterface($concrete)` kinda thing.

instance

`App::instance($abstract, $instance)`

Sharing through `bind` and the other sharing methods will eventually achieve the same thing, but doing this is a shortcut to the desired endpoint: the instance you need is in `$app->instances`. Use this when you have an instance ready to go, or the other methods if you are using string classnames or closures.

`$instance` should **not** be a closure as it is returned as-is in `make`.

The use-case for instance, from the look of it, is that when asked (through type hinting usually) for an instance of `Class`, return `$instance`. Good for mocking Eloquent models:

```
<?php $mock = m::mock('Post'); $this->app->instance('Post', $mock); class SomeController { public function __construct(Post $post) { // $post is now the mocked version of the real Post class } }
```

alias

`App::alias($abstract, $alias)`

Allows you to alias a fully-qualified type to a shorter version. So you can alias `\My\Namespace\Is\Deep\MyClass` to `MyClass` by calling `App::alias('\My\Namespace\Is\Deep\MyClass', 'MyClass');` and then later you can just call `App::make('MyClass');`

This function does not actually store any bindings – just the fact that something called `$abstract` in the IoC container can be found by asking for `$alias`. It's also naïve in that you can pass anything in and it does no checking. Presumably the eventual call to `make` will do the checking that something *can* be resolved.

make

`App::make($abstract[, $parameters = array()])`

Get the instance of Class that you know about. If you don't have one registered (either with `App::instance()`, `App:share()` or `App:singleton()`, etc.) then try to just resolve the class the best way you know how (i.e. instantiate and use reflection to also try to resolve its dependencies).

This could go all sorts of ways depending on how the binding is set up, but basically:

1. the alias is resolved if there is one
2. if there's already an instance stored in `$app->instances` that'll be returned, if not, continue
3. get the concrete version of the given abstract <— QUALIFY
4. if the concrete is buildable (QUALIFY) then `build` it, otherwise `make` it to get the actual instance to be used
5. if it's set to be shared, put the instance in `$app->instances` so next time we stop at 2.
6. fire callbacks, passing in the instance
7. return the instance

This appears to be the backbone of the IoC – at a guess, whenever Laravel needs to resolve anything (through either an explicit call to `App::make()` or typehints) it'll use this. In turn, this uses the various registration methods outlined above.

URL Generation

- `URL::current()`

  Returns the full URL of the current request, e.g. `http://localhost/path/to/page`

- `URL::full()`

  Returns the full URL of the current request with querystring appended, e.g. `http://localhost/path/to/page?foo=bar`

- `URL::previous()`

  Returns the full URL of the previous request (useful when redirecting maybe) – simply uses the value from the HTTP 'referer' header, so *should* include the querystring

- `URL::to($uri[, $parameters = array(), $secure = false])`

  Generates a full URL to the given URI path after the domain part. Does no checking of the validity of the URI passed.

- `URL::secure($uri[, $parameters = array(), $secure = false])`

Like `URL::to()` but generates HTTPS links

- `URL::route($route_name[, $parameters = array(), $secure = false])`

  Like `URL::to()` but pass in the name of a named route and it'll give the associated URI

- `URL::action($action[, $parameters = array(), $secure = false])`

  Like `URL::route()` but instead of passing in a named route, pass in a controller-action combination, e.g. `Controller@action`

- `URL::asset($uri[, $secure = false])`

  Looks like it works just like `URL::to()` in that is appends the passed URI to the full URL to the framework document root.

- `URL::secureAsset($uri)`

  Like `URL::asset()` but generates HTTPS links

- `url($uri[, $parameters = array(), $secure = false])`

  Alias of `URL::to()`

- `secure_url($uri[, $parameters = array(), $secure = false])`

  Alias of `URL::secure()`

- `route($route_name[, $parameters = array(), $secure = false])`

  Alias of `URL::route()`

- `action($action[, $parameters = array(), $secure = false])`

  Alias of `URL::action()`

- `asset($uri[, $secure=false])`

  Alias of `URL::asset()`

- `secure_asset($uri)`

  Alias of `URL::secureAsset()`

- `link_to($uri, $title[, $attributes = array(), $secure = false])`

  Generates an HTML anchor tag to the given URI (see `URL::to()`)

- `link_to_asset($uri, $title[, $attributes = array(), $secure = false])`

Generates an HTML anchor tag to the given asset URI (see `URL::asset()`)

- `link_to_route($route_name, $title[, $parameters = array(), $attributes = array()])`

  Generates an HTML anchor linking to the named route (see `URL::route()`)

- `link_to_action($action, $title[, $parameters = array(), $attributes = array()])`

  Generates an HTML anchor linking to the action (see `URL::action()`)

Service Providers

The `register` method of a service provider is called first. This method should do very little – simply add any bindings to the IoC container. The reason being that all service providers will be initialised (using `register`) in order, so a given service provider cannot guarantee that a service provider it may rely on is initialised yet, and thus cannot use it.

The next method that's called is `boot` . By the time this method fires, all of the service providers should be initialised, so things like `Route::get()` and `Queue::push()` can be used. Do the 'proper' initialisation of a service provider (such as adding routes and generally using other service providers) here.

Packages

There are various parts of putting together a package and its (main) service provider that aren't discussed very well in the documentation:

- `$this->package('vendor/package'[, $namespace = null, $path = null])`

  This sets up a package so that Laravel knows where to look for the config, views, etc. In general, just passing the vendor/package name to the call to `package` will suffice, but if your paths are non-standard or you want to use a specific view/config namespace, you can set them up here.

- `$this->commands(array)`

  This registers with Artisan the commands specified in the passed array (which should already be registered in the IoC container). The strings in the array should be the IoC keys you set up. This is pretty much a shortcut (due to common use-case) to calling `Artisan::resolve('ioc_binding_name')` , which is itself a shortcut of sorts to `Artisan::add($command_instance)` .

- `protected $defer = false`

This service provider attribute allows Laravel to know whether your service provider's loading can be deferred or not. Defaults to false, but set to true if you do a lot of bootstrapping maybe?

- `function provides()  array`

  Laravel calls this function to learn what services the service provider actually provides to the IoC container (before it actually does so). This is meant to be used in combination with the `$defer` attribute so that, for deferred service providers, the first time a provider specified in this array is used, Laravel can lazily register and boot the correct service provider.

Also not discussed in the documentation is that a package's views can be overridden like its config can. To override the config, `artisan config:publish vendor/package` can be called, which under the hood copies the package's `<package_root>/config/*` files to `app/config/packages/<vendor>/<package>/*`. There is now (all thanks to me) also a corresponding `artisan` command for publishing views: just use `artisan view:publish vendor/package`. **Do be careful with this though!**

Eloquent Models

Relationships

A default belongsToMany relationship will look for the table references by concatenating the two tables in question, both singular and listed in alphabetical order, with an underscore in between.

So `users` and `tags` are joined through `tag_user`.

Using relationships

In the following code:

- `Post` hasMany `Comment`
- `Comment` belongsTo `Post`
- `Post` belongsToMany `Tag`
- `Tag` belongsToMany `Post`

To get the collection associated with a `hasMany` or `belongsToMany`, or the model associated with a `belongsTo`, the relationship field can be used directly from the model:

```
<?php $post = Post::findById(1); $comments = $post-
>comments; // plural as it's a hasMany - but not
`$post->comments()` // $comments can now be iterated
through ?> <?php $comment = Comment::findById(1);
$post = $comment->post; // singular as it's a
belongsTo - but not `$comment->post()` // $post is an
instance of Post
```

In the case of altering the relationships (or working on the pivot table in a `belongsToMany` relationship), the set must be retrieved by calling the relationship as a function.

For `hasMany` and `belongsTo`:

```
<?php $post = Post::findById(1); $comment =
Comment::findById(1); $other_comment =
Comment::findById(2); // add to a hasMany set $post-
>comments()->save($comment); // both are models
$post->comments()->saveMany([$comment,
$other_comment]); // all are models // can also use
create/createMany to create and associate at the same
time $post->comments()->create(['title' => 'Some
Comment']); $post->comments()->createMany([['title'
=> 'Some Comment'], ['title' => 'Some Other
Comment']]); ?> <?php // associate parent in
belongsTo $comment = Comment::findById(1); $post =
Post::findById(1); $comment->post()-
>associate($post); // both are models $comment-
>save(); // must save after association (presumably
because it's a field on the table)
```

As far as I can tell, there is no way to remove a model from a `hasMany` set or remove the relation in a `belongsTo`. Calling `$comment->post()->associate()` will not work to set the relation to `NULL` and effectively sever the relationship as the `associate` method expects an `Eloquent\Model` instance. Similarly, there is no way to set a whole hasMany set in one go – `saveMany` will add multiple associations, but will not remove any one not passed in.

For `belongsToMany` (note how the functions use IDs, not models*):

```
<?php $post = Post::findById(1); $tag =
Tag::findById(1); // add tag to post $post->tags()-
>attach($tag->id); // uses the tag's ID - also this
code is probably better written as `$tag->getKey()`
// remove tag from post $post->tags()->detach($tag-
>id); // set all tags (will also remove any current
set but not passed in) $other_tag =
Tag::findById(2); $post->tags()->sync([$tag->id,
$other_tag->id]); // remove all tags from a post
$post->tags()->detach();
```

* Actually `attach` and `detach` can accept instances of `Eloquent\Model` but only if you're attaching or detaching one – they cannot be used if arrays are passed

Fetching

Using Eloquent with `where`, `orWhere`, `take`, etc. uses the query builder. Once a query is complete and you need to get the actual collection of models the following methods are available:

- `Model::all()` gets everything (minus trashed)
- `Model::where(/* condition */)->get()` gets the collection filtered by the query builder
- `Model::first()` gets the first result from the possible set that would be returned
- `Model::pluck($field)` returns a string representing the value in the passed field – if the query would have returned multiple results, the first is chosen and the value of its `$field` field is returned (i.e. as if `first` is chained before `pluck`)
- `Model::lists($field)` returns an array of the above – this time, multiple results are allowed, also duplicates will be returned if the database has duplicates
- `Model::distinct($field)->lists($field)` is a much more useful version of the above line – it'll strip duplicate field values and then return them in an array
- `Model::toSql()` is useful for debugging – return the current query as an SQL string to be executed

Nested Wheres

A few examples of converting SQL to Eloquent/Fluent query builder calls, as it's slightly counter-intuitive:

```
<?php // SELECT * FROM `table` WHERE `x` AND (`y` OR
`z`) Model::where(/* x */)->where(function ($query)
{ $query->(/* y */)->orWhere(/* z */); }); // SELECT
* FROM `table` WHERE `x` OR (`y` AND `z`)
Model::where(/* x */)->orWhere(function ($query) {
$query->(/* y */)->where(/* z */); });
```

Boot method

Like service providers, eloquent models have a `boot` method. This method is like a static constructor (as far as I can tell) in that it will only be called once, but it is also lazily-executed: it gets called the first time a model is instantiated. If a model should register its own event listeners, then it should be done here.

Example:

```
<?php class Post extends Eloquent { // called once
when Post is first used public static function
boot() { // there is some logic in this method, so
don't forget this! parent::boot();
Post::saving(function ($post) { // or something
$post->updated_at = new DateTime(); }); } // called
for every new instance of Post public function
__construct(PostRepository $posts) { $this->posts =
$posts; } }
```

Routing

Just some more detail on the less-discussed commands:

- `Route::when($pattern, $names[, $methods = null])`

Register a filter or set of filters against a URI rather than route.

Normally, you register filters (e.g. `csrf` and `auth`) on individual routes, or on a route group, but you can apply them to a given URI pattern. This may be useful, but may also break easily-modifiable URIs by meaning that a given URI must be changed in two places if it changes. The third parameter, `$methods`, can take an array of HTTP methods.

- `Route::pattern($key, $pattern)`

  Set a global `where` pattern for a given named parameter for all routes.

  In a route with named parameters, you can specify a regex requirement for each parameter with `where`. If the same named parameter is used in multiple routes, rather than re-specifying the `where` for each one, this method can be used to say "wherever you see `$key`, add a where constraint of `$where`"

- `Route::model($key, $model)`

  Much like for `Route::pattern`, register a given model for the specified named parameter for all routes. The value of the parameter will be interpreted as an `id` that should be used to search for the given model instance.

  `$model` should be a fully-qualified classname.

- `Route::bind($key, Closure $resolver)`

  In cases where an `id` lookup will not suffice, we can use a closure to look a given model up from the parameter's value instead. The value is passed into the closure and whatever the closure returns (should be an instance of `Eloquent\Model`) will be passed to the route's action method (closure or controller method).

  This method is very useful for looking up URI slugs, for example, as well as more complex lookups.

- `Route::controller($uri, $classname)`

  Wildcard routing to a controller using the passed URI as a prefix. The first segment that comes after `$uri` in the URI is turned into the method name on the controller, prefixed with the HTTP method (get, post, etc.). Examples for `Route::controller('user', 'UserController')`:

  - `GET /user` `UserController@getIndex() // (I think)`
  - `GET /user/list` `UserController@getList()`
  - `POST /user/edit/1` `UserController@postEdit(1)`

`$classname` should be a fully-qualified classname to the controller.

This can also be called as `Route::controllers(array $mapping)` which takes a single parameter which is an array that maps uri classname.

The general advice is not to use this though, instead going with explicit routing as much as possible.

- `Route::resource($uri, $classname)`

  Like `Route::controller` it allows a form of predefined routing to a controller. However, the `resource` method has very specific rules that follows RESTful principles. As such, it will automatically register routes like:

  - `GET /uri`
  - `GET /uri/{id}`
  - `POST /uri/{id}`
  - `DELETE /uri/{id}`

  `$classname` should be a fully-qualified classname to the controller.

  This method is only useful for setting up routing for an API really.

The View class

- `View::share()`

  Inject a view variable that will be available in all views.

- `View::name()`

  Register a named view. Like IoC container `alias` function, but works like named routes. Also can't be used with `View::make` but instead should be used with `View::of`. At least as far as I can tell.

- `View::of($view, $data = array())`

  Render (well return) a named view. Just like `View::make` but for named views. As far as I know. Not really sure about the use-case here, it's just something I saw and thought I should document.

Поделиться7