مع عدة Laravel للأصناف المتوافرة في حاوي خدمات تطبيقك. يأتي (static) "تُوفِّر الواجهات الساكنة واجهة "ساكنة (static) "الساكنة "وسطاء ساكنات Laravel تقريبًا. تمثل واجهات Laravel واجهات تمكنك من استخدام كل خاصيات المقتضبة والمُعبِّرة مع الحفاظ على قابلية (syntax) للأصناف الأساسية بحاوي الخدمات مما يوفر كل فوائد الصِّيغ (proxies الاختبار ومرونة أكبر من الدالات الساكنة التقليدية.

يمكننا الوصول لواجهة ساكنة Illuminate\Support\Facades. الساكنة مُعرَّفةٌ في مجال الأسماء Laravel كل واجهات بهذه الطريقة:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

متى نستخدم الواجهات الساكنة؟
دون حفظ أسماء Laravel للواجهات الساكنة فوائد كثيرة فهي توفِّر صيغةً مقتضبةً وسهلة التذكّر تسمح لنا باستخدام خاصيات PHP الأصناف الطويلة التي تسلتزم إضافتها وإعدادها يدويًا. علاوة على ذلك، اختبارهن أسهل بفضل استخدامهن الفريد لدوال الدَينامِيكيَة.

بما أن .(class scope) يجب الاحتياط عند استخدام الواجهات الساكنة حيث أنّ خطرهم الرئيسي هو زحف مجال الصَنف الواجهات السَاكنة سهلة الاستخدام ولا تتطلب إضافات، قد يسهل ترك صنفك يواصل التضخَم بشكل هام عند استخدام عدَة واجهات اذ ان ،(dependency injection) سَاكنة في صنف واحد. يمكن الحد من هذه المشكلة باستخدام إضافة الاعتماديات ضخمة تعلمك ان صنفك تضخَّم أكثر من اللازم. لذلك أعر انتباهًا خاصًا constructor التحذيرات المرئية التي تصدرها دالة بناء ضيَقًا (scope of responsibility) لحجم صنفك كي يظل مجال مسؤوليته.

من الأفضل إضافة عقود Laravel، ملاحظة: عند بناء حزمة طرف ثالث تتعامل مع Laravel (Laravel contracts) ذاته لن تستطيع استخدام مساعدي اختبار واجهات Laravel بدل استخدام الواجهات الساكنة لأنّ الحزم المبنيّة خارج Laravel الساكنة (facade testing helpers)

الواجهات وإضافة الاعتماديّة (dependency injection)
الصنف المضاف هي إحدى فوائد إضافة الاعتماديّة الأساسية. هذه (implementations) القدرة على استبدال تعاريف استخدام وتتحقق من استدعاء مختلف (stub) أو بذرة (mock) الخاصية مفيدة عند الاختبار بما أنك تقدر على إضافة غرض مُقلِّد الدوال في تلك البذرة.

مع ذلك، لمّا .(static class method) ليس من الممكن عادة استخدام الأغراض المُقلِّدة أو البذور على دالَة ساكنة فعلًا مستبينة من حاوي الخدمات، يمكننا اختبار (objects) كانت الواجهات الساكنة تستخدم دوالًا ديناميكية لتوكيل استدعاء كائنات الواجهات الساكنة تمامًا كما نختبر نسخة صنف مضاف. خذ مثلًا المسار الآتي:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

يمكننا كتابة الاختبار التالي للتحقق من استدعاء الدالة Cache::get بالمتغيَر الوسيط (argument) الذي توقعناه:

```
use Illuminate\Support\Facades\Cache;

/**
 * مثال دالة بانية بسيطة.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
         ->with('key')
```

```
            ->andReturn('value');

        $this->visit('/cache')
            ->see('value');
    }
```

الواجهات الساكنات والدوال البانية المساعدة (Helper Function)

عدة دوال بانية "مُساعدة" قادرة على تنفيذ مهام شائعة مثل توليد الواجهات Laravel إضافة للواجهات الساكنة، يتضمن تقوم عدَة دوال مُساعدة بنفس وظيفة الواجهة .HTTP أو إرسال ردود (jobs)، وإطلاق الأحداث، وإرسال الأعمال (views)، السَاكنة المناظرة. ففي المثال الآتي سيكون استدعاء هذه الواجهة هو نظير استدعاء الدالة المساعدة:

```
return View::make('profile');

return view('profile');
```

لا يوجد أي فرق تطبيقي بين الواجهات السَاكنة والدوال المُساعدة بتاتًا. تستطيع اختبار الدَالة ونظيرتها الواجهة الساكنة بنفس الطريقة بالضبط. فلنأخذ المسار التالي مثلًا:

```
Route::get('/cache', function () {
    return cache('key');
});
```

على الصَنف الكامن وراء واجهة الذاكرة المؤقتة الساكنة وراء الكواليس. لذا نستطيع get مُساعد الذاكرة المؤقتة سيستدعي التابع الذي نتوقَعه رغم أننا نستخدم الدالَة (argument) كتابة الاختبار التالي للتحقق من أن التابع استُدعِيَ بنفس المتغيَر الوسيط المساعدة:

```
use Illuminate\Support\Facades\Cache;

/**
 * مثال دالة بانية بسيطة.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $this->visit('/cache')
        ->see('value');
}
```

كيفية عمل الواجهات الساكنة

الواجهة الساكنة هي صنف يوفر الوصول لكائن من الحاوي. الآلية التي تسمح بحدوث هذا موجودة في Laravel، في تطبيقات Illuminate\Support\Facades\ الساكنة إضافة لأي واجهة تنشئها ستوسّع صنف Laravel واجهات Facade. الصنف الأساسي Facade.

لتأجيل الاستدعاء من الواجهة لكائن __callStatic() الأساسي الدالة السحرية Facade يستغل صنف الواجهة الساكنة المؤقتة. بنظرة خاطفة على هذه التعليمات يمكن Laravel مُستَبْيَن من الحاوي. في المثال أدناه يُرسَل استدعاء نظام لذاكرة Cache: اسُتدعِيَت على صنف ذاكرة التخزين المؤقتة get للمرء افتراض أن الدالة الساكنة

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
```

```
    /**
     * عرض ملف المستخدم الشخصي
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Cache::get('user:'.$id);

        return view('profile', ['user' => $user]);
    }
}
```

قرب الجزء العلوي من الملف. هذه الواجهة بمثابة وكيل للوصول Cache لاحظ كيف أننا "نستورد" واجهة ذاكرة التخزين المؤقتة (implementation) الكامن وراء واجهة Illuminate\Contracts\Cache\Factory. لتعريف الاستخدام سيمرر أي المؤقتة. إن نظرنا لصنف Laravel استدعاء نقوم به باستخدام الواجهة الساكنة للنسخة الكامنة وراء خدمة ذاكرة Illuminate\Support\Facades\Cache. سنرى أنه لا وجود للتابع الساكن get:

```
class Cache extends Facade
{
    /**
     * تحصل على اسم المُكوِّن المُسجَل
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }
}
```

وتُعرِّف التابع Facade صنف الواجهة الساكنة الأساسي Cache بدلًا عن هذا، تُوسِّع واجهة ذاكرة التخزين المؤقتة الساكنة ارتباط الذاكرة المؤقتة Laravel وظيفة هذا التابع هي رد اسم ارتباط حاوي الخدمات. يستبين .()getFacadeAccessor من حاوي الخدمات عندما يشير المستخدم لأي دالة ساكنة بواجهة ذاكرة التخزين المؤقتة الساكنة، ثم يُنفِّذ الدالة المطلوبة cache على ذاك الكائن (get في هذه الحالة).

الواجهات الساكنة في الوقت الحالي (Real-Time Facades)

يمكنك معالجة أي صنف في تطبيقك كما لو كان واجهة ساكنة باستخدام الواجهات الساكنة في الوقت الحالي. لتوضيح كيفية نحتاج لإضافة نسخة podcast لكي ننشر .publish التابع Podcast الاستخدام فلندرس بديلا. فلنفترض مثلًا أن لنَموذَجِنَا Publisher :

```
namespace App;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @param  Publisher  $publisher
     * @return void
     */
    public function publish(Publisher $publisher)
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
```

```
    }
}
```

الناشر (mock) تسمح لنا إضافة تعريف استخدام للناشر للتابع باختبار الدالة بسهولة على انفراد بما أننا قادرون على تقليد يمكننا الحفاظ على .publish في كل مرة نستدعي فيها التابع publisher المُضَاف. مع ذلك يستلزمنا أن نُمَرِّر نسخة من باستخدام الواجهات الساكنة (explicitly) صراحة Publisher نفس القدرة على الاختبار دون الاضطرار إلى تمرير نسخة بالوقت الحالي. لتوليد واجهات ساكنة بالوقت الحالي، أسبق مجال أسماء الصنف المُستَوْرَد بالكلمة Facades:

```
namespace App;

use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @return void
     */
    public function publish()
    {
        $this->update(['publishing' => now()]);

        Publisher::publish($this);
    }
}
```

أو اسم الصنف الذي يظهر بعد (interface) يستبين تعريف استخدام الناشر من حاوي الخدمات باستخدام جزء الواجهة المُدمجات لتقليد Laravel يمكننا استخدام مُساعدي اختبار .Facades استخدام واجهات ساكنة بالوقت الحالي بعد سابقة استدعاء هذا التابع عند الاختبار:

```
namespace Tests\Feature;

use App\Podcast;
use Tests\TestCase;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;

class PodcastTest extends TestCase
{
    use RefreshDatabase;
  /**
     * مثال اختبار
     *
     * @return void
     */
    public function test_podcast_can_be_published()
    {
        $podcast = factory(Podcast::class)->create();

        Publisher::shouldReceive('publish')->once()->with($podcast);

        $podcast->publish();
    }
}
```