# Multi-tenant Laravel in Production

**Samuel Štancl**
samuel@archte.ch
Founder, ArchTech

# Contents

# 1. Introduction

In this book, I'll dive into everything I've learned about building and running multi-tenant Laravel applications from the years of developing Tenancy for Laravel, our SaaS boilerplate, consulting with clients using the package in production, and more.

This book will cover creating multi-tenant applications in Laravel from a **real-world, practical** point of view.

As the name implies, it's focused on the challenges you may face running a multi-tenant application in production. That said, the book also goes into a lot of depth about how the package works in the first place, what multi-tenancy setup will work best for your use case, and how to structure your application.

In other words, you should find plenty of valuable info here regardless of whether you're just getting started with your app or if you're already running an application in production.

Let's put this book into context, compared to the other resources available for the package.

The package documentation serves as a simple introduction and technical reference for using the package.

The sponsor docs serve as a collection of directly usable things that you can use in your application in a pretty much drop-in way.

This book aims to go over the **real-world considerations** of building a production multi-tenant application.

Some chapters will give direct solutions to common issues/features. Other chapters will explain all the nuances to consider and show how I'd go about solving a particular problem. Seeing my thinking should give you a helpful reference for how to think about building multi-tenant applications.

This book has no required reading order – besides the *2. How the package works* chapter being strongly recommended to read in full to get a proper understanding of how the package works under the hood, if you're not completely familiar with that. With all of the other chapters, you can feel free to jump to whatever interests you most.

## 1.1. Terminology recap

This book will use the following terms with very specific meanings, so it's good to be familiar with them.

**Tenant identification** refers to identifying a tenant from a request using an identification middleware.

**Tenancy initialization** refers to initializing tenancy for a given tenant (= setting the tenant as the current tenant and generally using **tenancy bootstrappers** to make the application tenant-aware).

**Automatic mode** refers to running bootstrappers upon tenancy initialization to make the application tenant-aware.

**Manual mode** refers to *not* using bootstrappers upon tenancy initialization. Note that this is just a technicality and we don't really cover it in this book since few people use that setup. The

point is that you *can* use it, and it's good to know that identification and initialization can be decoupled, as well as the terminology we use for this.

**Tenancy bootstrappers** are classes that scope the application for a given tenant. Think setting cache key prefixes, changing the default database connection, adjusting filesystem paths, etc. These classes essentially transition the application in and out of a specific tenant's context.

**Tenants** are instances of your configured tenant model (see *2.1. Tenants*).

**Tenant key** is the unique identifier of a tenant – like a primary key, but used for the purposes of the package. In other words, you may use a different column than the primary key (though it's typically the same column) for your tenant key and the package will use exclusively that column when querying tenants.

**Route action** refers to the function that gets executed to process a request for a given route. Generally a controller method.

**Central app** refers to the part of the application that's central – no bootstrappers used, no tenant identification middleware used.

**Tenant app** refers to the part of the application that's in the tenant context. Typically the bulk of the application logic is in the tenant app.

## 1.2. Multi-tenancy vs multi-instance

Multi-tenant applications differ from multi-instance applications in that you only need a *single deployment* of the application to serve multiple tenants.

A multi-instance setup would be one where you have the application deployed separately for each client.

The massive advantage of multi-tenancy is that you only need to maintain a single deployment. This saves you from enormous infrastructure complexity at the cost of some application complexity.

That said, our package was specifically written with minimizing application complexity in mind – that's its **entire purpose**. The goal of the package is for it to be relatively easy to add it to an existing application without any large scale code changes.

That way, you get the lowest possible infrastructure complexity *and* very low application-level complexity at the same time.

# 2. How the package works

This section will explain what exactly happens when a request comes in and tenancy is initialized.

Before we get to that though, let's first clarify what exactly tenants are.

## 2.1. Tenants

Tenants are relatively simple models stored in the central database. Any model that implements the `Stancl\Tenancy\Contracts\Tenant` interface can be used as a tenant:

```
interface Tenant
{
    /** Get the name of the key used for identifying the tenant. */
    public function getTenantKeyName(): string;

    /** Get the value of the key used for identifying the tenant. */
    public function getTenantKey(): int|string;

    /** Get the value of an internal key. */
    public function getInternal(string $key): mixed;

    /** Set the value of an internal key. */
    public function setInternal(string $key, mixed $value): static;
}
```

Here's what the methods do:

`getTenantKeyName()` and `getTenantKey()` are a bit like `getRouteKey()` and `getRouteKeyName()`[1]. Route keys are used for finding models based on route parameters. Similarly, tenant keys are used for finding tenants *from any code in our package*. The reason for having an extra key is so that you can use any columns for primary keys[2] and route keys, without tenancy-related logic affecting either one.

`getInternal()` and `setInternal()` are used for storing "internal" properties on the tenant. For instance, the package may need to store `db_name` after a tenant database is created. There's no pre-defined list of these properties, and we reserve the right to add more of them in the future, so we use this special abstraction for storing them instead of requiring that you have a column for each one. The default `Stancl\Tenancy\Database\Models\Tenant` base model uses a JSON column for this.

### 2.1.1. Default Tenant model

Aside from the interface, the package also ships with a base model that implements the interface as well as many convenient quality-of-life features.

Odds are, your tenant model extends this base model – that's the recommended way of getting started with the package, documented for instance in the quickstart guide.

---

[1] See `Illuminate\Database\Eloquent\Model::getRouteKey()`

[2] Similarly to route keys, primary keys use `getKeyName()` and `getKey()` methods. This means that with route keys and tenant keys added, there are 3 types of unique keys you can use in your models.

Speaking of the quickstart guide, let's take a look at how the tenant model is defined there:

```
namespace App\Models;

use Stancl\Tenancy\Database\Models\Tenant as BaseTenant;
use Stancl\Tenancy\Contracts\TenantWithDatabase;
use Stancl\Tenancy\Database\Concerns\HasDatabase;
use Stancl\Tenancy\Database\Concerns\HasDomains;

class Tenant extends BaseTenant implements TenantWithDatabase
{
    use HasDatabase, HasDomains;
}
```

We can see that the model is extending the base Tenant model – `Stancl\Tenancy\Database\Models\Tenant` – and implementing the `TenantWithDatabase` interface. It also uses the `HasDatabase` and `HasDomains` traits.

Let's take a look at the interface first:

```
interface TenantWithDatabase extends Tenant
{
    /** Get the tenant's database config. */
    public function database(): DatabaseConfig;
}
```

This interface extends the original `Tenant` interface and tells the package that we want to use multi-database tenancy. The `database()` method is used by the package to configure the tenant's database connection.

The reason why this is a separate interface is that the package supports both single-database and multi-database tenancy. Therefore the base `Tenant` interface, as well as the base `Tenant` model, are unopinionated with regards to which tenancy setup you're using.

Now let's take a look at the traits:

```
trait HasDatabase
{
    use HasInternalKeys;

    public function database(): DatabaseConfig
    {
        /** @var TenantWithDatabase&Model $this */
        $databaseConfig = [];

        // ...

        return new DatabaseConfig($this, $databaseConfig);
    }
}
```

This trait is the counterpart to the `TenantWithDatabase` interface. It provides a default `database()` implementation that depends on the default implementation of the internal keys logic.

And the `HasDomains` trait:

```
trait HasDomains
{
```

```
    public function domains()
    {
        return $this->hasMany(config('tenancy.models.domain'),
Tenancy::tenantKeyColumn());
    }

    public function createDomain($data): Domain
    {
        $class = config('tenancy.models.domain');

        if (! is_array($data)) {
            $data = ['domain' => $data];
        }

        $domain = (new $class)->fill($data);
        $domain->tenant()->associate($this);
        $domain->save();

        return $domain;
    }
}
```

This simply defines the `domains()` relationship, as well as a convenient method for creating domains for tenants that can be called as:

```
$tenant->createDomain([
    'domain' => 'example.com',
    'is_primary' => true,
    'foo' => 'bar',
]);

// or
$tenant->createDomain('example.com');
```

Notice that the `HasDomains` trait doesn't have a counterpart interface. The reason for this is that there are more ways to work with domains than just having a `domains(): HasMany` relationship, and the package calls this method in only very few places. Therefore, to give you more flexibility with how you define domains for your tenant, it doesn't require a specific interface.

Now that we've covered what we're adding to the base `Tenant` model in our `App\Models\Tenant` model, let's take a look at the base model itself:[3]

```
class Tenant extends Model implements Contracts\Tenant
{
    use VirtualColumn,
        Concerns\CentralConnection,
        Concerns\GeneratesIds,
        Concerns\HasInternalKeys,
        Concerns\TenantRun,
        Concerns\InitializationHelpers,
        Concerns\InvalidatesResolverCache;

    protected static $modelsShouldPreventAccessingMissingAttributes = false;
    protected $guarded = [];

    public function getTenantKeyName(): string
    {
```

[3]The code below is a bit simplified – some methods and docblocks have been stripped out for brevity.

```
        return 'id';
    }

    public function getTenantKey(): int|string
    {
        return $this->getAttribute($this->getTenantKeyName());
    }
}
```

You can see that this class implements the `getTenantKey()`/`getTenantKeyName()` methods. It also uses the following traits:

- `VirtualColumn`[4]: this trait lets you set any property on a model, even if the respective column doesn't exist. When that happens, the property is stored in a JSON column (`data` in this case),
- `CentralConnection`: this trait forces the model to use the central connection connection, so that we can interact with it even when the default connection is set to a tenant connection,
- `GeneratesIds`: this trait generates UUIDs for the tenant's `id` column[5],
- `HasInternalKeys`: this trait provides the `getInternal()` and `setInternal()` methods for storing internal properties on the tenant. It essentially just prefixes the keys with `tenancy_` and stores them in the `data` column using the `VirtualColumn` logic,
- `TenantRun`: this trait provides the `run()` method for running code as the tenant. It's a helpful method for running something within a specific tenant's context, with a guarantee that it will revert to the previous context (which can be the central context or another tenant),
- `InitializationHelpers`: this trait provides the `$tenant->enter()` and `$tenant->leave()` methods. These methods are just aliases for methods on the `Tenancy` singleton,
- `InvalidatesResolverCache`: this trait invalidates the resolver cache when the tenant is saved. Resolvers are used by identification middleware.

### 2.1.2. Configuring the model

The last step to making the package use our tenant model is to set the `tenancy.models.tenant` config key to the fully qualified class name of our tenant model:

```
// config/tenancy.php
return [
    'models' => [
        'tenant' => App\Models\Tenant::class,
    ],
];
```

> In summary, tenants are models stored in the central database that must implement methods defined by the Tenant interface. For the package to use our model, we must configure it as the tenant model in tenancy.models.tenant.

## 2.2. Tenant identification

Now that it's clear *what* tenants are, let's talk about how they're identified, or even more generally: what tenant identification is.

[4] https://github.com/archtechx/virtualcolumn

[5] For more information about why we use UUIDs instead of regular autoincrement ids, see the *5.1. Tenant keys* chapter.

Tenant identification is the *first step of the tenancy logic* that takes place when a request comes in. It's essentially the process of figuring out which tenant the request is for – if any. This logic is generally done using **middleware**.

### 2.2.1. Middleware

The package comes with a few middleware that can be used for tenant identification:

- `InitializeTenancyByDomain`: identifies tenants using the domain the request is being made on,
- `InitializeTenancyBySubdomain`: identifies tenants using the subdomain the request is being made on,
- `InitializeTenancyByDomainOrSubdomain`: identifies tenants using the domain *or* subdomain the request is being made on,
- `InitializeTenancyByPath`: identifies tenants using a route parameter (`{tenant}` in the route definition),
- `InitializeTenancyByRequestData`: identifies tenants using request data (headers, query parameters, cookies, etc.),
- `InitializeTenancyByOriginHeader`: identifies tenants using the `Origin` header.

We expand on these in depth in the *3. Picking the right setup for your application* section.

Note that these middleware are just the ones that come with the package. You can easily create your own middleware for tenant identification if you need any custom logic.

There's some amount of complexity in these first-party middleware, but that's mostly to work well with our other abstractions like resolvers (see the next section), onFail logic, universal routes, and early identification[6]. But if you wanted to write an identification middleware out of the box, it'd essentially be:

```
namespace App\Http\Middleware;

class MyInitializeTenancyByDomain
{
    public function handle($request, $next)
    {
        $domain = $request->getHost();

        if ($domain = Domain::where('domain', $domain)->first()) {
            tenancy()->initialize($domain->tenant);
        } else {
            $this->onFail();
        }

        return $next($request);
    }

    protected function onFail()
    {
        abort(404);
```

[6] *onFail logic* refers to the behavior that should take place when the tenant cannot be identified using the middleware, as shown in the code example above. In all first-party middleware this behavior is configurable using static properties. *Universal routes* are routes that are accessible both in the tenant context and the central context – see the next section for an explanation. *Early identification* refers to executing the middleware before the controller constructor is executed, also explained in an upcoming section.

```
    }
}
```

### 2.2.2. Resolvers

Resolvers are an abstraction used in the identification middleware.

They accept data in some simpler form – for instance the domain identification middleware extracts the domain from the `Request` instance and passes it to the resolver as a string – and return a tenant.

There are a few reasons for resolvers being a separate abstraction:
1. Simplifying the middleware.
2. Reusing logic – different middleware may use the same resolver.
3. Cached lookup – resolvers can cache the `Tenant` instance for the data they were given, so that we can avoid making a query to the central database at the start of each request. Keep in mind that queries are fast, but establishing connections can be slightly slow (with some variations depending on your database setup and php-fpm configuration[7]), so it's wise to avoid doing so when possible.
4. Cache invalidation – The caching logic being moved to resolvers is also helpful for invalidation since it allows us to invalidate the cache for *every* resolver[8] whenever there's a change made to a tenant.

When writing custom identification middleware, it's ideal to use a resolver for the reasons mentioned above. Whenever possible, you should use an existing resolver to save yourself the trouble of writing one from scratch.

### 2.2.3. Universal routes

*Universal routes* refers to routes that are accessible in both the central context and the tenant context. Since the addition of early identification in version 4, the implementation has become more complex, but the best way to understand universal routes is to think about them the way they were implemented in v3: the `$onFail` being just `return $next($request)` – if we fail at identifying a tenant, we just carry on with the request execution, without running any tenancy logic.

### 2.2.4. Early identification

Early identification refers to running the tenancy middleware *before* controller constructors are executed.

That may be an accurate description of what early identification is, but it doesn't actually tell us much, so let's try another way.

In Laravel, there are two middleware stacks:
1. the route middleware stack – specific to a given route,
2. the kernel/global middleware stack – runs on *all* routes, and much sooner than route middleware.

[7] See *10.1. Persistent PDO connections*

[8] Resolvers are registered in the `tenancy.php` config

Normally you place the tenancy middleware in the route middleware stack, for example in `routes/tenant.php`:

```
Route::middleware([
    'web',
    Middleware\InitializeTenancyByDomain::class,
    Middleware\PreventAccessFromUnwantedDomains::class,
    Middleware\ScopeSessions::class,
])->group(function () {
    Route::get('/', function () {
        return 'Current tenant: ' . tenant('id') . "\n";
    });
});
```

However, this can cause issues if you have a controller that injects dependencies in its constructor[9] .

For instance, if you injected a class called `OpenAIClient` and wanted to use the tenant's OpenAI keys that you set as the `services.openai` config once tenancy is initialized, it would be still using your central OpenAI keys since the dependency got injected before tenancy got initialized.

The reason for this is that Laravel allows[9] configuring middleware in the controller constructor, which means that the constructor *has to* get executed before the route middleware.

Note that there are various ways to get around this, the best one being to just inject dependencies in *route actions* (= controller methods) if possible.

That said, there are cases where you *will* need early identification – in the form of running tenancy middleware in the global middleware stack – and Tenancy v4 implements that in a convenient way. You put your identification middleware in the kernel stack and then use these "flag" middleware (`tenant`, `central`, `universal`) to indicate whether a route should be run in the tenant context, the central context, or either one.

This is covered in a lot of depth in our documentation, so you should go read that page to properly understand early identification.

## 2.3. Tenancy initialization

We've already hinted at this in the previous section. Tenancy initialization is the step where we transition the application to the tenant context.

In the custom middleware above – `MyInitializeTenancyByDomain` – the middleware does two things:

1. Tenant identification (finding a `Tenant` using a `Domain` based on `$request->getHost()`)
2. Tenancy initialization (calling `tenancy()->initialize($domain->tenant)`)

The `initialize()` method is how we enter a tenant's context. Its counterpart is `tenancy()->end()` which switches back to the central context.

Under the hood, these two methods call the *tenancy bootstrappers*.

[9] It's actually a bit more complex than that since Laravel has taken steps to improve the situation, so now this only applies to controllers that are an instance of `Illuminate\Routing\Controller` *and do not* implement `Illuminate\Routing\Controllers/HasMiddleware`. This is covered in a lot of depth and with good visualizations on our documentation page about early identification.

### 2.3.1. Tenancy bootstrappers

A tenancy bootstrapper is a class that transitions the application's context to that of a given tenant. It's a class that implements the `TenancyBootstrapper` interface:

```
interface TenancyBootstrapper
{
    public function bootstrap(Tenant $tenant): void;

    public function revert(): void;
}
```

The `tenancy()->initialize()` method essentially loops over all registered bootstrappers and calls their `bootstrap()` method with the tenant as an argument.

The `tenancy()->end()` method does the same, but calls the `revert()` method.

Bootstrappers should be *fully reversible*, meaning that the state of the application after calling `revert()` should be the same as it was before `bootstrap()` was called.

## 2.4. Creating tenants

The package is configured in two main ways – simple configuration goes into the `config/tenancy.php` file, while more dynamic things are configured in the `TenancyServiceProvider`. That's also where the process of creating a tenant is configured:

```
// app/Providers/TenancyServiceProvider.php
class TenancyServiceProvider extends ServiceProvider
{
    public function events()
    {
        return [
            // Tenant events
            Events\CreatingTenant::class => [],
            Events\TenantCreated::class => [
                JobPipeline::make([
                    Jobs\CreateDatabase::class,
                    Jobs\MigrateDatabase::class,
                    // Jobs\SeedDatabase::class,
                    // Jobs\CreateStorageSymlinks::class,

                    // Your own jobs to prepare the tenant.
                    // Provision API keys, create S3 buckets, anything you want!
                ])->send(function (Events\TenantCreated $event) {
                    return $event->tenant;
                })->shouldBeQueued(false), // `false` by default, but you likely
want to make this `true` in production.

                // Listeners\CreateTenantStorage::class,
            ],
        // ...
        ];
    }
    // ...
}
```

You can see that by default, we enable the `CreateDatabase` and `MigrateDatabase` jobs. The `JobPipeline` is our own abstraction for chaining jobs into event listeners that are optionally queuable.

This means that if you wanted to also seed the tenant database upon tenant creation, you could just uncomment the `SeedDatabase` line. And you can add any custom jobs here.

### 2.4.1. TenancyServiceProvider

This `events()` method is also where events related to initializing tenancy are configured:

```
Events\TenancyInitialized::class => [
    Listeners\BootstrapTenancy::class,
],
Events\TenancyEnded::class => [
    Listeners\RevertToCentralContext::class,
],
```

Tenancy doesn't run bootstrappers directly, it simply dispatches a `TenancyInitialized` event when `tenancy()->initialize()` is called. The `BootstrapTenancy` listener configured here is what actually calls the bootstrappers. And similarly, `TenancyEnded` is dispatched when `tenancy()->end()` is called, and `RevertToCentralContext` calls `revert()` in the configured bootstrappers[10].

This is what we refer to as the event-based architecture of the package. Since multi-tenancy is a complex problem that may need to be addressed differently in each application, we make the package as configurable as possible – using the config file, various static properties across the codebase, and event listeners.

## 2.5. Summary

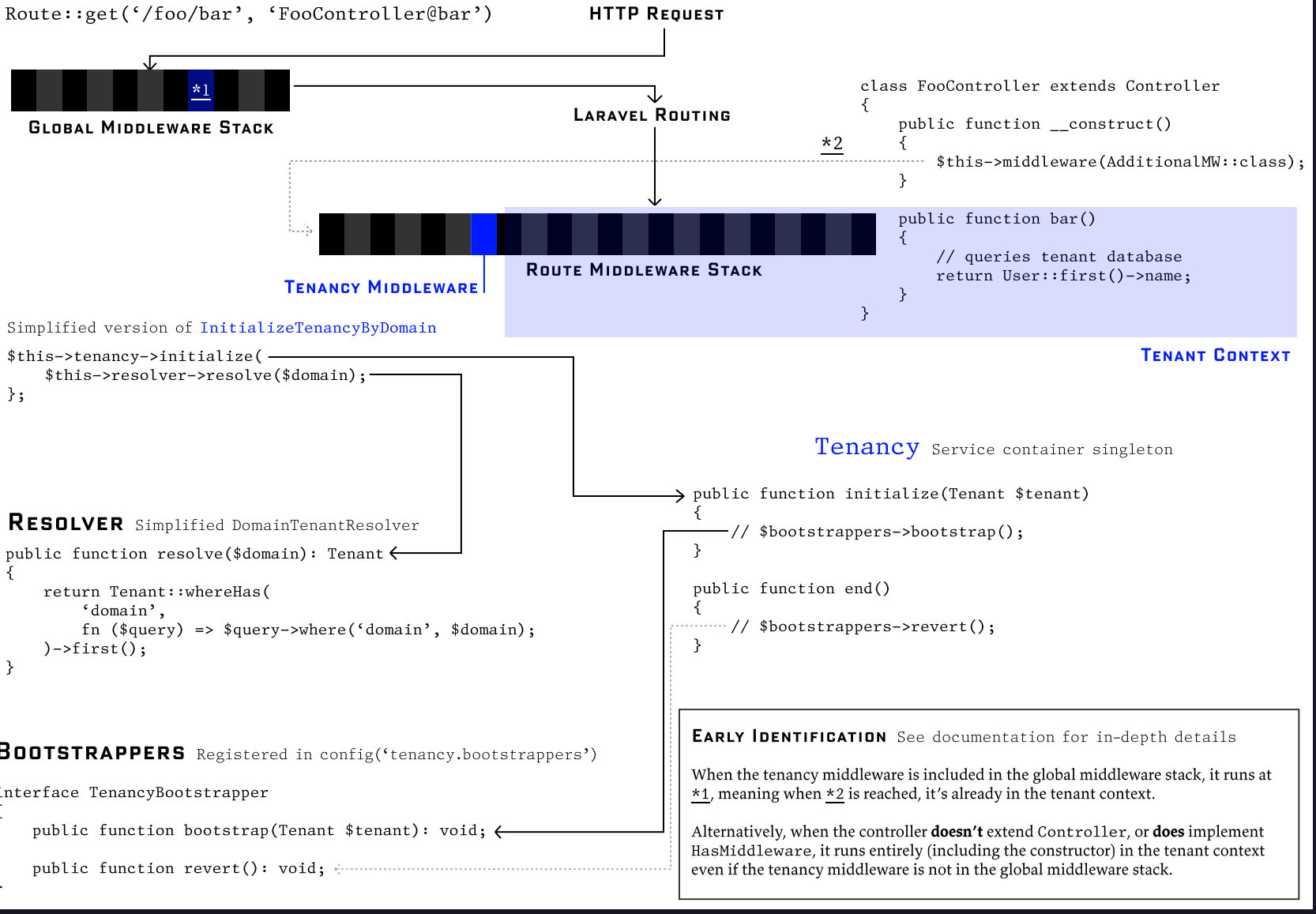
> In summary, tenants are models stored in the central database that must implement methods defined in the Tenant interface. Tenant identification is the process of figuring out which tenant the request is for, and it's generally done using middleware. Tenancy initialization is the step where we transition the application to the tenant context. This is done using the tenancy()->initialize() method, which calls the registered *tenancy bootstrappers*.

On the next page, you can see a schematic summarizing the tenancy logic that takes place as part of routing. It also includes a visualization of early identification, but going forward we are not going to cover it in too much depth. I'd highly recommend taking the time to read the *Early identification* page of the documentation to better understand some edge cases with Laravel's routing.

For an in-depth look at how the routing works in some edge cases, take a look at the *Universal routes* and *Early identification* pages of the package documentation.

Now that we got how the package works out of the way, we can talk about the things you bought this book for: production considerations, best practices, common pitfalls, security, and more.

[10] An interesting detail is that this listener runs the bootstrappers in reverse order. Since bootstrappers can depend on each other – based on the order they're defined in – they should also be reverted in an order that takes these dependencies into account.


Route::get('/foo/bar', 'FooController@bar')
HTTP Request
*1
Global Middleware Stack
Laravel Routing
class FooController extends Controller
{
public function __construct()
{
*2
$this->middleware(AdditionalMW::class);
}
Route Middleware Stack
Tenancy Middleware
public function bar()
{
// queries tenant database
return User::first()->name;
}
}
Tenant Context
Simplified version of InitializeTenancyByDomain
$this->tenancy->initialize(
$this->resolver->resolve($domain);
};
Tenancy Service container singleton
public function initialize(Tenant $tenant)
{
// $bootstrappers->bootstrap();
}
public function end()
{
// $bootstrappers->revert();
}
Resolver Simplified DomainTenantResolver
public function resolve($domain): Tenant
{
return Tenant::whereHas(
'domain',
fn ($query) => $query->where('domain', $domain);
)->first();
}
Bootstrappers Registered in config('tenancy.bootstrappers')
interface TenancyBootstrapper
{
public function bootstrap(Tenant $tenant): void;
public function revert(): void;
}
Early Identification See documentation for in-depth details
When the tenancy middleware is included in the global middleware stack, it runs at *1, meaning when *2 is reached, it's already in the tenant context.
Alternatively, when the controller doesn't extend Controller, or does implement HasMiddleware, it runs entirely (including the constructor) in the tenant context even if the tenancy middleware is not in the global middleware stack.