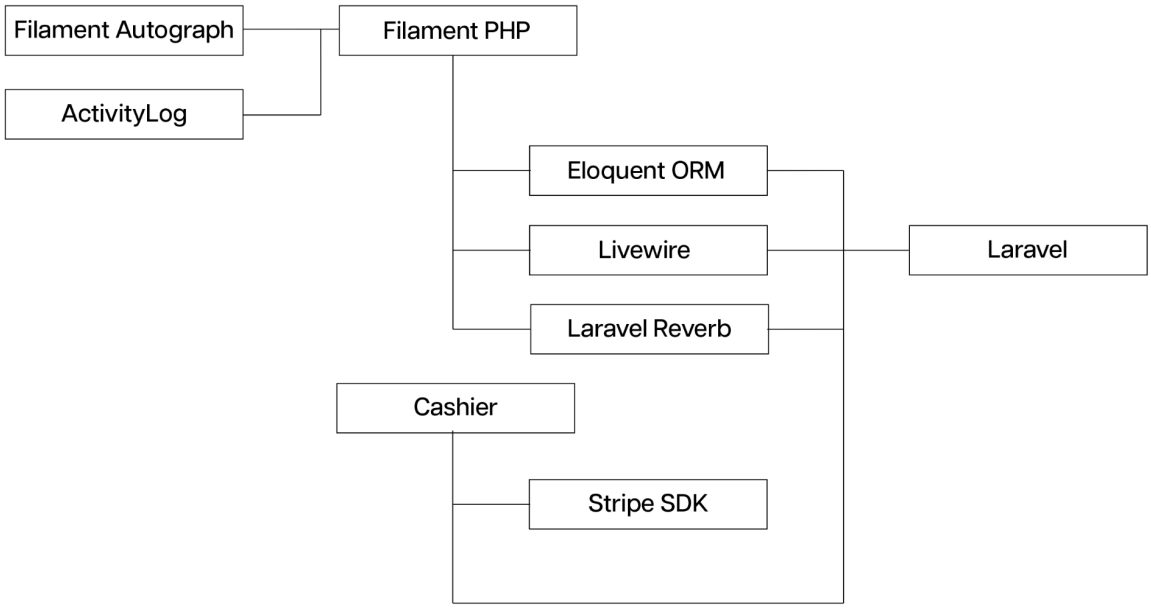
Introduction

SuccessManager is powered by the PHP programming language and the Laravel framework that promotes a MVC architecture. To simplify the development process and eliminate the repetitive task of creating dashboards, CRUD views and handling the inputs, another library that builds on top of Laravel called Filament PHP was utilized.

Filament PHP takes away the load of developing forms, designing input components, validating input and managing relationships. Furthermore, notifications and some authentication procedures are handled by Filament as well using custom override classes that extend the Filament core classes. In addition, the connection between the frontend SPA (single page application) that consists of processes such as routing, lazy loading of assets and XHR requests are also handled directly by Filament PHP through Laravel's Livewire set of packages.

The Dependency Tree



The project mainly consists of packages that are designed around Laravel. There are the two libraries Filament Autograph –to collect signatures drawn with a touchpad– and ActivityLog –that offers an inheritable trait for Model classes to allow for activity logging– that depend on Filament.

Filament also depends on the three main packages supplied by Laravel that are normally optional, Eloquent ORM –for database transactions–, Livewire –for server rendered HTML forms– and Reverb –for real time push notifications.

Cashier is an additional library that works with Filament to provide billing capabilities. It depends on the Stripe SDK and Laravel. It may be used standalone with Laravel, hence the nonexistent dependency relationship with Filament, but when together, it offers additional functionality.

Full dependency list is below and the full package list is compiled by the PHP package manager Composer accordingly at the time of installation.

```
"discoverydesign/filament-gaze": "^1.1", 1.1.3, updates to 1.1.12
"filament/filament": "^3.2", v3.2.102, updates to v3.2.117
"hasnayeen/themes": "^3.0", v3.0.23
"joaopaulolndev/filament-pdf-viewer": "^1.0", v1.0.2, updates to v1.0.6
"laravel/cashier": "^15.4", v15.4.1, updates to v15.4.3
"laravel/framework": "^11.9", v11.20.0, updates to v11.27.2
"laravel/reverb": "^1.0", v1.1.0, updates to v1.4.1
"laravel/sanctum": "^4.0", v4.0.2, updates to v4.0.3
"laravel/tinker": "^2.9", v2.9.0, updates to v2.10.0
"league/flysystem-aws-s3-v3": "^3.0", 3.28.0, updates to 3.29.0
"mohammadhprp/filament-ip-to-country-flag-column": "^1.0", v1.0.2
"nembie/iban-rule": "^1.0", 1.0.3
"owenvoke/blade-fontawesome": "^2.6", v2.6.0
"predis/predis": "^2.2", v2.2.2
"pxlrbt/filament-spotlight": "^1.2", v1.2.2
"rmsramos/activitylog": "^1.0", v1.0.4, updates to v1.0.7
"saade/filament-autograph": "^3.1", v3.1.0
"spatie/laravel-medialibrary": "^11.8", 11.8.2, updates to 11.9.1
"spatie/laravel-permission": "^6.9", 6.9.0
"stechstudio/filament-impersonate": "^3.13", 3.13, updates to 3.14
"stripe/stripe-php": "^13.18" v13.18.0
```

Structure of the Application

All services of the web app are handled by the ConsumerPanelService provider that extends Filament's \Filament\PanelProvider abstract class.

The panel function of the provider returns a panel object with SuccessManager's respective traits configured.

```
return $panel
    ->id( id: 'consumer')
    ->path( path: 'manager')
    ->default()
    ->tenant( model: \App\Models\Tenant::class, slugAttribute: 'slug')
    ->brandLogo( logo: "/images/Success_Manager-Logo-01.png")
    ->viteTheme( theme: 'resources/css/filament/consumer/theme.css')
    ->brandLogoHeight( height: "48px")
    ->colors([
        'primary' => Color::Blue,
    ])
```

Every function provided by Filament are functions that essentially return a modified copy of the calling object. This allows for chaining of these object methods to finally return a completely configured Panel object.

```
->discoverResources(in: app_path( path: 'Filament/Consumer/Resources'), for: 'App\\Filament\\Consumer\\Resources')
->discoverPages(in: app_path( path: 'Filament/Consumer/Pages'), for: 'App\\Filament\\Consumer\\Pages')
->pages([
    Pages\Dashboard::class,
])
->discoverWidgets(in: app_path( path: 'Filament/Consumer/Widgets'), for: 'App\\Filament\\Consumer\\Widgets')
->widgets([
    Widgets\AccountWidget::class,
])
```

Filament will accept 3 types of files that are either a Widget, Page or Resource that each have respective abstract classes defined within Filament's spec. The Panel object must be configured with search paths on where to look for these three types of files which each contain a PHP class.

Resources are a full model that contain full CRUD operations with a Page for each and any respective Widgets.

Pages are standalone pages that are not associated with a model that provide some database operations. These are usually one of the 4 CRUD operations.

Widgets are simple HTML components that are embedded into pages for easy presentation of information.

```
->tenantProfile( page: EditSyncedTenantProfile::class)
->middleware([
    EncryptCookies::class,
    AddQueuedCookiesToResponse::class,
    StartSession::class,
    AuthenticateSession::class,
    ShareErrorsFromSession::class,
    VerifyCsrfToken::class,
    SubstituteBindings::class,
    DisableBladeIconComponents::class,
    DispatchServingFilamentEvent::class,
    PermissionTenantChooser::class,
    EnsureNoCardSetup::class,
])
->tenantMiddleware([
    SetTheme::class
])
->authMiddleware([
    Authenticate::class,
]);
```

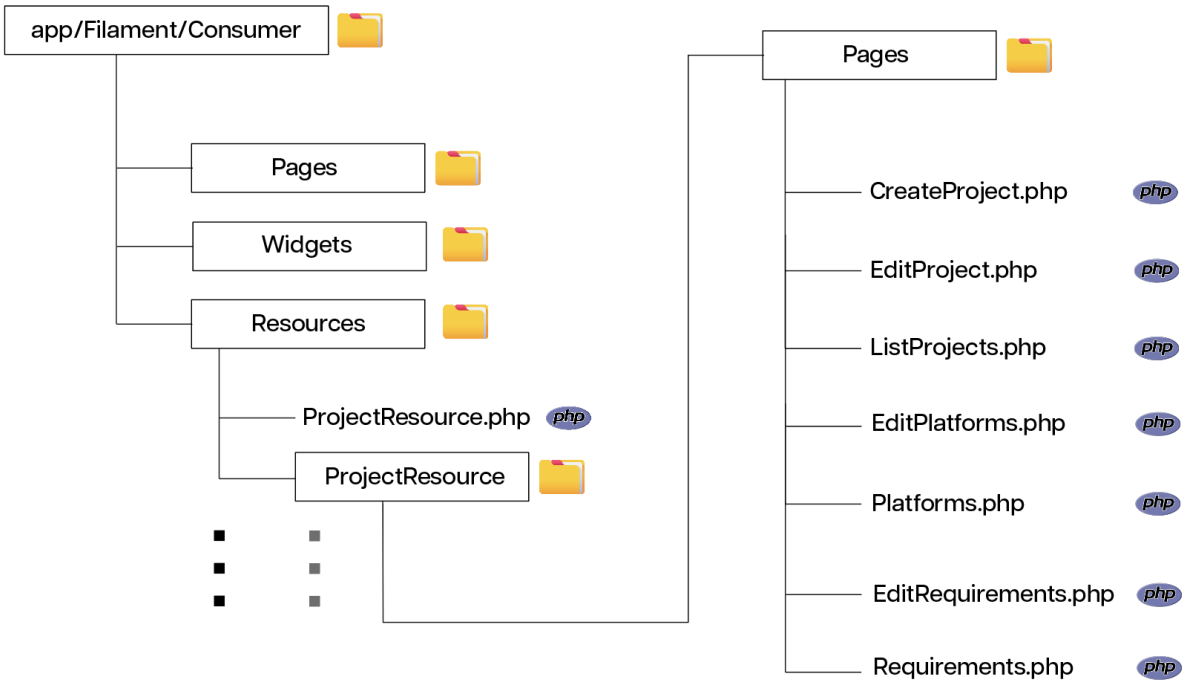
Panel must also be configured with Middleware that intercept the incoming HTTP requests to validate them against security and routing policies. Filament accepts three types of Middleware for authentication, for setting tenants and for all requests regardless respectively.

Each Middleware is a class that inherits Laravel's Http/Middleware class and overrides the "handle" function. Example is shown below.

```php
<?php

namespace App\Http\Middleware;

use ...

Yiğit Kerem Oktay
class
EnsureNoCardSetup
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
     */
    Yiğit Kerem Oktay
    public function handle(Request $request, Closure $next): Response
    {

        $card = CustomerPaymentMethod::where([
            ["creator_id", auth()->id()],
            ["stripe_ref", null],
            ["type", "stripe"],
        ])->first();

        if ($card == null || !$request->isMethod( method: 'get')) {
            return $next($request);
        } else {
            session()->put('redirect_url', $request->url());
            return redirect()->route( route: 'card-setup');
        }
    }
}
```

This example Middleware is one that checks if there is a pending card transaction and redirects the customer to the transaction page where they will have to either cancel or complete it before continuing to use the application. A middleware may call the $next method which was passed as an argument in the original method to let Laravel execute the remaining middlewares and if none are remaining, the app itself. If required, a middleware may also redirect to another page or throw an HTTP error to abort the request altogether.

Structure of a Resource



A Resource is a Filament's way of grouping together MVCs belonging to a single resource which is a model in the database. For each resource, a main class like the ProjectResource.php in this example must be created. Then, all pages relating to this resource must be grouped together in a ResourceNameResource/Pages/(Create/Edit/List)ResourceName.php file format like the example.

The ProjectResource.php file –and all the ResourceNameResource.php files– inherit the \Filament\Resources\Resource abstract resource class. These files override the methods listed in the table to provide specific functionality to the resources.

Method Name	Purpose	Returns	Example
form()	Rendering the CRUD form fields.	A Filament\Form object with the fields embedded into the schema property.	

table()	Rendering the table that displays all the records of the resource	A Filament\Table resource with the fields incorporated into the columns property.	
getRelations()	Setting the relationships the resource has to other resources	An array of Filament\RelationManager classes	
getPages()	Setting up routing for the CRUD and custom pages related to a resource	An array of Filament\Page objects.	

There exists more properties that can be overridden but they are much beyond the scope of this project and hence excluded from the documentation.

A resource is defined within this file and the properties set forth in this file are inherited in each of the Resouce's pages. The fields set here will be used as the forms in CR operations of CRUD, the table will be used as a list in the list page.

Take the example of the AgreementResource, it has all of its fields defined in the Resource file and all of its pages inherit the form from the resource.

```php
return $form
    ->schema([
        Forms\Components\TextInput::make( name: 'name')
            ->required()
            ->maxLength( length: 255),
        Forms\Components\ToggleButtons::make( name: 'status')
            ->label( label: "Status")
            ->options( options: AgreementStatus::class)
            ->inline()
            ->required(),
        Forms\Components\Textarea::make( name: 'description'),
        Forms\Components\Select::make( name: 'tenant_id')
            ->required()
            ->relationship( name: 'tenant', titleAttribute: 'legal_name')
            ->native( condition: false)
            ->searchable(),
        Forms\Components\Select::make( name: 'user_id')
            ->required()
            ->relationship( name: 'user', titleAttribute: 'first_name')
            ->native( condition: false)
            ->searchable(),
        Forms\Components\Select::make( name: 'project_id')
            ->relationship( name: 'project', titleAttribute: 'name')
            ->native( condition: false)
            ->searchable(),
        Forms\Components\Select::make( name: 'customer_subscription_id')
            ->relationship( name: 'customer_subscription', titleAttribute: 'name')
            ->native( condition: false)
            ->searchable(),
        Forms\Components\FileUpload::make( name: 'files')
            ->maxFiles( count: 1)
            ->maxSize( size: 5000)
            ->visibility( visibility: 'private')
            ->directory( directory: 'contracts')
            ->acceptedFileTypes(['application/pdf'])
            ->required()
    ]);
```

form() method of AgreementResource

```php
return $table
    ->columns([
        Tables\Columns\TextColumn::make( name: 'name')
            ->searchable(),
        Tables\Columns\TextColumn::make( name: 'tenant.legal_name')
            ->numeric()
            ->sortable(),
        Tables\Columns\TextColumn::make( name: 'user')
            ->label( label: 'Authorised Representative')
            ->formatStateUsing(fn(Agreement $agreement): string => $agreement->user()->first()->getFilamentName())
            ->sortable(),
        Tables\Columns\TextColumn::make( name: 'status')
            ->badge()
            ->sortable(),
        Tables\Columns\TextColumn::make( name: 'created_at')
            ->dateTime()
            ->sortable(),
        Tables\Columns\TextColumn::make( name: 'updated_at')
            ->dateTime()
            ->sortable()
            ->toggleable(isToggledHiddenByDefault: true),
    ])
```

table() method of AgreementResource

Only override we make is during the creation of Agreements where we run additional code after saving the newly created model and send notifications using the afterSave() method.

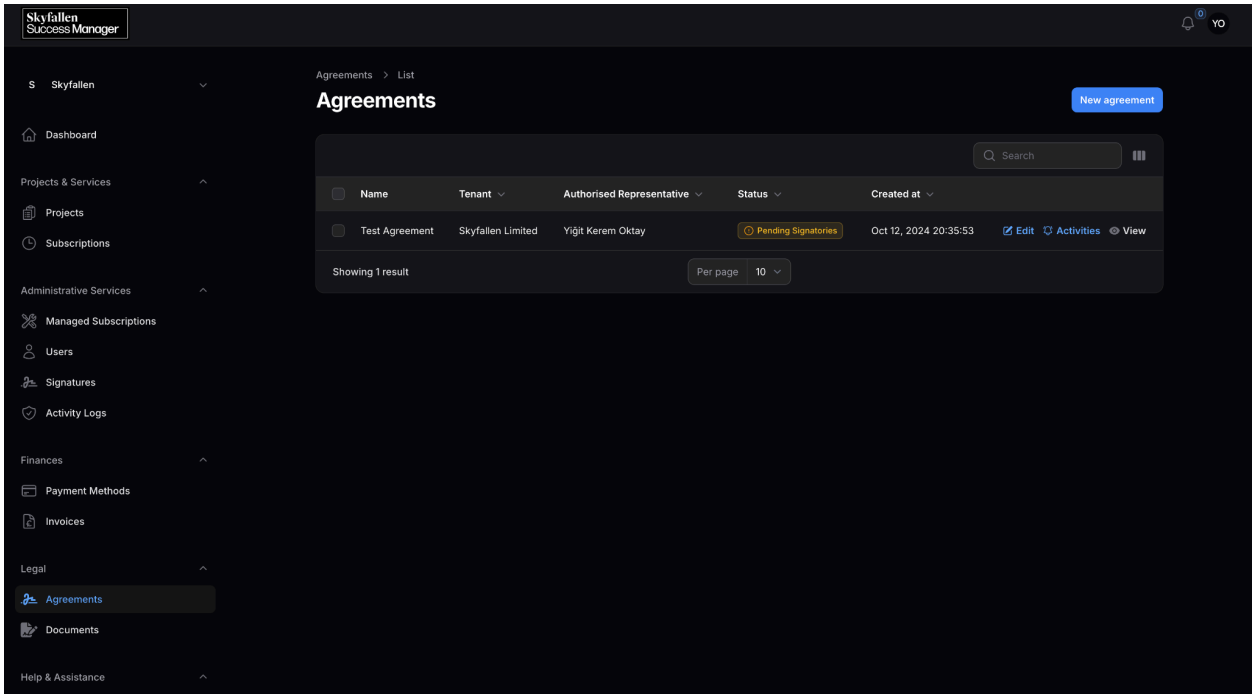
```php
class CreateAgreement extends CreateRecord
{
    protected static string $resource = AgreementResource::class;

    no usages  Yiğit Kerem Oktay
    protected function handleRecordCreation(array $data): Model
    {
        return Agreement::create($data);
    }

    no usages  Yiğit Kerem Oktay
    protected function afterSave(): void
    {
        $recipients = $this->record->user()->first() ?? $this->record->tenant()->first()->users()->get();

        Notification::make()
            ->title( title: 'A new agreement is requested.')
            ->body( body: "Agreement " . $this->record->name . " was requested for " . $this->record->tenant()->first()->name . '.')
            ->icon($this->record->status->getIcon())
            ->actions([
                \Filament\Notifications\Actions\Action::make( name: 'view')
                    ->label( label: "View Agreements")
                    ->url(AgreementResource::getUrl())
            ])
            ->color($this->record->status->getColor())
            ->sendToDatabase($recipients)
            ->toEmail($recipients)
            ->broadcast($recipients);
    }

}
```

Beyond this override, the EditAgreement & ListAgreements files are as-is from the Filament provided stub (template)

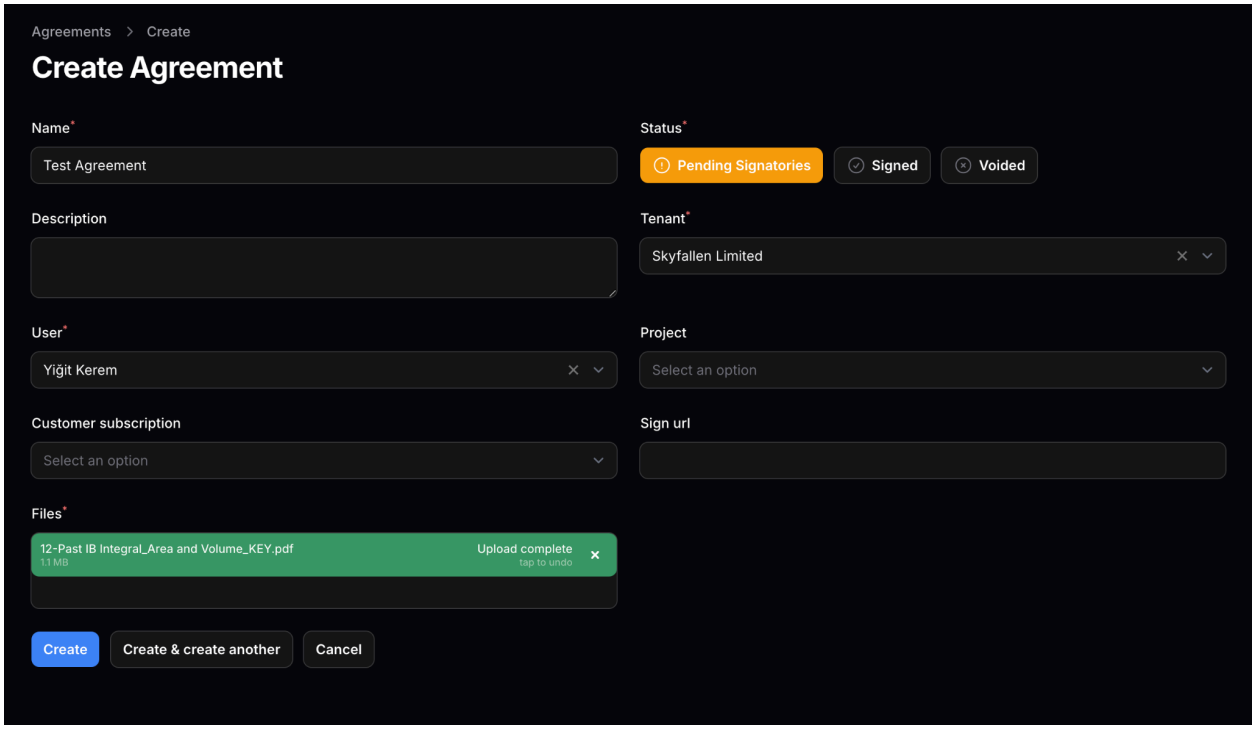
```php
<?php

namespace App\Filament\Consumer\Resources\AgreementResource\Pages;

use App\Filament\Consumer\Resources\AgreementResource;
use Filament\Actions;
use Filament\Resources\Pages\EditRecord;

Yiğit Kerem Oktay
class EditAgreement extends EditRecord
{
    protected static string $resource = AgreementResource::class;

    Yiğit Kerem Oktay
    protected function getHeaderActions(): array
    {
        return [
            Actions\DeleteAction::make(),
        ];
    }
}
```

```php
<?php

namespace App\Filament\Consumer\Resources\AgreementResource\Pages;

use App\Filament\Consumer\Resources\AgreementResource;
use Filament\Actions;
use Filament\Resources\Pages\ListRecords;

1 usage  Yiğit Kerem Oktay
class ListAgreements extends ListRecords
{
    protected static string $resource = AgreementResource::class;

    Yiğit Kerem Oktay
    protected function getHeaderActions(): array
    {
        return [
            Actions\CreateAction::make(),
        ];
    }
}
```

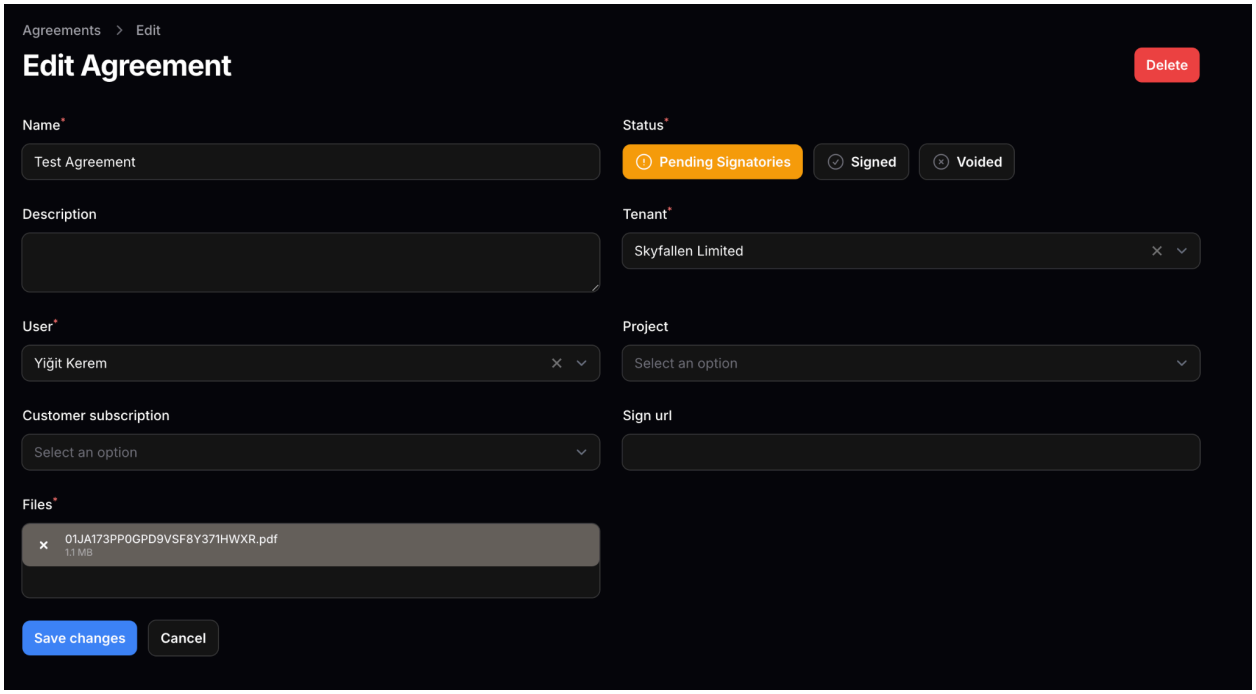
These all translate into the agreement pages and are routed simply.



ListAgreements Page



CreateAgreement Page



EditAgreement Page

Structure of a Resource Page

Each page for a resource houses traits that do not apply to all of a model's operations. One model may incorporate different forms for creating and updating records or may need to perform additional tasks before rendering the record in a read page.

This requires this functionality to be embedded to the Page field instead of the Resource page so that the modifications only apply to specific operations related to a resource.

Let's go back to the CreateAgreement example for instance. During the creation of an Agreement, the relevant customer must be notified of the creation whilst no notification is required during update because the update happens when the user signs the agreement and the user knows they have signed the agreement. The notification would be redundant.

For this case, we create an override method in CreateAgreement.php:

```php
no usages  Yiğit Kerem Oktay
protected function afterSave(): void
{
    $recipients = $this->record->user()->first() ?? $this->record->tenant()->first()->users()->get();

    Notification::make()
        ->title( title: 'A new agreement is requested.')
        ->body( body: "Agreement " . $this->record->name . " was requested for " . $this->record->tenant()->first()->name . '.')
        ->icon($this->record->status->getIcon())
        ->actions([
            \Filament\Notifications\Actions\Action::make( name: 'view')
                ->label( label: "View Agreements")
                ->url(AgreementResource::getUrl())
        ])
        ->color($this->record->status->getColor())
        ->sendToDatabase($recipients)
        ->toEmail($recipients)
        ->broadcast($recipients);
}
```

The afterSave() method is called after the page executes the creation method. We can access the created record using the calling object's $record property ($this→record). This snippet will use Eloquent ORM to check if the record is related to a specific user and if not will consider all the members of the respective tenant as recipients. It will then send a notification to all recipients.

Please note that this modification is not applied to all operations but only Creation operations since it is defined here.

It can be said that this override is not common as only 30-40% of all pages in the app contain some kind of override whereas the remaining 60-70% are exactly the same as the Filament provided stub.

GUI Complexities

The amount of data and relationships handled through the application presents a unique challenge where information from multiple resources must be incorporated into a single view of the application for the user to be able to make educated decisions.

There are three ways this app makes use of advanced GUI techniques to achieve this:

1. Modals
2. Repeaters

Modals

Instead of redirecting the user to a completely different page, modals are utilized for actions that generate a new resource using data already available within a specific instance of a resource. An example of this is generating invoices or payments from a project.

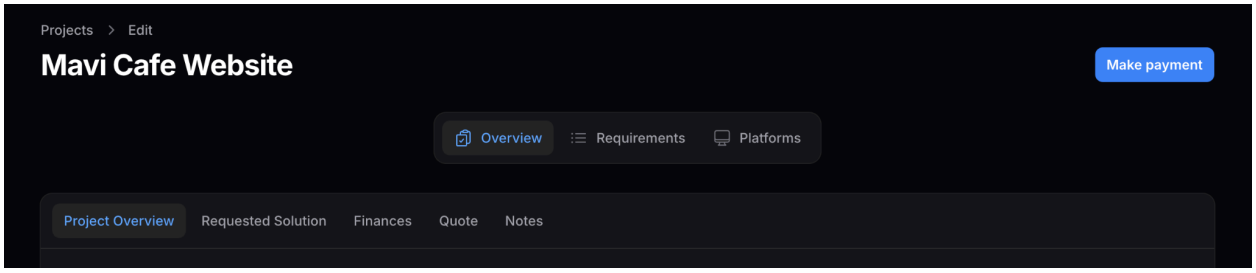
The project object already has properties that define the amounts of payments and the description of what this payment entails. The time and date of the payment is the time the user initiates the transaction. Only thing required from the user is at this point a signature and a payment method choice which can simply be asked using a modal instead of redirecting them to a page where they fill out details about the payment from the beginning.

Filament makes this problem very simple to approach. A form must be defined and associated into an action, then the respective code to handle the submission must be put into place.

To start, we tap into the EditRecord Page of the ProjectResource. There, we will override the getHeaderActions() method to add a new action to the Edit Page.

```
2 usages  Yiğit Kerem Oktay
class EditProject extends EditRecord
{
    protected static string $resource = ProjectResource::class;

    no usages
    protected static ?string $navigationLabel = 'Overview';

    protected static ?string $navigationIcon = 'heroicon-o-clipboard-document-check';

    Yiğit Kerem Oktay
    public function getTitle(): string
    {
        return $this->record->name;
    }

    Yiğit Kerem Oktay
    protected function getHeaderActions(): array
    {
        return [
            Action::make( name: 'makePayment')
                ->fillForm(fn (Project $record): array => [
                    'upfront_fee' => ($record->status->value === "upfront_payment_pending" ? 1 : -1) * $record->upfr
                    'fee' => $record->fee,
                    'additional_fee' => $record->additional_fee,
                    'total_due' => $record->status->value === "upfront_payment_pending" ? $record->upfront_fee + $re
                ])
                ->modalHeading( heading: 'Complete the payment')
                ->modalDescription( description: "Choose the bank account you have initiated a transfer from or select a
                ->form([
                    Select::make( name: 'payment_method')
                        ->label( label: 'Choose a payment method')
                        ->native( condition: false)
                        ->searchable()
                        ->options(Filament::getTenant()->payment_methods()->pluck('name', 'id'))
```

Code from ProjectResource/Pages/EditRecord



The added button

Then, a new form is defined:

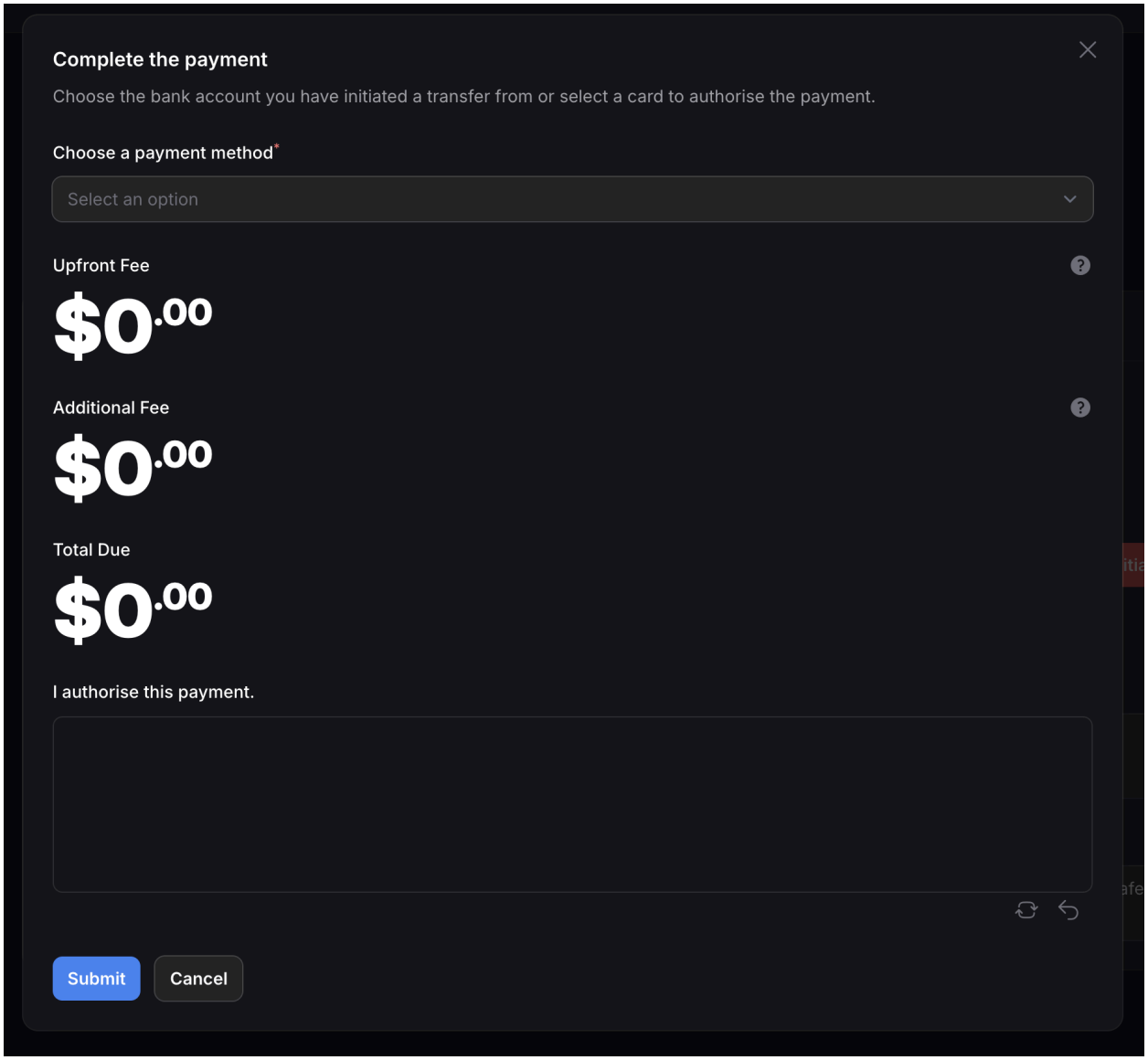
```
Select::make( name: 'payment_method')
    ->label( label: 'Choose a payment method')
    ->native( condition: false)
    ->searchable()
    ->options(Filament::getTenant()->payment_methods()->pluck('name', 'id'))
    ->required(),
Group::make([
    PriceField::make( name: 'upfront_fee')
        ->label( label: "Upfront Fee")
        ->hintIcon( icon: 'heroicon-m-question-mark-circle', tooltip: 'This is portion of the fee that must be paid before work starts. It is includ
    PriceField::make( name: 'additional_fee')
        ->label( label: "Additional Fee")
        ->disabled()
        ->hintIcon( icon: 'heroicon-m-question-mark-circle', tooltip: 'This is portion of the fee that is paid to third party providers and is exclu
])->hidden(fn (Project $project): bool => $project->status->value !== "upfront_payment_pending"),
Group::make([
    PriceField::make( name: 'fee')
        ->label( label: "Project Fee")
        ->hintIcon( icon: 'heroicon-m-question-mark-circle', tooltip: 'This is the total project cost.'),
    PriceField::make( name: 'upfront_fee')
        ->label( label: "Upfront Fee")
        ->disabled()
        ->hintIcon( icon: 'heroicon-m-question-mark-circle', tooltip: 'This is portion of the fee that you paid before the work started.'),
])->hidden(fn (Project $project): bool => $project->status->value !== "pending_payment"),
PriceField::make( name: 'total_due')
    ->disabled()
    ->label( label: "Total Due"),
SignaturePad::make( name: 'signature')
    ->label( label: "I authorise this payment.")
```

The SignaturePad field is a premade component from an external library that returns a base64 png image of a signature drawn using a touchpad.

The form is prefilled with details:

```
->fillForm(fn (Project $record): array => [
    'upfront_fee' => ($record->status->value === "upfront_payment_pending" ? 1 : -1) * $record->upfront_fee,
    'fee' => $record->fee,
    'additional_fee' => $record->additional_fee,
    'total_due' => $record->status->value === "upfront_payment_pending" ? $record->upfront_fee + $record->additional_fee : $record->fee - $record->upfront_fee
])
```

Filament provides the current $record as a parameter to the fillForm method so we can access the project during the method. We return an array with the prefilled details such as the project's fee that will be presented to the user.



The Presented Modal

Then, the actions to handle the submission of this modal are defined in the action() method.

```
->action(function (array $data, Project $project): void {

    $payment_method = CustomerPaymentMethod::find($data['payment_method']);

    $stripe = new \Stripe\StripeClient(env( key: 'STRIPE_SECRET'));

    $invoice = "";
    $stripeCustomer = Filament::getTenant()->asStripeCustomer()->id;

    if($payment_method->stripe_ref){
        $invoice = $stripe->invoices->create([
            'customer' => $stripeCustomer,
            'currency' => $project->currency,
            'metadata' => [
                'payment_method_id' => $payment_method->id,
                'project_id' => $project->id
            ],
            'default_payment_method' => $payment_method->stripe_ref
        ])->id;
    } else {
        $invoice = $stripe->invoices->create([
            'customer' => $stripeCustomer,
            'currency' => $project->currency,
            'metadata' => [
                'payment_method_id' => $payment_method->id,
                'project_id' => $project->id
            ],
        ])->id;
    }
```

The form data is passed to the action as an array $data and the project is accessible as $project. This provides the user simplicity. Now there is no need to ask the user about the invoice details. On the call to payment processor's invoice creation method $stripe→invoices→create(), we can fetch the currency from $project→currency and combine the information from the form like the $payment_method we fetch from the database using the $data['payment_method'] provided by the form on the modal.

Repeaters

Some of the resources in the project have lots of related resources. The number of children in one to many relationships make viewing and modifying children difficult to the user. An example of this is the project requirements of a project. A user will need to be able to see 10-20 features of a project at once and edit them without revisiting multiple edit pages.

To achieve this, we use a repeater that is an abstract for a drag & drop cloneable form, provided by Filament. Repeaters will have a simple schema for each item (each child in the relationship) and during the save, are automatically saved as relationship objects into their respective separate tables.

To create the repeater, we use the Repeater::make(name) static function. This returns a Repeater instance that can be modified using instance methods. Note that this Repeater object conforms to a field so a repeater can be placed into any form regardless.

```php
return Repeater::make( name: 'requirements')
    ->disabled(fn(Project $project): bool => $project->status !== null && $project->status->getLockState())
    ->relationship( name: 'requirements')
    ->schema([
        ToggleButtons::make( name: 'required')
            ->label( label: "Is this a requirement?")
            ->options( options: YesNo::class)
            ->inline()
            ->default( state: true)
            ->required(),
        TextInput::make( name: 'name')
            ->label( label: "Name this requirement")
            ->required(),
        TextArea::make( name: 'description')
            ->label( label: "Describe this requirement")
            ->required(),
        Group::make()
            ->schema([
                TextInput::make( name: 'fee')
                    ->label( label: "Additional fee")
                    ->numeric()
                    ->disabled(!auth()->user()->manager),
                TextInput::make( name: 'additional_days')
                    ->label( label: "Additional days")
                    ->numeric()
                    ->disabled(!auth()->user()->manager),
            ])
            ->columns( columns: 2)
            ->hidden(!auth()->user()->manager)
    ])
    ->mutateRelationshipDataBeforeCreateUsing(function (array $data): array {
        $data["adder_id"] = auth()->id();
        return $data;
    })
    ->orderColumn( column: 'sort')
    ->collapsible()
```

The below is used to tell Filament that this repeater belongs to a relationship called ‘requirements’.

```
->relationship( name: 'requirements')
```

Where relationship ‘requirements’ is defined in the Laravel Eloquent model of Project as:

```
no usages   Yiğit Kerem Oktay
function requirements()
{
    return $this->hasMany( related: ProjectRequirement::class);
}
```

app/Http/Models/Project.php

This will make sure Filament saves the items in this repeater to the correct database table.

Then, fields for each requirement are specified. Note that these correspond to the fields in the requirements table.

```
->schema([
    ToggleButtons::make( name: 'required')
        ->label( label: "Is this a requirement?")
        ->options( options: YesNo::class)
        ->inline()
        ->default( state: true)
        ->required(),
    TextInput::make( name: 'name')
        ->label( label: "Name this requirement")
        ->required(),
    TextArea::make( name: 'description')
        ->label( label: "Describe this requirement")
        ->required(),
    Group::make()
        ->schema([
            TextInput::make( name: 'fee')
                ->label( label: "Additional fee")
                ->numeric()
                ->disabled(!auth()->user()->manager),
            TextInput::make( name: 'additional_days')
                ->label( label: "Additional days")
                ->numeric()
                ->disabled(!auth()->user()->manager),
        ])
        ->columns( columns: 2)
        ->hidden(!auth()->user()->manager)
])
```

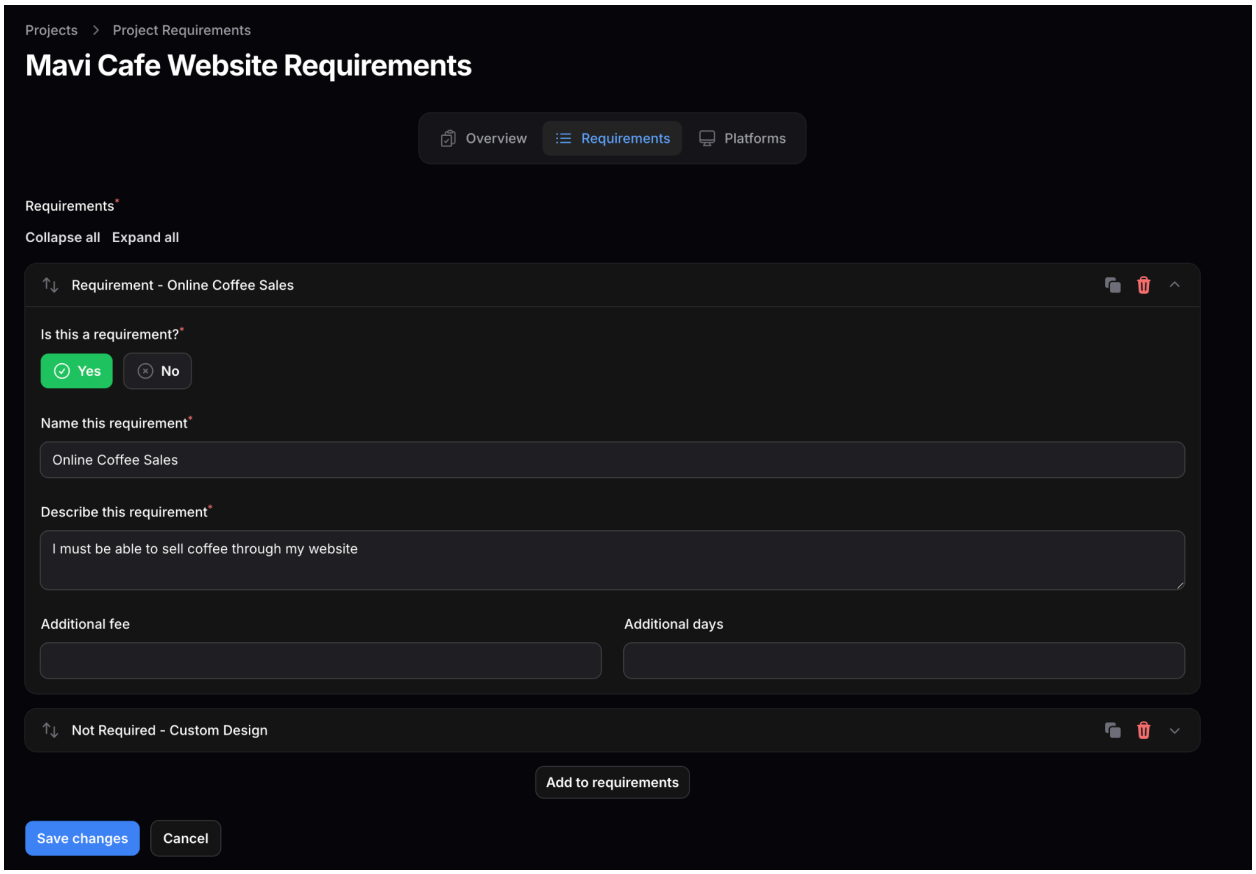
Since we also want to know who added a requirement, before Filament saves the repeater data, we alter each item's data array by an override to add the "adder_id" key.

```
->mutateRelationshipDataBeforeCreateUsing(function (array $data): array {
    $data["adder_id"] = auth()->id();
    return $data;
})
```

We also set a few more parameters.

```
->orderColumn( column: 'sort') // Use the 'sort' column in the database to decide on the sort order
->collapsible() // Allow the user to collapse items
->collapsed() // Collapse items by default
->cloneable() // Allow user to duplicate item
->itemLabel(fn(array $state): ?string => ($state['required'] ? "Requirement - " : "Not Required - ") . $state['name'] ?? null) // Set the title of an item
->required(); // make it required
```

The following repeater is rendered.



Database Complexities

The high number of models also make the database relationships much more difficult to manage but Laravel comes to the rescue. There are three different difficulties with the database:

1. Enums

2. Many to many relationships + Pivot fields

Enums

Due to the nature of the models in this project, there are lots of different enum properties such as statuses or yes / no fields. One major example of this is the project status where there are multiple properties attached to a project's status, namely:

1. Human Readable Name
2. Color
3. Icon
4. Quoted State (Is the current state before a the quote step or after)
5. Lock State (Does this state lock the project on the customer side for any updates)

The best solution to this is using enums. Enums are a PHP data type. Let's look at the example of a ProjectStatus enum at app/Enums/ProjectStatus:

```php
<?php

namespace App\Enums;

use Filament\Support\Contracts\HasColor;
use Filament\Support\Contracts\HasIcon;
use Filament\Support\Contracts\HasLabel;

5 usages  Yiğit Kerem Oktay +1
enum ProjectStatus: string implements HasIcon, HasLabel, HasColor
{
    6 usages
    case Submitted = 'submitted';
    6 usages
    case InReview = 'in_review';
    6 usages
    case Accepted = 'accepted';
    6 usages
    case Rejected = 'rejected';
    6 usages
    case UpfrontPaymentPending = 'upfront_payment_pending';
    6 usages
    case InProgress = 'in_progress';
    6 usages
    case PendingPayment = 'pending_payment';
    6 usages
    case Completed = 'completed';

    Yiğit Kerem Oktay
```

We inherit Filament's traits to be able to use this enum in its fields and then define the cases in the database.

```
Yiğit Kerem Oktay
public function getColor(): string|array|null
{

    return match ($this) {
        self::Submitted => "warning",
        self::InReview, self::InProgress => "info",
        self::Accepted, self::Completed => "success",
        self::Rejected, self::UpfrontPaymentPending, self::PendingPayment => "danger",
    };
}

Yiğit Kerem Oktay
public function getIcon(): ?string
{
    return match ($this) {
        self::Submitted => "heroicon-o-arrow-right-end-on-rectangle",
        self::InReview => "heroicon-o-arrow-path",
        self::Accepted => "heroicon-o-bell",
        self::Rejected => "heroicon-o-exclamation-circle",
        self::UpfrontPaymentPending => "heroicon-o-circle-stack",
        self::InProgress => "heroicon-o-command-line",
        self::PendingPayment => "heroicon-o-banknotes",
        self::Completed => "heroicon-o-check-circle",
    };
}
```

Then, for each case and each tied property, we define return values like seen above. Also, like below, we can combine this information with the current session data. For example, the project will never be locked if the current user is a manager.

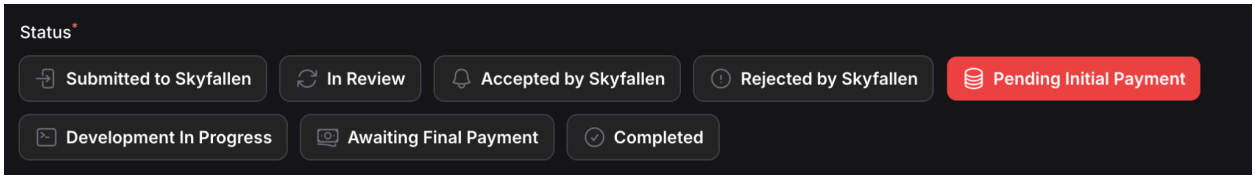
```
no usages  Yiğit Kerem Oktay
public function getLockState(): bool
{
    return match ($this) {
            self::Submitted, self::Rejected => false,
            self::InReview, self::Accepted, self::UpfrontPaymentPending, self::InProgress, self::PendingPayment, self::Completed => true,
        } && !auth()->user()->manager;
}
```

This is incorporated into the fields for a seamless experience for the user.

```
ToggleButtons::make( name: 'status')
    ->label( label: "Status")
    ->options( options: ProjectStatus::class)
    ->inline()
    ->required()
    ->disabled(!auth()->user()->manager)
    ->columnSpanFull(),
```

ProjectResource.php, form() method

Final output:



Many to Many Relationships

The need for tenancy requires a difficult many-to-many relationship where many tenants must be matched with many users. This is handled using a pivot table which contains a user ID and a tenant ID.

For this, we create a pivot table with the special name, tenant_user.

```
return new class extends Migration {
    /**
     * Run the migrations.
     */
    Yiğit Kerem Oktay *
    public function up(): void
    {
        Schema::create( table: 'tenant_user', function (Blueprint $table) {
            $table->foreignId( column: 'tenant_id')->references( column: 'id')->on( table: 'tenants')->onDelete( action: 'cascade');
            $table->foreignId( column: 'user_id')->references( column: 'id')->on( table: 'users')->onDelete( action: 'cascade');
            $table->enum( column: 'role', ['admin', 'manager', 'user']);
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    Yiğit Kerem Oktay
    public function down(): void
    {
        Schema::dropIfExists( table: 'tenant_user');
    }
};
```

Thanks to this special name, in the Laravel model definition of a tenant and a user, we can define the following relationships:

```
Yiğit Kerem Oktay
public function users(): BelongsToMany
{
    return $this->belongsToMany( related: User::class);
}
```

Tenant.php

```
Yiğit Kerem Oktay
function tenants()
{
    return $this->belongsToMany( related: Tenant::class);
}
```

User.php

This feature allows the code to easily access this relationship. For example when we send a project invoice notification to tenant members, we first get a project's tenant, then its members as an array without needing to think about the pivot table ourselves.

```
$recipients = $project->tenant()->first()->users()->get();

Notification::make()
    ->title( title: 'Project invoice was paid.')
    ->body( body: "Invoice for project " . $project->name . " was paid and the project is now " . $project->status->getLabel() . '.')
    ->icon($project->status->getIcon())
    ->actions([
        \Filament\Notifications\Actions\Action::make( name: 'view')
            ->label( label: "View Project")
            ->url(ProjectResource::getUrl( name: "edit", [ "record" => $project->id, "tenant" => $project->tenant()->first()->slug ]))
    ])
    ->color($project->status->getColor())
    ->sendToDatabase($recipients)
    ->toEmail($recipients)
    ->broadcast($recipients);
```

APIs & Webhook Complexities

This project requires connection to two main external services, an IDP and a Payment Processor. The IDP is a custom service that provides an API into users, tenants and their relations whereas the payment processor allows for invoices to be created and financial transactions to be conducted.

There are two major difficulties:

1. Fetching IDP authentication details securely and setting them in the database
2. Getting updates from Stripe payment processor API in real time using a Webhook

IDP Authentication

This project uses a SSO implementation that is akin to the Oauth protocol where the security is established by a combination of short and long lived tokens exchanged very quickly.

1. A redirect is made to the IDP providing a public app ID, "SuccessManager".

```php
function index(Request $request)
{
    return redirect( to: env( key: "SIS_UI") . "/authorise/cas/" . env( key: "SIS_APP_ID"));
}
```

2. IDP returns a short 30 second token only accessible with a private app key only known by SuccessManager's owner.

```php
$resp = Http::post( url: env( key: "SIS_IDMS") . "/api/cas/session", [
    "code"   => $request->code,
    "secret" => env( key: "SIS_APP_SECRET")
])->json();

if ($resp["status"] == "error") {
    abort( code: 403);
}
```

3. App exchanges the short-lived token for an access token.
   See above.
4. The access token is used to fetch tenants, memberships & user profile.

```
$cas_tenant = Http::post( url: env( key: "SIS_IDMS") . "/api/cas/tenant", [
    "token" => $resp["token"]["value"],
])->json()["tenant"];
```

```
$cas_user = Http::post( url: env( key: "SIS_IDMS") . "/api/cas/user", [
    "token" => $resp["token"]["value"],
])->json()["user"];
```

```
$cas_membership = Http::post( url: env( key: "SIS_IDMS") . "/api/cas/membership", [
    "token" => $resp["token"]["value"],
])->json()["membership"];
```

5. Associate the remote models with local database tables by checking if any local record exists matching the remote identifier stored at the database column “cas_id”.

```php
$cas_tenant["cas_id"] = $cas_tenant["id"];
$cas_tenant["slug"] = Str::slug($cas_tenant["name"], separator: "-");

$tenant = Tenant::updateOrCreate(["cas_id" => $cas_tenant["cas_id"]], $cas_tenant);

$user->tenants()->detach();

$tenant->users()->attach($user);

$tenant->createOrGetStripeCustomer();

$tenant->updateStripeCustomer([
    "address" => $billing_details["address"],
    "email"   => $billing_details["email"],
    "phone"   => $billing_details["phone"],
    "name"    => $cas_tenant["legal_name"]
]);

setPermissionsTeamId($tenant->id);

$permissions = explode( separator: ",", $cas_membership["permissions"]);

foreach ($permissions as $permission) {
    try {
        Permission::create(['name' => $permission]);
    } catch (PermissionAlreadyExists $e) {
        continue;
    }
}

$user->syncPermissions($permissions);

auth()->login($user);

return redirect( to: '/manager');
```

Payment Processor Webhooks

Due to the nature of banking operations, financial transactions are first requested but are usually not instantaneously fulfilled. Usually a charge is first authorized but a bank has to let the merchant capture it and this process may take a few hours. Until this is done, a merchant does not know if the transaction did go through.

To detect these events, Stripe fires webhooks (POST requests with a secret key) to a public url served by our app with the payload. Our app must listen to these requests, verify the signature, classify the event and act accordingly.

Laravel's Cashier will handle the signature verification for us, given we provide the key in the applications environment variables.

Then, all requests are routed to a specific function where we:

1. Classify event types

```
    if ($event->payload['type'] === 'invoice.finalized') {
```

2. Match with local records

```
$tenant = Tenant::where('stripe_id', $event->payload['data']['object']['customer'])->first();

if($tenant !== null){
```

```
$invoice = $event->payload['data']['object'];

$project_id = null;

if(isset($invoice['metadata']['project_id'])){
    $project = Project::find($invoice['metadata']['project_id']);
    $project_id = $project ? $project->id : null;
}

$payment_method_id = null;

if(isset($invoice['metadata']['payment_method_id'])){
    $payment_method = CustomerPaymentMethod::find($invoice['metadata']['payment_method_id']);
    $payment_method_id = $payment_method ? $payment_method->id : null;
}
```

3. Update local records accordingly

```
$db_invoice = Invoice::create([
    'tenant_id' => $tenant->id,
    'name' => $invoice['number'],
    'description' => 'Automatically generated PDF invoice from Stripe.',
    'amount' => $invoice['amount_due'],
    'status' => 'pending',
    'stripe_ref' => $invoice['id'],
    'customer_payment_method_id' => $payment_method_id,
    'project_id' => $project_id,
    'currency' => $invoice['currency'],
]);
```

4. Store any file attachments

```php
$path = storage_path( path: Str::random( length: 64).".pdf");

$client = new Client();

$client->request( method: 'GET', $invoice['invoice_pdf'], ['sink' => $path]);

$url = Storage::putFile( path: 'invoices', new File($path));

Storage::disk( name: 'local')->delete($path);

$db_invoice->files = $url;

$db_invoice->save();
```