OGP / UiApi

UI API package for Laravel 12 (PHP 8.3+). It provides:

- Generic CRUD JSON API for your Eloquent models
- Component configuration service to generate UI payloads from simple JSON view configs

Package namespace: `Ogp\UiApi`. Auto-discovered provider: `Ogp\UiApi\UiApiServiceProvider`.

Installation

1) Require the package

```
composer require ogp/uiapi
```

2) (Optional) Publish config and view configs

- View configs (installs sample JSON files to `app/Services/viewConfigs`):

```
php artisan vendor:publish --tag=uiapi-view-configs
```

3) Clear caches

```
php artisan optimize:clear
```

Routes

By default routes are registered under `/{prefix}` where `prefix = config('uiapi.route_prefix', 'api')`.

- Component Config Service (CCS)
    - GET `/{prefix}/ccs/{model}` → returns component settings for the given model and `component` key
        - Required: `component=listView` (or another key present in your view config)
        - Optional: `lang` — defaults to `dv` if omitted (supports `en`, `dv` where applicable)
- Generic API CRUD (GAPI)
    - GET `/{prefix}/gapi/{model}` → list/index
    - GET `/{prefix}/gapi/{model}/{id}` → show
    - POST `/{prefix}/gapi/{model}` → create
    - PUT `/{prefix}/gapi/{model}/{id}` → update

- DELETE `/{prefix}/gapi/{model}/{id}` → delete

Verify:

```
php artisan route:list --path=api
```

Model Name Normalization

Both CCS and GAPI accept flexible model names in the URL path. The resolver normalizes common multi-word inputs to StudlyCase and checks these namespaces in order: `Ogp\UiApi\Models\`, then `App\Models\`.

- Accepted examples all resolve to `EntryType`:
  - `entry_type`, `entry-type`, `entry type`, `entryType`
- Ambiguous forms like `entrytype` (no delimiters) cannot be reliably split; prefer one of the delimited forms above when your model is multi-word.

Query Parameters (GAPI)

Supported by `GenericApiController` and `BaseModel` scopes:

- `columns`: comma list of selected columns; relation tokens supported (e.g., `country.name_eng`).
- `with`: eager-load relations, supports `relation` or `relation:col1,col2` to constrain selected columns.
- `pivot`: eager-load belongsToMany relations including pivot data.
- `search`: OR groups separated by `|`, each group has comma pairs `column:value`.
- `searchAll`: global search across the model's searchable fields.
- `withoutRow`: exclude rows matching criteria (`column:value`, supports OR groups).
- `filter`: advanced filters
  - AND across pairs: `a:x,b:y`; OR within a field: `a:x|y`
  - Relation fields: `relation.field:value`
  - Special values: `null`, `!null`, `true`, `false`, `''` (empty)
- `sort`: comma list; `-field` for DESC (e.g., `-created_at,name`).
- `pagination`: `off` to disable pagination.
- `per_page`: items per page.
- `page`: page number.

Responses:

- Paginated: `{ data: [...], pagination: { current_page, last_page, per_page, total } }`
- Non-paginated: `{ data: [...] }`

View Configs (UI Component Settings)

View configs are JSON files per model (lowercase filename, no hyphens, no underscores or spaces) and can define multiple views (e.g., `listView`). Default path: `app/Services/viewConfigs`.

Common keys:

- `components`: components to assemble. Currently `table` and `filterSection` are supported.
- `columns`: the fields to include for the view. Use relation tokens like `country.name_eng`.
- `columnCustomizations`: per-column overrides for labels, visibility, display types, etc.
- `filters`: whitelisted fields for filter UI.
- `per_page`: default page size for component settings.
- `lang`: allowed UI languages for this view (e.g., `["en","dv"]`). If the request `lang` isn't allowed, the service returns an informative message.

Example (`app/Services/viewConfigs/person.json`):

```
{
  "listView": {
    "components": {
      "table": {},
      "filterSection": { "buttons": ["submit", "clear"] }
    },
    "columns": [
      "country_id", "id", "country.name_eng", "first_name_eng",
      "first_name_div", "gender", "is_in_custody", "date_of_birth"
    ],
    "columnCustomizations": {
      "is_in_custody": {
        "title": { "en": "ISINCUSTODY", "dv": "" },
        "sortable": true,
        "hidden": false,
        "displayType": "checkbox",
        "inlineEditable": true,
        "displayProps": { "color": "primary" }
      }
    },
    "filters": ["gender", "country_id", "date_of_birth"],
    "per_page": 11,
    "lang": ["en", "dv"]
  }
}
```

NoModel View Configs

In some views, you can build UI payloads without an Eloquent model by setting `noModel: true` on the view config block. In this mode, the returned payload is derived entirely from the JSON config — specifically from `columns`, `columnsSchema`, optional `columnCustomizations`, `filters`, and `per_page`.

- What changes in noModel:
  - No model resolution occurs; `apiSchema()` is not required.
  - `columnsSchema` is mandatory and drives headers, filter metadata, and language support.
  - Relation fields can be declared directly via dot tokens as keys in `columnsSchema` (e.g., `"country.name_eng"`). Their schema entries (e.g., `hidden`, `sortable`, `type`, `displayType`,

`lang`) will be used for headers and filtering.
    - Language handling honors `lang` on each column in `columnsSchema`; only columns supporting the requested `lang` are included.

- Component assembly remains the same: component templates in `src/Services/ComponentConfigs/*.json` are still used and merged with your view config inputs to produce `componentSettings`.

Datalink in noModel mode

The `datalink` section of a component template controls how the data-fetch URL is included in the payload.

- When a template section sets `"datalink": "on"`, the CCS generates the URL automatically based on:
    - Selected `columns` (after language filtering), including relation dot tokens.
    - Relation dot tokens become `with` segments: for example, `country.name_eng` yields `with=country:name_eng`.
    - Pagination is appended using `per_page` from the view config (or request override).
    - The base path uses the package route prefix: `/{prefix}/gapi/{Model}` where `{prefix} = config('uiapi.route_prefix', 'api')`.
- When a template section sets `"datalink": "off"`, the CCS omits the `datalink` key from the section payload.
- When a template section sets `"datalink": <object|string>`, the CCS uses that value directly, allowing custom URLs or structures without auto-generation.

Example minimal noModel view block:

```
{
  "listView2": {
    "noModel": true,
    "components": { "table": {}, "filterSection": { "buttons": ["submit",
"clear"] } },
    "columns": ["id", "country.name_eng", "first_name_eng"],
    "columnsSchema": {
      "id": { "hidden": true, "key": "id", "label": { "en": "ID" }, "type":
"number", "sortable": true, "lang": ["en", "dv"] },
      "country.name_eng": { "lang": ["en"], "hidden": true },
      "first_name_eng": { "key": "first_name_eng", "label": { "en": "First
Name" }, "type": "string", "displayType": "text", "sortable": true, "lang":
["en"] }
    },
    "per_page": 25,
    "lang": ["en", "dv"]
  }
}
```

Notes:

- If `lang` requested is not in the view's `lang` array, CCS returns a message and an empty `data` array.
- Dot-key `columnsSchema` entries are fully honored (hidden, sortable, type, displayType, inlineEditable, displayProps, lang).
- `columnCustomizations` still apply and override schema-derived defaults in the final headers.

Component configuration templates live in the package at `src/Services/ComponentConfigs/*.json` and are referenced by keys in your view config `components` block (e.g., `table`, `filterSection`). `ComponentConfigService` builds per-component payloads based on these templates + your model's `apiSchema`.

Fetch Component Settings

```
curl "http://localhost/api/ccs/person?component=listView&lang=en"
```

If `lang` is omitted, it defaults to `dv`.

Creating a Model with apiSchema

Your models should expose an `apiSchema()` method that returns a schema with a `columns` map. The package resolves models by checking `Ogp\UiApi\Models\{Model}` first, then `App\Models\{Model}`.

Fastest path: extend the package base model so all query scopes and helpers are available.

Minimal example (`App\Models\Book`):

```php
<?php

namespace App\Models;

use Ogp\UiApi\Models\BaseModel;

class Book extends BaseModel // important to extent BaseModel
{


    public function apiSchema(): array
    {
        return [
            'columns' => [
                'id' => [
                    'hidden' => true,
                    'key' => 'id',
                    'label' => ['en' => 'ID'],
                    'type' => 'number',
                    'displayType' => 'text',
                    'sortable' => true,
                ],
                'title' => [
                    'hidden' => false,
```

```
                    'key' => 'title',
                    'label' => ['en' => 'Title'],
                    'type' => 'string',
                    'displayType' => 'text',
                    'sortable' => true,
                    'filterable' => [
                        'type' => 'search',
                        'value' => 'title',
                    ],
                ],
                'author_id' => [
                    'hidden' => true,
                    'key' => 'author_id',
                    'label' => ['en' => 'Author'],
                    'type' => 'number',
                    'displayType' => 'text',
                    'filterable' => [
                        'type' => 'select',
                        'mode' => 'relation',
                        'relationship' => 'author',
                        'itemTitle' => ['en' => 'name_eng'],
                        'itemValue' => 'id',
                        'value' => 'author_id',
                    ],
                ],
            ],
        ];
    }

    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

Notes:

- `label` can be a string or a per-language map. `lang` on a column can restrict language availability for UI.
- `displayType` helps the UI render (e.g., `text`, `chip`, `date`, `checkbox`).
- `filterable` defines filter UI. `type` can be `search` or `select`; `mode` can be `self` or `relation`.
- Relation tokens like `author.name_eng` are supported in `columns` and the service.

Example Data Requests

- List with relation fields, sort, and pagination:

```
curl "http://localhost/api/gapi/book?
columns=id,title,author.name_eng&with=author:name_eng&id&sort=-
created_at&per_page=10"
```

- Fetch a single record with a relation:

```
curl "http://localhost/api/gapi/book/42?columns=id,title&with=author"
```

- Create:

```
curl -X POST "http://localhost/api/gapi/book" \
  -H "Content-Type: application/json" \
  -d '{"title":"My Book","author_id":1}'
```

- Update:

```
curl -X PUT "http://localhost/api/gapi/book/42" \
  -H "Content-Type: application/json" \
  -d '{"title":"Updated"}'
```

- Delete:

```
curl -X DELETE "http://localhost/api/gapi/book/42"
```