Ontwikkeling

Bij de aanschaf van onze Makelaar plugin ontvang je een standaard thema, deze kan je als child-theme naast je hoofd thema installeren. Hierdoor kan je je hoofdthema up-to-date houden en verlies je geen instellingen van onze plugin.

Binnen dit thema werk je aan de ontwikkeling van de betreffende pagina's, zoals het overzicht en de detailpagina van de objecten. Je schrijft zelf de HTML, CSS, PHP en eventueel extra Javascript om de pagina's vorm te geven.

> Je kan deze pagina's niet bewerken vanuit WordPress, ook niet met page builders zoals Elementor.

In de onderstaande beschrijvingen wordt alleen de module voor woningen uitgelicht, de werking is hetzelfde voor alle andere modules.

Daarnaast worden er veel velden gebruikt in de voorbeeldcode, het kan zijn dat er voor de koppeling met jouw CRM pakket een (net iets) ander veldnaam gebruikt moet worden. Bijvoorbeeld bij het gedeelte "Afhankelijkheden van andere velden" wordt er gebruik gemaakt van de velden `koopprijs` en `koopconditie`, dit is voor de koppeling met het CRM van Realworks.

> Alle velden voor de koppeling met jouw CRM pakket vind je onder Velden.

Templates

Het standaard thema is opgedeeld in verschillende templates.

Template	Beschrijving
wonen/**archive.php**	Wordt door WordPress geladen op de archiefpagina van woningen.
wonen/**search.php**	Bevat het zoekformulier op de archiefpagina. Aparte template i.v.m. live zoeken.
wonen/**loop.php**	Bevat de resultaten op de archiefpagina. Aparte template i.v.m. live zoeken.
wonen/**item.php**	Compacte weergave van een woning, gebruikt vanuit *loop.php* om resultaten te renderen.

Template	Beschrijving
wonen/**map-info.php**	Inhoud van een Google Map info venster. Wordt ingeladen door de Google Maps ondersteuning binnen de plugin.
wonen/**none.php**	Wordt ingeladen vanuit *archive.php* als er geen zoekresultaten zijn.
wonen/**single.php**	Wordt door WordPress geladen wanneer een enkele woning wordt bekeken.

Om het template *wonen/item.php* te renderen vanuit een ander template kan het volgende gebruikt worden:
`<?= Wonen::template('item'); ?>`

Naast de template bestanden worden er standaard de volgende bestanden automatisch ingeladen. Bijvoorbeeld wanneer de plugin gekoppeld wordt met het Realworks CRM: *realworks/functions.php* en *wonen/functions.php.* De eerst genoemde om code van toepassing op alle onderdelen van de plugin te plaatsen en de tweede functions is specifiek voor de module woningen.

> WordPress laadt de archive, single en de functions in omdat we via de geavanceerde instellingen van de plugin het pad aangeven naar de templates. Als je het thema anders op wil bouwen moet je daarom de paden aanpassen.

Globals

Binnen de loop worden zonder zoekopdracht alle woningen opgehaald en op het moment dat de bezoeker een zoekopdracht instelt worden hier de woningen getoond die voldoen aan die zoekopdracht. In deze loop is de huidige woning in de iteratie beschikbaar in de variabele `$woning`. Het is over het algemeen niet nodig om deze variabele expliciet als `global` te declareren wanneer een template gebruikt wordt, de plugin zal dit voor zijn rekening nemen. Met deze variabele kan je informatie van de woning tonen in de item.php.

> Let op: Wanneer er geen gebruik gemaakt wordt van een template is de globale variabele `$woning` leeg. We raden daarom het gebruik van templates aan.

Caching

We raden aan om een WordPress caching plugin te gebruiken om het laden van de site te versnellen. Voor het ophalen van de data voor alle woningen zijn veel database query's nodig, wat alles bij elkaar enkele tientallen milliseconden in beslag kan nemen.

Om dit te versnellen, raden we aan om templates waarin gegevens van een woning worden opgevraagd te cachen. Bijvoorbeeld binnen de loop tijdens het weergeven van de woningen:
`<?= Wonen::template('item')->cache(); ?>`

> Templates worden gecached op basis van de naam en ID van de huidige woning.

Weergave van velden

Het kan voorkomen dat er specifieke eisen zijn om velden op een bepaalde manier weer te geven, soms afhankelijk van andere velden en soms omdat je een prijs als een prijs wil formatten. Om te voorkomen dat deze logica overal door het thema opnieuw geschreven wordt biedt de plugin een aantal manieren om dit op een prettige manier te laten verlopen, waarbij alles op een centrale plek aangepast kan worden.

Een veld uit onze plugin is een object en kan naast het terug geven van zijn waarde nog veel meer informatie terug geven. Zo kunnen de volgende methods worden gebruikt:

```
$woning->plaats; // Het volledige object voor het veld 'plaats'
$woning->plaats->render(); // De plaats van de woning na transformaties
$woning->plaats->value(); // De waarde van het veld plaats
$woning->plaats->label(); // Label van het veld plaats
$woning->plaats->is('Leeuwarden'); // Is de plaats Leeuwarden?
$woning->plaats->is('Leeuwarden', 'Groningen'); // Is de plaats Leeuwarden of Groningen?
$woning->plaats->isnt('Leeuwarden'); // Is de plaats niet Leeuwarden?
$woning->plaats->isEmpty(); // Heeft de plaats geen waarde?
$woning->plaats->hasValue(); // Heeft de plaats een waarde?
```

> **TIP:** Doordat onze plugin gebruik maakt van Magic Methods (__toString()) kunnen we voor het tonen van de plaats ook simpel het volgende schrijven `<?= $woning->plaats; ?>`. Voor een simpele echo is `->render()` niet nodig.

Veld types

Elk veld heeft een bepaald type zodat het weergeven ervan automatisch gebeurt. Ieder van de types heeft verder specifieke methodes als hulp. Voor de onderstaande veldtypes worden letterlijke voorbeelden in de objecten gezet, er zijn echter geen velden die letterlijk ‘list’, ‘boolean’, ‘date’, ‘dateTime’, ‘integer’ of ‘double’ heten. Deze velden referen naar velden met dat type waarde. De velden die geschikt zijn voor jouw CRM pakket vind je allemaal terug onder het hoofdstuk 'velden'.

Arrays

```
$woning->list->render($separator = ', ');
$woning->list->count();
$woning->list->all();
$woning->list->first();
$woning->list->last();
$woning->list->implode($glue);
```

In plaats van komma gescheiden waardes kan ook gebruik worden gemaakt van een lijst. Verander hiervoor de formatter in `list`, waarbij als standaard een `<ul>` wordt gebruikt. Optioneel kunnen twee argumenten worden opgegeven om de list template en item template aan te passen, in het voorbeeld hieronder zijn de standaard templates opgegeven:
`$woning->list->formatter('list')->render('<ul>#{items}</ul>', '<li>#{item}</li>');`

Booleans

```
CustomPost\Formatter\BooleanFormatter::setTrue($true);
CustomPost\Formatter\BooleanFormatter::setFalse($false);
CustomPost\Formatter\BooleanFormatter::setTrueFalse($true, $false);
```

```
$woning->boolean->render($true= null, $false = null);
$woning->boolean->boolValue();
$woning->boolean->isTrue();
$woning->boolean->isFalse();
```

Datums

Voor datums wordt gebruik gemaakt van de Carbon library. Voor het weergeven wordt de PHP functie strftime gebruikt en niet de Carbon API, vanwege localization ondersteuning in `strftime`. Naast de volledige Carbon API zijn de volgende methods beschikbaar:

```
$woning->date->render($format = '%e %B %Y'); // Velden zonder tijd
$woning->dateTime->render($format = '%e %B %Y, %R'); // Velden met tijd
$woning->date->date(); // Alias voor `value`
```

Integers

```
$woning->integer->render($decimals = 0, $decimal = ',', $thousands = '.');
$woning->integer->n('berging', 'bergingen'); // Kiest de juiste vorm
```

Doubles

```
$woning->double->render($decimals = 2, $decimal = ',', $thousands = '.');
```

Bedragen

```
CustomPost\Formatter\MoneyFormatter::setCurrency($currency);
CustomPost\Formatter\MoneyFormatter::setDecimals($decimals);
```

```
$woning->prijs->render($currency = null, $suffix = null, $decimals = null);
```

Weergave aanpassen

Voor ieder veld kan echter de weergave ervan worden aangepast. Als voorbeeld zetten we het veld *plaats* om in hoofdletters:

```
Wonen::formatter('plaats', function ($value)
{
    return strtoupper($value);
});
```

> Standaard wordt het veld *plaats* al slim omgezet in de juiste vorm. Bovenstaande geldt als voorbeeld en kan beter worden bereikt met CSS.

Afhankelijkheden van andere velden

In sommige gevallen is voor het bepalen van de weergave meer informatie over de woning nodig. Het volgende voorbeeld toont hoe de koopprijs wordt weergegeven in combinatie met de koopconditie, maar alleen als de prijs mag worden weergegeven.

```
Wonen::formatter('koopprijs', function ($value, $field)
{
    if ($field->post()->prijsTonen->isTrue())
    {
        return $field->formatter('money').' '.$field->post()->koopconditie;
    }
    else
    {
        return 'Prijs op aanvraag';
    }
```

```
});
```

Waar het eerste argument dus de waarde van het veld is, is het tweede argument het veld zelf. De woning is van daaruit op te vragen a.d.h.v. `$field->post()`.

In bovenstaande voorbeeld wordt ook duidelijk dat we terug kunnen vallen op de valuta weergave door gebruik te maken van de formatter 'money' : `$field->formatter('money')`, zoals het *koopprijs* veld origineel zou worden weergegeven.

> De volgende formatters zijn beschikbaar: *string, bool, date, datetime, double, float, integer, array, list, money*.

Formatter class

Als niet alleen de waarde van het veld moet worden getransitioneerd maar bijvoorbeeld ook het label moet worden afgeleid van een ander veld, kan een class worden gebruikt om de weergave te beinvloeden:

```
class KoopprijsFormatter extends CustomPost\Formatter\DelegateFormatter
{
    public function label ()
    {
        return ucfirst($this->post()->koopprijsVoorvoegsel->render()) ?: parent::label();
    }

    public function render ()
    {
        if ($this->post()->prijsTonen->isTrue())
        {
            return $this->formatter('money').' '.$this->post()->koopconditie;
        }
        else
        {
            return 'Prijs op aanvraag';
        }
    }
}
Wonen::formatter('koopprijs', 'KoopprijsFormatter');
```

De waarde van het veld is beschikbaar via `$this->value` en de woning zelf via `$this->post()`.

Label aanpassen

In het vorige voorbeeld wordt een class gebruikt om het label af te leiden van een ander veld. In veel gevallen zal het label echter niet afhankelijk zijn van een waarde en kan een simpelere constructie worden gebruikt:

```
Wonen::label('koopconditie', 'Conditie');
```

Values aanpassen

Als een veld een beperkt aantal values heeft waarvan de weergave moet worden aangepast, gebruik dan de volgende constructie:

```
Wonen::values('koopconditie', array(
    'kosten koper' => 'k.k.',
    'vrij op naam' => 'v.o.n.',
));
```

Hierdoor zal het veld *koopconditie* met waarde *kosten koper* worden weergegeven als *k.k.*, enz. Waardes waarvoor geen transformatie is opgegeven zullen zonder wijziging worden weergegeven.

> Let er op dat je de waardes aanpast die zijn weggeschreven in de database. Voor de koppeling met het Realworks CRM gaat het bijvoorbeeld om de values `KOSTEN_KOPER` en `VRIJ_OP_NAAM`.

Listings

Vaak moet een lijstje van bepaalde eigenschappen worden gegeven. Daarbij moet worden gecontroleerd of iedere eigenschap wel een waarde heeft om zo geen lege rijen te tonen. Dit is waar listings gebuikt kunnen worden:

```
<?= $woning->listing('prijs', 'woning.perceelOppervlakte', 'woning.aantalKamers'); ?>
```

Bovenstaande snippet geeft een lijst met de opgegeven velden in een lijst, met hun bijbehorende labels en alleen als de velden een waarde hebben. Optioneel kan er een titel worden weergegeven welke voor de lijst komt te staan:

```
<?= $woning->listing(...)
    ->title('<h3>Eigenschappen</h3>'); ?>
```

Lege listings

Wanneer geen van de velden een waarde heeft wordt de listing in zijn geheel niet weergegeven. Om toch bijvoorbeeld een melding te geven kan het volgende worden gebruikt:

```
<?= $woning->listing(...)
    ->ifEmpty('<p>Geen eigenschappen bekend</p>'); ?>
```

> Als op deze manier een melding wordt ingevoegd wordt ook de titel getoond.

Templates

Standaard wordt een definition list `<dl><dt>#{label}</dt><dd>#{value}</dd></dl>` structuur gebruikt om de velden weer te geven. Dit kan echter naar wens worden aangepast, als volgt:

```
<?= $woning->listing(...)
    ->before('<div>')->between('<hr>')->after('</div>')
    ->item('<div>#{label}: #{value}</div>'); ?>
```

Template globaal aanpassen

Binnen een site zal vaak dezelfde opmaak gebruikt worden. Het is mogelijk om dit globaal te wijzigen:

```
CustomPost\Formatter\Listing::templates(array(
    'title' => '<h2>#{title}</h2>',
));
```

De volgende templates zijn aan te passen:

Template	Standaard waarde
title	`'#{title}'`
before	`'<dl>'`
item	`'<dt>#{label}</dt><dd>#{value}</dd>'`
between	`''`
after	`'</dl>'`
empty	`'#{message}'`

Specifiek per veld

Soms moet een veld op een andere manier worden weergegeven dan de overige velden. Om dit te bereiken kan het `item` template hiervoor specifiek per veld worden opgegeven:

```
<?= Wonen::listing('woning.voorzieningen')
    ->before('<ul>')->after('</ul>')
    ->item('<li>#{value}</li>')
    ->field('woning.voorzieningen', '#{items}')
?>
```

Naast een template string kan ook een render callback functie worden opgegeven, waarbij in de callback alles voor een normaal veld handmatig moet worden gedaan (alles wat normaal via het `item` template wordt bereikt moet dus nu worden herhaald).

Aangepaste labels

We hebben al gezien hoe het label van een veld kan worden gewijzigd. Omdat een aangepast label soms alleen nodig is op bepaalde plekken is het mogelijk om dit per listing aan te passen:

```
<?= $woning->listing(array(
    'prijs',
    'woning.perceelOppervlakte' => 'Oppervlak',
    'woning.aantalKamers',
)); ?>
```

Voor *woning.perceelOppervlak* zal vervolgens binnen de listing het label *Oppervlak* worden gebruikt, de overige velden behouden hun originele label.

Extra data meegeven

Soms is het nodig om per eigenschap een class op te geven, voor bijvoorbeeld een icoontje. Geef hiervoor in plaats van een enkel label een array op met de benodigde data.

```
<?= $woning->listing(array(
    'woning.perceelOppervlakte' => array('label' => 'Perceel', 'class' => 'expand', 'suffix' => ''),
    'woning.aantalKamers' => array('label' => 'Kamers', 'class' => 'building-o', 'suffix' => array(' kamer', '
kamers')),
))
->item(
    '<dt><i class="fa fa-#{data.class}"></i> #{label}</dt>
     <dd>#{value}#{data.suffix}</dd>'
); ?>
```

Met de key `'label'` kan het label als voorheen worden overschreven, de overige keys zijn beschikbaar via `#{data.key}`.

> De array notatie voor `'suffix'` geeft de twee waardes op voor enkelvoud en meervoud.

Render argumenten opgeven

Geef aangepaste argumenten op om een veld te renderen door in de lijst met data de key `'render'` op te geven als array van de argumenten.

```
<?= $woning->listing(array(
   'datumInvoer' => array('render' => '%B %Y'),
)); ?>
```

Formatter type aanpassen

Het type formatter kan worden aangepast door in de lijst met data de key `'formatter'` op te geven met de naam van de gewenste formatter.

```
<?= $woning->listing(array(
   'woning.voorzieningen' => array('formatter' => 'list'),
)); ?>
```

Item template aanpassen

We hebben al gezien hoe `$listing->field($name, $template)` kan worden gebruikt om de template specifiek voor een item aan te passen. Voor gemak kan het ook via de data key `'template'` worden bereikt. In het volgende voorbeeld gebruiken we al het bovenstaande om een lijst weer te geven waarbij alle waardes van het veld *woning.voorzieningen* geintegreerd zijn in de lijst zelf:

```
<?= $woning->listing(array(
   'woning.perceelOppervlakte',
   'woning.aantalKamers',
   'woning.voorzieningen' => array('template' => '#{value}', 'formatter' => 'list', 'render' => '#{items}')
))
->before('<ul>')->after('</ul>')
->item('<li>#{value}</li>')
?>
```

Macros

Vaak worden stukken code vaker gebruikt en is het gewenst om logica uit de templates te houden. Hiervoor kunnen zogenaamde macros worden toegevoegd, welke dan als method op iedere woning kunnen worden aangeroepen.

```
/* wonen/functions.php */
    Wonen::macro('similar', function ($woning, $amount = 3)
    {
        return Wonen::search(array($woning->plaats), array('posts_per_page' => $amount));
    });

    /* wonen/single.php */
    // Resultaat wordt gecached, herhaaldelijk opvragen geeft identiek resultaat
    $similar = $woning->similar;

    // Voert macro iedere keer uit<
    $similar = $woning->similar();

    // Geef aangepaste argumenten op
    $similar = $woning->similar(10);
```

Customs

Macros kunnen ook een veld als resultaat geven. In dat geval kan de macro net als ieder ander veld worden gebruikt, het is dus een soort alias naar een ander veld. Zo is het *prijs* veld wat standaard beschikbaar is ook een alias naar *koopprijs* of *huurprijs*, afhankelijk van welke beschikbaar is. Het voordeel hiervan is dat hierdoor `$woning->prijs->label()` automatisch aangeeft of het om een koop- of huurprijs gaat.

Bij het gebruik van een macro is het echter van belang dat er altijd een veld als resultaat wordt gegeven, `null` of alleen een waarde als resultaat zal fouten opleveren. Om dit te voorkomen kunnen er custom velden worden gedefinieerd:

```
Wonen::custom('soort', function ($woning)
{
if($woning->woning->woonhuis->soort->hasValue())
{
    // Een veld als resultaat werkt gelijk als bij macros
    return $woning->woning->woonhuis->soort;
}
else if ($woning->bouwgrond->huidigeBestemming->hasValue())
```

```
{
    // Met custom velden kunnen we echter ook direct een waarde voor<
    // het veld opgeven. Het label wordt afgeleid van de naam van het
    // veld, in dit geval wordt het Soort.
    return 'Bouwgrond';
}

// Zonder een waarde als resultaat geven is gelijk aan `return null`,
// waarbij wordt aangenomen dat het veld leeg is.
}, 'string');
```

Het laatste argument, in bovenstaande voorbeeld `'string'`, bepaalt het type van het veld. Omdat `'string'` standaard is kan het in dit geval worden weggelaten.

Custom veld in een listing

Een andere reden om een custom te maken is het tonen van informatie die binnen een collection staat. Om de informatie binnen een collection te tonen zal je namelijk een loop op moeten zetten, er kunnen namelijk meerdere van zijn. Een voorbeeld van een collection is voor het veld `$object->object->functies` binnen de module voor bedrijfspanden voor de gebruikers van het CRM van Realworks. Een object kan meerdere functies hebben en voor iedere functie zijn er kenmerken die je wil kunnen tonen.

Zo wordt het aantal verdiepingen in de feed toegestuurd per functie van het object. Als je het aantal verdiepingen van het volledige object wil tonen zal je daarom een loop op moeten zetten over de functies van het object. Vervolgens moet je voor elke functie het aantal verdiepingen bij elkaar op tellen.

Dit is standaard niet mogelijk binnen een listing en als je de waarde buiten een listing wil tonen zal je eerst de hele loop moeten schrijven. Als je het veld op meerdere plekken wil tonen moet je ook nog eens de loop op meerdere plekken zetten. Een custom is daarom een goede uitkomst.

Hieronder zie je een voorbeeld om het aantal verdiepingen van een bedrijfsobject te tonen door middel van het veld `$object->aantalVerdiepingen`. Je ziet de loop over de functies en voor iedere functie tellen we het aantal verdiepingen bij elkaar op en wanneer het aantal hoger is dan 0 dan wordt de waarde in de `return` gezet.

```
Bog::custom('aantalVerdiepingen', function($object) {
    $verdiepingen = 0;

    foreach ($object->object->functies as $functies) {
        $verdiepingen += $functies->overige->aantalVerdiepingen->value();
        $verdiepingen += $functies->kantoorruimte->verdiepingen->value();
        $verdiepingen += $functies->bedrijfsruimte->kantoorruimte->verdiepingen->value();
```

```
    $verdiepingen += $functies->winkelruimte->aantalVerdiepingen->value();
}

return ($verdiepingen > 0) ? $verdiepingen : null;
});
```

In een listing kan je het bovenstaande veld als volgt gebruiken:

```
<?= $object->listing(array(
    'aantalVerdiepingen' => 'Verdiepingen',
)); ?>
```

Querying

Het opvragen van woningen is ondersteund via `Wonen::query()`, waarbij via het eerste optionele argument de `WP_Query` argumenten kunnen worden opgegeven.
`$woningen = Wonen::query(array('posts_per_page' => 5));`

Het resultaat is een Collection van posts.

Sorteren

Sorteer de resultaten door gebruik te maken van de `orderby` optie van `WP_Query.` De beschikbare sorteervolgordes zijn `asc` voor oplopend (A-Z) en `desc` voor aflopend (Z-A) en kunnen worden opgegeven door het veld en de ordering te scheiden met een dubbele punt. Meerdere velden kunnen worden opgegeven door ze te scheiden met een komma.
`$woningen = Wonen::query(array('orderby' => 'prijs:desc,plaats:asc'));`

Zoeken

Om op woningen met specifieke eisen te zoeken, kan `Wonen::search()` worden gebruikt. Vervolgens kan de zoekquery worden opgebouwd via chained method calls. Het is mogelijk om de aanroep tot `search()` weg te laten, alle ondersteunde calls zijn ook direct onder `Wonen` beschikbaar.

```
// Alle woningen met plaats Leeuwarden
$woningen = Wonen::search()->where('plaats', 'Leeuwarden');
$woningen = Wonen::search()->where('plaats', '=', 'Leeuwarden');
$woningen = Wonen::where('plaats', 'Leeuwarden');

// Uitgebreid zoeken met meer eisen
```

```
$woningen = Wonen::whereNotNull('plaats')->where('koopprijs', '>', 100000)->whereSql('datumInvoer', '>
DATE_SUB(NOW(), INTERVAL 1 WEEK)');
```

De volgende methodes zijn beschikbaar om zoektermen op te geven:

Methode	Beschrijving
`where('veld', '=', $value)`	Geef op dat `veld` een bepaalde waarde heeft. Operator `=` is standaard en kan worden weggelaten.
`whereBetween('veld', $min, $max)`	Geef op dat `veld` tussen `$min` en `$max` moet zijn.
`whereNull('veld')`	Geef op dat `veld` geen waarde mag hebben.
`whereNonNull('veld')`	Geef op dat `veld` een waarde moet hebben.
`whereSql('veld', 'sql')`	Gebruik een ruwe SQL clause als zoekopdracht.
`matching($woning->field, $operator = '=')`	Zoek op woningen met dezelfde waarde als `$woning->field`, optioneel kan een operator worden opgegeven.
`without($woning, ...)`	Neemt `$woning` niet mee in de resultaten.

Daarnaast zijn de volgende methodes beschikbaar voor het opgeven van WordPress query argumenten, om bijvoorbeeld het aantal resultaten en de sorteervolgorde aan te passen:

Methode	Beschrijving
`amount($n)`	Beperkt het resultaat tot `$n` woningen.
`all()`	Geef op dat alle woningen opgehaald moeten worden.
`order($field, $direction = 'asc')`	Geef sorteervolgorde op.
`ascending($field)`	Geef oplopende sorteervolgorde op.
`descending($field)`	Geef aflopende sorteervolgorde op.
`arg($key, $value)`	Geef direct een WordPress query argument op.
`args($args)`	Geef direct WordPress query argumenten op.

Nesting

Zoektermen kunnen worden genest om te schakelen tussen `and` en `or`. Standaard moet aan alle zoektermen worden voldaan, maar door eerst `push('or')` en vervolgens `pop()` aan te roepen hoeft alleen aan minimaal één van de zoektermen ertussenin voldaan te worden.

```
// Zoek woningen met een plaats, van de laatste week, met een koopprijs van meer dan een ton of een huurprijs
van meer dan 600 euro.
$woningen = Wonen::whereNotNull('plaats')
  ->push('or')
  ->where('koopprijs', '>', 100000)->where('huurprijs', '>', 600)
```

```
    ->pop()
    ->whereSql('datumInvoer', '> DATE_SUB(NOW(), INTERVAL 1 WEEK)');
```

Soortgelijke woningen zoeken

Het is eenvoudig gemaakt om voor de huidige woning, dus bijvoorbeeld binnen de template file `single.php`, te zoeken naar woningen in dezelfde plaats. Geef hiertoe alleen het veld in:

```
// Zoek drie woningen met dezelfde plaats als de huidige woning, maar goedkoper
$woningen = Wonen::matching($woning->plaats)->matching($woning->prijs, '<')->amount(3);
```

Woningen tellen

Soms is het alleen het aantal resultaten nodig.
`$aantal = Wonen::whereSql('datumInvoer', '> DATE_SUB(NOW(), INTERVAL 1 WEEK)')->total();`

Performance

We ontwikkelen de plugins altijd door, dit heeft er in geresulteerd dat het laden van de plugin tot 2.5 keer sneller is dan voor voorheen. Dit is bereikt door het laden van de modules Wonen, BOG, Nieuwbouw en A&LV uit te stellen tot het moment waarop ze als eerste gebruikt worden, om zo onnodig werk te voorkomen. Dit vereist dat ook het definieren van formatters en overige acties met betrekking tot velden moet worden uitgesteld tot wanneer een component wordt geladen. Hiervoor moet het manueel includen van de verschillende functions.php bestanden uit *active-theme/functions.php* worden verwijderd zodat deze bestanden vervolgens on-demand worden ingeladen. Let hierbij op dat in deze bestanden de acties, die stonden gepland in de `init` hook, direct uitgevoerd dienen te worden, `init` is op dit moment namelijk al uitgevoerd geweest.

> Wanneer de manuele includes niet verwijderd worden zal alles blijven werken, echter blijft de snelheidswinst dan ook grotendeels uit.

Modules uitschakelen

De plugins zullen normaal alle beschikbare modules laden, terwijl ze wellicht niet allemaal gebruikt zullen worden. Pas hiertoe voor bijvoorbeeld het Realworks CRM *realworks/functions.php* aan en schakel de modules hier uit door middel van:

```
Realworks::disable('wonen');
Realworks::disable('bog');
Realworks::disable('nieuwbouw');
```

Vervolgens zullen de modules niet meer worden ingeladen. Merk op dat de modules hierdoor ook niet langer beschikbaar zijn in de admin omgeving van de plugin.

Zoekvelden

Eerder heb je over het instellen van zoekvelden gelezen in het admin gedeelte van onze plugin, in dit gedeelte van de documentatie leggen we uit hoe je de ingestelde zoekvelden kunt koppelen aan de code in de `search.php` en daarnaast leggen we uit hoe je de zoekvelden kunt gebruiken.

Zoekvelden koppelen

In het admin gedeelte van de plugin kan je zoekvelden aanmaken, de naam die je het zoekveld daar geeft zorgt voor de koppeling. Een aantal voorbeelden.

Status: `<?php Wonen::form()->dropdown('status'); ?>`
Plaats: `<?php Wonen::form()->dropdown('plaats'); ?>`
Land: `<?php Wonen::form()->dropdown('land'); ?>`
Type: `<?php Wonen::form()->dropdown('type'); ?>`

Let er daarom goed op dat je de naam een duidelijke, maar simpele naam geeft.

> Een naam met spaties, hoofdletters of bijzondere karakters kan voor onverwachte problemen zorgen.

Type zoekvelden

Zoekvelden aangemaakt vanuit de admin kunnen op verschillende manieren worden weergegeven.

Dropdown

`<?php Wonen::form()->dropdown('naam', $options = []); ?>`

De volgende mogelijkheden heb je in de options array:

Optie	Type	Standaard	Beschrijving
`'emptyLabel'`	string	`''`	Label van optie om zoekveld leeg te laten.
`'showCounts'`	bool	`false`	Weergave van aantal resultaten per optie.

Optie	Type	Standaard	Beschrijving
`'hideWhenNoResults'`	bool	`false`	Verberg een optie als er 0 resultaten zijn.
`'hideWhenNotExists'`	bool	`false`	Verberg een optie als de waarde nooit resultaten zal geven.
`'select'`	array	`[]`	Extra HTML attributen voor de `<select>`.
`'option'`	array	`[]`	Extra HTML attributen voor `<option>` elementen.
`'emptyOption'`	array	`[]`	Extra HTML attributen voor `<option>` elementen, specifiek voor lege optie.

```
<?php Wonen::form()->minDropdown('naam', $options = []); ?>
<?php Wonen::form()->maxDropdown('naam', $options = []); ?>
<?php Wonen::form()->minMaxDropdown('naam', $options = []); ?>
```

Optie	Type	Standaard	Beschrijving
`'emptyLabelMin'`	string	`''`	Label van optie om zoekveld leeg te laten voor minimum dropdown.
`'emptyLabelMax'`	string	`''`	Label van optie om zoekveld leeg te laten voor maximum dropdown.

Checkboxes & Radio buttons

```
<?php Wonen::form()->checkboxes('naam', $options = []); ?>
<?php Wonen::form()->radios('naam', $options = []); ?>
```

Optie	Type	Standaard	Beschrijving
`'showCounts'`	bool	`false`	Weergave van aantal resultaten per optie.
`'hideWhenNoResults'`	bool	`false`	Verberg een optie als er 0 resultaten zijn.
`'hideWhenNotExists'`	bool	`false`	Verberg een optie als de waarde nooit resultaten zal geven.
`'label'`	array	`[]`	Extra HTML attributen voor `<label>` elementen.

Optie	Type	Standaard	Beschrijving
'input'	array	[]	Extra HTML attributen voor <input> elementen.
'count'	array	[]	Extra HTML attributen voor de om aantal resultaten in te tonen.

Verborgen input

```
<?php Wonen::form()->hidden('naam', $options = []); ?>
```

Optie	Type	Standaard	Beschrijving
'input'	array	[]	Extra HTML attributen voor de <input>.

> Extra HTML attributen kunnen worden opgegeven via de opties, zoals in de tabellen staat beschreven. Voor nodes die specifiek zijn voor een bepaalde optie worden de placeholders {{label}}, {{value}} en {{count}} ingevuld met de label, waarde en aantal resultaten voor de optie. Dit kan bijvoorbeeld worden gebruikt om extra data-attributen op te geven welke door JavaScript plugins worden gebruikt om de weergave van de zoekvelden aan te passen.

Vrije input

Zoekvelden hebben een vast aantal opties en accepteren geen aangepaste waardes. Om bijvoorbeeld te kunnen zoeken op adres moet handmatig de query worden aangepast. Als HTML fragment kan het volgende worden gebruikt:

```
<input type="text" name="adres" value="<?= Wonen::form()->value('adres'); ?>">
```

Om vervolgens het adres op de query toe te passen moet een event listener worden toegevoegd om de query te wijzigen:

```
Wonen::listen('search.before', function($form, $query)
{
    if ($address = $form->value('adres'))
    {
        $query->compare('adres', 'like', '%'.str_replace(' ', '%', $address).'%');
    }
});
```

De query kan in alle mogelijke constructies worden beinvloed:

Operator	Code
IS NULL	`$query->isNull('veld')`
IS NOT NULL	`$query->isNotNull('veld')`
BETWEEN	`$query->between('veld', $min, $max)`
SQL	`$query->sql('veld', '> NOW()')`
=, <>, <, >, !=, …	`$query->compare('veld', $operator, $value)`

Een geneste groep kan worden begonnen met `$query->push('and|or')`, waarna opvolgende declaraties binnen die groep worden toegepast. Een groep moet worden afgesloten met `$query->pop()`.

Nieuwe types toevoegen

Voor meer controle over de HTML kunnen nieuwe types worden gemaakt. De standaard types zijn makkelijk aanpasbaar door ze te subclassen en voor de gewenste methods een aangepaste implementatie te schrijven.

```
class CustomRenderer extends CustomPost\Search\Renderers\Dropdown
{
    // ...
}


Wonen::form()->render('naam', new CustomRenderer($options = []));
```

Door de subclass te registreren onder een bepaalde identifier kan het nieuwe/aangepaste type eenvoudiger worden gebruikt:

```
CustomPost\Search\Form::register('identifier', 'CustomRenderer');


Wonen::form()->identifier('naam', $options = []);
```

Daarnaast is het ook mogelijk om bestaande types aan te passen zodat je meer vrijheid hebt in de vormgeving. In het onderstaande voorbeeld kan je zien hoe je een `span` toe kan voegen om het label binnen `checkboxes` en `radios`, hiermee kan je bijvoorbeeld met een `::before` of `::after` de input vervangen voor een mooiere vormgeving.

```
// Checkboxes
use \CustomPost\Search\Option;
```

```
class CustomCheckboxes extends \CustomPost\Search\Renderers\Checkboxes {
    protected function renderLabel(Option $option) {
        $option->setLabel('<span>' . $option->getLabel() . '</span>');
        parent::renderLabel($option);
    }
}

// Radios
class CustomRadios extends \CustomPost\Search\Renderers\Radios {
    protected function renderLabel(Option $option) {
        $option->setLabel('<span>' . $option->getLabel() . '</span>');
        parent::renderLabel($option);
    }
}
CustomPost\Search\Form::register('checkboxes', new CustomCheckboxes());
CustomPost\Search\Form::register('radios', new CustomRadios());
```

> Voor meer informatie wordt aangeraden om contact met ons op te nemen. De standaard types bieden ruime mogelijkheden om via CSS aan te passen, alleen in uitzonderlijke gevallen zal een aangepast type noodzakelijk zijn.

Zoekopties vanuit Javascript

In enkele gevallen kan het handig zijn om alle opties van een zoekveld in Javascript beschikbaar te hebben, bijvoorbeeld om afhankelijke velden handmatig bij te kunnen werken als er geen gebruik wordt gemaakt van live bijwerken. Een geneste structuur, vooral geschikt voor een zoekveld met afhankelijkheden, kan in JSON formaat worden verkregen via:

```
Wonen::form()->options('plaats')->json();
```

Dit geeft een JSON object waarbij alle opties onderverdeeld zijn onder de afhankelijke waarde waar de optie bij hoort. Als voorbeeld een zoekveld plaats waarvan de waardes afhankelijk zijn van het geselecteerde land:

```
{
   "nederland": [
      {"value": "leeuwarden", "label": "Leeuwarden"},
```

```
        {"value": "groningen", "label": "Groningen"}
    ],
    "duitsland": [
        {"value": "berlijn", "label": "Berlijn"}
    ]
}
```

Het is echter ook mogelijk om alle opties in een vlakke lijst te krijgen:

```
Wonen::form()->options('plaats')->flat()->json();
```

```
[
    {"value": "leeuwarden", "label": "Leeuwarden", "parents": {"land": "nederland"}},
    {"value": "groningen", "label": "Groningen", "parents": {"land": "nederland"}},
    {"value": "berlijn", "label": "Berlijn", "parents": {"land": "duitsland"}}
]
```

> In plaats van de opties in JSON representatie kan ook een PHP array worden opgevraagd via `get()` in plaats van `json()`.

Voorbeeld

Als voorbeeld een manier om een dropdown voor het zoekveld plaats bij te werken met de opties behorende bij een bepaald land, nadat een ander land is geselecteerd. We geven hierbij alle opties in bovenstaande geneste structuur op via een HTML data-attribuut, zodat dit vervolgens vanuit Javascript beschikbaar is.

```
<?php Wonen::form()->dropdown('plaats', array(
    'emptyLabel' => 'Plaats',
    'select' => array(
        'data-options' => Wonen::form()->options('plaats')->json(),
    ),
)); ?>
```

Het onderstaande is de minimaal benodigde Javascript om de dropdown bij te werken:

```
$('#land').change(function() {
    var land = $(this), plaats = $('#plaats');

    plaats.empty().append($('<option></option>').attr('value', '').text('Plaats'));
```

```
    $.each(plaats.data('options')[land.val()] || [], function(option) {
        plaats.append($('<option></option>').attr('value', option.value).text(option.label));
    });
});
```

Resultaten sorteren

Binnen het Admin gedeelte van de documentatie heb je kunnen lezen hoe je sorteer opties kunt toevoegen, dit doe namelijk via de geavanceerde instellingen van de plugin. Om de bezoeker van de website de mogelijkheid te geven om te switchen tussen de door jou ingestelde opties moeten we een `orderby` veld vullen met opties.

Een lijst van de ingestelde sorteeropties is beschikbaar via `Wonen::form()->orderings()` (voor de module Wonen) en kan bijvoorbeeld worden gebuikt om een dropdown weer te geven:

```
<select name="orderby">
    <?php foreach (Wonen::form()->orderings() as $ordering): ?>
        <option value="<?= $ordering['orderby'] ?>" <?= selected($ordering['current']); ?>>
            <?= $ordering['label'] ?>
        </option>
    <?php endforeach; ?>
</select>
```

Als je in plaats van een dropdown buttons wil gebruiken kan bijvoorbeeld een verborgen input worden ingevoegd waarvan de waarde met JavaScript wordt aangepast:

```
<input type="hidden" name="orderby" id="orderby" value="<?= Wonen::form()->ordering(); ?>">

<?php foreach (Wonen::form()->orderings() as $ordering): ?>
    <a class="orderby" data-orderby="<?= $ordering['orderby'] ?>"><?= $ordering['label'] ?></a>
<?php endforeach; ?>
```

```
$(document).on('click', 'a.orderby', function() {
    var orderby = $(this).data('orderby'), current = $('#orderby').val();

    if (current !== orderby) {
        $('#orderby').val(orderby).trigger('change');
    }
```

```
}
```

> In de vormgeving ben je vrij, zolang je maar de `name` `orderby` mee stuurt met de zoekopdracht. De plugin sorteert vervolgens de resultaten.

Hulpmiddelen

De plugin bevat ingebouwde hulpmiddelen die je kunnen helpen bij het ontwikkelen van je thema, hier lees je meer over deze hulpmiddelen.

Woningen overslaan

Het kan voorkomen dat je bepaalde woningen niet wilt importeren, bijvoorbeeld als ze vertrouwelijk zijn of wanneer de publicatiedatum in de toekomst ligt. Dit kan worden bereikt door een filter toe te voegen welke bepaalt of de woning moet worden overgeslagen.

In het onderstaande voorbeeld worden woningen die vertrouwelijk zijn en woningen met een publicatiedatum in de toekomst overgeslagen.

```
Wonen::filter('updater.skip', function($skip, $woning)
{
    // Skip deze woning wanneer vertouwelijk op 'ja' is ingevuld
    if ($woning->vertrouwelijk->value()) {
        return true;
    }

    // Skip deze woning als de publicatiedatum in de toekomst ligt
    if ($woning->publicatiedatum->hasValue()) {
        if ($woning->publicatiedatum->date()->startOfDay() > \Carbon\Carbon::now()) {
            return true;
        }
    }
    return $skip;
});
```

De bovenstaande code kan je toevoegen binnen het onderstaande bestand, dit is een voorbeeld voor de gebruikers van het CRM van Realworks:
`/wp-content/themes/*active-theme*/realworks/functions.php`

Het mechanisme werkt via de WordPress `add_filter` API, het eerste argument `$skip` is standaard `false`. Want zonder dit filter willen we immers alle woningen uit de feed importeren. Dit is de reden dat we `return true` doen op het moment dat een woning aan onze eisen voldoet.

> Wanneer je van de laatste functie in de bovenstaande code gebruikt controleer dan of je de volgende code bovenaan je functions.php hebt staan: `use Carbon\Carbon;`

Post Collections

Ingebouwde query resultaten zijn van het type `QueryCollection`, een gebruiksvriendelijke interface in plaats van het standaard WordPress `WP_Query` object. Dit biedt een aantal voordelen, zo zijn alle methods uit Illuminate's Collection class beschikbaar, en blijven ook alle properties en methods van `WP_Query` direct beschikbaar.

```
$woningen = Wonen::search(...);


$woningen->count(); // Aantal woningen in resultaat, rekening houdende met pagination
$woningen->total(); // Totaal aantal gevonden resultaten, zonder pagination
$woningen->has(); // Of er resultaten zijn
$woningen->isEmtpty();
$woningen->first();
$woningen->last();
$woningen->random();
$woningen->sample(3); // Collection van 3 random woningen
```

> Het `WP_Query` object is beschikbaar via `$woningen->getQuery()`, al zijn alle properties en methods dus ook beschikbaar op de Collection zelf.

The loop

De Collection biedt een PHP iterator om de WordPress loop constructie niet handmatig uit te hoeven schrijven:

```
while ($woningen->next()):
    // ...
```

```
endwhile;
```

Dit is equivalent aan de loop constructie:

```
while ($woningen->have_posts()): $woningen->the_post();
    // ...
endwhile;
wp_reset_postdata();
```

Media

Voor iedere woning kan de bijbehorende media eenvoudig worden opgevraagd d.m.v. `$woning->media()` met eventueel een argument om query opties in te geven:

```
$media = $woning->media(array('posts_per_page' => 3));
```

Ook dit levert een Collection op.

Groepen

Vanuit jouw CRM wordt media onderverdeeld in een aantal groepen. Alle media in een bepaalde groep kan eenvoudig worden opgevraagd, als volgt:

Methode	Groepen
`$woning->afbeeldingen()`	HoofdFoto, Foto, Plattegrond
`$woning->fotos()`	HoofdFoto, Foto
`$woning->plattegronden()`	Plattegrond
`$woning->brochures()`	Brochure
`$woning->groep('HOOFDFOTO')`	Hoofdfoto afbeelding
`$woning->groep('FOTO')`	Foto afbeeldingen
`$woning->groep('PLATTEGROND')`	Plattegrond afbeeldingen
`$woning->groep('DOCUMENT')`	PDF bestanden
`$woning->groep('VIDEO')`	MP4 bestanden

Methode	Groepen
`$woning->groep('CONNECTED_PARTNER')`	Media van 3e partijen (bv. floorplanner / virtuele tour)

Omdat er logischerwijs maximaal één brochure gekoppeld is, is ook `$woning->brochure()` beschikbaar welke `null` geeft als er geen brochure beschikbaar is.

Voor iedere andere combinatie van groepen kan `$woning->groep(...)` worden gebruikt, met als eerste argument een groep of array van groepen.

Groepen - Skarabee

Methode	Groepen
`$woning->pictures()`	HoofdFoto, Foto
`$woning->floorplans()`	Plattegrond
`$woning->documents()`	Documenten

Caching

Iedere keer als een van bovenstaande methods wordt aangeroepen zal opnieuw het resultaat uit de database worden opgevraagd. Om dit te voorkomen zijn de methods afbeeldingen, fotos, plattegronden, brochures en brochure ook beschikbaar als properties, waarbij alleen bij het eerste gebruik de resultaten zullen worden opgehaald.

```
foreach ($woning->afbeeldingen as $afbeelding):
    // ...
endforeach;
```

Doordat de `QueryCollection` wordt gebruikt, werkt bovenstaande voorbeeld impliciet de loop af en zal achteraf automatisch de postdata worden herstelt.

Dit voorbeeld gebruikt een `foreach` loop in plaats van `while` zoals bij de Collections documentatie. Dit is omdat hiermee een nieuwe loop gestart wordt, waarbinnen `$woning` niet langer beschikbaar is. Bij gebruik van `while ($woning->afbeeldingen->next())` zal na de eerste afbeelding `$woning` dus `null` zijn.

Overig

Code	Omschrijving

`$woning->prijs`	De *koopprijs* of *huurprijs* van de woning, afhankelijk van welke beschikbaar is.
`$woning->isKoop`	Bepaalt of de woning een koopprijs heeft.
`$woning->isHuur`	Bepaalt of de woning een huurprijs heeft.
`$woning->hasLocation`	Bepaalt of voor de woning locatiegegevens bekend zijn.
`$woning->adresPlaats`	Het adres van de woning, inclusief plaats. Houdt ook rekening met woningen in het buitenland, waarbij de adresgegevens in andere velden staat opgeslagen.

Javascript

Vanuit de plugin is standaard ondersteuning om het zoeken interactief te maken. Het meegeleverde thema bevat enkele Javascript bestanden, in dit gedeelte van de plugin willen we duidelijk maken hoe deze kunnen worden gebruikt en wat alle opties zijn.

Template	Beschrijving
js/wonen/**archive.js**	Geregistreerd als `wonen-archive` en te enqueuen vanuit *wonen/archive.php*.
js/wonen/**single.js**	Geregistreerd als `wonen-single` en te enqueuen vanuit *wonen/single.php*.
js/**googlemap.js**	Google Maps plugin. Geregistreerd als `realworks-googlemap` en als dependency op te geven aan `wonen-archive` en `wonen-single`.

Live bijwerken

Een van de voornaamste features van de plugin is het direct bijwerken van resultaten nadat een zoekveld van waarde is veranderd. Het principe is eenvoudig: vanuit Javascript worden alle wijzigingen opgemerkt en wordt het zoekformulier via AJAX verzonden, waarna het wordt verwerkt op de server. Deze stuurt een JSON response met de templates die vernieuwd moeten worden terug, zodat de nieuwe templates vervolgens in de DOM worden ingevoegd. Ook pagination links worden onderschept en verwerkt op dezelfde manier.

Een bijkomend voordeel van deze methode is dat een lange, lelijke en onduidelijke query string wordt voorkomen, doordat het formuleer wordt geëncodeerd in een hash welke aan de URL wordt toegevoegd. Hierdoor blijft de browsergeschiedenis intact en wordt bij het vernieuwen van de pagina de gewenste zoekopdracht weer uitgevoerd.

De plugin wordt ingeladen door `custompost-liveform` toe te voegen als script dependency, waarna initialisatie als volgt gebeurt:

```
var form = new CustomPost.LiveForm(options);

// Event handlers toevoegen

form.init();
```

Optie	Standaard	Beschrijving
`'container'`	`'#entity-search-form'`	DOM element of selector voor `<form>` container.
`'els'`	*Zie Templates*	Bepaalt welke templates moeten worden bijgewerkt in welke containers.
`'pageLinkEls'`	`'.pagination a'`	Selector voor pagination links.

Templates

Voor het bijwerken van de pagina worden de benodigde templates op de server gerendered en wordt de parent DOM vervangen met de nieuwe content. Via de optie `'els'` kan worden opgegeven welke templates moeten worden bijgewerkt en de selector van de parent element, met een Javacript object met als key de naam van de template en als value de selector:

```
{
   search: '#entity-search',
   loop: '#entity-results'
}
```

Vanuit wonen/archive.php moeten deze templates dus als volgt worden gebruikt, waarbij de wrapper alleen de template mag bevatten:

```
<div id="entity-search">
   <?= Wonen::template('search'); ?>
</div>
```

Er kunnen indien nodig dus nog meer templates worden ingeladen en vervangen, door ze in `'els'` op te geven. De template met zoekvelden wordt bijgewerkt om zo het aantal resultaten per optie bij te werken.

Events

Om bepaalde acties uit te voeren wanneer specifieke events optreden kunnen event listeners worden toegevoegd. Zo moeten bijvoorbeeld Javascript widgets welke van toepassing zijn op elementen in de te vervangen templates iedere keer opnieuw worden geinitialiseerd, omdat de volledige DOM structuur vervangen wordt.

Event	Argumenten	Beschrijving
`'load'`	`data|null`	Uitgevoerd bij initialisatie en nadat bijwerken is voltooid.
`'updated'`	`data`	Uitgevoerd nadat bijwerken is voltooid.
`'refresh'`		Uitgevoerd wanneer een refresh wordt gestart.
`'before:replace'`	`data`	Uitgevoerd net voordat de templates worden ingevoegd in hun DOM parents.
`'after:replace'`	`data`	Uitgevoerd direct nadat de templates zijn ingevoegd.
`'after:append'`		
`'request:data'`	`data`, `request`	Geeft de mogelijkheid om extra data aan de request toe te voegen door `data` argument aan te passen.
`'change:page'`		Uitgevoerd wanneer van pagina wordt gewisseld.

De API voor het gebruik van events is als volgt:

```
form.on('event', function(e, ...) {
    // ...
});

form.off('event');
```

DOM Event delegation

We raden aan om DOM event listeners aan document toe te voegen en vervolgens te filteren op het gewenste element. Hierdoor hoeft de listener maar één keer worden toegevoegd en niet bij iedere 'load' op de vernieuwde elementen:

```
$(document).on('click', 'a', function(e) {
    // ...
});
```

Pagination

Links om van pagina te wisselen moeten ook worden onderschept om een pagina reload te voorkomen. Met de optie `'pageLinkEls'` kan de selector voor pagination links worden opgegeven, waarbij de plugin ook pagina wijzigingen via AJAX in zal laden. Ook wordt het event `'change:page'` gegenereerd om een actie te ondernemen bij het wijzigen van de pagina.

Infinite scrollen

In plaats van pagination is het mogelijk om meer resultaten achteraf in te voegen, mogelijk automatisch wanneer de gebruiker het einde van de resultaten bereikt. Voeg hiertoe alleen een link in naar de volgende pagina, met de class `infinite-results` of een van de parents moet deze class toegekend krijgen. Extra woningen zullen vervolgens achteraan in `options.infinite.appendTo` (met standaardwaarde `#entity-items`) worden ingevoegd, waarbij per woning de events `before:append` en `after:append` worden uitgevoerd met het DOM element van de woning als extra parameter. Vanuit deze events kunnen bijvoorbeeld animaties worden opgegeven of overige widgets worden geinitialiseerd. Alle overige events zijn als volgt:

Event	Argumenten	Beschrijving
`'before:append'`	`item`	Uitgevoerd net voordat een woning wordt toegevoegd in de DOM.
`'after:append'`	`item`	Uitgevoerd direct nadat een woning is ingevoegd.
`'appended'`	`data`	Uitgevoerd direct nadat alle woningen zijn ingevoegd.
`'infinite:end'`	`data`	Uitgevoerd wanneer alle resultaten zijn ingeladen.
`'append:page'`		Uitgevoerd wanneer meer resultaten worden opgehaald.

De events `before:replace`, `after:replace`, `change:page` en `updated` komen te vervallen en verder zal het `load` event alleen bij initialisatie worden uitgevoerd en niet meer voor nieuwe resultaten.

> Als voor een woning ongeldige HTML wordt gegenereert zal er geen DOM element aangemaakt kunnen worden, met als gevolg dat er vanuit *liveform.js* een fout optreedt. Ga in dat geval na of alle elementen wel worden gesloten en of dit in de correcte volgorde gebeurt.

Voor ieder resultaat zal de template `item` worden gerendered en worden ingevoegd in de DOM. De template per woning kan worden aangepast door `options.infinite.template` te wijzigen in het gewenste template. Om infinite scrollen te bereiken moet een klik-event op de pagination link handmatig worden uitgevoerd, zodat automatisch de juiste pagina aan extra resultaten geladen wordt. De

pagination link moet verplicht zijn opgenomen in de DOM maar kan uiteraard verborgen worden indien gewenst. Voorbeeldcode om dit te bereiken is als volgt:

```
var scrolled = false, $window = $(window);
$window.scroll(function() { scrolled = true; });

setInterval(function() {
    if (scrolled && !form.isLoading() && $window.scrollTop() + $window.height() >= form.$el.offset().top +
form.$el.height()) {
        $('.infinite-results a, a.infinite-results').click();
    }
    scrolled = false;
}, 100);
```

Laad indicatie

Tijdens het wachten op resultaten van de server krijgt de container `<form>` de class `search-loading` toegewezen. Vanuit CSS kan dit bijvoorbeeld worden gebruikt om laad indicaties op het juiste moment zichtbaar te maken.

Widgets

Er zijn standaard een aantal widgets beschikbaar om de functionaliteit van zoekvelden uit te breiden.

Sliders

Om een slider weer te geven worden de opties van een dropdown gebruikt, er wordt een jQuery slider widget toegevoegd in de DOM welke in sync blijft met de dropdown. Hierdoor is het eenvoudig om de dropdown te blijven tonen voor bijv. mobiele devices en de slider alleen zichtbaar te maken voor desktops, aan de hand van responsive CSS rules. Voor gebruik van deze widget, voeg `custompost-slider` toe als script dependency en initialiseer de widget als volgt:

```
new CustomPost.Slider(options);
```

Optie	Standaard	Beschrijving
`'container'`	*Verplicht*	DOM element of selector voor label container.
`'type'`	`'min'`	Bepaalt het type slider, `'min'` of `'max'`.

Optie	Standaard	Beschrijving
`'select'`	`'select'`	DOM element of selector voor gekoppelde dropdown.
`'emptyLabel'`	`null`	Label wanneer geen minimum/maximum is ingesteld, standaard bepaald a.d.h.v. lege optie in gekoppelde dropdown.
`'optionLabelAttribute'`	`null`	Biedt de mogelijkheid om HTML label te gebruiken zoals opgegeven in data attribuut van de `<option>`. Standaard wordt simpelweg de optie's label gebruikt.
`'sliderOptions'`	`{}`	Geef extra opties op voor de jQuery slider widget.

Bij initialisatie wordt een extra wrapper toegevoegd aan de opgegeven container, met de volgende opmaak:

```
<div class="slider-container">
    <div class="slider-values">
        <div class="value low|high"></div>
    </div>
    <div class="slider"></div>
</div>
```

De class van `div.value` is `low` wanneer `'type' = 'min'`, voor `'type' = 'max'` wordt de class `high` gebruikt. Hierin wordt de huidige waarde van het veld weergegeven. Tijdens verplaatsen van de handle krijgt de container de class `tracking` toegewezen, en `tracking-low`/`tracking-high` voor respectievelijk `min`/`max` types.

Range sliders

Naast sliders met een enkele handle kan ook een dubbele handle worden gebruikt, om zowel een minimum als maximum op te geven. Range sliders moeten los van sliders worden opgegeven met `custompost-rangeslider` als dependency.

```
new CustomPost.RangeSlider(options);
```

Optie	Standaard	Beschrijving
`'container'`	*Verplicht*	DOM element of selector voor label container.
`'minSelect'`	`'select[id$="-min"]'`	DOM element of selector voor gekoppelde dropdown voor minimum waarde.

Optie	Standaard	Beschrijving
'maxSelect'	'select[id$="-max"]'	DOM element of selector voor gekoppelde dropdown voor maximum waarde.
'minLabel'	null	Label wanneer geen minimum is ingesteld, standaard bepaald a.d.h.v. lege optie in gekoppelde dropdown.
'maxLabel'	null	Label wanneer geen maximum is ingesteld, standaard bepaald a.d.h.v. lege optie in gekoppelde dropdown.
'optionLabelAttribute'	null	Biedt de mogelijkheid om HTML label te gebruiken zoals opgegeven in data attribuut van de <option>. Standaard wordt simpelweg de optie's label gebruikt.
'sliderOptions'	{}	Geef extra opties op voor de jQuery slider widget.

Nu zullen zowel div.value.low als div.value.high beschikbaar zijn, de eerste toont het label van de minimum waarde en de laatste bevat het label van de maximum waarde.

Show more

Deze widget is bedoeld in combinatie met een lijst van checkboxes/radios, om in eerste instantie slechts een beperkt aantal opties te tonen en de resterende opties pas later weer te geven.

De hoeveelheid altijd zichtbare opties kan worden ingesteld op een vast aantal, of afgeleid worden van de alfabetische volgorde van de opties. Zo kunnen primaire opties op alfabet voor secundaire opties worden geplaatst, waardoor automatisch de grens tussen primair en secundair bepaald kan worden.

De widget wordt ingeladen door custompost-showmore op te geven als script dependency, hoeft verder alleen te worden geinitialiseerd:

```
new CustomPost.ShowMore(options);
```

Optie	Standaard	Beschrijving
'container'	*Verplicht*	jQuery element of DOM selector voor label container.
'name'	*Verplicht*	Unieke naam van de lijst.
'moreText'	'More...'	Tekst welke wordt weergegeven als niet alle opties worden weergegeven.
'lessText'	'Less'	Tekst welke wordt weergegeven als wel alle opties worden weergegeven.

Optie	Standaard	Beschrijving
`'defaultAmount'`	`0`	Het aantal te tonen opties als primair, of `0` om af te leiden van de opties.

De naam is verplicht in verband met het onthouden van de state, waarop wordt teruggevallen bij herinitialisatie tijdens live bijwerken zodat de widget in de juiste state hersteld wordt.

Wat betreft HTML structuur wordt een lijst van labels verwacht, genest in een container. Bij initialisatie wordt een `<a class="show-more-toggle">` element toegevoegd aan het einde van de container, waarvan de click events de secundaire opties tonen/verbergen. Als alleen de primaire opties worden weergegeven wordt de tekst `options.moreText` weergegeven en heeft dit element de class `less`, bij tonen van de secundaire opties verandert de tekst in `options.lessText` en de class in `more`.

Google Maps

Vanuit de plugin is standaard ondersteuning voor het tonen van interactieve Google Maps widgets. Hiervoor moet vanuit de admin worden ingesteld dat GPS coördinaten worden gedownload tijdens het bijwerken van woningen. De Google Map widget is in te laden door `"crm"-googlemap` toe te voegen als script dependency. Voor gebruikers van het Skarabee CRM gebruik je de dependency `skarabee-googlemap`. Verder is de code beschikbaar in het standaard thema *js/googlemap.js* en kan daardoor naar wens worden aangepast.

Maps Javascript API key

Zoals je in het Admin gedeelte van de documentatie hebt kunnen lezen is het hebben van verschillende API key's noodzakelijk voor het gebruik van een Google Map. Je hebt daar gelezen dat een Geocoding API nodig is voor het ophalen van coördinaten en dat een Maps Javascript API nodig is voor het daadwerkelijk tonen van een Google Map. Deze keys kunnen niet gecombineerd worden omdat je voor beide key's een andere restrictie in moet stellen.

Key	Applicatie restrictie
Maps Javascript	HTTP verwijzingen (websites)
Geocoding	IP-adressen (webservers, cron jobs, enzovoort)

> Als je de Geocoding API alleen gebruikt in onze plugin is er geen Applicatie restrictie nodig, deze key is namelijk niet publiekelijk te bereiken.

Je hebt al gelezen hoe je de Geocoding API key moet verwerken in de plugin, maar het verwerken van de Maps Javascript API gebeurt binnen het thema en niet in het dashboard van de plugin.

In het standaard thema vind je in de javascript bestanden ook `googlemaps.js` dit is een erg uitgebreid script waarmee we het tonen van een Google map kunnen bewerkstelligen, hierbinnen moeten we de Maps Javascript key verwerken. Ga daarvoor naar de `load` functie:

```
load: function() {
    if (window.realworksInitMap) return;

    window.realworksInitMap = $.proxy(this.init, this);

    if (typeof google === 'undefined' || typeof google.maps === 'undefined') {
        var script = document.createElement('script');
        script.type = 'text/javascript';
        script.src = '//maps.googleapis.com/maps/api/js?callback=realworksInitMap';
        document.body.appendChild(script);
    } else {
        this.init();
    }
},
```

Op lijn 9 in de bovenstaande code zie je de verwijzing naar de source van het Google map script, dit is het script die tegenwoordig de Maps Javascript API key verwacht. Verwerk je key daarom als volgt in de source url:

```
script.src = '//maps.googleapis.com/maps/api/js?key=INSERT_KEY_HERE&callback=realworksInitMap';
```

De Maps Javascript API key is ook te beheren vanuit de admin van onze plugin, volg hiervoor deze stappen.

Resultaten pagina

In de template *wonen/archive.php* kan het volgende worden gebruik om vanuit Javascript de benodigde informatie in `Wonen.maps` beschikbaar te hebben.

```
<?php wp_localize_script('wonen-archive', 'Wonen', array(
    'maps' => Wonen::maps('all'),
)); ?>
```

Voor het weergeven van een Google kaart bij de zoekresultaten kunnen twee opties worden gekozen, namelijk alle resultaten tegelijk weergeven (zoals in bovenstaande voorbeeld door `'all'` op

te geven) of alleen de woningen op de huidige pagina tonen, waarbij `'all'` moet worden gewijzigd in `'paged'`.

Mocht je gebruik willen maken van een Google Map op de resultaten pagina dan zal je de verdere functionaliteiten hier zelf voor moeten ontwikkelen. Met de bovenstaande informatie kan je in iedergeval de woning data, waaronder de coordinaten beschikbaar maken op de pagina.

Enkel object

Voor de template *wonen/single.php* is de informatie beschikbaar via `$woning->map()`, in te voegen in de template (nadat de loop is gestart, zodat `$woning` gedefinieerd is) door middel van:

```
<?php wp_localize_script('wonen-single', 'Wonen', array(
    'map' => $woning->map(),
)); ?>
```

Om de kaart daadwerkelijk weer te geven moet deze worden geladen:

```
if (Wonen.map) {
    var map = new SingleMap({ data: Wonen.map });

    map.load();
}
```

Omdat de locatie van een woning niet bekend kan zijn, wordt gecontroleerd of de data wel beschikbaar is. Vanuit de template kan `$woning->hasLocation` worden gebruikt om dit te bepalen.

> Sommige thema’s tonen de kaart in een tabblad. Het is in zulke gevallen aangeraden om het laden van de kaart uit te stellen totdat het tabblad geactiveerd wordt, om het laden van de pagina te versnellen.

Woningen in de buurt

Ook zoeken naar woningen in de buurt van de huidige woning kan gemakkelijk worden bereikt:

```
<?php wp_localize_script('wonen-single', 'Wonen', array(
    'nearby' => $woning->nearby('10km')->all()->maps(),
)); ?>
```

De huidige woning is hier vervolgens bij inbegrepen, deze kan worden genegeerd door `all()->without($woning)` te gebruiken.

>

> In bovenstaande voorbeeld wordt `'10km'` gebruikt om duidelijk te maken dat het om een afstand gaat, echter wordt de eenheid "km" genegeerd.

Info vensters

Info vensters worden gerendered op de server en de template ervan is aanpasbaar in `wonen/map-info.php`

Geavanceerde instellingen

Om het gebruik van de Maps Javascript API vanuit de admin van onze plugin te beheren kunnen we de geavanceerde instellingen inzetten. Standaard ziet dat er als volgt uit en is er alleen ruimte voor een Geocoding API:

```
{
"updater": {
"sources": [
{
"user": "TestMakelaar",
"password": "rwauth abcd-1234-zyxw-9876-voorbeeld",
"offices": "",
"entities": [
"wonen"
]
}
],
"attempts": 5,
"schedule": {
"interval": "daily",
"time": "09:30",
"earliest": "08:30",
"latest": "23:59"
},
"notify": {
"enabled": true,
"email": "info@tussendoor.nl"
},
"executor": {
"memory-limit": "1024M"
}
},
```

```
"maps": {
"api-key": ""
}
}
```

Het gedeelte van de geavanceerde instellingen van onze plugin is behoorlijk flexibel opgezet waardoor we het maps gedeelte naar wens kunnen uitbreiden, zo kunnen we op onderstaande manier ook een Maps Javascript API key toevoegen:

```
"maps": {
"api-key": "GEOCODING_KEY_HERE",
    "javascript-key": "MAPS_JAVASCRIPT_KEY_HERE"
}
```

Vervolgens kunnen we vanuit het thema de Maps Javascript API key als volgt aanroepen, de classname is in onderstaande voorbeeld Eyemove. Dit is afhankelijk van het CRM dat je gebruikt:

```
Eyemove::config('maps.javascript-key')
```

Via localize_script op de archief- en detail-pagina van de objecten kunnen we vervolgens de Maps Javascript doorsturen naar de googlemaps.js. Bijvoorbeeld voor de archief pagina van woningen:

```
<?php wp_localize_script('wonen-archive', 'Wonen', array(
    'maps' => Wonen::maps('all'),
    'mapsApiKey' => Eyemove::config('maps.javascript-key'),
)); ?>
```

De key kunnen we binnen googlemap.js als volgt ophalen:

```
script.src = '//maps.googleapis.com/maps/api/js?key=' + Wonen.mapsApiKey + 'callback=realworksInitMap';
```

Revisie #63
Aangemaakt: 13 november 2020 15:54:03 door Tussendoor
Bijgewerkt: 16 december 2025 08:35:38 door Tussendoor