**Working with views** <https://drupal.org/documentation/modules/views>

The views module allows administrators and site designers to create, manage, and display lists of content. Each list managed by the views module is known as a "view", and the output of a view is known as a "display". Displays are provided in either block or page form, and a single view may have multiple displays. Optional navigation aids, including a system path and menu item, can be set for each page-based display of a view. By default, views may be created that list content (a *Node* view type), content revisions (a *Node revisions* view type) or users (a *User* view type). A view may be restricted to members of specific user roles, and may be added, edited or deleted at the views administration page.

For more technical users, views can be understood as a user interface to compose SQL-queries, pulling information (Content, Users, etc.) from the database and showing it on screen in the desired format.

The "building block" design of the views system provides power and flexibility, allowing parameters to be specified only when needed. While an advanced view may use all of the available parameters to create complex and highly interactive applications, a simple content listing may specify only a few options. All views rely on a conceptual framework that includes:

**Fields**

Fields, or the individual pieces of data being displayed. Adding the fields *Node: Title*, *Node: Type*, and *Node: Post date* to a node view, for example, includes the title, content type and creation date in the displayed results

**Relationships**

Relationships, or information about how data elements relate to one another. If relationship data is available, like that provided by a CCK *nodereference* field, items from a related node may be included in the view

**Arguments(D6)/Contextual filters(D7)**

Arguments, or additional parameters that dynamically refine the view results, passed as part of the path. Adding an argument of *Node: Type* to a node view with a path of "content", for example, dynamically filters the displayed items by content type. In this example (shown with Clean URLs enabled), accessing the view through the path "*http://www.example.com/content/page*" displays all posts of the type "page", the path "*http://www.example.com/content/story*" displays all posts of the type "story", and "*http://www.example.com/content*" displays all posts regardless of type)

**Sort criteria**

Sort criteria, which determine the order of items displayed in the view results. Adding the sort criteria *Node: Post date* (in descending order) to a node *view*, for example, sorts the displayed posts in descending order by creation date

**Filters**

Filters, which limit items displayed in the results. Adding the filter *Node: Published* (and setting it equal to "Published") to a node view, for example, prevents unpublished items from being displayed

**Displays**

Displays, which control where the output will be seen. Every view has a default display, which doesn't actually display the view anywhere but is used to hold the default settings for the view. This default display is also used when the view is called programmatically without specifying another display. Much more useful to users are the page display, which gives a view a path and allows it to be the primary content of a page, or the block display which allows it to appear as secondary content on other pages.

**Header**

Header, which allow you to add by default one or more text area above the views output.

**Footer**

Footer, which allow you to add by default one or more text area beneath the views output.

**Empty Text**

The Empty Text content will be displayed, when you choose in the Arguments Section "Action to take if argument is not present" the option "Display empty text".

Parallels between the components in the Views interface and an SQL query:

| **SQL Query** | **Views Component** |
| --- | --- |
|  |  |
| SELECT n.title, u.name | fields |
| FROM {node} n base table | view type |
| INNER JOIN {users} u ON n.uid = u.uid | relationship |
| WHERE n.status = 1 | filter |
| AND u.uid = arg(1) | argument |
| ORDER BY n.changed DESC | sort |

# Understanding Views versus custom code

This page discusses the advantages and disadvantages of using the Views module versus custom coding your queries and presentation.

## Custom code advantages

Having custom code offers the following advantages:

* Complete control over the data fetched. Some SQL queries are difficult to replicate in Views.
* Some opportunities for query optimization might exist when writing custom code. Views is as efficient as a general tool can be, but the specific details of your target query might mean that further optimization is possible.
* Complete control over presentation. You can specify all the presentation details for your data.
* Complete knowledge of your code. If you write it yourself you know what it contains. Compare this to the task of studying the Views codebase.
* Simplicity. In many cases all you need is a menu, a callback and a simple theme function to render the fetched data. In less then 30 lines of PHP you can fetch and display a list of nodes. Compare that to the huge amount of views-code, including plugins and interfaces to manage a similar list.
* Revision control systems like CVS, SVN. If your view is stored in the database you risk losing it, if the database becomes corrupt. Note that you can [export views to PHP code and put them in custom modules](http://drupal.org/node/272912), though, so this one equals out.

## Views advantages

Using the Views module offers the following advantages:

* Change behavior without changing code. You can alter your application at runtime without worrying about breaking code or making a PHP syntax error. (Note: since views that are stored in the database do not have revision control, changes made like this cannot simply be undone... be warned)
* Many options for presentation. Different view types mean that you can change the presentation of the data you fetch.
* Black box. You don't need to write or read any PHP code to get started with Views.
* Data safety. Malicious users can do a lot of damage without proper data safety testing. Views goes through the trouble to scrub all data it presents, so you can feel assured that as long as the module providing the data properly implements the Views API, its data will be safe.
* Open architecture. If you are a developer, you can write to the Views API and add features, presentation styles and behaviors to Views in custom modules. These are then reusable in other views that you later create.
* Community benefit. You will be the recipient of an enormous amount of Views code that is being contributed and improved by the community. There will be Views features available tomorrow that weren't here today. There will be people fixing Views bugs while you sleep. This won't be the case for your custom code.
* Reuse views. Build a basic view that does something useful, export it, and then use it on the next site you build.
* Tight integration with other modules that you might already be using. Every contrib module can potentially extend Views, so make sure and take inventory of what modules you plan to use when evaluating Views on its own.
* Panels integration. Panels is especially powerful and the Views integration is very good.
* Programmatic handling of views. You can instantiate Views objects in PHP code and manipulate them in many ways. The Views API provides several points of access to the Views object, so if you are really unhappy with what you can achieve with the Views UI, you can take control of the object itself.
* Documentation. There is more documentation about Views than there is about your custom code.
* Page, menu and block integration. Views gives your page views and block views, and lets you instantiate menu items as well.
* Exposed filters: Views gives you a lot of control over creating form elements to filter your view. Writing FAPI code is bulky and verbose. Creating all those dropdown selects, autocomplete text fields and checkboxes is a lot of work. You can still use form alter to get to the exposed filters submission and validation handlers, so you don't lose any control by going with Views here.

Once you've created a view, you can customize it further - you can control the view's

* title
* format (output style) - choose from
* jump menu
* grid
* HTML list
* table
* unformatted
* rss
* row style
* fields
* full nodes
* teasers
* RSS feed items
* fields
* filters (selecting which content will appear in the view)
* sorting
* header
* footer
* pager

In addition, there are advanced options to let you control contextual filters, relationships between content types, caching, and theme templates.

# Query Settings

Some Views may contain duplicate results. There is an option that can help remove duplicates. Open upu the "Advanced" section of the view's administration page and in the "Other" section are "Query Settings".

Here there is a checkbox for "Distinct" and another one that appears if it is checked, "Pure Distinct". It may not be clear which to use, but for a new view, it is best to check both options. Some sites that have upgraded from a previous version of Views may have problems with pure distinct, so this option is disabled by default.

# Format

The **Format** setting controls the overall format for the list of information in your view.

By default, you can choose from these formats:

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **What it does** | **HTML tags** | **Settings** |
| Grid | shows the item in a grid with two or more items across | <table><tr><td> | number of columns in the grid; whether to fill the grid from left to right (horizontally) or top to bottom (vertically) |
| HTML list | uses HTML tags to create a numbered or bulleted list | <ul><li> or <ol><li> | ordered (numbered) or unordered (bullet) list |
| Jump menu | creates a drop-down list with an optional Go button to let users jump to a particular item | <form><select> | button text for the Go button; label for the selector; text for the pre-selected option in the selector |
| Table | shows the items in a table, with a separate column for each field you want to display | <table><tr><td> | for each field in the table, you can select which column will hold it (so you can combine fields into one column), how to align text in the column, the separator between items in the column, whether the column is sortable, whether it's the default sort column, and whether to hide the column if it's empty |
| Unformatted list | displays unformatted divs | <div> | (none) |

You may have additional options if you've installed other modules, such as [Views Slideshow](https://drupal.org/project/views_slideshow), [Calendar](https://drupal.org/project/calendar), or [GMap (Google Maps)](https://drupal.org/project/gmap) - to name just a few.

In addition to the Settings options shown above, most formats let you specify whether to add default CSS classes to each row of output (giving you extensive control over CSS theming for each row) and whether to add striping CSS classes (to make it easier to add striping or special CSS for the first and last rows).

## What's the difference between Grid and Table?

A **grid** combines an entire row in each cell, like this:

|  |  |  |  |
| --- | --- | --- | --- |
| Row 1 Oslo museum visit | Row 2 Singapore concert | Row 3 Chicago walking tour | Row 4 Montevideo parade |
| Row 5 Budapest concert | Row 6 London lecture | Row 7 Dakar park visit | Row 8 Dublin gallery opening |
| Row 9 Brisbane park visit | Row 10 San Francisco museum visit | Row 11 Victoria movie | Row 12 Osaka walking tour |

A **table** separates fields into columns, like this:

|  |  |  |
| --- | --- | --- |
| **Title** | **City** | **Event** |
| Row 1 | Oslo | museum visit |
| Row 2 | Singapore | concert |
| Row 3 | Chicago | walking tour |
| Row 4 | Montevideo | parade |
| Row 5 | Budapest | concert |
| Row 6 | London | lecture |
| Row 7 | Dakar | park visit |
| Row 8 | Dublin | gallery opening |
| Row 9 | Brisbane | park visit |
| Row 10 | San Francisco | museum visit |
| Row 11 | Victoria | movie |
| Row 12 | Osaka | walking tour |

# Grid (output format)

The **grid** style displays each row of your view within a grid. You can customize the **number of columns**; the default is 4 columns.

You can also choose **horizontal** or **vertical** alignment for the grid.

* **Horizontal alignment** displays items starting in the upper left and moving right.
* **Vertical alignment** displays items starting in the upper left and moving down.

A grid looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| row 1 | row 2 | row 3 | row 4 |
| row 5 | row 6 | row 7 | row 8 |
| row 9 | row 10 | row 11 | row 12 |
| row 13 | row 14 | row 15 | row 16 |

The above uses the **horizontal** alignment, where rows are added into the grid from left to right.

With a **vertical** alignment, rows will be placed from top to bottom, like this (your **row** results will appear as columns):

|  |  |  |  |
| --- | --- | --- | --- |
| row 1 | row 5 | row 9 | row 13 |
| row 2 | row 6 | row 10 | row 14 |
| row 3 | row 7 | row 11 | row 15 |
| row 4 | row 8 | row 12 | row 16 |

You can also choose to **group a field** from the Fields Section. This grouping field will be displayed as a header, and all rows will be displayed beneath it.

This style uses a row style to determine what each row will look like.

The [Format page](https://drupal.org/node/2288017) explains the difference between Grid format and Table format.

# HTML list (output format)

The **HTML list** format displays each row of the view as part of an HTML list. For example:

* Row 1
* Row 2
* Row 3
* Row 4

Click a **List type** option to specify which type of list you want:

* Select **Unordered list** to display the list using bullets
* Select **Ordered list** to display the list using numbers

# Theming ajaxified view/view block

Views module provides powerful end flexible tools to create custom template files for each element, but as it is using jQuery in few places, it is necessary to keep template structure in order, to make sure all JS and Ajax facilities will work correctly. This document describes relation between markup of Views and jQuery code which is attached to view which is using Ajax.

## Required class for view container view-dom-id-{name/md5} - why and what for?

view-dom-id-{name/md5} - provided to our template with $classes variable is a main selector inside of ajax\_view.js, which means that is required for views module to run ajax properly:

/\*\*

\* Javascript object for a certain view.

\*/

var selector = '.view-dom-id-' + settings.view\_dom\_id;

this.$view = $(selector);

The reason why this class sometimes contains weird md5 hash instead of view name is explained in [views\_block\_info()](https://api.drupal.org/api/views/views.module/function/views_block_info/7) function.

block.module has a delta length limit of 32, but our deltas can unfortunately be longer because view names can be 32 and display IDs can also be 32. So for very long deltas, change to md5 hashes.

And here’s PHP code responsible for it:

foreach ($keys as $delta) {

if (strlen($delta) >= 32) {

$hash = md5($delta);

$hashes[$hash] = $delta;

$items[$hash] = $items[$delta];

unset($items[$delta]);

}

}

In conclusion: **always output $classes variable to make sure ajax will init properly!**

## Markup for views exposed filters

When view will be successfully targeted via DOM Id, it will attempt to find exposed filters using jQuery children(); method, and standard view classes:

/\*\*

\* Add the ajax to exposed forms.

\*/

this.$exposed\_form = this.$view.children('.view-filters').children('form');

this.$exposed\_form.once(jQuery.proxy(this.attachExposedFormAjax, this));

This forces us to keep nesting of this element in one proper way only,  
**.view-filters** element must be a children of **.view-dom-id-{name/md5}** element, and the must be rendered inside of **.view-filters** element. Next thing is submit button - it just needs to be placed anywhere inside of element:

Drupal.views.ajaxView.prototype.attachExposedFormAjax = function() {

var button = $('input[type=submit], button[type=submit], input[type=image]', this.$exposed\_form);

button = button[0];

this.exposedFormAjax = new Drupal.ajax($(button).attr('id'), button, this.element\_settings);

};

## Pagination

/\*\*

\* Attach the ajax behavior to each link.

\*/

Drupal.views.ajaxView.prototype.attachPagerAjax = function() {

this.$view.find('ul.pager > li > a, th.views-field a, .attachment .views-summary a')

.each(jQuery.proxy(this.attachPagerLinkAjax, this));

};

Requirement for pagination is much simpler, because it is targeted with jQuery find(); method, so it just needs to be anywhere inside of **.view-dom-id-{name/md5}** wrapper, **ul** must have **.pager** class,**li** must be a child of **ul**, **a** must be a child of **li**. That’s all.

## Template/markup

To make sure Ajax will work properly we don’t need to output entire view markup, this is all we need:

<div class="<?php print $classes; ?>">

<div class="views-filters">

<?php

// our exposed form must be wrapped with .views-filters class

print $exposed;

?>

</div>

<div class="view-content">

View content goes here, or anywhere.

</div>

<?php

// pager can be placed anywhere inside of main view wrapper

print $pager;

?>

</div>

And this is sample of our HTML output:

<div class="views-dom-id-{this-is-view-dom-id}">

<div class="views-filters">

<form>

<input type="submit" />

</form>

</div>

<div class="view-content">

<p>VIEWS CONTENT</p>

</div>

<ul class="pager">

<li>

<a>Page 1</a>

</li>

</ul>

</div>

Hope I didn't missed anything and it will help a little to all Themers :\_)