Plone

Blocks: Grid based layouts

Martin Aspeli, Geir Bækholt, Alexander Limi, Laurence Rowe

29 October 2014

Plone

Executive Summary

Status quo

Plone currently relies primarily on Zope Page Templates and METAL macros to render pages. Typically, a view or template will invoke the “master” macro in *main_template* to pull in Plone’s layout, filling various slots to supply the body text, add Javascript or CSS resources, and sometimes override the right/left columns. To customise the layout, an integrator has to provide an alternative *main_template*. Unfortunately, main_template is complex and has much boilerplate, making customisations brittle. Furthermore, some layout information is contained in the individual views, resulting in a need to customise many templates, especially in cases where views do not follow established conventions.

To make it easier to change the look-and-feel of a Plone site without customising *main_template*, Plone 3 uses viewlets extensively. Viewlet managers with names such as “HTML head” and “footer” are included in *main_template*, allowing Plone and third party products to include snippets of HTML in the output by registering viewlets for those managers. Many customisation tasks can be achieved by adding or hiding viewlets. However, working with viewlets is cumbersome and requires an understanding of Zope 3 browser components and ZCML, and it is difficult to move viewlets freely around the page.

Finally, Plone does not currently have a good solution to allow content authors to create composite pages with complex layout. There are a number of composite page add-on products, but none provide a particularly compelling user experience.

Proposal

Plone will gain a new layout system, where re-usable *tiles* are placed upon a layout. Tiles can be used to render images, movie clips, polls, search widgets, folder listings and any number of other things. They are affixed either to a global layout, which is managed as a simple HTML document, or placed in the body text of content objects that support layout. Tiles can be simple snippets of HTML, or more complex, persistent components with individual settings.

This approach has a number of advantages:

- Site layout and composite page layout can be managed with HTML only, using rich through-the-web editors.
- Every content object will have a canonical “no layout” view that includes only the semantic HTML to render that object.
- Tiles will be easy to write and distribute, and existing viewlets and portlets can be made available as tiles.
- Tiles can be individually and aggressively cached, and will support for ESI (Edge Side Includes).

Plone

Implementation

This section describes how Blocks will be implemented.

Guiding principles

When we started thinking about this system, we had a few guiding principles in mind:

- HTML is good! You can represent a lot of information in plain HTML, and there are lots of great tools out there, from Dreamweaver to lxml, that work well with HTML. Properly structured (X)HTML is easy to parse, inspect and understand.
- Don't be too clever! Too much magic makes the system hard to understand. By keeping the system refreshingly dumb, it becomes easier to understand and manipulate.

Deco

Deco is the code name for the proposed next-generation Plone rich content editing tools and UI. It has a separate proposal with UI mock-ups and rationale behind the concepts. Deco refers to grids, panels, layouts and tiles from a UI perspective. It is probably useful to read the Deco proposal before reading this one.

Core concepts

Blocks is based on the following core concepts.

Grid

Deco will use a CSS grid framework such as *Blueprint* to manage visual page layout. This will break the page up into regions of designated proportions.

The grid will be composed of <div /> elements pulled both from the main layout and the page layout (see below). Note that Blocks is not concerned with CSS or visual look and feel.

Panel

A *panel* is a region within the grid that can contain a number of tiles, assigned through the web using a rich editing GUI. For example, a standard three column layout may have three central panels - left, middle and right - as well other panels covering the header and footer.

A panel is simply a <div /> with an appropriate HTML id. A layout can contain an arbitrary number of panels, panels can contain an arbitrary number of tiles, and tiles may be placed outside panels.

Each panel has a name. If the same panel is defined in a page layout and the current site layout, the panel in the page layout will override the panel in the site layout.

Tile

A *tile* is an item placed upon a layout, such as a search box widget, a form field, an image, an inline movie player, or indeed any other visual component. Tiles will often be placed within panels (especially for tiles placed by the user through a GUI), but may exist anywhere in the rendered layout.

The simplest tile is simply an HTML snippet or template and is referenced by name only. Some tiles will have associated persistent settings, which are configurable through the Plone GUI. In this case, there will be one persistent object containing these settings per instantiation of the tile.

Tiles are individually traversable and can be fully rendered independently of a layout simply by knowing its URI. When rendering a tile, the browser will typically see a full HTML document, including a <head /> and <body />. When the tile is inserted into the head or body of a page, only the relevant portions are included.

When a tile is placed onto a panel, this fact will be recorded using an <a /> tag with appropriate *rel* and *href* attributes. During page rendering, this will be replaced by the rendered tile.

In Plone, new types of user-insertable tiles are installed with associated metadata, such as an “add” permission (e.g. some tiles can only be added by the administrator), a name and a title. Once installed, a sufficiently privileged user will be able to insert the tile into any layout using the Deco GUI.

Site layout

The overall layout of the page is stored as plain HTML, with panels defined by appropriate <div /> tags. The administrator will be able to assign tiles to the global layout through the Deco GUI.

A number of layouts may be available, and new layouts may be installed through add-on products, or even created through the web. (As an implementation detail, layouts are registered as named utilities.)

At any point in time, one layout is chosen as the default, although a view will be able to signal that it requires a different layout (via an adapter). This may for example be used to render a “fullscreen” view of images that omits most layout.

A new default layout may be configured in any “component site”. This allows a different layout (with different tiles) to be created in a particular folder, for example.

Page layout

Views of content objects may provide their own panels. This is simply a matter of providing the appropriate <div /> elements in their output.

Content types that opt into “layout editing mode” will need to provide a field into which the page layout is stored. As with the site layout, tiles are placed into panels using a rich GUI, but stored as simple <a /> tags.

The layout editor provides the facility for users to choose an appropriate layout template, and can save a defined layout as a new template. (As an implementation detail, templates are registered as named local utilities.)

(Note that even if a content type does not use the rich layout editor, it can still render panel <div /> elements in its view.)

Existing components

The following components are currently used in Plone. Blocks will still use most of these, but may combine them in different ways.

Views

When publishing an object at a given URL, Zope will usually discover a *view* (which is either named in the URL or looked up as a default view for a given context) and call it. The result is normally a string of HTML.

With Blocks, the views of content objects will typically be standalone HTML pages that contain semantic HTML describing the object. They will not concern themselves with site layout or site-wide tiles (e.g. the search box, or the footer), although they may contain panels and tiles defined in the page layout.

Macros

Zope Page Templates use *macros* to allow template authors to share page elements. In Plone, most views invoke the *master* macro from *main_template*. This renders most of Plone's UI, and includes *slots* for the body text, portlet columns, HTML head and so on. Views will fill these slots as appropriate.

Blocks will eschew the use of macros for layout. Macros may still be useful where views want to share template logic, but this will be an implementation detail only. Furthermore, Blocks will not be tied to ZPT. Views and tiles will be free to use whatever templating language is most appropriate.

Content providers

A *content provider* is a basic view component described by the *IContentProvider* interface from *zope.contentprovider*. This contains two methods - *update()* and *render()*, in order to support two-phase rendering and inter-provider communication, whereby all content providers are first updated before they are all rendered.

In reality, Zope cannot guarantee two-phased rendering of content provides. Since the use cases are limited and the potential performance overhead is tangible, Blocks will not attempt to support full two-phase rendering for tiles at this time.

Viewlets

A *viewlet manager* is a type of content provider that can discover and render viewlets. *Viewlets* are special content providers registered for a particular viewlet manager, using ZCML. The default viewlet manager implementation looks up viewlets in the component registry and then updates and renders each viewlet in turn.

Blocks will provide support for viewlet managers and their viewlets through special type of tile, which can be moved around the layout like any other. However, Blocks should make many of the default viewlet managers in Plone obsolete: the placement and configuration of things like the footer text, the search box, or the global tabs will be better achieved using simple tiles.

Viewlets will still be useful for use cases where a third party product needs to "plug in" a new snippet declaratively. For example, the Iterate product marks working copies with a particular marker interface, and registers a viewlet to display a notice to the user that he is editing a working copy for this interface only. Clearly, it would not be the administrator's responsibility to place a tile for this message in the layout, although he may be responsible for deciding where this type of information will appear. It is envisaged that viewlet managers will be be given semantic names, such as “info box”, rather than names based on where they appear (such as “above content title”).

Portlets

A *portlet manager* is analogous to a viewlet manager, except that it renders *portlets*. Portlets are similar to viewlets, but consist of two components: a persistent object that represents the *assignment* of a portlet to a particular context (e.g. a folder) and any settings associated with that portlet; and a *renderer* for that assignment.

Whereas viewlets are configured in the component registry by a developer, portlets are instantiated and configured through the web, much like content objects. The portlet manager uses a pluggable algorithm to discover which portlets to render at any given time.

In Blocks, portlets will be recast as tiles with persistent configuration. They can be placed on the site layout or attached to a page. When placed on the site layout, the assignment object will be stored as a named local utility. When attached to a page, the assignment object will be stored in an annotation. This is similar to, but simpler than, the current portlet storage infrastructure. Portlets written for Plone 3 should work as tiles with little or no migration required.

Blocks will do away with group portlets except on a designated dashboard page. Content-type portlets will be managed by providing standard tiles on default layout templates. This is in part to simplify the interaction model, and in part to encourage pages that cache well.

Tile addressing

All tiles must be individually addressable via a URI. This is in part to facilitate the rendering process, and in part to support ESI, where a cache server may compose a partially cached page by re-rendering one or more tiles individually. When published independently of the page layout, a tile should render to a valid HTML page.

Different tile implementations may use different URI naming schemes. However, the two most common types of tiles (in Plone, at least) will be transient tiles (similar to views or content providers) and persistent tiles (those that are tied to a particular persistent settings object, similar to portlets).

Transient tiles

A simple tile may be addressed as http://example.com/++tile++/mytile, where ‘mytile’ is the name of the content provider.

Here, the context is assumed to be the Plone site root (at http://www.example.com). If the tile needs to be context-aware, the context in which it is being rendered may need to be included in the URI before the ++tile++ namespace adapter. When saved to a layout, the tile may be stored with prefix that is relative to the context (e.g. *./++tile++/mytile*) or the Plone site root (/++tile++/mytile). A transform will turn this into an absolute URI (since the Plone site root may not be mounted on the root of the domain) early in the rendering process.

Persistent tiles

A tile with settings is addressed by including the name of the persistent object that contains the settings, e.g. http://www.example.com/++tile++navtree/. In this example, the default view of this tile would be rendered. If there was another view, it could be addressed as http://www.example.com/++tile++navtree/alternative-view.

Plone

Tile representation

Blocks will primarily deal with these two types of tiles, but does not preclude other representations.

Transient tiles

This type of tile is similar to a basic browser view, and is registered in the component registry as a (named) adapter factory.

Transient tiles can be created in several ways, e.g.:

- As a simple template on the filesystem. The developer will register a directory for tiles using a *<plone:tileDirectory />* ZCML directive. This can contain plain HTML, or Chameleon (or ZPT) templates. The filename will be used as the tile name.
- Through-the-web, using a dedicated GUI where a tile can be written as a TAL template. This results in a local adapter registration.
- With ZCML, using a *<plone:tile />* directive. This may reference a template and/or a class that implements IBrowserPage, and include some tile-specific metadata to allow the tile to be discovered by the Deco GUI.
- With Grok-like conventions, where a class is “grokked” for directives. This is analogous to registering the component with ZCML, but does not involve a separate configuration file.

Transient tiles may be assigned to site or page layouts freely. They will be instantiated on demand, much like any view or content provider.

A transient tile may need some configuration options. These could be passed as query string parameters and embedded in the tile link when it is saved into the page or site layout.

Persistent tiles

This type of tile is similar to a portlet. It is implemented as a traversable, persistent object that contains tile data, and a view of this object, which is looked up in the adapter registry much like a portlet renderer is now. This means that existing portlets should be re-usable as persistent tiles.

If the ++tile++ traversal namespace is given the id of a tile (e.g. .../++tile++mytile), it will look for a persistent tile with the given id. If the tile is being rendered in the context of the portal root (e.g. http://www.example.com/++tile++mytile), the tile settings with id ‘mytile’ will be looked up as a named local utility. This allows tiles to be assigned to (and overridden at) local component sites. If the context is a content object (e.g. http://www.example.com/some-context/++tile++mytile), then the tile settings with id ‘mytile’ will be retrieved from an annotation on the context.

Once the settings object has been located, traversal will continue either to the default tile view or a named view, if one is given in the URI. This will then be instantiated as a multi-adapter, much in the same way that portlet renderers are now.

Persistent tiles can be registered in two main ways:

- With ZCML specifying the name of a class that contains tile settings, and a class or template implementing the renderer. If the settings object contains a schema, a form will be auto-generated from this, though the form can be overridden by specifying a particular for implementation.
- Using Grok-like conventions instead of ZCML.

Persistent tiles must be instantiated and saved (either to a local utility in the case of a tile on the site layout, or an object annotation in the case of a page layout) before they can be placed into a layout. This will be managed using the layout mode GUI.

Tile type information

Both persistent and transient tiles may be instantiated using the layout mode GUI. In order to present the appropriate tiles to insert, this will query all utilities providing the *ITileInfo* interface. Such utilities will normally be installed locally into the site, using GenericSetup, and may contain:

- A title and description
- An “add” view that can be used to configure the tile and obtain the appropriate tile URI
- An “edit” view that can be used to configure the tile after it has been added
- Tile permissions to control which roles can insert the tile into the layout
- The types of layout (site or page) that the tile is applicable for

As far as possible, these will use sensible defaults. For example, it should be possible to register a tile type using only its name (for transient tiles) or class (for persistent ones) and a title.

Tile introspection

Tiles should support runtime introspection, including:

- The attribute of a tile rendering class that contains the tile’s template (normally ‘index’). This should make visual customisation more predictable.
- Where (in which package and files) the tile is registered.
- The name of the type information utility.

Page composition

So far, we have described how tiles are implemented by a developer and assigned to a layout by a user. Next, we must consider how tiles are actually rendered.

Blocks takes a multi-phased approach to rendering. In the default implementation, it is envisaged that all of these will take place within the Zope publisher, but for some deployments, it may be desirable to push some of these out into WSGI middle or an external cache server (ESI).

For the purposes of this description, let’s assume that a site visitor has requested the URI http://www.example.com/some-context/@@view.

Phase 1: Page layout

The first step is to render the page layout, independently of the site. This simply involves publishing the view, which should return a simple HTML page containing the semantic information for this content object. The page title should be contained within the *<title />* tag, and other metadata should be represented by *<meta />* tags.

If applicable, the page body may include a number or panel *<div />* elements with the appropriate ids and layout-specific classes. If the page uses tiles in its internal layout (whether hard-coded into a template or assigned by a content author using layout mode) they are rendered as link elements.

The following example shows a page that that contains two panels, *main* and *left*, each with one or more tiles, as well as some static body text. It has opted into layout mode and requested the current default layout:

```
<html>
  <head>
    <title>Some context</title>
    <meta name=”description” content=”This is the description of my page” />
    <script type=”text/javascript” src=”/portal_javascripts/my.js” />
    <link rel=”stylesheet” type=”text/css” href=”/portal_css/my.css” />
    <link rel=”layout” href=”/++layout++” />
  </head>
  <body>
    <div id=”panel-main”>
      <p>Welcome to my cool page.</p>
      <p>Below is an embedded movie attached to this page with a persistent tile</p>
      <a rel=”tile” href=”++tile++movie1/” />
      <p>And here is a folder listing using a transient tile and a relative URL</p>
      <a rel=”tile” href=”++tile++/folderlisting” />
    </div>
    <div id=”panel-left”>
      <p>I’m overriding the left column here, and inserting a global transient tile</p>
      <a rel=”tile” href=”/++tile++navtree/expanded”  />
    </div>
  </body>
</html>
```

Things to note:

- This is pure, semantic HTML
- The page include local and global, persistent and transient tiles in *<a />* tags.
- There are two panels - main and left. These both have a place in the site layout.
- The page names the site layout it would prefer to use via a *<link />* with *rel=”layout”*.

Phase 2: Site layout

Next, the site layout will be rendered. The page above has opted into the default (unnamed) layout. It could have specified another layout, e.g. with:

```
<link rel=”layout” href=”/++layout++alternate-one” />
```

(As an implementation detail, the layout to use would be looked up as a utility with the name “alternate-one”).

Let’s assume the site layout looks as follows:

```
<html>
  <head>
    <link rel=”tile” href=”/++tile++/mergedresources” />
    <!-- global resources and head elements go here -->
  </head>
  <body>
    <div id="panel-main">
      <!-- The main panel will replace this div -->
    </div>
    <div id="panel-left">
      <p>You are here:</p>
      <a rel="tile" href="/++tile++navtree"  />
    </div>
    <div id="panel-right">
      <p>Latest news:</p>
      <a rel="tile" href="/++tile++news/simple"  />
    </div>
  </body>
</html>
```

Things to note:

- This is a also pure HTML
- There are three panels: *main* (which is empty), *left* and *right*

The bodies of the two layouts are then merged, giving preference to panels in the page layout. The result of this phase may be:

```
<html>
  <head>
    <link rel="tile" href="/++tile++/mergedresources" />
    <!-- global resources and head elements go here -->
  </head>
  <body>
    <div id="panel-main">
      <p>Welcome to my cool page.</p>
      <p>Below is an embedded movie attached to this page with a persistent tile</p>
      <a rel="tile" href="++tile++movie1/" />
      <p>And here is a folder listing using a transient tile and a relative URL</p>
      <a rel="tile" href="++tile++/folderlisting" />
    </div>
    <div id="panel-left">
      <p>I'm overriding the left column here, and inserting a global transient tile</p>
      <a rel="tile" href="/++tile++navtree/expanded"  />
    </div>
    <div id="panel-right">
      <p>Latest news:</p>
      <a rel="tile" href="/++tile++news/simple"  />
    </div>
  </body>
</html>
```

Note that if no panels can be found the page layout, then it is assumed that the whole contents of the *<body />* tag should be placed inside the *main* panel. This makes it possible to re-use views not written with panels in mind.

Phase 3: Head merge

In this phase, the *<head />* of page is merged into the layout. This may include some transforms, for example to affix style sheets to the site root without requiring the view to prefix everything with the site root URL. Some types of tags, such as <title /> and <base />, will be replaced if they exist in both the page and site layout, with the page contents taking precedence. Other tags, such as <script /> or <link />, will be merged with page content appended to the site layout.

The output of this phase could be:

```
<html>
  <head>
    <link rel=”tile” href=”/++tile++/mergedresources” />
    <!-- global resources and head elements go here -->
    <title>Some context</title>
    <meta name=”description” content=”This is the description of my page” />
    <script type=”text/javascript” src=”http://www.example.com/portal_javascripts/cachekey/my.js” />
    <link type=”text/css” href=”http://www.example.com/portal_css/cachekey/my.css” />
  </head>
  <body>
    <div id=”panel-main”>
      <p>Welcome to my cool page.</p>
      <p>Below is an embedded movie attached to this page with a persistent tile</p>
      <a rel=”tile” href=”++tile++movie1/” />
      <p>And here is a folder listing using a transient tile and a relative URL</p>
      <a rel=”tile” href=”++tile++/folderlisting” />
    </div>
    <div id=”panel-left”>
      <p>I’m overriding the left column here, and inserting a global transient tile</p>
      <a rel=”tile” href=”/++tile++navtree/expanded”  />
    </div>
    <div id=”panel-right”>
      <p>Latest news:</p>
      <a rel=”tile” href=”/++tile++news/simple”  />
    </div>
  </body>
</html>
```

Phase 4: Tile render

In this phase, each tile is asked to render itself, before the *<a rel="tile" ... />* tag is replaced by the rendered tile, wrapped in a *<div class="tile" />* with a unique id (unless in the *<head />*). This makes it easier to uniquely identify tiles in the final rendered page, which may look like this:

```
<html>
  <head>
   <!-- Tile: mergedresources -->
    <link rel="styleshet" type="text/css" href="http://www.example.com/++resource++navtree.css" />
    <link rel="styleshet" type="text/css" href="http://www.example.com/++resource++news.css" />
    <script type="text/javascript" href="http://www.example.com/news.css" />
    <!-- global resources and head elements go here -->
    <title>Some context</title>
    <meta name="description" content="This is the description of my page" />
    <script type="text/javascript" src="http://www.example.com/portal_javascripts/cachekey/my.js" />
    <link type="text/css" href="http://www.example.com/portal_css/cachekey/my.css" />
  </head>
  <body>
    <div id="panel-main">
      <!-- The main panel will replace this div -->
    </div>
    <div id="panel-main">
      <p>Welcome to my cool page.</p>
      <p>Below is an embedded movie attached to this page with a persistent tile</p>
      <div class="tile" id="tile-local-movie-1">
        <embed ... />
      </div>
      <p>And here is a folder listing using a transient tile and a relative URL</p>
      <div class="tile" id="tile-local-folderlisting-1">
        <dl> ... </dl>
      </div>
    </div>
    <div id="panel-left">
      <p>I'm overriding the left column here, and inserting a global transient tile</p>
      <div class="tile" id="tile-local-navtree-1">
        ...
      </div>
    </div>
    <div id="panel-right">
      <p>Latest news:</p>
      <div class="tile" id="tile-global-news-1">
        ...
      </div>
    </div>
  </body>
</html>
```

At this point, we have a full page that includes all rendered tiles. On subsequent requests, it may be possible to cache the output of one or more of the earlier phases and only apply the latter stages, either inside Zope, in WSGI middleware or in a cache server.

Client-side rendering

It would be possible to move parts of the rendering process to the browser. For example, imagine that we had a title that contained the user name. This could be represented as a client-side tile like this:

```
<a rel="client-tile" href="/username" />
```

This could then be replaced by the logged in user's name by JavaScript, if the user name was held in a cookie. This type of rendering may be useful to make pages more cacheable, or to make use of browser features in rendering the page.

Client-side tiles would require some Java Script infrastructure, although this may just boil down to some patterns for use with jQuery.

Special tiles

Blocks is expected to ship with (but not to be dependent on) a number of special "utility" tiles, some of which have been referenced in the sections above. These could include:

- A *viewletmanager* persistent tile, which can be configured to render any known viewlet manager. This allows third party products to "plug in" snippets using the established viewlets mechanism, but allows the administrator to control the placement of such viewlets.
- A *fieldview* persistent tile, which can be configured to render a field of a content object when the content author drags a field onto the layout.

Backwards compatibility

This section will outline approaches to backwards compatibility with existing browse components.

Existing views

It should be possible to provide a (deprecated) main_template implementation that results in a semantic page similar to the first stage of rendering outlined above. It may be necessary to apply some transforms on the resulting HTML to cover common conventions. Some pages may require code changes to render properly.

Existing viewlets

Existing viewlets should continue to work. There will be a special type of tile that renders a viewlet manager with all its viewlets, which may be placed anywhere on the site layout.

It should also be trivial to convert a viewlet to a transient tile, with minimal or no changes to code or templates.

Existing portlets

Existing portlets should continue to work as persistent tiles. The existing *<plone:portlet />* registration directive and *portlets.xml* GenericSetup handler will be kept for backwards compatibility. However, a *<plone:tile />* directive applied to a persistent tile will probably be simpler and require less boilerplate.

Portlet add- and edit- forms will be optional (and inferred) by Blocks if not registered, but the ones currently used for portlets should continue to work.

The existing IPortletManager content provider and storage implementation will be replaced by the simpler tile storage model for global and local tiles, although persistent personal and group tiles for the dashboard may continue to use the older portlet storage model.

Frequently Asked Questions

How will tiles be customised?

The two standard types of tiles – persistent and transient tiles – will be implemented along the lines of the standard Zope 3 UI components: views and viewlets. As such, the "view" of a tile will be looked up in the adapter registry as a multi-adapter that includes the context and request, allowing custom versions of a tile for different types of context or request/layer.

In practice, customisation will be supported in a number of ways:

- Through-the-web, using an enhanced version of the *portal_view_cusomizations* tool.
- On the file system, registering override components using ZCML and/or Grok-like semantics.
- As files in a special directory. The directory will be registered (in ZCML) as a "tile overrides" directory where tiles will provide a particular browser layer, and contain template or HTML files that override tiles based on their filename.

What happened to content-type and group portlets?

Blocks aims to do away with complicated merging algorithms that are difficult to understand and requires complex storage models. This means that content-type and group portlets will not be supported in the same way that they are in Plone 3.

Content-type portlets can be simulated by associating a default page layout containing a predefined set of tiles with a given content type and/or category. The content author will then be able to move these around as required when editing the page. There will not be a way to add portlets to all pages of a given type/category after those pages have been created already.

Group portlets will have no equivalent in Blocks, except where a tile renders itself differently depending on the group of the current user. However, the dashboard should be improved to support tiles assigned to groups.

What happened to portlet inheritance and blocking?

Again, complex inheritance and block rules are confusing and almost impossible to make flexible enough. Blocks allows any page layout to override panels from the site layout, which effectively is the same as blocking all portlets in a given column. However, this is a per-page concept and does not inherit to pages in sub-folders.

Note that section-specific site layouts may be used to override the tiles shown in any folder (and its sub-folders). Similarly, pages that request a different layout to the default may gain a different configuration of tiles. The key conceptual difference here is that the set of available layouts is centrally managed, as opposed to being built up from a combination of local portlet assignments and blocking configuration.