CanIt Branding Guide

for Version 10.2.5
AppRiver, LLC
19 September 2018

1 Introduction

CanIt has a facility called *branding* that lets you alter the look and feel of the web-based interface. You can brand CanIt to look like the rest of your web site, and make it fit seamlessly into the site. Generally, you can create your branding so that it survives CanIt upgrades.

1.1 Target Audience

CanIt's web interface is written in PHP. Before you tackle a branding project, you should:

- Be very familiar with HTML.
- Have experience programming in PHP.

2 File Layout

All of CanIt's PHP files live under a single directory. On CanIt appliances, this directory is **`/var/www/canit`**. On Red Hat machines, it's likely to be **`/var/www/html/canit`**. In this manual, we will refer to the directory containing the files as ***web_root***.

The files are laid out as follows:

- **`.php`** files directly under ***web_root*** are small stubs that serve the actual CanIt web pages.
- ***web_root*/classes** – object definitions for various CanIt objects such as users, domain rules, custom rules, etc.
- ***web_root*/images** – static GIF, PNG and JPEG image files.
- ***web_root*/js** – various JavaScript files.
- ***web_root*/manual** – online HTML versions of the manuals.
- ***web_root*/pages** – the actual implementation of CanIt web pages.
- ***web_root*/site** – a directory for placing site-specific configuration files.
- ***web_root*/themes** – a directory of *theme* subdirectories. A theme specifies the branding (or "look-and-feel") of CanIt.
- ***web_root*/themes/langs** – language files for translating the CanIt web interface into another language.

As you customize the look of CanIt, you'll work mostly in the **`site`** and **`themes`** subdirectories of ***web_root***. But you may occasionally need to look at files in the other directories.

3 Configuration Files

When CanIt starts up, it reads the following configuration files, in order:

1. ***web_root*/config.php**

2. ***web_root*/config-pro.php**
3. ***web_root*/config-domain-pro.php** (CanIt-Domain-PRO only.)
4. ***web_root*/config-appliance.php** (CanIt appliances only.)
5. ***web_root*/site/config.php** (if it exists.)
6. ***web_root*/site/*hostname*/config.php** where *hostname* is the server name sent by the Web browser.

CanIt then loads the appropriate language file.

Next, CanIt looks for additional configuration files as follows:

1. If the directory ***web_root*/config.d** exists, then CanIt reads each file in that directory whose name matches ***.php**. The files are read in the same sorted order as output by **ls**.
2. If the directory ***web_root*/site/config.d** exists, then CanIt reads all the ***.php** files in that directory in sorted order.

4 Creating a Theme

To create a theme, you need to create a subdirectory under ***web_root*/themes** containing your theme files. If your directory is named ***web_root*/themes/*theme_name***, then we refer to *theme_name* as the *name* of your theme.

4.1 theme_info.php

The theme subdirectory must contain a file called **theme_info.php**. The file must define a function called ***theme_name*_config** that takes no arguments and returns an array. Here is an example from the **modern** theme:

```
<?php
function modern_config() {
    return array(
        'name' => 'Modern',
        'menu_location' => L("to the left"),
        'page_template' => 'page_template.php',
        'login_template' => '',
        'simple_template' => '',
        'files' => 'themes/modern'
        );
}
?>
```

The array returned by the ***theme_name*_config** function is called the *Configuration Array*. The Configuration Array can contain the following keys. Keys that must be present are marked "required".

- **name** (required). The name of the theme as it should be displayed in the Web interface. Note that the internal name is simply the name of the subdirectory under **themes/**, but the display name can be whatever you like.
- **menu_location** (required). The location of the main menu relative to the page content. It should be something like "above" or "to the left", and should be wrapped in a call to the **L()** function so that it can be localized.
- **page_template** (required). The file name of the actual page template. This template is used to render interior pages. The file name is interpreted within the theme directory.
- **login_template** (optional). The file name of a template that renders the login page. If you do not supply a name (or supply the empty string), the default ***web_root*/themes/common_login.php** template is used.
- **simple_template** (optional). The file name of a template that renders the "simplified" interface. If you do not supply a name (or supply the empty string), the default ***web_root*/themes/common_simple_gui.php** template is used. The **simple_template** is also used to render the following pages:
  - The page that allows users to pick up stripped-and-held attachments.
  - The page that lets users vote as spam/ham from links in the email footer.
  - The URL Proxy landing page.

- The pages that show messages or trap contents that do not require authentication (these are typically accessed by clicking on links in the Pending Notification email body.)

  The **simple_template** should differ from the main template in that it should not display any menu items or the “Logged in as” and “Viewing stream” components.

- **files** (required). The name of a directory (relative to ***`web_root`***) in which to search for files (image files, CSS files, etc.) This will be discussed in depth in Section 5.24. Note that you should *always* use a relative rather than absolute directory for **files**.

- **page_template_dir** (optional). The name of a directory within the theme directory (it must be a relative path.) If you supply this key, then you can use different templates on a per-page basis. For example, the page **`rules.php`** will use the template **`rules.php`** within the **page_template_dir** if it exists and is readable. Otherwise, the standard **page_template** is used.

- **preferences** (optional). If you wish to make aspects of the page layout dynamic based on user-preferences, the **preferences** key should be an array of possible preferences. These are fully described in Section 7.

4.2 The Page Template

The file named by the **page_template** key in the Configuration Array is the template used for rendering most CanIt pages. It is simply an HTML file with some embedded PHP calls.

The page template is invoked with a variable called **`$t`** bound to an object called *the T*. Your template should *only ever call methods on the T*. You should *not* be tempted to look at the innards of the CanIt PHP code and make calls to other CanIt functions. Treat the CanIt PHP code as opaque.

Take a look at the **`modern/page_template.php`** and **`postmodern/page_template.php`** files for examples of how templates are laid out.

5 T Methods

The T has the following public methods. You should only call these methods; *do not* access internal T data structures or private methods.

5.1 head_title()

Calling **`$t->head_title()`** produces HTML code suitable for insertion within the TITLE tags. Here’s how you might use it in a template:

```
<html>
<head>
   <title><?php print $t->head_title(); ?></title>
```

5.2 head_content()

Call **`$t->head_content()`** just before you close the HEAD tag. For example:

```
    <?php print $t->head_content(); ?>
</head>
```

5.3 bodytag

You should not start your body with an explicit BODY tag. Instead, use **`$t->bodytag()`**. You need to call this method because CanIt sometimes adds some attributes to the body tag. Example:

```
<?php print $t->bodytag(); ?>
<!-- Above line emits a <BODY> tag -->
```

5.4 logged_in_as

This method returns the name of the logged-in user. Special HTML characters are replaced with HTML entities. If there is no logged in user, then the blank string (”) is returned. Example:

```
<p>You are logged in as: <b><?php print $t->logged_in_as(); ?></p>
```

5.5 streamname()

This method returns the name of the current stream. Special characters are replaced with HTML entities. Example:

```
<p>Viewing stream: <?php print $t->streamname(); ?></p>
```

5.6 streaminfo()

This method returns information about the current stream, including showing stream inheritance. Special characters are replaced with HTML entities. Example:

```
<p>Stream Details: <?php print $t->streaminfo(); ?></p>
```

5.7 stream_switch_form()

This method returns HTML code for producing the “Switch Stream” form. In CanIt-Domain-PRO, it also renders the Switch Realm form (if the user has access to other realms. Usage is very simple:

```
<?php print $t->stream_switch_form(); ?>
```

5.8 quicklink_menu

This method returns HTML code for producing the “quick links” menu. If there are no quick links, the empty string is returned. The resulting HTML code looks something like this:

```
<ul class="quicklink_menu">
<li>...</li>
<li class="active">...</li>
<li>...</li>
</ul>
```

That is, the quick links menu is an unordered list of class **`quicklink_menu`**. Each link is an LI element. The active element has the class **`active`**. Here's a usage example. Note that we don't create a table row if there are no quick links.

```
<?php $ql = $t->quicklink_menu();
      if ($ql != '') {
          print("<tr><td>$ql</td></tr>\n");
      }
?>
```

5.9 menu

The **`menu`** method is the most complex of the T methods. It can be invoked in three different ways:

1. Calling **`$t->menu()`** yields HTML code representing the navigation menus as nested UL elements.
2. Calling **`$t->menu(n)`** yields HTML code representing *just* the *n*th menu from the top as a single UL element. **`$t->menu(0)`** returns the top-level menu, **`$t->menu(1)`** returns the second-level menu, and so on. (Currently, CanIt has at most three levels of menu.) If the menu with the specified level does not exist, the empty string is returned.
3. Calling **`$t->menu(n, m)`** yields HTML code for menus *n* through *m* as nested UL elements. Thus, **`$t->menu(0, 1)`** returns the top-level and second-level menus as nested UL elements, but ignores the third-level menu.

The results of calls to **`menu`** look something like this:

```
<!-- Output of $t->menu() -->
<ul class="menu1">
  <li>...</li>
  <li>...</li>
  <li class="active">...</li>
    <ul class="menu2">
      <li>...</li>
      <li class="active">...</li>
        <ul class="menu3">
          <li>...</li>
          <li>...</li>
          <li class="active">...</li>
        </ul>
    </ul>
  <li>...</li>
</ul>

<!-- Output of $t->menu(1) -->
<ul class="menu2">
  <li>...</li>
  <li class="active">...</li>
</ul>

<!-- Output of $t->menu(0,1) -->
<ul class="menu1">
  <li>...</li>
  <li>...</li>
  <li class="active">...</li>
    <ul class="menu2">
      <li>...</li>
      <li class="active">...</li>
    </ul>
  <li>...</li>
</ul>
```

Each menu is an UL element; it has the class **`menu`*`N`***, where *N* is 1 for the top-level menu, 2 for the second-level menu and so on. Within a menu, each entry is an LI element, with the active menu entry having class **`active`**.

5.10 helpbox

The **`$t->helpbox()`** method returns HTML code for the online help and online documentation links. Usage is simple:

```
<?php print $t->helpbox(); ?>
```

5.11 language

The **`$t->language()`** method returns the language code currently in use. It is all lower-case and consists of an IETF language tag. Typical tags are `en-us`, `en`, `pt`, `fr`, `de` and `es`, denoting US English, English, Portuguese, French, German and Spanish, respectively.

5.12 title

The **`$t->title()`** method returns the page title suitable for use within the document body. For example:

```
<h1><?php print $t->title(); ?></h1>
```

5.13 body

The **`$t->body()`** method returns most of the actual page content. Very simply, where you want all the page content to appear, use:

```
<?php print $t->body(); ?>
```

5.14 guide_links

The method **`$t->guide_links`** returns a UL element of class **`guides`** containing the links to online documentation and the "Technical Support" link. Each link is an LI element. Where you want the links to appear, use:

```
<?php print $t->guide_links(); ?>
```

5.15 quicklink_form

This method returns HTML code for implementing the "Add to Quick Links" form. The FORM element has class **`quicklink-form`**. Use this code where you want the form to appear:

```
<?php print $t->quicklink_form(); ?>
```

5.16 powered_by_with_version

This method returns HTML code for printing the "Powered by CanIt" link, typically in the page footer. Note that *you may not remove this link*. All of your page templates *must* include it somewhere. Usage:

```
<?php print $t->powered_by_with_version(); ?>
```

5.17 powered_by_no_version

This is similar to **`powered_by_with_version`**, but does not include the product version number. Typically, this is used on the login page since we may not wish to advertise the version number to non-authenticated users. Usage:

```
<?php print $t->powered_by_no_version(); ?>
```

5.18 theme_selection_menu

If you wish to allow users to switch themes on the fly, include a call to this method somewhere in your template. It produces HTML code to create a pulldown selection widget for switching themes. Usage:

```
<?php print $t->theme_selection_menu(); ?>
```

5.19 is_guest

The method **`$t->is_guest()`** returns true if the user is not authenticated (for example, if you have enabled unauthenticated voting and the user clicks a voting link, or a user releases a message by following a link in a notification email message.)

5.20 is_root

The method **`$t->is_root()`** returns true if the user has "root" access. In CanIt-Domain-PRO, this means that the user is either the site administrator or a realm administrator.

5.21 is_super_root

The method **`$t->is_super_root()`** returns true if the user is the site administrator. In CanIt-Domain-PRO, this means that the user has "root" access in the **base** realm.

5.22 product_name

The method **`$t->product_name()`** returns the product name (CanIt-PRO or CanIt-Domain-PRO).

5.23 product_version

The method **`$t->product_version()`** returns the product version number.

5.24 find_file

The method **`$t->find_file("filename")`** searches for a file in a specified set of directories and returns the *first* one found. If the file is not found at all, returns **`"filename"`** unchanged.

Suppose the CanIt URL is: **`http://server.example.org/dir/index.php`** and that your theme's **file** attribute is **`themes/mytheme`**. Then the file *filename* is looked for in the following locations under ***web_root***:

1. **themes/site/themes/mytheme/server.example.org/dir/*filename***
2. **themes/site/themes/mytheme/dir/*filename***
3. **themes/site/themes/mytheme/server.example.org/*filename***
4. **themes/site/themes/mytheme/*filename***
5. **themes/site/files/server.example.org/dir/*filename***

6. **`themes/site/files/dir/`*`filename`***
7. **`themes/site/files/server.example.org/`*`filename`***
8. **`themes/site/files/`*`filename`***
9. **`themes/mytheme/server.example.org/dir/`*`filename`***
10. **`themes/mytheme/dir/`*`filename`***
11. **`themes/mytheme/server.example.org/`*`filename`***
12. **`themes/mytheme/`*`filename`***
13. **`themes/files/server.example.org/dir/`*`filename`***
14. **`themes/files/dir/`*`filename`***
15. **`themes/files/server.example.org/`*`filename`***
16. **`themes/files/`*`filename`***

If you make use of this function in your templates, you can allow simple customizations such as changing a logo or a CSS file based on the server host name or path (which you can make with a symbolic link.)

That is, rather than hard-coding:

```
<img src="themes/mytheme/some_image_file.gif">
```

You should use:

```
<img src="<?php print $t->find_file("some_image_file.gif"); ?>">
```

6 Advanced T Methods

The methods in this section are for the more adventurous. We *do not* recommend using them unless you are very comfortable with programming in PHP.

6.1 get_pref

The method **`$t->get_pref(`*`name, default`*`)`** returns the value of theme preference *name*, or *default* if no value has been set. (If *default* is not supplied, the empty string is used in its place.)

This method is meant for use with themes that support user-preferences; see Section 7 for details.

6.2 menu_items

The method **`$t->menu_items(`*`n`*`)`** returns all entries in menu *n* as a PHP array. Each returned element is an array with the following keys:

- **key** — a unique identifier within this menu.
- **url** — the URL to link to.

- **title** — the suggested value for the “title” attribute.
- **text** — the text of the menu item.
- **active** — true if the menu item is currently active; false otherwise.

6.3 menu_structure

The method **`$t->menu_structure()`** returns an array describing the full menu structure. Each element of the array is a has containing the following keys:

- **key** — a unique identifier within this menu.
- **url** — the URL to link to.
- **title** — the suggested value for the “title” attribute.
- **text** — the text of the menu item.
- **target** — the desired attribute for the “target” attribute. If this is set to the empty string, you should not include a “target” attribute in the rendered HTML.
- **active** — true if the menu item is currently active; false otherwise.
- **sub** — an array of menu items describing the next-level menu associated with this upper-level **key**. If there is no next-level menu, then **sub** is an empty array.

6.4 menupath

The method **`$t->menupath()`** returns an array consisting of the path to the current page. Each entry corresponds to the **key** hash in a menu (see **`menu_items`** above.) The first entry of the array corresponds to the key in menu 0, the next to the key in menu 1, and so on.

6.5 stash

Page authors can “stash” various bits of information on a page. The method **`$t->stash(`*`key`*`)`** returns the stashed item associated with *key*. If there is no such stash item, then the empty string is returned.

7 Theme Preferences

Advanced theme authors may wish to make the page layout dynamic based on user preferences. If the Configuration Array contains a **preferences** key, then it should be an array of possible preferences. Each element of the array is a *preference descriptor* that contains the following elements:

- **name** — an internal computer name for the preference. The name should contain only lower-case letters, numbers and underscores. Note that each theme’s preferences are in a separate namespace, so different themes can have the same preference name without interfering with one another.
- **description** — a human-readable description of the preference.
- **default** — the default value for the preference if none has been set by the user.

- **type** — a string describing the possible type of the preference. The **type** string must take one of the following forms:

  1. **YESNO** or **BOOL** — the preferences is a yes/no preference. The user will be presented with "Yes" and "No" radio buttons, which set the value of the preference to 1 and 0, respectively.
  2. **LIST choice1 choice2 ...** — The preference must be one of the space-separated words following LIST. The user is presented with a pulldown menu to select a choice.
  3. **INT min max** — The preference must be an integer ranging from *min* to *max*.
  4. **FLOAT min max** — The preference must be a number ranging from *min* to *max*.
  5. **STRING len** — The preference must be a string, up to *len* characters long.

If a theme has user-settable preferences, then the **Preferences** page shows them at the end of the preference list, using the entry type appropriate for the **type** of the preference.

8 Menus

CanIt allows you to customize the menus (to add new pages, for example.)

Adding pages to CanIt requires intimate knowledge of the inner workings of the PHP code. The internals can change from release to release; AppRiver cannot offer support for custom pages or custom menu entries.

8.1 URL Keys

CanIt stores *URL Keys* in the menu structure rather than actual URLs. This allows you to change where a menu entry points simply by changing the URL key.

All URLs in the menu system are specified as relative URLs. For example, use **`rules.php`** and not **`/canit/rules.php`** or **`http://server.example.com/canit/rules.php`**.

In general, we recommend that a URL of **`somepath.php`** be given a URL key of **`somepath`**. The internal CanIt code follows this convention everywhere.

To set up your URL keys, create a configuration file (we recommend **`site/config.d/10_url_keys.php`**) with contents similar to the following:

```
<?php
global $Config;
$Config['URL:somepath'] = 'somepath.php';
$Config['URL:another'] = 'another.php';
?>
```

For a URL of ***something*.php**, use a URL key of ***something*** and make an assignment to **$Config['URL:*something*']**.

8.2 Menu Items

A menu entry is represented as a PHP array. The array has the following elements:

- **text** — The text to display for the menu item.

- **key** — An identifier for the menu item. The key must be unique for all menu items at a given level.

- **after** — This element is optional. If supplied, it names the **key** of the menu entry *after which* to place this menu entry. If **after** is set to the special value `*FIRST*`, then the menu entry is placed at the very beginning of the menu. If **after** is not set, then the menu entry is placed at the very end of the menu.
- **url** — This is the URL Key of the page to go to when the menu entry is clicked.
- **target** — This is the desired value of the "target" attribute of the rendered menu link. *You should use **target** sparingly, if at all.*
- **permissionlist** — An array of permission names. If the current user has *any* permission in the list, access to the entry will be granted. Otherwise, the menu entry will not be shown. Consult the PHP source code for a list of permissions.
- **type** — A string or an array of strings. The possible string values are:
  - **`only_root`** — Only show the menu entry to site administrators (and in CanIt-Domain-PRO, realm administrators.)
  - **`only_super_root`** — Only show the menu entry to site administrators.
  - **`has_subrealms`** — (CanIt-Domain-PRO only). Only show the menu entry to realm administrators who have sub-realms.
  - **`sub`** — Only show the menu entry if at least one sub-item would be accessible to the user.
  - **`func:`*`some_name`*** — Call the global function **`menu_callback_`*`some_name`*** and only show the menu entry if it returns a true value.

  If an array of strings is provided for **type**, then the menu entry is shown only if *all* string tests pass.
- **feature_required** — If supplied, this is the name of a "Feature" that must be enabled for the menu entry to be shown. Consult the PHP source code for a list of feature names.
- **title** — This is the text to put in the "title=XXX" attribute of the `<a>` tag.
- **disabled** — If supplied and given a true value, the menu entry will not be shown.

8.3 Manipulating Menu Entries

Menu entries can only be added when a configuration file is read. By the time the page template is invoked to render content, it is too late to add menu items (although existing items can be disabled prior to rendering the menu.) For this reason, you should place code that creates menu items in a site configuration file. **`site/config.2/20_menu_items.php`** is a reasonable place.

A global variable called `$CANIT_MENU` holds the menu structure. This object has the following public methods:

8.3.1 Adding a Top-Level Menu Item

To add a top-level menu item, call `$CANIT_MENU->add(`*`$hash`*`)` where *$hash* is an array as described in Section 8.2. Here is an example:

```
<?php
# Don't forget that $CANIT_MENU is global, so declare it!
global $CANIT_MENU;
$CANIT_MENU->add(array('key' => 'mypage',
                       'text' => L("My Entry"),
                       'url' => 'mypage',
                       'type' => 'only_root',
                       'title' => L("Secret page for admins")));
?>
```

The example code above assumes that `$Config['URL:mypage']` has been set appropriately. Note that English messages are wrapped in a call to the global function `L` like this: `L(``Some-Message'')`. These messages can be localized for the user's selected language.

8.3.2 Adding a Lower-Level Menu Item

To add a lower-level menu item, call `$CANIT_MENU->add_under(`*`$path`*`, `*`$hash`*`)` where *$path* is an array of keys starting from the top-level entry and continuing down, and *$hash* is an array as described in Section 8.2. Here is an example:

```
<?php
global $CANIT_MENU;
$CANIT_MENU->add_under(array('rules'),
                       array('key' => 'newrule',
                             'text' => L("My Rule"),
                             'after' => 'domains',
                             'url' => 'myrule',
                             'title' => L("New Rule Type")));
?>
```

Assuming that the top-level “Rules” menu has a key of `rules`, this adds a menu entry under that menu. If the `rules` menu has a menu entry with a key of `domains`, then the new entry is placed immediately after that entry.

8.3.3 Disabling a Menu Item

To disable a menu item, call `$CANIT_MENU->disable(`*`$path`*`)` where *$path* is an array of keys starting from the top-level entry and continuing down to the desired menu item. Here are some examples:

```
<?php
global $CANIT_MENU;
# Disable Rules : Networks
$CANIT_MENU->disable(array('rules', 'networks'));

# Disable the entire Preferences menu
$CANIT_MENU->disable(array('prefs'));
?>
```

You can also disable a menu from *within* your page template by calling the method `$t->menu_disable_item(`*`$path`*`)`

9 Localizing Messages

To translate messages from English into the user's selected language, create or edit a file called **site/langs/*code*.php** where *code* is the language code (lower-case) such as "fr" or "it" for French and Italian, respectively.

The language file simply assigns translations to the global `$CANIT_MSG` associative array. For example, your language file might look like this:

```
<?php
global $CANIT_MSG;
$CANIT_MSG["Welcome"] = "Bienvenue";
$CANIT_MSG["Waiter, some water, please!"] =
   "Gar&ccedil;on, un peu d'eau, s'il vous pla&icirc;t!";
?>
```

Notice that the translated messages use HTML entities such as `&ccedil;` and `&icirc;` rather than `ç` or `î` directly. You should *always* use HTML entities and *never* use ISO-8859-1 or UTF8-encoded source files.

10 Setting the Default Theme

Normally, CanIt uses the **postmodern** theme as its default theme. To change this, edit ***web_root*/site/config.php** and set:

`$Config['Theme:Name'] = '`*`theme_name`*`';`

Replace *theme_name* with the name of your theme.

11 Restricting Access to Themes

By default, CanIt's theme-switching pulldown allows access to all themes in the **themes** subdirectory. You can explicitly list the themes that should be allowed as follows in ***web_root*/site/config.php**:

```
$Config['Themes:permitted'] = array('modern', 'postmodern');
```

If you set **$Config['Themes:permitted']**, it should be set to an *array* of allowed theme names.

If you want to allow access to all themes except for some, you can use:

```
$Config['Themes:denied'] = array('modern');
```

Set **$Config['Themes:denied']** to an *array* of denied theme names. Note that **Themes:permitted** overrides **Themes:denied**. If it is set, then **Themes:denied** is ignored.

12 Making Themes Customizable

CanIt supports *customizable themes*. These are themes in which the base appearance is specified by the theme, but which permit users to change various aspects of the appearance (such as colors and logos) via the CanIt Web interface.

Creating a customizable theme is not much harder than creating a non-customizable theme. We recommend creating the non-customizable theme first and then modifying it to be customizable.

12.1 Adjusting `theme_info.php`

To make a theme customizable, you need to add a `customizable_properties` member to the Configuration Array. This member is an hash of name/value pairs, each of whose keys is any name you choose (except for **`css`**, which is reserved) and each of whose values is a hash describing the customizable element.

A customizable element can have one of three types:

1. `image`-type properties allow users to upload custom logos.
2. `color`-type properties allow users to set colors of various page elements.
3. `text`-type properties allow users to customize various snippets of text.
4. `css`-type properties allow users to add arbitrary CSS code to the stylesheet. User-supplied CSS always comes last when CanIt generates the dynamic CSS.

   You should have at most *one* `css`-type property.

12.1.1 Custom Images

A customizable `image` property must contain the following hash elements in its value:

- `desc` — a human-readable description of the element.
- `type` — the value of this key must be `image`.
- `limits` — a four-element hash with elements `min_width`, `min_height`, `max_width` and `max_height` specifying the minimum and maximum permissible image dimensions.
- `default` — a file name (*not* a path name) which is passed to **`$t->find_file()`** to load a static image if the user has not uploaded a custom image.

12.1.2 Custom Colors

A customizable `color` property must contain the following hash elements in its value:

- `desc` — a human-readable description of the element.
- `type` — the value of this key must be `color`.
- `default` — the default color of the HTML element(s).
- `css` — the CSS that should be emitted to customize the appearance of the elements. Within the CSS string, use **`%s`** to substitute the color value.

12.1.3 Custom Text Fragments

- `desc` — a human-readable description of the element.
- `type` — the value of this key must be `text`.
- `default` — the default text if the user does not supply a customized version.

Note that in addition to the hash members described above, any element may optionally contain a `base_only` key. If this is set to true, then in CanIt-Domain-PRO, the element will be customizable only in the `base` realm. Use this (for example) for elements that appear on the login page: Before anyone has logged in, the system is implicitly in the `base` realm and only customizations created in the `base` realm can apply.

12.1.4 Custom CSS Code

A `css` property must contain the following hash elements in its value:

- `desc` — a human-readable description of the element.
- `type` — the value of this key must be `css`.
- `default` — must be set to the empty string.
- `css` — must be set to the empty string.

12.2 An Example

Here is a small example of a **`theme_info.php`** file with customizable theme properties:

```
<?php
function example_config() {
  return array(
    'name' => 'Example',
    'files' => 'themes/example',
    'customizable_properties' => array(
      'logo' => array('desc' => L("Logo"),
                      'type' => 'image',
                      'limits' => array('min_width' =>50, 'max_width'=>160,
                                        'min_height'=>50, 'max_height'=>160),
                      'default' => 'logo_example.png'),

      'header_bg' => array('desc' => L("Background color of top bar"),
                           'type' => 'color',
                           'default' => '#3399ff',
                           'css'  => "#header { background-color: %s;}\n")
        ));
}
?>
```

In this example, there are two customizable elements: **logo** and **header_bg**. Let's examine them:

- **logo** is of type `image` and allows the user to upload an image ranging in size from 50×50 to 160×160 pixels. The description on the customization page reads "Logo" and if the user does not upload an image, the default `logo_example.png` image will be used.
- **header_bg** is of type `color`. If the user specifies a non-default color, then the CSS code: **`#header { background-color: `*`color`*`;}`** will be emitted.

12.3 Making the Templates Customizable

To make your theme templates customizable, you simply need to perform the following steps:

1. Make sure that you specify colors *only* in CSS files. Make sure that the **`customizable_properties`** array specifies the correct CSS fragments to override your colors. CanIt will automatically generate a CSS file that comes last (after all your theme's CSS files) to override colors based on user customization.

2. When you wish to display an image, use the **`$t->custom_property`** method to obtain the image. In the example above, the logo image corresponds to a property called **`logo`**, so your template might look like this:

   ```
   <?php
       $logo = $t->custom_property("logo");
       print "<img src=\"$logo\" alt=\"Logo\">";
   ?>
   ```

The **`$t->custom_property("logo")`** call generates the correct filename for the image—it will use the default **`logo_example.png`** image if the theme has not been customized, or a dynamically-generated image if it has.

Similarly, within your theme, use **`$t->custom_property(`*`key`*`)`** to obtain custom `text` values. *Note that CanIt returns whatever text the user has entered. Depending on your theme, you may wish to escape the returned text with* ***`htmlspecialchars`*** *to avoid cross-site scripting attacks.*

13 Advanced Theming: Views

A given theme may support multiple *views*. By default, CanIt supports two views: **`standard`** and **`mobile`**. If a theme supports a **`mobile`** view, then CanIt automatically uses it if it detects that the browser window's width or height is 400 pixels or less.

To indicate that your theme supports multiple views, add an element to the Configuration Array called **`views`**. It should be an array of supported views. For example:

```
<?php
function example_config() {
  return array(
    'name' => 'Example',
    /* ... */
    'views' => array('standard', 'mobile'),
    /* ... */
  );
}
```

Next, the following keys can have array values rather than being simple scalars. The keys of the arrays should be view names:

- **`login_template`**
- **`menu_location`**
- **`page_template_dir`**
- **`page_template`**
- **`simple_template`**

Here's an example:

```
<?php
function example_config() {
  return array(
    'name' => 'Example',
    'views' => array('standard', 'mobile'),
    'page_template' => array('standard' => 'page_template_standard.php',
                             'mobile'   => 'page_template_mobile.php'),
    /* ... etc ... */
  );
}
```

The final step is to create the templates and CSS files appropriate to each view.