Template Overview

Each Squarespace Template is made up of a series of predefined folders and files structured very similarly to a static website. This section will explain the purpose of each file and how they are organized into folders.

Languages and Filetypes

Your Squarespace website will contain regular web files like CSS and JavaScript. In addition, Squarespace is built to recognize a few special file types:

JSON Template Files

Squarespace template files are written in JSON Template, also known as JSON-T. It is an easy to use, easy to read, minimalist template language. JSON-T extends HTML, so you can open these files as HTML files in your editor. JSON-T files have different extensions depending on the type of file, for example *.list, .item,* and *.region*. See the section below on Template Directories for a discussion of the different file types.

LESS Files

Template LESS files (.less) are processed through theLESS preprocessor. LESS extends CSS with dynamic behavior such as variables, mixins, operations and functions. (Note: we also run .css files through the LESS preprocessor in order to parse special styles for our Style Editor.)

Template File Structure

At the very minimum, your template needs a *.region* file and a*template.conf*. The *site.region* is typically the main template file for your website. It's your website's *index.html*, or if you're familiar with Wordpress, *index.php*. The *template.conf* is the main configuration file for your site. It tells Squarespace how the rest of your website is configured.

Squarespace template files are organized using the following folder (or directory) structure at the root of your site:

- assets design assets — *example: images, fonts and icons*
- blocks block files — *navigation.block*
- collections collection files — *[collection].list, [collection].item, [collection].conf*
- pages static page files — *[static].page, [static].page.conf*
- scripts Javascript files — *site.js*
- styles stylesheet files — *styles.css, styles.less*
- [root] sitewide files — *site.region, template.conf*

Each template is made up of a collection of template files. Template files fall into three categories: site-wide files, collection-specific files, and block-specific files. Templates may or may not include all of the folders listed above, and missing folders can be added using SFTP or Git.

Template Directories

A deeper look at the (pre-defined) folders that are used to structure the files in every Squarespace template.

/assets/

This folder is a general purpose storage folder for template assets: Images, Icons, Web Fonts, etc. There is a 1MB file size limit on each individual file uploaded to /assets. After uploading, your template assets are accessible via:

```
http://yoursitename.com/assets/your-asset-file.png
```

/blocks/

In this folder, you will find the templates and configuration files for Squarespace block files. Block files as pertaining to templates let you create reusable partials or 'includes'. These are primarily used to create and configure navigations.

- .block - Block template files. (JSON Template)
- .block.conf - Block template configuration settings. (JSON)

See Blocks for more details.

/collections/

In this folder you can define the presentation for the various kinds of content in your template. You can have an unlimited number of collection types in each site. Each collection can be configured to support specific post types and can be sorted chronologically (like a blog) or user ordered (like a gallery). Each collection must have a configuration file and at least one `.list` or `.item` file.

All .list and .item files are written in JSON Template.

- .list - Templates for the list view of a collection. Example: blog.list templates a list of blog posts. This is the default view of every collection. (JSON Template)
- .item - Templates for the individual page view (permalink) of a collection item. Example: blog.item templates a single blog post page. If a .list file is not provided, the .item will become the default template for the collection (redirecting visitors on the first individual item page instead of the item list page). (JSON Template)
- .conf - Contains the configuration settings for a collection. There is one configuration file for each collection. (JSON)

See Collections for more details.

/pages/

This folder contains static site HTML pages. These are a special type of collection that do not inherently have data of their own. These pages can not be modified within the Squarespace interface by the end user.

- .page - HTML markup for a page. (JSON Template)
- .page.conf - Contains configuration settings for a page. (JSON)

/scripts/

This folder contains the scripts for your website. To include a script in your template, in the <head> area of your template, use:

```
<squarespace:script data-preserve-html-node="true" src="your-script.js" combo="true"/>
```

This syntax will ensure that your script is loaded via the proper URL, and Squarespace will automatically attempt to merge multiple scripts into a single download if the combo parameter is set to true.

See Custom Javascript for more details.

/styles/

This folder contains the CSS styles for your website. All Squarespace CSS is processed using less.css syntax. To include CSS files in your template, edit the template.conf and list the files in the order you wish to include them. For example:

```
"stylesheets" : [ "global.css", "my-site.less", "another-style.less" ]
```

All stylesheets added to the template.conf in this way will be merged and served as one CSS file automatically. You do not need to add any stylesheet links to the HEAD section of your site.

NOTE: If you add a file named reset.css to the /styles folder it will automatically be added to the top of the site.css file and itshould not be listed in the stylesheets variable in thetemplate.conf file.

/site.region

Typically this file is used as the global site wrapper template – containing the site header, footer, and sidebars. Every template must have at least one .region file. Simple templates will have a single .region, more advanced templates will have multiple .region files describing header, body, and footer variants. *Regions files live in the root directory of a template.*

See Layouts & Regions for more details.

/template.conf

Contains the configuration settings for the template. This is where you can name your template, specify layouts, add navigation sections, specify stylesheets to be rolled up into site.css, and other general site options. *Template configuration files must live in the root directory of a template.*

See Template Configuration for more details.

What is JSON?

JSON (JavaScript Object Notation) is a minimal, readable format for structuring data. It is used primarily to transmit data between a server and web application, as an alternative to XML. Squarespace uses JSON to store and organize site content created with the CMS.

Example

Append `?format=json-pretty` to the URL of any page on your Squarespace site and you'll be able to view the JSON data for the site..

Keys and Values

The two primary parts that make up JSON are keys and values. Together they make a key/value pair.

- Key: A key is always a string enclosed in quotation marks.
- Value: A value can be a string, number, boolean expression, array, or object.
- Key/Value Pair: A key value pair follows a specific syntax, with the key followed by a colon followed by the value. Key/value pairs are comma separated.

Let's take one line from the JSON sample above and identify each part of the code.

```
"foo" : "bar"
```

This example is a key/value pair. The key is "foo" and the value is "bar".

Types of Values

- Array: An associative array of values.
- Boolean: True or false.
- Number: An integer.
- Object: An associative array of key/value pairs.
- String: Several plain text characters which usually form a word.

Numbers, booleans and strings are self-evident, so we'll skip over those sections. Arrays and Objects are explained in more depth below.

Arrays

Almost every blog has categories and tags. In this example we've added a categories key, but the value might look unfamiliar. Since each post in a blog can have more than one category, an array of multiple strings is returned.

```
```

```
"foo" : {
  "bar" : "Hello",
  "baz" : [ "quuz", "norf" ]
}
```

Objects

An object is indicated by curly brackets. Everything inside of the curly brackets is part of the object. We already learned a value can be an object. So that means "foo" and the corresponding object are a key/value pair.

```
"foo" : {
  "bar" : "Hello"
}
```

The key/value pair "bar" : "Hello" is nested inside the key/value pair "foo" : { ... }. That's an example of a hierarchy in JSON data.

Recap

We said at the beginning of this tutorial that JSON is a minimal data format. Just by learning a few key principles you can decode an entire site's worth of JSON. And now you can apply that knowledge to developing Squarespace sites with JSON-T, the template language behind the Squarespace Developer platform.

UPDATED: FEBRUARY, 24 2016

Templating Basics

Squarespace sites store their content using a JSON data dictionary, which you can see by adding '?format=json-pretty' to the end of any site URL. This page will show some examples of how we use JSON-T to display JSON data on template pages.

> NOTE: The examples on this page require an understanding of JSON key/value pairs. If you aren't familiar with what JSON Data is, please see theintroduction to JSON page.

Rendering JSON Data

In order to show data on a page, you first need to scope into the appropriate section of JSON Data. Then you can display data by using curly brackets around any JSON key inside that section. In this basic example, we'll show a Page Title:

```
// JSON Data

{
  "collection" : {
    "title" : "Page Title",
    "description" : "This is the page description."
  }
}



<!-- Template code (JSON-T) -->

{.section collection}
  <h1>{title}</h1>
  <p>{description}</p>
{.end}



<!-- Rendered values in HTML -->

<h1>Page Title</h1>
<p>This is the page description.</p>
```

Handling an Array

When scoping into a section with multiple items, commonly known as an array, JSON-T uses the syntax `{.repeated section key}{.end}` instead of `{.section key}{.end}`. Every piece of code within the opening and closing scope tags will be repeated

for each item in the array.

```
// JSON Data

{
  "items" : [
    {
      "title" : "First Item",
      "description" : "This is the first item description."
    },
    {
      "title" : "Second Item",
      "description" : "This is the second item description."
    },
    {
      "title" : "Third Item",
      "description" : "This is the third item description."
    }
  ]
}



<!-- Template code (JSON-T) -->

{.repeated section items}
  <article>
    <h1>{title}</h1>
    <p>{description}</p>
  </article>
{.end}



<!-- Rendered values in HTML -->

<article>
  <h1>First Item</h1>
  <p>This is the first item description.</p>
</article>
<article>
  <h1>Second item</h1>
  <p>This is the second item description.</p>
</article>
<article>
  <h1>Third Item</h1>
```

```
  <p>This is the third item description.</p>
</article>
```

Render Data from Multiple Sections

In your template you'll more than likely have a need to scope into multiple sections on any given page. You can do this by ending one scope tag and entering another.

```
// JSON Data

{
 "website" : {
  "siteTitle" : "My Website",
  "baseUrl" : "http://developers.squarespace.com"
 },
 "collection" : {
  "title" : "Page Title",
  "description" : "This is the page description."
 }
}



<!-- Template code (JSON-T) -->

{.section website}
 <header>
  <h1><a href="{baseUrl}">{siteTitle}</a></h1>
 </header>
{.end}
{.section collection}
 <section>
  <h1>{title}</h1>
  <p>{description}</p>
 </section>
{.end}



<!-- Rendered values in HTML -->

<header>
 <h1><a href="http://developers.squarespace.com">My Website</a></h1>
</header>
<section>
```

```
  <h1>Page Title</h1>
  <p>This is the page description.</p>
</section>
```

Section with no Data

If a section or repeated section has an empty value, nothing inside the scope tags will be outputted.

```
// JSON Data

{
  "collection" : { }
}



<!-- Template code (JSON-T) -->

{.section collection}
  <section>
    <h1>{title}</h1>
    <p>{description}</p>
  </section>
{.end}



<!-- Rendered values in HTML -->
```

Using an Or Statement

If a section or repeated section has no value, an or statement outputs alternative markup to express that empty value.

```
// JSON Data

{
  "items" : [ ]
}
```

```
<!-- Template code (JSON-T) -->

{.repeated section items}
  <article>
    <h1>{title}</h1>
    <p>{description}</p>
  </article>
{.or}
  <p>There are no items here.</p>
{.end}



<!-- Rendered values in HTML -->

<p>There are no items here.</p>
```

Using Dot Notation

You can add any JSON value to your template by writing the key and its parent object in dot notation. This method of scoping only defines the scope for this one single value, so no end tag is required in JSON-T.

```
// JSON Data

{
  "collection" : {
    "title" : "Page Title",
    "description" : "This is the page description."
  }
}



<!-- Template code (JSON-T) -->

<h1>{collection.title}</h1>
<p>{collection.description}</p>



<!-- Rendered values in HTML -->

<h1>Page Title</h1>
<p>This is the page description.</p>
```

Referencing the Scope

Using scope reference, written as `{@}`, allows you to reference the key you're scoped into. This is like `(this)` in JavaScript.

```
// JSON Data

{
  "collection" : {
    "title" : "Page Title",
    "description" : "This is the page description."
  }
}



<!-- Template code (JSON-T) -->

{.section collection}
  {.section title}<h1>{@}</h1>{.end}
  {.section description}<p>{@}</p>{.end}
{.end}



<!-- Rendered values in HTML -->

<h1>Page Title</h1>
<p>This is the page description.</p>
```

Using an If Statement

If statements check to see if a section exists without scoping into that section.

```
// JSON Data

{
  "items" : [
    {
      "title" : "First Item",
      "description" : "This is the first item description."
```

```
    },
    {
      "title" : "Second Item",
      "description" : "This is the second item description.",
      "featured" : true
    },
    {
      "title" : "Third Item",
      "description" : "This is the third item description."
    }
  ]
}
```

```
<!-- Template code (JSON-T) -->

{.repeated section items}
  {.if featured}
    <article class="featured-post">
      <h1>{title}</h1>
      <p>{description}</p>
    </article>
  {.or}
    <article>
      <h1>{title}</h1>
      <p>{description}</p>
    </article>
  {.end}
{.end}
```

```
<!-- Rendered values in HTML -->

<article>
  <h1>First Item</h1>
  <p>This is the first item description.</p>
</article>
<article class="featured-post">
  <h1>Second item</h1>
  <p>This is the second item description.</p>
</article>
<article>
  <h1>Third Item</h1>
  <p>This is the third item description.</p>
</article>
```

UPDATED: FEBRUARY, 12 2016

Using Git

This page covers the basics of using Git on your Squarespace site. For a more in-depth guide to using Git check out the Pro Git Book, which is available for free online at git-scm.com.

Tools You'll Need

- Squarespace Developer Site
- Terminal application

What is Git?

Git is a version control system. It keeps track of the code you write on your site. Whenever you make a change, it is committed to your Git repository (even if you upload those changes through SFTP), which allows you to do things like access older versions of your site, or identify when a bug was created.

Should you use Git?

That depends on your workflow. Git can make a lot of things, like cloning a template to your computer, easier. And it can be a great way to manage the problems that occur when multiple people are editing code on one site. If you're a single developer working on a site, it may be easier to use SFTP to download and upload files.

Downloading and Installing Git

Mac: There are several ways to install Git. One easy way is to use the graphic installer, which you can download here. Once the download is complete, open the .pkg file and follow the instructions to install it.

Once the install is complete, open up your Terminal application, type in the following command and hit enter:

```
git --version
```

If the install was successful, you should see a message that tells you which version of Git you are currently running, like this:

```
git version 1.7.9.6 (Apple Git-31.1)
```

Cloning a Template

To clone your template, log into your Squarespace site and go to the Developer tab. Under SFTP details there will be a line that says "Repository." Copy the URL listed there.

In your Terminal application, type in the following command and hit Enter:

```
git clone [your-repository.git]
```

You will get a prompt for your username and password. They are the same credentials you use to log into your Squarespace site. Once you type in your credentials, the template will begin downloading. This may take a few minutes.

Changing a Template

Once you've made changes and you'd like to push them to your site, you'll need to type in a few more terminal commands. The first simply navigates to the folder where your files are stored.

```
cd template
```

Then you'll need to add the files you want Git to track for this commit.

```
git add site.region
```

Then you'll need to commit your changes. You can see in the command below that there is a space to add a message to each commit. These messages can be extremely helpful if you need to revert back to an older version of your code.

```
git commit -m "Changing something in site.region"
```

And, finally, push the changes to your site.

```
git push
```

UPDATED: FEBRUARY, 19 2016

Template Configuration

The template configuration file (template.conf) contains template meta data (template name, author), defines page layouts, navigation areas, and the stylesheet loading order. All .conf files are written in the JSON format.

Example Template Configuration File (template.conf)

```
{
  "name" : "Template Name",
  "author" : "Author Name",

  "layouts" : {
    "default" : {
      "name" : "Default",
      "regions" : [ "site" ]
    }
  },

  "navigations" : [ {
      "title" : "Main Navigation",
      "name" : "mainNav"
  }, {
      "title" : "Secondary Navigation",
      "name" : "secondaryNav"
  } ],

  "stylesheets" : [ "global.less", "typography.less" ]

}
```

Configuration Options

name

The name of the template. Displayed in the Template and Developer tabs. (Required)

author

The author of the template. Displayed in the Template and Developer tabs. (Required)

layouts

Site layouts that consist of one or more regions. Defines the overall HTML markup. See Layouts & Regions. (Required: default layout must be defined)

layouts > name

The name of the layout option as it appears in the user-editable layout select field.

layouts > regions

List of region files to combine into layout (in the order they should be combined).

navigations

Configures the top level navigation sections visible in the Navigation section of the interface. See Menus & Navigation.

navigations > title

The name of the layout option as it appears in the user-editable layout select field.

navigations > name

The navigation ID. Used to access navigation data in navigation tags and elsewhere.

stylesheets

List of your stylesheet. Stylesheets will be compiled into site.css following the ordering here.

NOTE: If you add a file named reset.css to the `/styles/` folder it will automatically get added to the top of the `site.css` file. It should not be listed in the "stylesheets" variable in `template.conf`.

UPDATED: FEBRUARY, 11 2016

Layouts & Regions

Site layouts define the HTML 'wrapper' for your site... everything from <!doctype html> to </html>.

Single Layout Sites

At it's simplest, a layout is one file (typically named site.region). This is used as the main template (like index.php in Wordpress).

Example simple layout file (site.region):

```
<!doctype html>
<html>
  <head>
    {squarespace-headers}
  </head>
  <body class="{squarespace.page-classes}" id="{squarespace.page-id}">

    <header id="header">
      <h1><a href="/">{website.siteTitle}</a></h1>
      <squarespace:navigation navigationId="mainNav" template="navigation" />
    </header>

    <main id="canvas">
      <section id="page" role="main" data-content-field="main-content">
        {squarespace.main-content}
      </section>
      <aside id="sidebar">
        <squarespace:block-field id="sidebarBlocks" label="Sidebar Content" />
      </aside>
    </main>

    <footer id="footer">
      <squarespace:block-field id="footer-blocks" columns="12" label="Footer Content" />
    </footer>

  </body>
</html>
```

Example layout definition in template.conf:

```
...
```

```
"layouts"  : {
  "default" : {
    "name" : "default"
    "regions" : [ "site.region"]
  }
}

...
```

Multiple Layouts

While single layout sites are the norm and work perfectly well for most sites, some more advanced sites may require different thinking - that's why you can enable multiple layouts in a Squarespace template.

Consider the case where the homepage has a different layout than the sub pages. Let's say that the homepage is full-width and the sub pages have a sidebar... so the header and footer regions are the same, but the middle content section is different.

Step 1: Create Multiple Region Files

First, create the shared layouts:

- header.region *(only code from the site header)*
- footer.region *(only code from the site footer)*

Then create the two different regions for the middle content section:

- full-width.region *(no sidebar in the markup)*
- sidebar.region *(contains sidebar markup)*

Step 2: Configure Layouts

Set up the multiple layouts in your template configuration file (template.conf):

```
...

"layouts" : {
  "default" : {
    "name" : "Sidebar",
    "regions" : [ "header", "sidebar", "footer" ]
   },
  "homepage" : {
    "name" : "Full Width",
    "regions" : [ "header", "full-width", "footer" ]
```

```
    }
  },

  ...
```

Step 3: Set Default Layouts

Layouts can be set per page (via Page Settings in the interface), but you can make things easier for the user by setting the default layouts for specific types of pages.

There are three options for setting default layouts:

1. Site-wide default layout – the layout called "default" in template.conf will be the site-wide default template.
2. Collection-specific default layouts – you can specify a default layout for each type of collection in the collection configuration file (collectionName.conf) using the "layout" variable.
3. Folder-specific default layouts – you can specify a default layout for pages/collections within a folder in the folder configuration file using the "layout" variable. Folder defaults override collection defaults.

You can also specify the default homepage layout. If you add a layout variable named "homepage". See example above. The homepage default layout overrides all other default layouts.

UPDATED: FEBRUARY, 03 2016

Template Partials

Partials allow you to reuse code inside your template, instead of having redundant code scattered throughout your files. This can make your template easier to maintain.

Creating a Partial

To create a partial add a .block file to your /blocks/ folder. This file can contain any kind of code used in template files, most commonly HTML and JSON-T.

To use a partial in a template add a JSON-T formatter with the block file name inside the tag, like the code example below demonstrates.

```
{@|apply some-block.block}
```

When Should I Use Partials?

Think of partials like classes in CSS. A class is typically something that is reused across your site; classes are also broad and can be used in analogous contexts. If you will use a piece of code in your template more than once in a similar setting you may want to consider making it a partial.

Example

For the purpose of example, let's build a line of metadata that can be used in several different collections on both .list and .item pages.

```
item-meta.block
---------------------------
<div class="item-meta">
    <p>Added on {addedOn|date %B %d, %Y} by {author.displayName}</p>
</div>
```

This partial can be rendered in either a .list or .item file in any collection type. The output changes depending on the context. For instance, we can render it in the following ways.

```
blog.list:
---------------------------
{.repeated section items}
  {@|apply item-meta.block}
{.end}
```

```
blog.item:
---------------------------
{.section item}
  {@|apply item-meta.block}
{.end}

gallery.list:
---------------------------
{.repeated section items}
    {@|apply item-meta.block}
{.end}
```

NOTE: Partials are dependent on scope. If you break up your site into modules, it allows you to use them in several different contexts. To learn how scoping works visit this page.

UPDATED: MAY, 27 2016

Menus & Navigation

Take control of the navigation sections in the interface and create custom navigation templates for use throughout your template files.

Navigation Configuration

Navigation sections are defined in the `template.conf` file. These define the sections shown in the Navigation area of the interface.

You can define more than one navigation per template using`navigations`. Each navigation is given a unique name which is then used when including it in your `HTML`.

```
...

"navigations" : [
  {
    "title" : "Main Navigation",
    "name" : "mainNav"
  },
  {
    "title" : "Secondary Navigation",
    "name" : "secondaryNav"
  }
],

...
```

Navigation Templates

Navigation templates are defined as `.block` files. Navigation templates define the `HTML` markup for the navigation and are binded to navigation data via the Squarespace navigation tag.

Example navigation template:

```
<nav>
  <ul>

    {.repeated section items}
    <li class="{.section active} active-link{.end}">
```

```
      {.section collection}
        <a href="{fullUrl}">{navigationTitle}</a>
      {.end}

      {.section externalLink}
        <a href="{url}"{.section newWindow} target="_blank"{.end}>{title}</a>
      {.end}

    </li>
    {.end}

  </ul>
</nav>
```

Folders

If you have defined a `folder.conf` file in your `/collections/`folder, users will be able to add folders to the navigation sections. They can also be used to include automatic sub menus (see the Sub Menus Section below).

The folder structure can be utilized in the navigation template to create drop downs as indicated in the following example:

```
<nav>
  <ul>

    {.repeated section items}
    <li class="{.section active} active-link{.end}
      {.if folderActive} active-link active-folder{.end}">

      {.folder?}

        <a>{collection.navigationTitle}</a>
          <ul class="subnav">
            {.repeated section items}
              {.collection?}
                <li class="{.section active} active-link{.end}">
                  <a href="{collection.fullUrl}">{collection.navigationTitle}</a>
                </li>
              {.end}
              {.section externalLink}
                <li class="external-link">
                  <a href="{url}"{.section newWindow} target="_blank"{.end}>{title}</a>
                </li>
```

```
            {.end}
          {.end}
        </ul>

      {.or}

        {.section collection}
          <a href="{fullUrl}">{navigationTitle}</a>
        {.end}

        {.section externalLink}
          <a href="{url}"{.section newWindow} target="_blank"{.end}>{title}</a>
        {.end}

      {.end}

    </li>
    {.end}

  </ul>
</nav>
```

NOTE: Folder depth in Squarespace is currently limited to one ... folders currently cannot be nested inside other folders.

UPDATED: FEBRUARY, 11 2016

Folders & Indexes

Folders group pages and collections together for use in submenus, drop-down menus, and/or index pages.

Folder Configuration

To add folders on your site, create a folders.conf file in your collections folder. The standard settings for a folder.conf file are below.

```
{
 "title" : "My Folder",
 "folder": true,
 "addText": "Add Page",
 "icon" : "folder"
}
```

Folders in Navigations

The contents of a folder can be added to a site navigation in one of two ways:

1. A dropdown menu beneath a main navigation item.
2. A submenu, which is separate from the main navigation template and is displayed on pages within the folder.

For more information on using folders in navigations see the Menus and Navigation page.

Indexes

An index is a folder with a main page that allows developers to aggregate data from the collections and pages contained within it.

Examples of the indexes used in Squarespace templates include, see the homepages on the Avenue and Flatiron templates.

Creating an Index

To create an index, add an index.conf file to your collections folder. The standard settings for an index.conf file are below.

```
{
 "title" : "My Index",
 "folder" : true,
 "index" : true,
```

```
  "addText" : "Add Collection",
  "acceptTypes" : [ "page", "gallery" ],
  "icon" : "index",
  "indexType" : "grid",
  "fullData" : false
}
```

To make an index page template, add an index.list file to your collections folder.

The JSON data that's returned on an index page includes a unique array nested inside the collection object titled "collections." The code sample below shows a pared down version of the data, along with template code to loop through each collection.

> NOTE: This sample shows the JSON return when ["FullData": true] is set. This will return all nested item data. If you don't need the full data, setting this value to "false" will improve page load speeds.

JSON

```
{
  "collection" : [ {
    "title" : "My Index",
    "collections" : [ {
      "title" : "Blog",
      "items" : [ {
        "title" : "First Post"
      }, {
        "title" : "Second Post"
      } ]
    }, {
      "title" : "Gallery",
      "items" : [ {
        "title" : "First Photo"
      }, {
        "title" : "Second Photo"
      } ]
    }, {
      "title" : "Events",
      "items" : [ {
        "title" : "First Event"
      }, {
        "title" : "Second Event"
      } ]
    } ]
  } ]
}
```

HTML

```
{.section collection}
  <h1>{title}</h1>
  {.repeated section collections}
    <h2>{title}</h2>
    {.repeated section items}
      <ul>
        <li>{title}</li>
      </ul>
    {.end}
  {.end}
{.end}
```

Compiled HTML

```
<h1>My Index</h1>

<h2>Blog</h2>
<ul>
  <li>First Post</li>
  <li>Second Post</li>
</ul>

<h2>Gallery</h2>
<ul>
  <li>First Photo</li>
  <li>Second Photo</li>
</ul>

<h2>Events</h2>
<ul>
  <li>First Event</li>
  <li>Second Event</li>
</ul>
```

Collections

A site can have an unlimited number of collection types. Each collection can be configured to support specific post types and can be sorted chronologically (like a blog) or user ordered (like a gallery).

System Collections

Squarespace creates and maintains several collection types in the system you can use in your site without needing the files in your template. We call these "system collections." To add a system collection to your site, add an array to your `template.conf` file that specifies which collections you'd like to use.

```
"systemCollections" : [
  "album",
  "blog",
  "events",
  "gallery",
  "products"
]
```

System collections require no coding or maintenance by the developer. The system collection code is maintained by Squarespace and is not available for customization. For information about custom collections, see below.

Creating Custom Collections

To create a custom collection on your site you have to create a configuration file (.conf) and a .item and/or .list file.

Collection Configuration (collection.conf)

Contains the configuration settings for a collection. There is one configuration file for each collection.

```
{
  "title" : "Blog",
  "ordering" : "chronological",
  "addText" : "Add Post",
  "acceptTypes" : [ "text" ]
}
```

Configuration Options

title: The name of the collection as it will appear in the "Add New Page" dialog.

ordering: The method of ordering for the collection. Available options: chronological, user-orderable, calendar.

addText: Specifies the text used in the "add" button in the Squarespace interface. It is also used in empty collection message when a collection does not contain any items.

acceptTypes: Specifies the post types allowed in this collection. Available: text, image, video.

Collection List Views (collection.list)

This is the default view of every collection and shows all posts in that collection. For example, blog.list templates a list of blog posts.

Collection Item Views (collection.item)

Item views are templates for the individual pages of a collection item (permalink). Example: blog.item templates a single blog post page.

> TIP: Exclude the .list file to start on item view
>
> If a .list file is not provided, the .item becomes the default template for the collection, thus starting viewers on the first individual item page instead of the collection's list page.

UPDATED: MARCH, 07 2016

What is JSON-T?

JSON Template (JSON-T) is a minimal but powerful template language, designed to be paired with a JSON dataset. This data is provided by Squarespace and is dynamically generated, containing all of your site content.

Squarespace uses JSON-T to transform data into a web page. This process is called "rendering" the web page. The renderer combines data from the CMS, also known as the "context" with the JSON-T code to create the HTML output. That HTML is then sent to your browser and displayed.

Here is a high-level overview of JSON-T and how it works:

JSON-T uses a special syntax to mark where data will be inserted into the page. For example: `{foo}`. These are called JSON-T tags. There are two main types of tags in JSON-T, *variables* and*directives*.

- Variables are used to insert data into the page. They tell the renderer what data to render. This is a variable tag: `{foo}`
- Directives are like commands. They tell the renderer how to render a section of JSON. You can identify them because they use an extra dot, and they often come in pairs. For example: `{.section foo} expand foo {.end}`.

Variable Substitution

Variables inject data from the JSON context onto the page. To inject the value of `baz`, put it within curly braces:

```
<!-- will print Hello -->

{baz}

<!-- given this JSON context -->

{ "baz" : "Hello" }
```

You can also *drill down* into the JSON structure using dot notation. When looking up a variable name, we start at the top level of the context and pick the nested object that corresponds to each part of the variable.

```
<!-- will print Hello -->

{foo.bar.baz}

<!-- given this JSON context --->

{
 "foo": {
```

```
    "bar": {
      "baz": "Hello"
    }
  }
}
```

Built-in Directives

Directives are built-in language constructs that typically start with a period (.). The two main directives used in JSON-T aresections and repeated sections.

`{.section foo}` starts a section named foo. The namecorresponds to a JSON key. The section is expanded if the key is present and not false. Sections are closed with a `{.end}` directive.

```
<!-- will print Hello -->

{.section foo}
    {bar}
{.end}

<!-- given this JSON context -->

{
  "foo": {
    "bar": "Hello"
  }
}

<!-- but prints nothing if "foo" is missing -->
```

`{.repeated section bar}` starts a repeated section named bar. The name corresponds to a JSON key, whose value is a list of dictionaries or strings. The section is expanded once for each element of the list. The special variable `{@}` represents the value of the context at the current scope, or currently active part of the context.

```
<!-- will print Hello World -->

{.repeated section bar}
  {@}
{.end}
```

```
<!-- given the JSON context -->

{
  "bar": ["Hello", "World"]
}
```

Putting it All Together

Combining these simple constructs together with your HTML, CSS and Javascript is how a Squarespace Template is built, using the CMS to provide the data and information that we render in our template files.

REFERENCE: JSON-TEMPLATE REFERENCE WIKI

UPDATED: AUGUST, 02 2016

JSON-T System Variables

System Variables are JSON-T variables that cannot be found in the context, but are essential for most websites. These variables should be rendered onto a page just like normal context-backed variables. They represent core system level information, such as system headers, the main content of a page, and page classes used to control how a website functions.

Squarespace Headers

```
{squarespace-headers}
```

Injection point for Squarespace scripts, system generated meta tags, and user content from the 'Header Code Injection' point found in the 'Code Injection' tab.

Squarespace Page ID

```
{squarespace.page-id}
```

Adds a page specific unique id - for use on `<body>` tag.

Squarespace Page Classes

```
{squarespace.page-classes}
```

Adds site and page specific classes - for use on `<body>` tag.

Squarespace Main Content

```
{squarespace.main-content}
```

Injection point for page templates in site region files. This is where the collection or page content will be injected.

Squarespace Post Entry

```
{squarespace-post-entry}
```

Injection point for user content. Typically placed after each item in a blog collection.

Squarespace Footers

```
{squarespace-footers}
```

Injection point for tracking scripts and user content from the 'Footer Code Injection' point found in the 'Code Injection' tab.

UPDATED: FEBRUARY, 24 2016

JSON-T Directives

Directives in JSON-T are like commands. They tell the renderer how to render a section of JSON. You can identify them because they use an extra dot, and they often come in pairs (ex. '{.section foo}').

Section

Sections do most of the work in JSON Template. They allow you to “zoom in” on a part of the JSON context, removing duplicated dot notation prefixes throughout a section of code. Here are two examples, one using a `section` and one without:

```
<!-- JSON-T using section -->

{.section website}
  {siteTitle}
{.end}

<!-- JSON-T using dot notation -->

{website.siteTitle}
```

There are two things you need to know about Section and Repeated Section:

1. The content inside a section will only display if the section exists
2. Sections define scope, meaning they set the root for any data tags within

Repeated Section

```
<!-- super basic repeated section example -->

{.repeated section items}
  If there are any items, repeat this info for each item
{.end}
```

NOTE: Inside a `repeated section` you can number each item with an instance of `{@index}` which outputs the current index of the item. Index numbers start with `0`.

Or Clause

You can use an `or` statement within a `section` or `repeated section` to display something if the condition of the section or repeated section is not met ... for example, if the section is missing or false:

```
<!-- super basic or statement -->

{.section item}
  Item exists.
{.or}
  Item does not exist.
{.end}
```

Alternates With Clause

Within a `repeated section`, the `alternates with` allows you to insert delimiters between each item that is repeated. Handy for commas, slashes or even horizontal rules, this block is expanded in between every pair of repeating sections.

```
{.repeated section items}
  This stuff shows for each item.
{.alternates with}
  ------ {# show this dashed line in between each item}
{.end}
```

Var Directive

The `var` directive allows you to temporarily store a value for reference later. This can be useful along with `sections` to keep code concise and legible.

```
{.var @myTitle website.siteTitle}
```

A `var` allows you to create a variable and then use it later within any `section` or `repeated section` within the current JSON-T file. Here's an example:

```
{.var @firstImg items.0}
{.var @pageThumb collection.mainImage}
```

```
{.section collection}
  <div class="wrapper{.if @firstImg} has-image{.end}">
    <div class="thumbnail-container">
      <img {@pageThumb|image-meta} data-load="false" />
    </div>
  </div>
{.end}
```

Comment Directive

Occasionally, you may want a comment that would not end up rendering in your HTML. This JSON-T comment is going to accomplish that for you. There are two types of comments available, single line and multiline.

```
{# This is a single line comment}

{##BEGIN}
    This is a longer,
    mulitline comment
{##END}
```

UPDATED: APRIL, 06 2016

JSON-T Predicates

Predicates are special Directives used like sections, except they don't correspond to a variable in the context. They allow you to test for certain system defined features.

The format for a predicate is `{.predicate-name?} ... {.end}`. You can generally spot a predicate by the trailing question mark.

Collection Predicates

These tags are used in collection template files (`.list`, `.item`).

Main Image Predicate

```
{.main-image?}
  <!-- code here -->
{.end}
```

Tests the presence of a Main Image for a collection or item.

Excerpt Predicate

```
{.excerpt?}
  <!-- code here -->
{.end}
```

Tests the presence an excerpt for an item.

Comments Enabled Predicate

```
{.comments?}
  <!-- code here -->
{.end}
```

Tests if comments are enabled for a particular item.

Disqus Enabled Predicate

```
{.disqus?}
  <!-- code here -->
{.end}
```

Test is Disqus comments have been enabled (only useful in items with comments enabled).

Video Predicate

```
{.video?}
  <!-- code here -->
{.end}
```

Tests if an item within a Gallery Collection is a video.

Even Predicate

```
{.even?}
  <!-- code here -->
{.end}
```

Tests if an item's index position is even.

Odd Predicate

```
{.odd?}
  <!-- code here -->
{.end}
```

Tests if an item's index position is odd.

Equal Predicate

```
{.equal? arg1 arg2}
  <!-- code here -->
```

```
{.end}
```

Tests if two arguments are equal. If true, anything between the condition (`{.equal?}`) and `{.end}` will be outputted. If either of your arguments could contain a space, use a different delimiter to ensure an accurate return like so: `{.equal?:arg1:arg2}`.

Navigation Predicates

These predicates are only used within navigation (`.block`) files.

Collection Predicate

```
{.collection?}
 <!-- code here -->
{.end}
```

Tests if a navigation item is a collection.

External Link Predicate

```
{.external-link?}
 <!-- code here -->
{.end}
```

Tests if a navigation item is an external link.

Folder Predicate

```
{.folder?}
 <!-- code here -->
{.end}
```

Tests if a navigation item is a folder.

JSON-T Formatters

Formatters can be used to control the formatting of data for a given variable.

The syntax of a formatter is `{variable-name|formatter-name}`.

JSON

```
{variable|json}
```

Output the variable as JSON data.

Prettified JSON

```
{variable|json-pretty}
```

Output the variable as formatted JSON data.

Date/Time Formatters

Date

```
{addedOn|date %A, %B %d}
```

Format a date in-line using the YUI date format.

Timesince Date

```
{addedOn|timesince}
```

Displays the time/date in a term relative to right now. Also referred to as 'time ago' or 'twitter-style' date format.

String Formatters

Slugify

```
{variable|slugify}
```

Converts a human readable string variable to lowercase, removes non-word characters (alphanumerics and underscores) and converts spaces to hyphens. For example, "Test Example" would become "test-example".

Smartypants

```
{variable|smartypants}
```

Translates plain ASCII punctuation characters into “smart” typographic punctuation HTML entities (source).

URL Encode

```
{variable|url-encode}
```

Encode a variable so it can be safely used in a URL. For instance, "Test Example" would become "Test%20Example".

HTML

```
{variable|html}
```

Escape a string to ensure it is valid HTML.

HTML Tag

```
{variable|htmltag}
```

Escapes html tags and quotes (<, >, &, ")

HTML Attribute

```

```

```
{variable|htmlattr}
```

Escape a string to ensure it can be used in an attribute reference (src="X", for instance) within an HTML tag.

Twitter Links

```
{variable|activate-twitter-links}
```

Make links and @names link to the appropriate places on twitter. For instance, @squarespace (the string) would become@squarespace.

Safe

```
{variable|safe}
```

Make the variable 'safe' by stripping away unsafe HTML, such as injected script tags.

Item Count

```
{items|count}
```

Output the length of the items array

UPDATED: AUGUST, 02 2016

JSON-T Helpers

Helpers are custom made JSON-T Formatters that expand into a pre-formatted section of code specifically for a Squarespace site.

These tags are used in collection template files (.list, .item). The @ symbol in the code samples refers to the local scope, so these code samples can be used inside of a {.section item} or{.repeated section items}. Alternatively, if you're working on an item page the @ symbols can be replaced by the word item.

Item Classes

```
{@|item-classes}
```

Adds useful item-specific classes (ex. author, category, tag, etc.).

Share Button

```
{@|social-button}
```

Includes pre-built share button that can be configured in the 'Share Button' tab.

Comments

```
{@|comments}
```

Includes pre-built comments that can be configured in 'General' settings tab.

Comment Link

```
{@|comment-link}
```

Injects [X] Comments or Disqus comment text

Comment Count

```
{@|comment-count}
```

Generates pluralized comments message (ex. '3 Comments'), or 'No Comments'.

Squarespace Simple Like Button

```
{@|like-button}
```

Includes pre-built simple like button that can be configured in 'General' settings tab.

Image Attributes

```
{@|image-meta}
```

Used to add attributes to an image tag (focal point data, image source, original image dimensions).

Product Price

```
{@|product-price}
```

Injects the price of an item in a products collection. This tag also formats the price into a currency format.

Product Variants, Quantity, Add to Cart Button

```
{@|product-status}
```

Pulls in the variants, quantity, and add to cart button for products.

UPDATED: FEBRUARY, 19 2016

Open Block Field

The Squarespace system provides a large number of blocks that can be added (using the CMS) to Pages, Blog Posts, and Open Block Fields present in a template. All system blocks have default templates that render the contents of the block.

Squarespace Block Field Tag

Open Block Fields are open areas in a template (provided by a developer) into which a user can add any system block, and use the same LayoutEngine grid-based layout system that is used in Pages and Blog Posts. Open Block Fields are ideal for site footers and blog sidebars.

To specify an Open Block Field in a template, use the Squarespace Block Field tag:

```
<squarespace:block-field id="blockField1" columns="12"/>
```

Each block field must have a unique `id`. The columns attribute can be set to either `1` or `12`. It is usually best to set this to 12, but you can set it to 1 when you want to limit the field to only a single column of blocks (in a sidebar for example).

Locking the Layout

A locked layout allows developers to place system blocks on a site without giving front-end users the ability to add additional blocks or change the layout of the open block field. Front-end users still have to ability to edit the content of a locked block.

Adding the `lock-layout="true"` attribute to an open block field will lock the layout. *It's important to note that you should only lock the layout of an open block once you've already added the blocks with example content using the CMS.* Otherwise, you will inadvertently lock yourself out of the open block.

```
<squarespace:block-field id="blockField1" columns="12" lock-layout="true"/>
```

Customizing Blocks

We currently do not support customizing the HTML markup of our system blocks.

UPDATED: FEBRUARY, 02 2016

Navigation Tag

Navigation tags in Squarespace are a particular type of .block file that creates and displays navigation areas in a template that are revealed in the CMS 'Pages' section of the site.

Site Navigation

To include a navigation in your template, use the Squarespace`navigation` tag:

```
<squarespace:navigation navigationId="mainNav" template="main-navigation" />
```

Squarespace navigation tags bind navigation data (navigationId) to a navigation template (block file). The `navigationId` must match the "name" of one of the navigations defined in`template.conf`. And the template must match the filename of a`.block` template in the /blocks folder. For instance, if you specify`template="mynav"`, you need a template called `mynav.block`in the/blocks folder.

Folder Navigation

Sometimes referred to as a "Sub menu," a `folder-navigation` tag will automatically generate links to all other pages within the same folder when viewing a page inside a that folder. To add a sub menu to your template, you need to use a `folder-navigation` tag pointing to a `.block` file containing the markup as shown here:

```
<squarespace:folder-navigation template="submenu" />
```

This tag is different from a normal navigation tag because it automatically pulls navigation data from the page being viewed, and there is no `navigationId` attribute necessary. It also includes active page variables so you can set active states on the active page links within your `.region` file.

UPDATED: JULY, 06 2016

Category Tag

Category Tags in Squarespace are a way to dynamically display the categories from the collection being viewed. Handy for a Blog or Events collection and can assist in navigating large collections.

Similar to a Navigation Tag, you can bind the category information to a template (`.block` file) to create the markup of the categories (when you do, the tag should be self-closed). Although, the `template` attribute is optional, a Squarespace Category Tag requires the `collection` attribute be populated with the collection url ID as shown in this example:

```
<squarespace:category collection="{urlId}" template="category-slugs" />
```

As an alternative, you can omit the template and just create the markup inline by using a non-self-closing version of the Squarepace Category tag and putting your markup inside as shown in this fine example:

```
{.section collection}
  ...
  <squarespace:category collection="{urlId}">
    {.if categories}
      <div class="category-wrapper">
        <span class="category">{.repeated section categories}{name}{.alternates with}, {.end}</span>
      </div>
    {.end}
  </squarespace:category>
  ...
{.end}
```

A Category Tag can also be a great way to 'filter' large galleries of work and can be used in either a `.region` or `.list` file.

UPDATED: JULY, 06 2016

Custom Query Tag

A query allows you to display items from any collection on any page of your site. A Squarespace Query can be filtered by several parameters.

> Important: the contents of a `<squarespace:query>` is not cached and may increase page load time. There is a hard limit of eight queries per page and 100 results per query.

Querying Items

To initiate a Squarespace Query, add the query tag to your collection page with one more parameters. All parameters are optional except for `collection`. All code inside of the opening and closing query tags will inherit the scope set by the parameters.

```
<squarespace:query collection="new-blog" limit="10">

  <ol>
    {.repeated section items}
      <li><a href="{fullUrl}">{title}</a></li>
    {.end}
  </ol>

</squarespace:query>
```

Parameters

- `collection=""` The collection URL ID (slug) to query*(required)*
- `limit=""` The number of items you'd like returned *(number, max 100)*
- `skip=""` The number of returned items you'd like to skip*(number, max 10)*
- `category=""` Only return items that have this category*(comma delimited)*
- `tag=""` Only return items that have this tag *(comma delimited)*
- `featured=""` Filters the return to show only Featured posts.*(boolean)*

Example

This example demonstrates how to create a featured item module that is flexible enough to be used on any .list file. To achieve that flexibility we'll pull in the collection URL dynamically.

```
blog.list
--------------------------

<squarespace:query collection="{.section collection}{urlId}{.end}" featured="true" limit="5">
```

```
  <div class="featured-wrapper">
    {.repeated section items}
    <div class="featured-post">
      {.main-image?}<img {@|image-meta} />{.end}
      <h2>{title}</h2>
    </div>
    {.end}
  </div>

</squarespace:query>
```

UPDATED: APRIL, 07 2016

Custom JavaScript

You can use any custom javascript file or library in your Squarespace Template. Squarespace provides a script loader that minifies and combines your custom scripts, which you can use if you like. Alternatively you can package your scripts using the preprocessor of your choice.

Using the Script Loader

Squarespace's script loader minifies your code and allows you to combine all of your JavaScript files into one, cutting down on `HTTP` requests. Loading JavaScript in Squarespace is very similar to the standard script tag syntax in `HTML`. The syntax is outlined in the code sample below.

```
<squarespace:script src="plugin.js" combo="true" />
<squarespace:script src="site.js" combo="true" />
```

NOTE: The script's `src` path is relative to the template `/scripts`folder.

Using your own Javascript Preprocessor

You can use any javascript workflow tool, like NPM, gulp, browserify or webpack. You can include the compiled code in your `/scripts` folder, and link to those files using regular script tags, or the Squarespace Script Tag.

Including External Third Party Libraries

You can use libraries hosted on an external server or CDN, but we recommend having a local fallback in case the CDN isn't available. Here's an example:

```
<script src="//code.jquery.com/jquery-2.2.1.min.js"></script>
<script>
  window.jQuery || document.write('<script src="scripts/jquery-2.2.1.min.js"><\/script>')
</script>
```

UPDATED: FEBRUARY, 25 2016

ImageLoader

Squarespace comes with a number of built-in facilities for managing images that are uploaded to our system. After uploading an image into a collection, Squarespace automatically creates multiple copies of the image with different sizes. Our image loader will then help render images at the proper size, even on retina displays.

The ImageLoader can also be used to fit or fill an image inside parent container, where it automatically determines which image size to use depending on the current dimensions of the container.

ImageLoader Basics

To eliminate having to guess the appropriate size of an image at load time, you can instead use the Squarespace ImageLoader formatter. Within your template, in the context of an item on our system that is an image, using:

```
<!-- Add the imageLoader JSON-T formatter to an img tag -->
<img {@|image-meta} />
```

will output a tag that has various Squarespace data embedded in the image tag, including dimensions, focal point, and a CDN-ready URL.

```
<!-- img tag is rendered with additional attributes -->
<img class="thumb-image loaded" data-src="http://static1.squarespace.com/static/577e80de20099e0c9b
```

By using this syntax, when the page loads, our imageLoader will inspect the image element and inject the appropriate size image. This allows you use specify properties like height="X", width="X" or CSS on the image. The loader will inspect the tag and always load the appropriate image size.

Running and Refreshing ImageLoader

In order to leverage the imageLoader built into Squarespace, we've got to initialize the script. Basically, we rely on the `data-src` attribute being present as a hook which is automatically generated for us when `{@|image-meta}` is used.

The Squarespace ImageLoader automatically loads the appropriate version of any image depending on its context. However, this context might change due to a window resize or a Style Editor change and the image might require an update. The following example script will run imageLoader on load and resize.

```
// Load all images via Squarespace's Responsive ImageLoader
function loadAllImages() {
  var images = document.querySelectorAll('img[data-src]' );
```

```
  for (var i = 0; i < images.length; i++) {
    ImageLoader.load(images[i], {load: true});
  }
}

// The event subscription that loads images when the page is ready
document.addEventListener('DOMContentLoaded', loadAllImages);

// The event subscription that reloads images on resize
window.addEventListener('resize', loadAllImages);
```

Bypassing ImageLoader

If you would like to load one of these image sizes explicitly, use the following syntax from within an image context:

```
<img src="{assetUrl}?format=300w" />
```

In this case, the image will be the 300w image. Note that this syntax bypasses our imageLoader, and is not recommended for general use. The available image sizes are:

```
original, 1500w, 1000w, 750w, 500w, 300w, 100w
```

Collection/Item Thumbnails

In Squarespace, all Collections and Items can have a thumbnail image associated with them. Sometimes this thumbnail will be implicitly chosen, as is the case with an image item, and sometimes this thumbnail must be explicitly added, as is the case with Collections.

```
{.main-image?}
  <img {@|image-meta} />
{.or}
  <img src="/assets/default-image.png" />
{.end}
```

Cropping with Content Fit and Content Fill

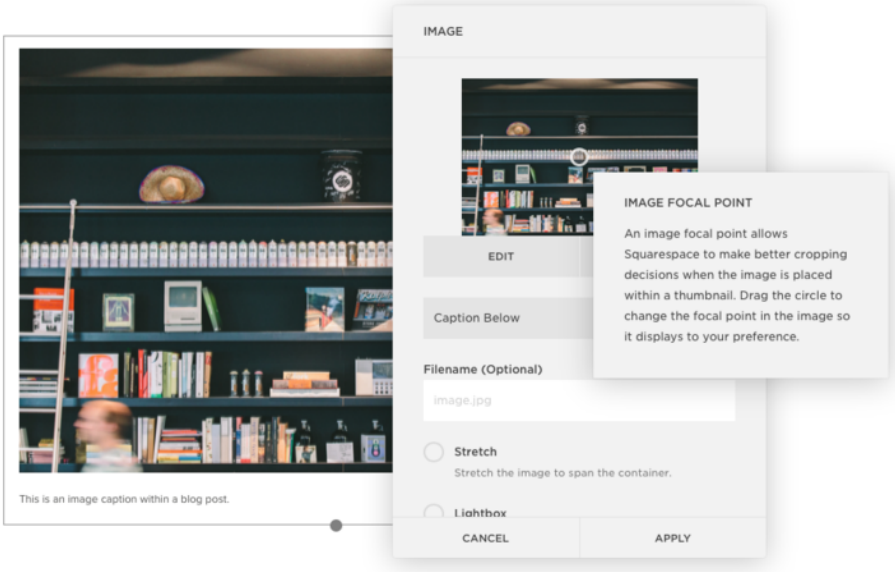
The Squarespace ImageLoader can also handle cropping of an image around a focal point. In CSS you can use `background-size: cover` and `background-size: contain` to tell a background image to fill or fit its element. With the ImageLoader we achieve the same behavior and get the benefits of loading the correct size image by adding a container element.

```
<div class="content-fit">
  <img {@|image-meta} />
</div>

<div class="content-fill">
  <img {@|image-meta} />
</div>
```

NOTE: When using either `content-fit` or `content-fill` wrapper classes, the containing element must have a width and height defined.

Focal Point

You can set the focal point for any image you upload – the point around which the system will always focus the image when using`content-fill`.



Original Dimensions

The following syntax can be used if you need to get hold of the original dimensions of the uploaded image:

```
{.main-image?}
  <img src="{fullUrl}?format=750w" data-image-dimensions="{originalSize}" />
{.end}
```

For example, `data-image-dimensions={500x300}` is created for a 500px by 300px image. You can also break dimensions as follows:

```
{.main-image?}
  <img src="{fullUrl}?format=750w" data-image-width="{originalSize|width}" data-image-height="{orig
{.end}
```

UPDATED: JULY, 14 2016

Local Development

Now you can work on template code locally using the Squarespace Local Development Server. The local development server is a command line tool that sets up a test server on your computer, allowing you to see changes to your template before making them "live."

> NOTE: Prior to developing locally, you'll need a website set up with developer mode enabled. (See Getting Started)

Installing the Dev Server

To begin working locally, you first must download and install the Squarespace Local Development Server. This tool is available viaNode Package Manager (NPM). Here are the steps required:

1. Install NPM: If you don't have NPM installed you'll need to set that up first, by following the NPM installation guide. You'll also need to ensure you have the appropriate permissions configured for NPM.
2. Install squarespace-server: You can do this with the following NPM command:

```
npm install -g @squarespace/squarespace-server
```

Using the Dev Server

First you need to have your template in Developer Mode, and have cloned your site into a folder using Git. (See Getting Started):

```
git clone https://site-name.squarespace.com/template.git
cd template
```

Then, to start Dev Server, run the following command:

```
squarespace-server https://site-name.squarespace.com
```

After starting `squarespace-server`, you will see the following screen:

```
Starting Development Server
```

```
      _
    / /    _
   / / / / \
  / / / / / |   Squarespace Inc.
 | / / / / /    Local Development Environment
  \_/ / / /
      / /
     -

Version 0.1.00
Listening on port 9000
```

The start-up message displays both the version of squarespace-server, and the port on which the server is running.

Now you can open your browser to your site by typing in the urlhttp://localhost:9000. Any changes you make to your local template files will be visible when you refresh your browser.

Advanced Use

You can view all the options with the command:

```
squarespace-server --help
```

If you have a site password, or your site is in trial mode, you'll need to log into your site via the dev server. You can do that with the following command:

```
squarespace-server https://site-name.squarespace.com --run-authenticated
```

Here's a list of some of the more useful options:

```
-h --help                    Show this screen.
-d --template-directory=PATH   Path to template source.
-p --port=PORT               Port that server listens on.
--run-authenticated          Login when starting server.
--verbose                    Verbose logging.
```

Help and Feedback

If you have questions about working locally, or would like to provide feedback on Dev Server, please contact support and be sure to let them know you're using the Squarespace Local Development Server.

UPDATED: AUGUST, 03 2016

Custom Post Types

You can add custom content fields to any post type in Squarespace. This allows a developer to create posts that serve a very specific use case that is not handled by Squarespace's default post types.

This page will outline how to create a post and add it to a collection.

Creating a Custom Post Type

In your template.conf file add a key called "customTypes." Here are the options for the customTypes configuration.

title (required)
The title that your users will see. This will show up in the CMS when you're adding a new post.

name (required)
The name is a way to identify your custom post type in your template code.

base (required)
The Squarespace post type you'd like to extend. The options for this field are:

- "image"
- "text"
- "video"

fields
The data that you'd like to add to the base post type is defined here. Each data field requires a name, title, and type. The type options are:

- "text"
- "wysiwyg"
- "image"
- "checkbox"
- "gallery"

Creating a Custom Post Type

Say, for instance, you run a blog that occasionally has sponsored posts. The code below will show you how to add a "Call to Action Link" to a custom "Sponsored Blog Post" type.

template.conf

```
"customTypes" : [
  {
    "title" : "Sponsored Blog Post",
    "name" : "sponsoredBlogPost",
    "base" : "text",
```

```
    "fields" : [
      {
        "name" : "ctaLink",
        "title" : "Call to Action Link",
        "type" : "text"
      }
    ]
  }
]
```

Adding Your Custom Post Type to a Collection

Once you've created your custom post type you have to add it to a collection, or multiple collections, on your site. Let's take the "Sponsored Blog Post" example from the above section and add it to blog collections on your site.

To do this, we need to open the blog.conf file, which is located in the collections folder. Inside the blog.conf, there is a section called "acceptTypes," which defines the post types that are accepted in that collection. Add "sponsoredBlogPost" to that array.

[collection].conf

```
{
  "title" : "Blog",
  "ordering" : "chronological",
  "addText" : "Add Post",
  "acceptTypes": ["text", "sponsoredBlogPost"]
}
```

At this point your custom post type is ready to use. Given the example above, all you need to do is navigate to a blog in the Squarespace editor, add a custom post, and fill out your custom fields.

Templating Custom Data

The data from custom post types gets added on to a standard item. Once you add a post to your site, navigate to that page and look at it's JSON using the ?format=json-pretty search parameter. Each custom type will be an item with an additional section called "customContent." The structure will look like this (a bunch of JSON keys have been removed from the item for the sake of readability):

?format=json-pretty

```
"items" : [
  {
   "title" : "Sponsored Blog Post"
   "customContent" : {
    "ctaLink" : "https://www.squarespace.com/"
   }
  }
]
```

UPDATED: SEPTEMBER, 29 2015

Template Annotations

Template annotations allow Squarespace 7 to identify each element of your page – like the site title, or a blog post – and allow you to edit those fields right on the page. Annotations are added to your template's markup in the .region, .list, .item, and .block files.

This page will detail each of the template annotations and show examples of how you can add them to your template. Template annotations are optimizations to Squarespace's editing tools. Your template will still work without the annotations.

Content Field

The `data-content-field` data tag is used in various places throughout Squarespace sites. It has several possible values, listed below.

Site Title

```
<h1 data-content-field="site-title">{website.siteTitle}</h1>
```

A shortcut to edit the Site Title. When 'Edit' is clicked, the panel containing the Site Title is opened.

Main Content

```
<main role="main" data-content-field="main-content">{squarespace.main-content}</main>
```

Edit the main content of a page. This attribute usually correlates with the placement of the aria role=”main” attribute.

Navigation

```
<nav data-content-field="navigation-mainNav"></nav>
```

Identifies one or more of the navigation elements on your site. The navigation value has an optional secondary argument, separated by a hyphen, that accepts the id of a navigation.

Connected Accounts

```
<div class="social-links" data-content-field="connected-accounts"></div>
```

Edit your connected accounts.

Title

```
<h1 data-content-field="title">{item.title}</h1>
```

Edit the title of an item.

Location

```
<div data-content-field="location"></div>
```

Edit your website's location data.

Item ID

```
<article data-item-id="{item.id}"></article>
```

This attribute identifies an item that is editable. It usually goes in the .item and .list files that make up your collections.

Collection ID

```
<article data-collection-id="{collection.id}"></article>
```

This attribute identifies a collection that is editable. It usually goes in the .list file.

Open Block Fields

```
<squarespace:block-field id="footerBlocks" label="Footer Blocks" columns="12" />
```

This attribute provides a label for the open block field. If no label is provided, the label defaults to the ID of the block field.

Index Types

An Index is one or more pages that can be visually grouped together on a single page. There are currently two different index types, `grid` and `stacked`. Avenue is an example of a grid index while Pacific is an example of a stacked index. In your `index.conf`file in the `collections` folder you can specify which type of index you're using on your site, which will help the Squarespace content editor optimize the content editing on that page.

- Stacked: Pages within stacked indexes are edited directly on the index page.
- Grid: Pages within grid indexes are typically edited on the individual item page, rather than the index page.

Here's an example index.conf file taken from the pacific index:

```
{
  "title" : "Index",
  "folder": true,
  "fullData": true,
  "acceptTypes" : ["page", "gallery", "album"],
  "addText": "Add Section",
  "icon": "projects",
  "index" : true,
  "indexType" : "stacked"
}
```

UPDATED: FEBRUARY, 24 2016

Style Editor

The Style Editor allows a developer to isolate specific elements of the design and present options to the user in an easy-to-use interface. Using those options, a user can make presentational changes - or 'tweaks' - to those elements without having to know or edit CSS code.

Basic Syntax

Tweaks, which are added to a template's CSS or LESS files, are based on LESS variables and a JSON object that defines the style options. It looks like this.

```
<!-- tweak definitions -->

// tweak: { "type" : "value", "title" : "Page Width", "min" : 500, "max" : 1400 }
@pageWidth: 600px;

// tweak: { "type" : "color", "title" : "Page Background Color" }
@pageBackgroundColor: whitesmoke;


<!-- styles elsewhere using the tweak variables -->

.container {
  width: @pageWidth;
  background-color: @pageBackgroundColor;
}
```

Note: The Tweak syntax is dependent on being commented out. Ensure you have two forward slashes and a space preceding your tweak, exactly as shown above.

If you're not familiar with variables you can read more about them at LESSCSS.org.

Types of Tweaks

Value

A range slider, which returns a unit of measurement, will be added to the Style Editor. Developers can define the minimum and maximum values, default value, sensitivity of the slider as it is moved, and the number of steps included in the slider.

```
// tweak: { "type" : "value", "title" : "Logo Size", "min" : 50, "max" : 150, "step" : 5 }
@logoHeight: 75px;
```

```
.logo {
  max-height: @logoHeight;
}
```

Color

A color picker will be added to the Style Editor and will return whichever color format used in the CSS (e.g. hex, rgba).

```
// tweak: { "type" : "color", "title" : "Text Color" }
@textColor: #444444;

body {
  color: @textColor;
}
```

Typography

A series of typographic options that match the class's CSS properties will be added to the Style Editor.

```
// tweak: { "type" : "font", "title" : "Body Font" }
.body-font {
  font-family: Helvetica, Arial, sans-serif;
  font-weight: 400;
  font-style: normal;
  font-size: 16px;
  letter-spacing: 0em;
  line-height: 1.2em;
  text-transform: none;
}

<!-- assign the font mixin elsewhere -->
body {
  .body-font;
}
```

Checkbox

A class will be added to the body when a checkbox is checked. The active key/value pair defines whether or not a checkbox is checked or unchecked by default.

```
// tweak: { "type" : "checkbox", "title" : "Hide Social Icons", "active" : false }

.hide-social-icons .social-icons {
  display: none;
  visibility: hidden;
}
```

Dropdown

A class will be added to the body depending on which option is selected.

```
// tweak: { "type" : "dropdown", "title" : "Layout", "options" : [ "Left", "Center", "Right" ], "default" : "C

.layout-left .container {
  float: left;
}
.layout-center .container {
  margin: 0 auto;
}
.layout-right .container {
  float: right;
}
```

Image

An image upload field will be added to the Style Editor. This is mainly useful for purely presentational images that are set in the CSS, like background images.

```
// tweak: { "type" : "image", "title" : "Background Image" }
.bg-image {
  background-image: url('/assets/noise.png');
  background-size: cover;
  background-position: center center;
  background-attachment: fixed;
}

body {
  .bg-image;
}
```

Organization

As style options grow more numerous there are a few tools to help organize the style options for users.

Categories

Categories allow you to easily group several style options togather. The syntax is very simple, just add a key value pair to a tweak. If the values in two category key/value pair match they will be grouped together under that title.

```
// tweak: { "type" : "color", "title" : "Link Color", "category" : "Links" }
@linkColor: #FFF;

// tweak: { "type" : "color", "title" : "Link Hover Color", "category" : "Links" }
@linkHoverColor: #CCC;

.link {
    color: @linkColor;
    &:hover {
        color: @linkHoverColor;
    }
}
```

Targets

Targets are visual editing tools that pair an element with its style options. When users access the style editor they can click an element and see all the style options associated with it.

> This is a beta feature. Its syntax may change in the future and there may be bugs when implementing it.

Developers must specify which tweaks to target when a user clicks on an element. Here is a look at the basic syntax. Notice the target key/value pair at the end of each tweak.

```
// tweak: { "type" : "color", "title" : "Link Color", "target" : ".link" }
@linkColor: #FFF;

// tweak: { "type" : "color", "title" : "Link Hover Color", "target" : ".link" }
@linkHoverColor: #000000;

.link {
  color: @linkColor;
  &:hover {
    color: @linkHoverColor;
  }
}
```

The target property does not respect browser specificity rules. Target rules should always avoid ambiguity. For the best results use a single class for the target property and the same class for the tweak's styles.

Show Only When Present

The showOnlyWhenPresent property defines which style tweaks correlate with which HTML elements, and makes style tweaks visible only when those elements are present.

List all the elements that relate to that style tweak, comma separated, as the example shows below.

```
// tweak: { "type" : "color", "title" : "Sidebars Background Color", "showOnlyWhenPresent" : '#left-sideb
@sidebarsBackgroundColor: #FFF;
```

JavaScript Integration

Tweak values and events can be used in JavaScript. This is useful when your scripts access any style of any element that can be changed in the Style Editor.

Get Tweak Value

A tweak's value can be used in JavaScript in two steps. First, in your CSS or LESS document add a key/value pair to your tweak that sets js to true.

```
// tweak: { "type" : "color", "title" : "Base Text Color", "js" : true }
@textColor: #FFF;
```

Then, in your JavaScript, use the following line to get the value of the LESS variable.

```
var tweakValue = Y.Squarespace.Template.getTweakValue('textColor');
```

Tweak Change Event

JavaScript functions might need to execute on tweak change. For instance, if the JavaScript gets a value of an element that can be changed by tweak, the function will need to run again to detect the new value. The syntax below shows how to listed for the tweak:change event.

```
Y.Global.on('tweak:change', function (f) {
    console.log('You changed the ' + f.getName());
```

```
    // f.getName() in this example will return the title of the tweak.
});
```

Tweak Reset Event

All of the tweaks may be reset to their default values using the "Reset" button. The syntax below shows how to listen for the tweak:reset event.

```
Y.Global.on('tweak:reset', function (f) {
    console.log('You reset all the styles.')
});
```

UPDATED: FEBRUARY, 11 2016

Error Reporting

Good error reporting is essential when building and debugging templates. This page explains how to find errors in your template and CSS files.

CSS Errors

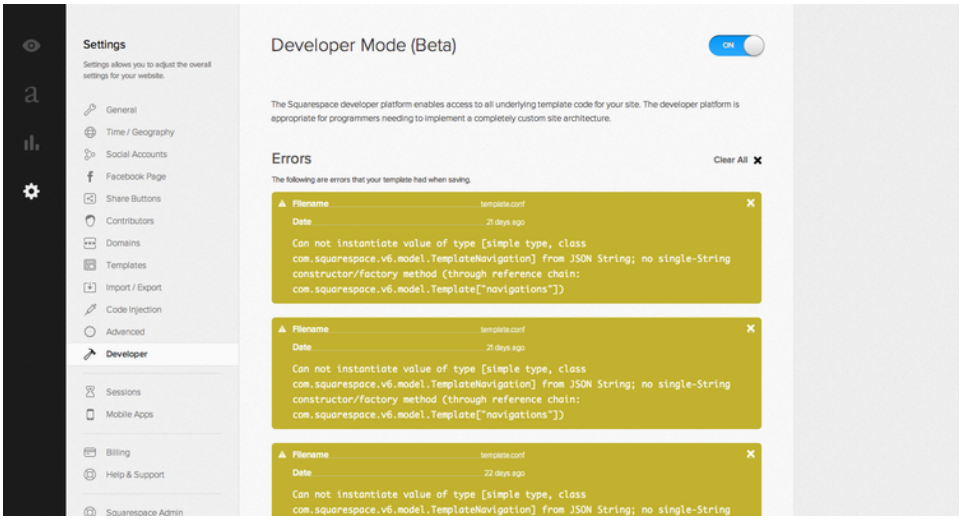
CSS errors (usually LESS processing errors) will be printed at the bottom of the compiled `/site.css` file found at your sites root. Scroll down to the end of the file to see any error messages. If your site suddenly appears to be rendering with no styles, it is a good time to check your `site.css`.

```
Error compiling LESS:

message: Syntax Error on line 425
line: 425

415:     .arrow-icon { cursor: pointer; background: none; width: auto; height: auto; text-indent: 0; }
416:     .arrow-icon.next-slide { background: none; }
417:     .slide-count { display: none; }
418:     .spacer { display: inline; }
419:     .grid-icon { cursor: pointer; background: none; width: auto; height: auto; text-indent: 0; }
420:   }
421: }
422:
423: .gallery-controls { color: @galleryControls;
424:   .arrow-icon { cursor: pointer background: url(/assets/left_icon.png) no-repeat center left; width: 13px
}

425:   .arrow-icon.next-slide { background: url(/assets/right_icon.png) no-repeat center right; }

426:   .spacer { display: none; }
427:   .slide-count { display: inline; font-style: italic; margin-left: 10px; font-size: 12px; }
428:   .grid-icon { cursor: pointer; display: inline-block; background: url(/assets/grid_icon.png) no-repeat c
-9999px; }
429: }
430: .icons-light, .gallery-icons-light {
431:   .gallery-controls {
432:     .arrow-icon { background-image: url(/assets/left_icon_light.png); }
433:     .arrow-icon.next-slide { background-image: url(/assets/right_icon_light.png); }
434:     .grid-icon { background-image: url(/assets/grid_icon_light.png); }
```

LESS Errors are returned in the compiled site.css file at the root of your site

Template Structure Errors

Template file errors (mostly errors in misformatted template configuration files) can be viewed in the Developer tab.



Template Structure Errors are returned in the Developer tab.

JSON-T Errors

Squarespace also reports JSON-T related errors on your front end site directly in a solid black screen. Lower impact JSON-T errors may show in your sites code, rendered inside a comment labeled SQUARESPACE_JSONT_ERROR.

JSONT Errors are hard to miss... they return a black screen of death on the pages they affect.

UPDATED: FEBRUARY, 10 2016

URL Queries

These query strings can be appended to any Squarespace collection URL.

JSON Format

```
http://your-site.squarespace.com/?format=json
```

Returns website and collection data in JSON format.

Prettified JSON Format

```
http://your-site.squarespace.com/?format=json-pretty
```

Returns website and collection data in a prettified JSON format.

Site CSS

```
http://your-site.squarespace.com/site.css?minify=false
```

Returns the compiled, non-minified CSS. The default view of /site.css is minified.

RSS Format

```
http://your-site.squarespace.com/?format=rss
```

Returns website and collection data in RSS format.

Main Content Format

```
http://your-site.squarespace.com/?format=main-content
```

Returns full generated html for pages (not collections).

Category Filter

```
http://your-site.squarespace.com/?category=[categoryName]
```

Returns list of collection items, filtered by given category.

Tag Filter

```
http://your-site.squarespace.com/?tag=[tagName]
```

Returns list of collection items, filtered by given tag.

Month Filter

```
http://your-site.squarespace.com/?month=[monthPublished]
```

Returns list of collection items, filtered by given month.

UPDATED: APRIL, 01 2016

Security Escalation

If you believe you may have discovered a security issue, please contact security@squarespace.com. You may encrypt any messages you send to this email address with OpenPGP using the public key listed here.

-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: GPGTools - https://gpgtools.org

mQINBFbqxUsBEACyTSGDjHUlJe3G5xizK7lzzlUKAjYLHXBbqnlh6lAprR1jA9nR
a4HEwY0A+IrMuNP3r5EZ/xn4LfuHaBkpfXoHyKQnjN5FULEsak5HN28VrlMuWQBJ
TrN7Mqne1FnsP64j5gvog/Uh23Ih2siCYLB4I7oow/saHKBUrT0wVEv357eqdile
i2geFYyxwzxs5mc6ub8nB3ZWpL/IJUMGtcBdI7XGvbbODxjOZRytPNrk9vH2eKM9
U/1A3Cjcli/YztHFxWpQeVtNTeM1IY/dYn3nHV/bmzAxJ/3XRB2SIgJPiBq7FXeu
JMAOfvuss4lMo7cfY0mW3PsMhDCQrSbZpFpZVmKGiD6NYYw4EYUydKlj85gTxKjE
eKQIFvHN6/ny6EPPOnpPK/VNpuN6zaAQskxz4ELlnFtmJ7/a4hBGdPhnbZr557QR
1gJNkLFVyxqjzkzNIWqsVvkyXmi+hjZCrrMJQ9rjlCZalXRt9n3mmz18VtJDCG+W
PQPS3P0d6chCuYyQ8tZFOxLC0kG+YOOk5XKYpF73cZU+ScGA14S2GRSBvckYFPqB
WeZjp6fT++2uCg4hj/pNazti2jlWYiCxI4+M3YRmw6Qj3XsY1EVJmoWlWDieVVga
R3bE7+Nma3SihsDUJnS5GZWWff+X/AnSfFgeS7B9pb+VGE9D2QuQqvqtlwARAQAB
tChTZWN1cml0eSBUZWFtIDxzZWN1cml0eUBzcXVhcmVzcGFjZS5jb20+iQI9BBMB
CgAnBQJW6sVLAhsDBQkHh3EABQsJCAcDBRUKCQgLBRYCAwEAAh4BAheAAAoJENH3
YezSrIJkNQMP/1t0t60/Y02V1z8fdjqEqZd8FcVxVAi7WAVDUoCFbTOpopLPryNX
zS3OIrTm0UekO0X0s/vCFOhYkP2jDtSLSak9EHJMQ8QeedHVqBq7L/Se4Yie5lZ4
DeO2gKPrA+ZeemUSdHFF4B6q3ZNTrggpwuhUsMevtdVOYihOYaBclq1l0wcEEEe2
+I9T7mDjuY3ERKTzN1mYLykYryn4Xi6YubWciCLwjwmnCMJuk0fMACVCKfCtCRaJ
PBerAJOVJZSS/JuNexLg5ooJoQzxEuWUHAt4ihWTfXusbgxXXLeWC+W2zXSkgQ9d
NluDuEsZlZ1Onc1hcF+VgZAdToTZl0jFXq7Bb8UkXNCYYKXNQP+mSEDbvfk8nB9t
6guf+HltyuLQ8CXTJEPu9fIedYC0YitEhCw5wXQtNU6PwWGOa4dP7ALIo22d2TEQ
kWMncrVG/7Gsxebb1e2pexIfvh6p3grTmT3D8wrWdrbx/L0WYZz8INsiDr289iWi
yfFqGF9M3lnqVL0vCG/WqF6F1j//KyBNoRLxiVr/oUPcnBH7cA8LdEaUX4krSV/U
P6ffLgz+zhDR+h2jMnFsUYpOXHcSfssB9B0ccJsA9iXGHlZ4d0XUiYKDbBTI2rhX
8lhPH+e+69vvRRKH5rThetEkCJut1U9JVS83r0NuPoxf0/8fHjiHX+1tuQINBFbq
xUsBEAC9p/O+anrlpqBrgFkirABy3pE8JqNmLdPK702Sxq+Ux8yTGoZ9BQPkM+EZ
mEnnVXQspkEVQhy7EShmah/QDGn8cLtz0mqcbpLYuqUHtbzcIXZv0COkupcSNcWG
K1rUorzf0cZ0LzCAVs/lyQ8IWE5eCAv+AoVfFA8AtrDrV4E9k/lxy+xA5xY9EaiF
3zcT61+1IQZDYD16h9PVNTAOUJHaAD9rGJrwG41QMZhdd7/tc2/u56nPd74ew2eq
1AtPskphsAd4cJ5UCuykzAzEpEm0yNF0RMzg407OpNoaFrfPQQk3iAtwD8Fl9Oh+
+GL3E20DDE3ZUiy+mnoHr/LKtBhmq5iGz7A6RlpW91tmQrbLX6pl5K3fHrpBHvKA
zV70nela2m1d9HQR7+4Ecn0Ecnc0MPnpQdCaO6BmsXxWODBymotRMnvf7cYRU93u
pJ/gEF85LG19qLKbNg0oTkS+QGwa2Ghs8jbm2V2nO4LBzN0DBK5uFIL7V+BAZ6iX
Uz9dNGyk6wx69/BtNm1FhIgG0hV2xndIktRklUbpm8QqkL8gkUFI/YI1eYxR22bF
jVPbn8/HmB/2ePxccpLCccdH28Ha/5xlexWM9cfq5bbuarA/quMuQ/2ZsnO0rC0d
hjfA+hkJ91/Z3tzzwrYmsU4307LWjY9Dnh7dD3St3jSHuP3TywARAQABiQIlBBgB
CgAPBQJW6sVLAhsMBQkHh3EAAAoJENH3YezSrIJkcoQP/1NStWZ3nwBiv2Qsh04k
Q6TLWj1Dv+4x1Cobu8iiqWDXRt+d26Q8WjAafQ4RmrTlUhjRqabyn+vCf4T12AGg

```
hbkQrkOnNUauy+U2ldpU1C/htrVPAPd9lCq9y/4KJGfW9EgEVlA7w+sq1ewqAteV
EQPuKpX3LP4iAtFG59gLAlGSv/8Et0/OtMKDD8c95ssEjilYON6WWDVHO2w6jJgo
uMClG6IS6+QHGEq7Gf1IE0xKl/HtDuvHLHXK8JZvwdigyUUSTcLyh7A5qAHrDs9E
zbUbwqL0xV1936odH0sGeOIwO3YA0zr+9g4RHooSqV++xX3b41WzqsBsOQKZ0IEP
jIuIv1HD9M2lu66uVM5+bFI2CAId+NRLNzg0WYYAjUwmb9Fi9Wd3ZzUa2RrM/x6C
sQ+db6Kz9zcXkVFMcNmhpz9SnVDUzpRd+nFOm5eOJprei3XDDx1EY2rpDb56m6R1
dpN3+u7SfjSorjTq7/AFCmj5jZEJ6Enr/04MfYxgyPOhSuIBMqCt/N78tsYFMXQz
3vAslDVGqsh2cd4LrIQIyziPlt5oK60BP6ZKaj1fKvTzvNAjnKufyodfw2tXHMzt
TYHf8XS41TvhE1QhMefHKmbhtpC7V1Y7hoRg6VJyTmvh1i9ZhNs9h+V363OmX2Hp
z5c37E/ulWfAJR2Q0pPu1K9A
=QTYu
-----END PGP PUBLIC KEY BLOCK-----
```

UPDATED: MARCH, 17 2016