Sitetheory

Release 1.0

Oct 15, 2019

Contents

Sitetheory is a robust and scalable framework to build beautiful and highly functional platforms. It is built to empower vendors, developers and design partners to rapidly create cool websites and powerful mobile apps. Sitetheory is the first vendor among many, with an admin website that functions as a website builder, CMS (Content Management System) and CRM (Customer Relationship Management). But other vendors can leverage the existing UI, API, Content Types, and Functionality to create their own identical (white labeled) or distinct platforms or just to make individual websites.

CHAPTER 1

Frontend

On the front end, all default Sitetheory themes use a proprietary Stratus Javascript framework to manage the UI/UX look, feel and functionality (custom themes can bypass everything if desired). Stratus has a few core features, and also loads other dependencies through Require.js. Via Require.js, Stratus loads some helpful javascript libraries used extensively throughout the site, including Underscore.js and Angular.js. Angular is used for model management (fetching and persisting data/entities from the API). Angular was chosen over React because it gives designers absolute and direct control over the look of everything from Twig template files (developers haven't hard coded bad design in obscure javascript files designers can't find). Empowering designers means faster and more beautiful design.

More details can be found at the Stratus Docs.

CHAPTER 2

Backend

On the backend, Sitetheory utilizes Symfony as a modern framework, Doctrine for database and entity management, and Twig for HTML templating. As a CMS framework, Sitetheory just manages page requests on the server side. The current URL determines the correct Content (page) to load. Each Content is associated with a specific Content Type, e.g. Article, Profile, Landing Page Stream, etc. Sitetheory will load the correct Controller for the current Content Type, as well as the appropriate Twig Template. Individual Themes or websites can extend or overwrite the Controller or Templates to allow for endless customization.

More details can be found in the documentation below.

CHAPTER 3

Versions

A new version will only be created at the moment when Backwards Compatibility is broken.

CHAPTER 4

Overview

@TODO: review and update to ensure this is all still valid based on changes to stratus and standards since this was written in 2017.

4.1 Introduction to Components

Sitetheory is designed to make it easy for designers to create beautiful websites that are highly interactive and functional. We wanted to separate design and code as much as possible, so that a designer could easily build HTML/CSS without having to stumble around intimidating code. And yet, we want to allow hard core developers unlimited creativity to implement complex javascript if necessary. To make this possible, we've adopted the Angular framework (full MVC) which has a great templating system for designers, with beautiful pre-built components for the most common use cases and a few components of our own for our custom needs. Then either the designer or a programmer can add logic to their design using Angular syntax, pull dynamic data from APIs, and create a rich experience. At any time the designer can go into an existing dynamic page, and easily edit the design without being too concerned about creating development bugs. If a developer needs to implement complex features, they have full access to the javascript through our Stratus framework, or they can use require.js to require third party libraries and implement any feature they want.

A component could be a simple display field to show the value of an entity, a text field that allows editing the value of an entity property, or it can be like a complex media selector that shows you all the elements you have selected and allows you to upload or select new media. Components render a template and add functionality to the page so the designer can control the user experience. Most components are set to auto-save changes, so the experience is much more responsive than traditional forms. Components are used extensively throughout the CMS admin and Live Edit mode.

See our Stratus documentation for specifics about how we use Stratus to manage the front end Javascript.

See our Stratus-Components documentation for specifics about custom components we already built.

4.2 Angular

Sitetheory implements Angular 1 to display and edit data on any website. Angular's Material.js also provides a lot of prebuilt components, general CSS and a Javascript framework to help rapid development and a general base for creating interactive sites. Generally speaking, Angular replaces Bootstrap and jQuery.

4.3 Implementing Components

4.3.1 Component Options

In addition to all the standard Angular options, the following options are the most common basic options used in our system.

- **ng-controller** (*string:required*) This tells Angular to use our standard *Generic* controller which fetches and binds the models to the current scope, e.g. *ng-controller="Generic"*. This standard *Generic* controller is good enough for most situations, but if you need a fully custom implementation, you can declare one in a <*script*> tag above (see example below).
- **data-target** (*string:required*) This is the name of the entity that the RESTful API will target, e.g. *User*.
- **ng-model** (*string:required*) This is the property that is being edited, e.g. *model.data.name*
- **data-api** (*json:optional*) This is an optional json array of settings that will be passed to the API via the Convoy's Meta property, e.g. *data-api='{"options":{"showRouting":true}}'*.

4.3.2 Component Properties Available

Inside an Angular controller scope the following objects, methods and properties can be accessed, e.g. <*span*>*{{ model.data.name }}*</*span*>

- **collection** (*object*) This is an object that is returned from the API when no specific ID is requested. It contains various methods and properties, including an array of models.
- **collection.meta** (*object*) This is meta data that was returned from the API with important information about the entity.
- **collection.models** (*array*) This is an array of models returned for a collection. The structure of each model is the same as when an individual model is returned
- **model**(*object*) This is an object that is returned from the API when a specific ID is requested. It shares the same data structure as an individual model inside a *collection.models*. It contains methods (e.g. save, fetch, sync) and all the *data* for the model's properties.
- **model.save** (*method*) Initiate this method to save a model.
- **model.fetch** (*method*) Initiate this method to refetch/refresh the model.
- **model.sync** (*method*) This is the manual method to interact with the API (not recommended). Save and Fetch use this method internally.
- **model.data** (*object*) This is where all the data for the model resides.

4.4 Examples

4.4.1 List

NOTE: below is sample HTML, but a lot of the outer HTML is reusable in Twig by extending the ListBase. The raw HTML will be shown first so you understand the big picture, and the Twig implementation will be shown second if you want .

RAW HTML

```
<!-- The ng-controller is the name of the API that will be called, e.g. ListApiController -->
<md-list ng-controller="Generic"
data-target="User" data-api='{"options":{"limitContext":true, "showProfile":true, "showMailLists":true}}'
layout-padding ng-cloak>

    <!-- Proggress Bar -->
    <md-progress-linear ng-if="collection.pending" md-mode="indeterminate"></md-progress-linear>

    <!-- Header -->
    <div layout="row">
        <div flex="5"></div>
        <div flex><h2>Name</h2></div>
        <div flex><h2>Profile</h2></div>
        <div flex><h2>Permissions</h2></div>
    </div>

    <!-- List Body with Repeating Rows -->
    <md-list-item
        ng-repeat="model in collection.models"
        layout="row"
        layout-xs="column"
        layout-sm="column"
        layout-align="space-between center"
        layout-wrap>

        <div flex="5">
            <md-button href="{{ collection.meta.attributes.editUrl }}?id={{ model.data.id }}" aria-label="edit" class="md-fab md-primary md-mini white-svg">
                <md-icon md-svg-src="/Api/Resource?path=@SitetheoryCoreBundle:images/icons/actionButtons/edit.svg"></md-icon>
            </md-button>
        </div>

        <div class="user" layout="column" flex>
            <h4><a href="{{ collection.meta.attributes.editUrl }}?id={{ model.data.id }}">{{ model.data.bestName }}</a></h4>
            <!-- Convert unix timestamp to readable date -->
            <div>Created {{ model.data.time*1000 | date:'medium' }}</div>
        </div>

        <div class="profile" layout="column" flex>
            <div>
```

(continues on next page)

(continued from previous page)

```
                <span ng-if="model.data.profile.lookupValues.gender">{{ model.data.profile.lookupValues.gender }}</span>
            </div>
            <div ng-if="model.data.profile.mailLists.length > 0">
                <span ng-repeat="mailList in model.data.profile.mailLists">{{ mailList.name }}<span ng-if="!$last">, </span></span>
            </div>
        </div>

        <div class="permissions" layout="column" flex>
            {{ model.roles.join(', ') }}
        </div>

        <md-divider md-inset ng-if="!$last"></md-divider>

    </md-list-item>
</md-list>
```

TWIG HTML By Extending the ListBase

```
{% extends 'SitetheoryCoreBundle:Core:ListBase.html.twig' %}
{% set stratusTarget = 'User' %}
{% set stratusApi = '{"options":{"limitContext":true, "showProfile":true, "showMailLists":true}, "q":"foo"}' %}
{% block listHeader %}
    <!-- HTML header-->
{% endblock listHeader %}
{% block listRow %}
    {% verbatim %}
    <!-- HTML for individual repeating rows with access to the `model` data -->
    <div><a href="{{ collection.meta.attributes.editUrl}}?id={{ model.data.id }}">Edit</a></div>
    <div>{{ model.data.bestName }}</div>
    {% endverbatim %}
{% endblock listRow %}
```

Javascript

If you need to define custom functionality, you can easily create a custom controller that utilizes the services of the default *Generic* controller. Then you either define the *ng-controller* manually, or if you are using the ListBase, you can define your own controller, e.g.:

```
{% set stratusController = 'FooController' %}`
{% block script %}

    {{ parent() }}

    <script>
    (function (root, factory) {
        if (typeof require === 'function') {
            require(['stratus'], factory);
        } else {
            factory(root.Stratus);
```

(continues on next page)

(continued from previous page)

```
        }
    }(this, function (Stratus) {
        Stratus.Events.on('initialize', function () {
            Stratus.Apps.Generic.controller('FooController', function ($scope,
↪$element, registry) {
                // Make API call to the target entity (registry prevents duplicate␣
↪calls)
                $scope.registry = new registry();
                // digests the HTML $element to find the data attributes defining the␣
↪options
                $scope.registry.fetch($element, $scope);

                // CUSTOM CODE BELOW HERE------------------

                // Make a Custom API call to some other User entity...
                // NOTE: there is no $scope passed in the fetch options, but we␣
↪define entity in $scope so {{ user }} can
                // be referenced in the angular HTML.
                $scope.user = $scope.registry.fetch({
                    // API Entity (required)
                    target:"User",
                    // Fetch one specific ID (optional)
                    id:1,
                    // Call the API and fetch an object on load (so you can save)␣
↪(optional)
                    manifest: false,
                    // Specify if the results should be stored in the registry (in␣
↪case you need something unique
                    decouple: true
                });
            });
        });
    }));
    </script>

{% endblock script %}
```

4.4.2 Edit

```
<!-- Targeting the Article entity API for the specified ID -->
<div ng-controller="Generic"
    data-target="Article"
    data-id="35558"
    data-manifest="true"
    layout-padding ng-cloak>

    <div layout="row" layout-xs="column" layout-sm="column" layout-align="space-
↪between center" layout-wrap>

        <md-progress-linear ng-if="model.pending" md-mode="indeterminate"></md-
↪progress-linear>

        {# Example: define variable for this scope #}
        <div flex="5"></div>
        <md-input-container flex="95" ng-show="model.completed">
```

(continues on next page)

(continued from previous page)

```
            <!-- set a variable unconnected to the model -->
            <md-switch ng-model="showHints">Hints</md-switch>
        </md-input-container>

        {# Example: listen to defined variable for this scope #}
        <div class="hint" ng-show="showHints" flex="100">
            This hint will show when showHints switch is true.
        </div>

        {# Example: help and generic input #}
        <stratus-help flex="5">Lorem ipsum dolor sit amet.</stratus-help>
        <md-input-container flex="95" ng-show="model.completed">
            <label>Title</label>
            <input ng-model="model.data.contentVersion.title" type="text" required>
        </md-input-container>

        {# Example: basic date picker #}
        <div flex="5"></div>
        <md-input-container flex="95" ng-show="model.completed">
            <label>Display Date</label>
            <md-datepicker ng-model="model.data.contentVersion.timeCustom"></md-
↪datepicker>
        </md-input-container>

        {# Example: Select with options hydrated from API #}
        <div flex="5"></div>
        <md-input-container flex="95" ng-show="model.completed">
            <label>Genre</label>
            {% verbatim %}
            <md-select
                ng-model="model.data.genre.id"
                ng-controller="Generic"
                data-target="SiteGenre"
                md-model-options="{trackBy: '$value.id'}"
                required>
                <md-option ng-repeat="option in collection.models" ng-value="option.
↪data.id">{{ option.data.name }}</md-option>
            </md-select>
            {% endverbatim %}
        </md-input-container>

        {# Example: auto-complete with chips #}
        <div flex="5"></div>
        <md-input-container flex="95" ng-show="model.completed">
            <md-chips
            ng-model="model.data.profile.mailLists"
            md-removable="true"
            placeholder="Add Mailing List"
            flex="100">
                {% verbatim %}
                <md-chip-template class="mailList">{{ $chip.name || $chip.data.name }}
↪</md-chip-template>
                <md-autocomplete
                    md-items="mailList in mailLists.filter(query)"
                    md-item-text="mailList.data.name"
                    md-selected-item="selected"
                    md-search-text="query"
```

(continues on next page)

(continued from previous page)

```
                    md-min-length="0"
                    md-no-cache="true"
                    placeholder="Pick a Mailing List">
                    <md-item-template>{{ mailList.data.name }}</md-item-template>
                    <md-not-found>No Mailing Lists Found...</md-not-found>
                </md-autocomplete>
                {% endverbatim %}
            </md-chips>
        </md-input-container>

        {# Example: Froala text editor #}
        <div flex="5"></div>
        <md-input-container flex="95" ng-show="model.completed">
            <label>Body</label>
            {# leave `froala` attribute empty to use default, provide value "froalaOptions" to use Stratus defaults, or pass in a JSON attribute of valid Froala options from their documentations #}
            <textarea froala="froalaOptions" ng-model="model.data.contentVersion.text"></textarea>
        </md-input-container>

        {# Example: Autosave is enabled by default in most contexts, but if you need to manually save the model you can do it this way #}
        <md-button aria-label="save" class="md-raised md-primary white-svg" ng-show="model.completed" ng-click="model.save()">Save</md-button>
    </div>
</div>
```

4.4.3 Fetch Content Pages

This is a simple way to fetch all types of Content pages (no restriction on `ContentType`, e.g. Articles and Profiles co-mingled)

```
<div id="list-container" ng-controller='Generic' ng-cloak
     data-target='Content'
     class="clearfix">

    <md-progress-linear md-mode="indeterminate" ng-show="collection.pending"></md-progress-linear>

    <div class="st-grid st-grid-tablet column20" ng-repeat="model in collection.models" ng-sanitize="true">

        <div class="related-item">
            {% verbatim %}<div class="related-image" style="background: url({{ model.data.version.images[0].url || '' }}) no-repeat center center; background-size: cover;">{% endverbatim %}
            <a ng-href="{% verbatim %} {{ model.data.routingPrimary.url }}{% endverbatim %}"><img src="{{ asset('bundles/sitetheorytemplate/images/common/shapeholder-square.png') }}"></a>
            </div>
            <div class="related-date font-primary" ng-bind="(model.data.version.timeCustom || model.data.time) |moment:{format:'MMMM Do YYYY'}"></div>
            <h2><a ng-href="{% verbatim %} {{ model.data.routingPrimary.url }}{% endverbatim %}" ng-bind="model.data.version.title"></a></h2>
```

(continues on next page)

(continued from previous page)

```
        </div>
    </div>
</div>
```

4.4.4 Fetch Only Articles

In the example above, if you wanted to only fetch the Articles you would target the Article ContentType only:

NOTE: You could specify any content type in the `data-target` field, e.g. `Profile`, `Event`, etc.

4.4.5 Fetch Articles by Tag

In the example above, if you wanted to only fetch the Articles associated with a specific Tag, you can modify the `data-target` like this: */Api/Tag/1/Article*

```
    data-target='Tag/1/Article'
```

Or Dynamically with a Twig Variable:

```
    data-target='Tag/{{ content.tags[0].id }}/Article'
```

If you wanted to fetch content for multiple tags, you can specify the tag IDs in a comma separated list. Note this just sends an API call with the query variables `/Api/Article?tags=[1,2]`:

```
    data-target='Article'
    data-api='{"t":"1,2"}'
```

This would give you everything that is assigned to a stream: /Api/Content/12345/Asset/Content This will take whatever tags the stream has, and do the same query, e.g. find Stream 12345 and get the content that are associated as assets (via the tags). You could also change this from Content to Media and it would find all media associated with Stream 12345.

In the examples above, if you want to limit the records returned or sort them, you can specify this in the `data-api` variables (See API Overview of Advanced Options.):

```
     data-api='{"limit":5, "sort":"title", "sortOrder":"ASC"}'
```

4.5 # VALIDATION

The *validate* directive enhances **'Angular's internal form<https://docs.angularjs.org/guide/forms>'_** by using the **'Angular ngMessages<https://docs.angularjs.org/api/ngMessages/directive/ngMessages>'_** system to allow custom validation in addition to the Angular defaults validation like *required*, *min*, *max*, *email*, etc. This *validate* directive adds several new validation methods that can be triggered for inputs by including the requirements as options.

- stringlarray *validateInvalid* One or more invalid values not allowed. Can include scope variables that will be evaluated, e.g. *validate-invalid='[model.data.nominatorName, "foo"]'*
- stringlarray *validateValid* One or more values that are valid.
- string *validateComparison* A scope variable comparison that will be evaluated, e.g. *model.data.nominatorName != model.data.nomineeName*. NOTE: if the comparison value evaluates the current model value, e.g. model.data.nomineeName this is evaluates after the viewValue is updated but BEFORE the model is updated, so it won't work with the timing.

The ng-message validate key will be set if a specific validation fails. If more than one validation scheme is set, we will also show if any of them fail: - *validateComparison*: if the comparison was false. - *validateInvalid*: if an invalid value was provided. - *validateValid*: if a valid value was not provided. - *validateAny*: if any of the validations fail.

Example:

```
<input name="nomineeName" ng-model="model.data.fooName" placeholder="" required␣
↪stratus-validate validate-comparison="model.data.foo != model.data.bar" validate-
↪invalid="['baz', 'rab']">
    <div ng-messages="Nominate.nomineeName.$error" ng-messages-multiple role="alert">
    <div ng-message="required">Please enter a name.</div>
    <div ng-message="validateComparison">Please do not nominate yourself.</div>
    <div ng-message="validateInvalid">Baz and Rab are not valid values.</div>
    <div ng-message="validateAny">Ya you really messed up.</div>
</div>
```

4.6 Content Types

4.6.1 What Is a Content Type

Every page is a content which is associated with a specific Content Type, e.g. Article, Map, Form, Video, etc. A Content Type is owned by a specific Vendor, and references a Controller that resides in a specific Bundle. The Controller is the PHP code that determines the actions for a specific page (and usually tells the page to display the content of the page according to the layout of a Twig template by the same name).

4.6.2 Restricting to Services

A Content Type is also associated with a specific service, and sites will have access to any Content Types belonging to services that they are subscribed to. For example, **Article** is Content Type that is available to everyone. But **CMS** is a service with a lot of Content Types that are only available to Sitetheory, and all those content types are the pages that power the CMS Control Panel, e.g. Dash, Aerial Menu, Content List, Editing pages, etc.

4.6.3 Functionality Content Types

Many Content Types are pages that interact with multiple entities, e.g. a Stream is a list page that shows all the content that is tagged to that Stream, which may include Articles, Videos, Maps, Images, etc. Or a User Sign-In page authenticates a specific User from the User entity, but it isn't creating or displaying information about that entity.

4.6.4 Entity Content Types

Other Content Types will actually be the entity itself, e.g. the details page of an Article is an Article entity that displays just that one article's content. Entity Content Types must have define an Entity associated with it, so that multiple instances of this entity can be persisted to the database, e.g. if you plan on writing more than one article each article will be a separate database record for each.

4.7 Content Versioning

Versioning allows an entity's revision history to be tracked and to control which version is published. The best example of this in action is seen when editing content (i.e. pages or modules on a site). Content versioning is triggered under

two cases: 1) when Content is edited by a new user or 2) it has been more than a set period of time (e.g. 30 minutes) since the last time it was saved. If either condition is met, a new unpublished version of the content is created. The new version can be previewed on the site in "Preview" mode, but will not appear on the live site until the version is "Published".

4.7.1 Best Version

The website automatically chooses the best version of Content (and other versionable entities) to display based on the context. On the live site, the system always chooses the most recent published version (not in the future), which means it shows whatever version was manually published. But if you are browsing the site in "preview mode" (e.g. you click preview in the admin and the site has a preview bar in the header) then the site will show the last "edited" version (whether or not it was published). This lets you preview your content before publishing. For obvious reasons, the admin control panel will always show the last edited version (just like preview mode on the live site) so that you can see the current state of changes. If you need to edit the published version, you will view the version history and find the version marked as published, then edit that version (which will create a new version based on the published version).

4.7.2 How Versions Work in the Code

All pages of the website are a `Content` which has an association with on or more `ContentVersion` entities. These `ContentVersion` entities (and the content specific entities that are associated with that version) contain the unique content. See the overview of how Pages work for more details.

Since versioning is a key part of the CMS, our framework makes it easy to make an entity "versionable". See the **:namespace:'SitetheoryCoreBundle/Entity/Content/ContentVersion'** or **:namespace:'SitetheoryCoreBundle/Entity/Design/Design'** as examples of how to implement this in different ways. We use special Traits to make this easy, and minimize redundant code.

4.7.3 Interacting with Versions

There are two ways to interact with versions: Fetching and Editing. The versionable entity can be setup to do one or both of these things, depending on needs of the parent.

1. **Fetch the Version** A versionable entity may only interface with a parent entity when the parent needs to fetch the data (e.g. displaying the correct version of information on the website).
2. **Edit the Version** Or a versionable entity may also need to be edited in conjunction with the parent.

4.7.4 Types of Versionable Entities

Independent Version

Any entity can be made to be versionable, and doesn't require that it is accessed by one single parent. Other entities that call it can use it's repository to find the live or preview version of the entity in question. An example of an independent entity is the `Design` entity. There is only one Design instance at a time, but there may be many versions.

Fixed Version

Fixed Version entities are versionable entities that need a fixed reference id (e.g. multiple Routing records point to one Content), but you need that entity to be versionable. To accomplish this, you will set a fixed parent entity (e.g.

Content) that references a single versionable entity of itself (e.g. ContentVersion). The version holds all the data for the fixed entity. The parent version will then dynamically find the right version to edit or display.

4.7.5 Parents of Versionable Entities

Versionable entities are usually referenced by one or more parents. And the parent may reference multiple associated entities, which may include one or more versionable entities. For example, the `Site` entity references `Design` and `Settings`. If `Design` is versionable, there won't be a specific association. `Site` will have a OneToMany association with all the `Design` versions, but it will only have a container for the single `Design` version (e.g. $site->design). This will have to be manually associated based on the version that is requested (e.g. Live, Preview, or version ID). The versionable entity's repository (`DesignRepository`) will use the VersionRepositoryTrait which will include the necessary method to associate the correct version based on the live or preview mode, i.e. associateVersion().

4.7.6 Editing Versionable Entities

When it comes to editing the versionable entity, they can be edited independently like any other entity (if that makes sense). So in cases like `Design` it makes sense to have an independent form to edit the design fields, without any reference to the `Site`. And likewise, if you edit the `Site` you don't need to edit the `Design`. So nothing more needs to be done.

But in cases you may want to edit both entities in the same form, or there is a fixed relationship between the parent and the versionable entity. In those cases, the parent will need to register the entities that it wants to display in it's own editing form (e.g. this is how `Content` references `ContentVersion`). The parent entity will add a method for `getEntityVersion()` that returns an array with the key of the property containing the entity and the value including the full path to the entity namespace.

4.7.7 Publishing Versionable Entities

If an entity is only going to be edited from it's single parent (e.g ContentVersion) the custom form type for that versionable entity should not have a timePublish() field, because the parent will manage that when it dynamically loads the versionable entities that are registered in the getEntityVersion() method (this is all handled in the EditControllerBase).

But if an entity is going to be edited independently (e.g. Design), then it needs it's own publishTime in the custom form type. This will work great, because all publishing does is set a timestamp.

4.7.8 How To Make an Entity Versionable

See the tutorial on /1.0/Tutorials/How-To-Make-An-Entity-Versionable for specific examples and instructions.

4.8 Features

4.8.1 Platform

Sitetheory is the primary vendor of the CMS, but it will have a site like any other client. And the admin that all its clients use will be pages from its website (associated with Sitetheory's siteId #1). But any client could theoretically be subscribed to the "platform" features, and get their own pages as well. This would allow designers and developers to have their own clients using their version of the pages.

4.8.2 Accounts

Shared Accounts

- All accounts are shared across the entire platform, but restricted to one or more sites, with specific privileges granted by the administrator.
- If an account is created on one site, and then the user visits another site on the Sitetheory platform, the user can register on the new site. But instead of creating a new account, the user's existing account (associated with the email) will be granted basic access to the second site. This allows them to keep a single identity across all the sites. There could be edge cases where a user may want a different profile for a special site (e.g. to mask their identity), in that case they can create a new account for a different email address (but that should be rare, and is a small inconvenience for the overall benefit of a shared account system).
- By default a new account has basic access to the site that it was created on, but the administrator can promote the user to be a moderator on the site, or an administrator with varying degrees of CRUD access on a page by page basis, or an administrator over multiple sites.

4.8.3 Administration

Multiple Sites

A user may create one or sites and have access to edit them all by viewing their Sites list. A site may be shared with one or more additional users, so that the owner can give different levels of administrative access to a team of people. And other users can share their site with you so that you have specific permissions to edit their site. All accounts are equal, so you can promote a visitor's account to become a moderator or an administrator of your site, or of multiple sites.

Billing

- A user may create one or more billing methods, which can be associated with one or more sites. This allows an agency, broker, or other organization to manage the billing for multiple sites.
- Subscriptions will be managed for each site, and billed to the billing record selected. This allows each site to have unique subscriptions, but they can all associate with a single billing record. Or each site could be associated with different combinations of billing records. This allows billing to be kept up to date in one location, but serve multiple sites.
- Access to billing records may also be shared with one or more additional users if you want to delegate billing updates to other members of the team.

4.8.4 Editing

Versioning

- When you are save a page the CMS will create a new version if it's been more than 30 minutes since you last saved or if it was last saved by someone else. This ensures that you don't overwrite other people's work and that you have a record of previous versions that you can revert back to or reference if necessary.
- You can view all the previous revisions of a page in the versions tab. You can load any version to reference previous work or revert to that version. If you edit a previous version it will save as a new version, which will be the latest staged version. If you want to publish a previous version (make it the live version), you simply click the publish button and select the current date and time.

Auto Save

The CMS will also auto save your changes as long as you are on the page. That means if you are working on a page, and your power goes out, your changes will still be saved (even if you never manually saved). These auto-saved changes will create new versions if it's been more than 30 minutes since you manually saved. So when you first start working a new version will be saved. And then after you have been working for 30 minutes a new version will be saved. If you make a mistake 15 minutes later, this will allow allow you to review previous versions.

Publishing

- You can easily see if you are editing the live published version of a page by looking at the publish button. If this version is the latest published version, the button will be green and will say "Published". Otherwise the publish button will be orange to alert you that there are unpublished changes.
- When you publish your changes, you can select a date to publish. This will default to the current time, if no publish date is set. But it will only publish on this date if you click the "Save and Publish". If you select a date, and only click the "Save" button, it will not be published and the date will not be saved.
- If you made changes and published, and then realized it was a mistake and want to rollback to a previously published version, you can simply delete the publish date and click "Save and Publish". This will unset the publish date, and the latest published version of the page will be used instead. This will still be the latest "staged" version, and you can publish it again when you are ready.

Duplicate

If you want to make another page similar to an existing page you can duplicate it. This can be done while editing an existing page, by clicking the duplicate button. If you have made any unsaved changes to your page before you duplicate it, they will be saved to the existing record before a new duplicate page is created.

Routing

For SEO and Human Optimization, every page can have one or more Friendly URLs ("routes"). Addtional routes are aliases that will redirect to the primary route.

4.8.5 Lists

Filtering

- You can filter list pages with one or more keywords. Individual keywords will be treated as additional requirements that limit the search, so a search for 'mango good' will only return records that have both those words in any combination at least one of the searchable fields (a single field must contain both words, you cannot have 'mango' in the title and 'good' in the article. To search for the exact phrase "good mango" surround the words with quotes.
- You can combine individual words and phrases in one search.

Field Specific Filters

You can do advanced searches on one or more specific fields by using a special field syntax `FIELD[=]VALUE`, where `FIELD` is the field name (or a registered alias) and `VALUE` is the value (one or more words). The comparison can be:

- **exactly equals: `[=]` or `[!=]`** Example: `title[=]foo bar stache` *(the title is exactly "foo bar stache")*
- **contains: `[:]` or `[!:]`** Example: `title[:]foo` *(the title contains "foo" anywhere, e.g. "foobar" or "barfoodo")*
- **greater or less than: `[>]` or `[<]` `[>=]` or `[<=]`** Note: if searching a time field, the human readable formats will be converted to a unix time stamp. Example: `time[>]2015-05-01`
- **regular expression: `[?]` or `[!?]`** Note: reserved Regular Expression special characters need to be commented out with a backslash "". Examples:

  `title[?]^foo[a-z]+ar` (the title starts with "foo" followed by any character a-z followed by "ar", e.g. "foobar" or "foojar") `title[!?]\(copy\)$` (anything with a title that doesn't end in "(copy)")
- **in list: `[#]` or `[!#]`** ***(the value is in the list of options)*** Note: the value should be a comma separated list. Example: `id[#]1,2,3` *(id equals 1,2 or 3)*

Multi Part Filters

**title[:]foo bar time[>]2014-10-14** - finds where title contains "foo bar" **and** time is greater than the date _**baz shazam title[:]foo bar**_ - finds where content includes baz and shazam in any field **and** "foo bar" only in the title field.

4.8.6 Customization

The framework allows you to customize the generic PHP controller or Twig template for any content type by adding an identical file to the client's site in the relevant version `/var/wwww/vhosts/{ID}/v/1/0/src/` directory. Design Templates can also be customized in the same way by adding files to the `Sitetheory/Template{TEMPLATE-NAME}Bundle/src/` directory. Individual pages can have a unique controller only for that content ID by adding a similar file with the additional content ID appended to the name.

Learn more about File Customizations.

4.9 Pages

Every page or module is a *Content* entity associated with a specific ContentType. ContentTypes are unique content like an Article, Profile, Form, Video, etc. The Content is a simple generic record that exists to provide a permanent id for each page. This permanent ID is necessary because a content (i.e. the page or content) can be versioned and each version will have a different ID, so there has to be one unifying ID that all other associated entities can reference.

4.9.1 Content Associations

Content (content)

The *Content* entity is a versionable entity, so *Content* which has a lot of associations with other entities that provide additional information about the page, but these are the most important associations to understand how a page is a built and how it interacts with the site

ContentType (content.contentType)

The *ContentType* entity defines the type of content of each Content, e.g. Article, Form, Map, Video, etc. These Content Types are defined in a master Content Type list, and are available based on the services that a site is subscribed to. The Content Types point to the Controller and Twig Templates that render the specific page. The ContentType also determines the correct Meta entity to associate with the version.

Routing (content.routing)

The *Routing* entity defines Friendly URLs that point to a specific Content (page). One or more Routing entities can be associated with a Content. There will always be one primary route and additional routes will be aliases that redirect to the primary route.

ContentVersion (content.version)

The *ContentVersion* entity is the versionable part of the *Content* entity, whichthat contains the majority of the information about a *Content* which can change, e.g. Title, Author, Date, Content, Images, etc. Content will automatically version when new users edit them or after a specific period of time (or when manually requested), and a new *ContentVersion* is cloned from the current *ContentVersion*. This keeps a revision history for every page. When previewing a site, the **last modified version** is used, but when viewing the live site, the current **Published** version is used.

The *ContentVersion* contains the standard fields that most pages or modules will need, e.g. Title, Author, Date, main Content, main Image, etc. This is done so that it is easier to reference the content of each entity in a list without having to attach a lot of other associated entities. These ContentVersion fields should be used by the Content Type whenever possible, i.e. some Content Types like "Article" may not even have any custom fields in the associate Article (Content Type) entity. But many Content Types do need unique fields, so all ContentVersions reference a related Content Type entity.

ContentVersion Associations

`Tag` The Content can be associated with one or more tags that are themselves associtaed with one or more Streams, (list pages that display all the content associated with the tags).

`ContentShell` A content can specify a unique template shell design to use (the look of the container around the content), and this preference is associated with the ContentVersion so that it can be previewed before the ContentVersion is published.

`ContentLayout` A content can specify a unique layout design to use (the look of the content area), and this preference is associated with the ContentVersion so that it can be previewed before the ContentVersion is published.

Meta (content.version.meta)

The *Meta* entity is a dynamic attachment point for different ContentType entities where unique data is stored for this type of content (e.g. *Article* will have slightly different data storage needs than *Profile* so we the meta lests us keep the *ContentVersion* entity focused on just the most commonly stored data relevant to all Content). *Content.ContentType* specifies which kind of Meta entity should be joined and then our code finds finds the correct ContentType entity (e.g. *Article*) and joins it to the ContentVersion. Therefore the meta entity iterates a new record everytime the *ContentVersion* changes.

4.10 Samples

4.10.1 Signup Form for External API

NOTE: Use the "Sitetheory Registration" component for capturing signups locally.

This is a sample form for submitting to an external third-party API for processing the signup and returning success. This is useful if you have your data stored elsewhere, and/or want to create custom functionality (e.g. customized email response). A great solution would be to use a Heroku node.js server to quickly setup logic to process the API call that we make below.

Javascript

```
{% block script %}
    {{ parent() }}

    {# SIGNUP FORM #}
    <script>
        (function (root, factory) {
            if (typeof require === 'function') {
                require(['stratus', 'underscore', 'angular', 'angular-material'], factory);
            } else {
                factory(root.Stratus);
            }
        }(this, function (Stratus, _) {

            Stratus.Controllers.SignupController = function ($scope, $element, $http, $attrs, $window) {

                $scope.options = {
                    pattern: {
                        email: /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/,
                        zip: /[a-zA-Z0-9 \-]{5,}/
                    },
                    // URL to
                    url: 'https://api.thirdpartydomain.com/signup',
                    response: {
                        success: 'Thanks for signing up! We\'ll be in touch shortly.',
                        error: 'Sorry ;( looks like there was an error saving your info. Please email us directly so we can help.'
                    },
                    redirect: {
                        // NOTE: browsers block popups so this isn't advised
                        popup: false,
                        url: false,
                        config: null // 'width=400,height=500,toolbar=no,menubar=no,scrollbars=yes,resizable=yes'
                    }
                };

                // Merge Custom Options
                if($attrs.options) _.extend($scope.options, JSON.parse($attrs.options));
```

(continues on next page)

(continued from previous page)

```

                $scope.response = '';
                $scope.status = null;

                $scope.data = {
                    email: '',
                    zip: ''
                };


                $scope.submit = function(form) {
                    var prototype = {
                        method: 'POST',
                        url: $scope.options.url,
                        data: JSON.stringify($scope.data)
                    };
                    $scope.status = 'sending';
                    $http(prototype).then(
                        // Success
                        function successCallback(response) {
                            if (response && (response.status === 200)) {
                                if($scope.options.redirect.url) {
                                    if ($scope.options.redirect.popup) {
                                        var win = $window.open($scope.options.
↪redirect.url, '_blank', $scope.options.redirect.config);
                                        if(win) win.focus();
                                    } else {
                                        $window.location($scope.options.redirect.url);
                                    }
                                }
                                $scope.response = $scope.options.response.success;
                                $scope.status = 'success';
                            } else {
                                $scope.response = $scope.options.response.error;
                                $scope.status = 'error';
                            }
                        },
                        // Error
                        function errorCallback(response) {
                            $scope.response = $scope.options.response.error;
                            $scope.status = 'error';
                        }
                    );
                }

            };
        }));
    </script>

{% endblock script %}
```

Twig

```
{% block registrationForm %}

```

(continues on next page)

(continued from previous page)

```
<form name="Signup" ng-submit="submit(form)" ng-controller="SignupController" options='{"redirect":{"url":"https://secure.actblue.com/contribute/page/bncdec", "popup":false}}' ng-class="status" ng-cloak>

    <md-progress-linear md-mode="indeterminate" ng-show="status === 'sending'"></md-progress-linear>
    <p class="message" ng-show="response.length" ng-bind-html="response"></p>
    <ul class="listInline divCenter fontSecondary">

        {{ registrationFormBefore|default('')|raw }}

        {% verbatim %}
        <li>
            <md-input-container>
                <label>Email</label>
                <input name="email1" type="email" ng-pattern="options.pattern.email" ng-model="data.email" required>
                <div ng-messages="Signup.email1.$error" role="alert">
                    <div ng-message-exp="['required', 'pattern']">
                        Please enter a valid email.
                    </div>
                </div>
            </md-input-container>
        </li>
        <li>
            <md-input-container>
                <label>Zip</label>
                <input name="zip" ng-pattern="options.pattern.zip" ng-model="data.zip" required>
                <div ng-messages="Signup.zip.$error" role="alert">
                    <div ng-message-exp="['required', 'pattern']">
                        Please enter a valid zip code.
                    </div>
                </div>
            </md-input-container>
        </li>
        {% endverbatim %}
        <li>
            <button type="submit" class="btn fakeFormSubmit" ng-disabled="Signup.$invalid">{{ textSubmit|default('Count Me In') }}</button>
        </li>

        {{ registrationFormAfter|default('')|raw }}

    </ul>
</form>
{% endblock registrationForm %}

<div id="footerJoinForm" class="joinForm purple" ng-cloak>
    {% set registrationFormBefore = '<li><div class="starLeft"></div></li><li><h1>Add Your Name</h1></li>' %}
    {% set registrationFormAfter = '<li><div class="starRight"></div></li>' %}
    {{ block('registrationForm') }}
</div>
```

Count Up

A counter that changes a number from a start to an end value. You can also tell countUp to animate other elements like a progress bar.

This sample code is using a Custom API to fetch custom data *results.count* which is set to fetch on load and then every 10 seconds afterwards.

```
<div ng-controller="CustomApi" options='{"controller":"/people/count", "onLoad":
↪"fetch", "onTime": {"time": "10s", "method":"fetch"}}'>
    <div id="progressBar" class="positionLeftTop salmon" style="max-width: 100%"></
↪div>
    <div id="totalSignUp" class="borderDashed fontSecondary salmonText" count-up␣
↪start-val="0" end-val="results.count" count-instance="countUp" related-target=
↪"progressBar" related-style="{ width: (100*(frameVal/500000))+'%' }" duration="1.5"␣
↪decimals="0" scroll-spy-event="elementFirstScrolledIntoView" scroll-spy></div>
</div>
```

4.11 Streams

4.11.1 Overview

Streams are pages that list all of the content that share similar tags. When Articles, Profiles, Events, or other content are associated with a tag, that content will appear on Stream pages that are associated with those tags. Streams come in many different layouts to fit different needs, e.g. a landing page with a big slideshow and other dynamic modules below, a blog style page with images and words flowing down the page, a grid of photos, or a simple compact text list, etc.

CHAPTER 5

Standards

5.1 Overview

Our Standards are carefully established to provide a decrease in faux pas while increasing legibility, maintainability, and scalability.

5.1.1 Symfony Framework

Symfony is an MVC framework we utilize to maintain optimal abstract logic. The internal application kernel maintains requests, responses, and configurations, then sends this information off to a Controller for business logic. Any common routines utilize dependency injection for optimal usage. With that in mind, we utilize controllers, services, and bundles based on particular feature sets to maintain a modular design. The only bundle that we have dependence on, at this time, is the CoreBundle. This may change down the road as more of these pieces get fragmented out.

5.1.2 Doctrine ORM

You may ask yourself, what is an ORM? For that I say, an Object Relational Manager is a system that maintains object persistence without directly querying and storing each property and their constraints separately. They also allow the use of multiple querying languages to handle said persistence.

5.1.3 Release Cycles

We use Semantic Versioning to determine when and how to set version numbers.

5.2 Styling Standards

5.2.1 CSS Standards

We suggest you follow the styling guide located at: https://github.com/necolas/idiomatic-css and declare files in the normal methods for Twig using Assetic in Sitetheory.

```
<link rel="stylesheet" href="{{ asset('bundles/sitetheorystratus/stratus/bower_
↪components/angular-material/angular-material' ~ minified ~ '.css') }}">
```

So, ideally, it will use the unminified version when you're in design mode, otherwise the min version will be supplied to production.

5.2.2 LESS Standards

Using LESS is always a bit more complex, but allows for a level of dynamism that can provide a much simple updating, alteration, and maintenance scheme in the long run. For example:

```
background: url('@{asset}/bundles/sitetheorytemplatesencha/images/socialSlash.png')␣
↪no-repeat right center;
```

We highly recommend using LESS, when applicable. Compilation and compression of these files works out of the box in most Sitetheory contexts.

5.2.3 Twig Compilation

This methodology allows for your LESS files to easily compile and compress appropriately within Sitetheory's ecosystem.

```
{% stylesheets '@SitetheoryTemplateBundle/Resources/public/css/common.less' filter=
↪'less' filter='?uglifycss' filter='cssrewrite' %}
    <link rel="stylesheet" href="{{ asset_url }}">
{% endstylesheets %}
```

5.3 Template Standards

5.3.1 Twig

We use Twig because it's awesome. See: https://twig.symfony.com/doc/2.x/

Twig Extensions

You can create useful methods that extend Twig with our own filters and functions. See Sitetheory/CoreBundle/Twig/Extensions/UtilityExtension.php (and other related). These need to be registered in the bundle's Resources/config/services.yml

Troubleshooting in Twig

It's often helpful to dump variables inside a Twig template, so that you know what variables exist, and what the values are. If you are in ?mode=dev (and you have the proper developer credentials) you will see a Symfony profiler bar, where dumps appear. In normal PHP, this is invoked like this:

```
if(function_exists('dump')) dump('some string', $someObject, $otherArray);
```

In Twig, you can dump a variable to the profiler bar like this:

```
{% dump someVariable %}
```

Dumping in Twig Extension

Extensions exist within their own scope, and so if you do a dump() within a PHP file that the Extension calls, you might expect it to appear in the profiler, but it never makes it out of this scope. So the solution is to include (temporarily) the TwigExtensionTrait in your extension and make it's dump function available to Twig.

```
class FooExtension extends \Twig_Extension
{
    // Include the Trait with the dump variable and method
    use TwigExtensionTrait;

    // Register the dump method along with the rest
    public function getFunctions() {
        return [
            // Existing Method
            new \Twig_SimpleFunction('foo', [$this, 'getFoo']),
            // Custom Dump Method: This should be called within twig, using {% dump␣
↪extensionDump() %}
            // TODO: deactivate when not testing
            new \Twig_SimpleFunction('extensionDump', [$this, 'getExtensionDump'])
        ];
    }

    // The existing function you want to dump from.
    public function getFoo() {
        // Set a Variable you want to dump from the twig
        $this->addDump($myObject);
    }

}
```

Then in the Twig template you can dump what you previously registered within the Extension.

```
{% dump extensionDump() %}
```

CHAPTER 6

Tutorials

6.1 Quick Overview

6.1.1 Technical Background

The current website is built on the Sitetheory.io platform, and implements Angular as a javascript framework to handle model management and interact with the Sitetheory API. Sitetheory is built using PHP on the Symfony framework and utilizes Doctrine for database and entity management. Sitetheory is a CMS and framework that manages page requests on the server side. The current URL determines the correct Content (page) to load. Each Content is associated with a specific Content Type, e.g. Article, Profile, Landing Page Stream, etc. Sitetheory will load the correct controller for the current Content Type, as well as the correct Twig template. As a designer, you don't need to worry about most of that, you can just customize the template for a given page. You'll have access to do fancy stuff through Angular if you need to create lists or inputs. And if you need to get more fancy, you can code custom javascript or load third party libraries directly from the template.

See the [Sitetheory Docs](http://docs.sitetheory.io/index.html) for more details, and from there access specifics about the API and the Stratus javascript framework. Sitetheory is a private SaaS platform for building simple or complex websites. The platform is still in development, so documentation is incomplete, but we're working to improve it every day.

6.1.2 Theming

Anything can be customized, by adding a custom version of the file in your vhost by mimicing the exact file structure of the content type ([vendor]/[bundle]/Resources/views/[file]).

Customize Shell

The site is assigned to a specific theme, which in this case is the Custom theme (which is a blank slate that allows easy customization). By default every page is assigned to a basic shell design (*shell.html.twig*) which determines the look/feel of the theme. But you can edit any given page (from the CMS Admin > Content > Edit > Settings) and select

an alternative shell that is available for this theme (You can customize existing shells or create new shells for a theme as well).

If you are assigned to the Custom theme, you can create a customize version of the *shell.html.twig* inside the *TemplateCustomBundle*, which is located in *Sitetheory/TemplateCustomBundle/Resources/views/shell/shell.html.twig*. This extends Sitetheory's base shell template [*Sitetheory/TemplateBundle/Resources/views/shell.html.twig*](https://github.com/gutensite/Sitetheory/blob/1.0/src/Sitetheory/TemplateBundle/Resources/views/shell.html.twig) which already establishes the main structure of a template. This core shell template in turn that extends the HTML base template [*Sitetheory/CoreBundle/Resources/views/base.html.twig*](https://github.com/gutensite/Sitetheory/blob/1.0/src/Sitetheory/CoreBundle/Resources/views/base.html.twig). Reviewing the parent templates will show you which Twig blocks can be extended.

Customize Pages

Every page is a "Content" which is assigned to a Content Type, e.g. Article, Profile, etc. To customize the look of all Articles, just add a file called */Sitetheory/ArticleBundle/Resources/views/Article.html.twig*. To customize one specific Article, get the unique content ID of that article (from the admin url ?id=xxxx, or in the dev toolbar) and add a file called */Sitetheory/ArticleBundle/Resources/views/Article[ID].html.twig* where "[ID]" is the content ID.

6.1.3 Workflows

Create New Custom Page

1. Create New Page.** Create new "Article" in the Sitetheory Admin > [Content section](https://admin.sitetheory.io/Content)
2. Customize Design.** Create a custom template file for that specific Article (see example above).

6.2 How To Add Menus to Templates

6.2.1 Menu Component

We provide an easy way for a designer to insert the menu into any part of the Twig template design and determine how the menu looks and functions.

Twig Method

{{ menu(content) }}

Caching

Currently, we cache the menu in the RAM (Environment) so you can call upon it multiple times without causing extra overhead. TODO: We need to create or delete/recreate a menu.json cache whenever a link is created, edited, or deleted in the main menu (this could be a Menu or MenuLink listener).

Twig Extension Location

Twig Extension: Sitetheory/MenuBundle/Twig/Extension/MenuExtension.php Twig Template: Sitetheory/MenuBundle/Resources/views/MenuExtension.html.twig Twig Menu Component Macros: Sitetheory/MenuBundle/Resources/views/components.html.twig

Usage

In a twig template we will call the menu function with desired options and directly insert the HTML into the page. {{ menu(content) }} or {{ menu(content, {'scope': 'primary', 'limit': 6}) }}

Returns

This will return standard formatted HTML that can be easily styled in CSS depending on the *type* of menu requested, or it will return an array if you have specified output='array'.

Arguments

-*content* (mixed obj or integer) [required] - This should either be the content object which is available in the twig template which describes the current page. Less common, it may also be an integer specifying a menu ID (if we specify a menu ID it will load that menu instead of the main menu which is default). The *content* object (if supplied) is used by the function to determine the current page's position in the menu, to set which section (and subsection) we are in (set active classes on the nested links or main level).

-*options* (object) [optional] - An object of options to override the defaults.

Options

Note: all options will have defaults that may be overwritten by the specified menu "type" (since each will have different use cases). But of course the actual value may be modified by the manual options specified by the calling script.

'scope' (str) [default: primary] - a string defining the scope of the menu to create. Options include: primary, section.

- "primary" (default) - will find all top level links with no parents. Will default to type=simple, and return 1 level deep only by default. But you can specify other menu types and depths.
- "section" - will find the section where the link to the current page is nested, and will return only that section (not counting the top link). If you get a section link you can also call {{ getMenuSectionName() }} and {{ sectionLink() }} (see other methods below).

'type' (str) [default: simple] - a string defining the type of menu to create. Options include: simple, nested, sitemap, dropdown.

-"simple" (default) - This is a simple menu list typically used primarily for a main menu links in header. It is automatically limited to one level, which by default will be the top level (if *parent* is defaulted to null or 0). [related defaults: 'depth'=1;'count'=6;]

-"accordion" - This works like a standard nested according menu (usually used for the sidebars to display section links because header shows main top level links). You can click to open subsections (if any nested menu links exist), when clicking a link on one level, it closes any other subsections that were previously open. [related defaults: 'parent'="section"; 'depth'=3, 'count'=null; 'action'='click']

-"nested" - This is identical to "accordion" (reuse same logic) but there are no actions to open/close, it's permanently open. It's commonly used to for mobile menu drawer. [related defaults: 'parent'="section"; 'depth'=3, 'count'=null; 'action'='open']

-“sitemap” - A list with a column for each top level link, with children nested below in column (mostly used for sitemaps in the footer). [related defaults: ‘depth‘=2; ‘count‘=6]

-*dropdown* - This uses the Angular dropdown menu, which has slightly different HTML than the other menu types (e.g. md-menu tags). See Angular dropdown for reference. [related defaults: ‘depth‘=2; ‘count‘=6; ‘action‘=”hover”]

‘depth‘ (int) [default: 1] - A depth of 1 means we only display menu links at the top level (usually parent=null or 0, but could be all links of a different parent if a parent is set for the section). While a depth of 2 would fetch links nested under each main link. [Requirement: depth cannot exceed 4 under any circumstances]

‘startDepth‘ (int) [default: 2] - A starting depth of 1 means we are displaying top level links and their children, while starting depth of 2 means we are only displaying the links of nested elements. This only applies to menu scope of “section” (i.e. a primary menu must always start at top level). That means that for a section menu, level 2 is the children of the current primary menu section.

‘parent‘ (mixed int or str) [default: null] - This defaults to null which means it will get all top level links without a parent. If another integer is specified, it will find links nested under the specified link ID (if it exists). Alternatively the value of “section” can be passed in to tell the script to fetch all links for the current main section. That means the current page (denoted by *content*) will be used to find the current main website section and we will only fetch the links that are nested under the current section. Section is defined as the highest level related link where parent=null or 0, e.g. If you have a site will main links: About, Resources, Products, each of those links are “sections” with parent=0 and if they have nested links, a “section” value would find all links underneath the “About” section.

‘limit‘ (int) [default: 6] - This limits the total number of links for the top level. There is no limit for subsequent levels. This is most used when a designer needs the ability to limit how are displayed in a main header links.

‘action‘ (str) [default: click] - Specify the type of action to trigger the opening of a menu subsection. Nested (accordion) menus should default to “click” while the Angular dropdown will default to “hover”. The option for “open” should only be used by the “nested” menu type if you want the nested menu structure to be fixed open without any opening/closing capabilities. Options include: “click”, “hover”, “open”.

‘output‘ (str) [default: html] - specify whether you want to return finished HTML or the raw array of links. Options include: “html”, “array”.

‘menu‘ (str) [default: null] - specify a specific menu id that you want to fetch, if none specified, it will find the “main” menu.

‘template‘ (str - default: ‘SitetheoryMenuBundle::MenuExtension.html.twig’) - specify an alternative template Alias (vendorBundle syntax) to use for rendering HTML of the menu. The default used is very flexible and can be easily styled in the CSS for every type of menu.

‘components‘ (str - default: ‘SitetheoryMenuBundle::components.html.twig’) - specify an alternative components template that is used for the repeating menu elements. This is useful if you just want to customize part of the menu.

‘ulClass‘ (str - default: null) - specify additional custom CSS Class names for the ul (all menu types except dropdown).

‘liClass‘ (str - default: null) specify additional custom CSS Class names for the li (all menu types except dropdown).

‘menuClass‘ (str - default: null) - specify additional custom CSS Class names for the md-menu (dropdown type only).

‘menuContentClass‘ (str - default: null) - specify additional custom CSS Class names for the md-menu-content (dropdown type only).

‘menuItemClass‘ (str - default: null) - specify additional custom CSS Class names for the md-menu-item (dropdown type only).

6.2.2 Other Features:

Styling - The HTML for "simple", "accordion" and "sitemap" are all identical, but they just change styling based on CSS. The CSS is already in the common.css file. The appropriate type class should be set on the parent container based on the "type" name, e.g. *.menu-simple*, *.menu-sitemap*, *.menu-nested*, and *.menu-accordion*, *.menu-dropdown*.

The layout is set in the MenuBundle/Resources/views/MenuExtension.html.twig (which can be customized for a specific template). But most of the elements are actually in the MenuBundle/Resources/views/components.html.twig, which can also be customized for a template, just point the options to that custom file, e.g. {{ menu(content, {'components': 'SitetheoryTemplateCustomBundle::components.html.twig'}) }}. That's is the guts of the styling. However, if you just want to include some extra classes, you can see the options above to include classes in the <li> and <a>.

Section Name - In cases where we use a section menu (e.g. {"scope": "section"} on a sidebar) we often want to know what section we are in (e.g. to put the name above the menu).

Section Name - In cases where we use a section menu (e.g. 'parent'="section" on a sidebar) we often want to know what section we are in (e.g. to put the name above the menu). So when we fetch that, we insert that information into the Twig Environment for the designer to access in the template. {{ section }} will contain an object that includes {'name', 'url'}. For getting section name,url we need to call {% set sectionName = sectionName(content,parent) %} where content can be object or menuId and parent will be nestParentId of the section for which we want to show the section name.

Active Menu - The method needs to determine which menu link is currently active for the current page, as well as all the related parents up to level 1 (so we can set an active class on the each active link). So we check the *content* and find the menu link that points to the current page. Then we keep make a list of that link ID and all the link IDs of it's parent up to level 1. When we create the HTML we add the "active" (if it's an active link) and "activeParent" (if it's a parent, not the actual active link) class to each link in that nested tree and make sure that accordion menus stay open if it has the active class.

The menuLinks array will specify *active* = true if the current link is active, and 'activeParent'=true if the current link is a parent of an active link (up the tree). So HTML should add the appropriate classes and styles for active links versus the parent of active links. Most likely you'll want them all to say 'active' and just style them differently.

Actions - For accordion ng-click and ng-class should add class .see-children only to the parent <li> of the link clicked.. There should be ng-click to open on levels 1-3. Clicking another menu open should close (collapse) all other menus already open. When a link is clicked with an ng-click (opening up a submenu) it should add the "active" class and remove the active class from all others at this current level or in other branches (keeping the active on it's own parent so it stays open and shows where we are in the menu).

Nesting Levels - HTML should dynamically add the relevant level number in nested menus, e.g. list-level3 (so we can style)

HTML Output - All the menu types share the same HTML except Dropdown uses Angular dropdown md-menu and md-link tags. Below is the recommended structure of the menus (which is already styled in the common.css).

NOTE: The menuHelper->getMenuLinksNested() function we use, actually gets the FULL menu (4 levels deep) and stores that in a cache. Then each menu that is requested, is parsed from that. This is necessary so that we can find the right "section" of a page that might be nested. Even though we only want to show one or two levels publicly, we still need to get the full menu so we can find that info.

6.2.3 Menu Section Component

If you have loaded a "section" menu e.g. *{{ menu(content, {'scope': 'section'}) }}* then you can also get the section Name and full Link object, in case you want to create a header, or cookie crumbs.

Get Section Name

You can just get the section name as a string. .. code-block:: html+twig

linenos

<h2 class="section-name">{{ menuSectionName() }}</h2>

Get Section Link

You can get the entire Link object of the current section in case you want to get the name, route, and even all the children to cycle through and create a cookie crumb.

```
{% set menuSectionLink = menuSectionLink() %}
<h2 class="menu-section-name"><a href="{{ menuSectionLink.route }}">{{
↪menuSectionLink.name }}</a></h2>
```

6.2.4 Menu HTML Component

If you have a manual array of menuLinks objects, and you just want the get the HTML, you can pass those in to this function and get the results.

```
    {{ menuHtml($menuLinks, {'scope': 'section', 'type': 'simple'}) }}
```

6.3 How To Create Admin Pages

Admin pages are going to be created slightly differently than regular pages, i.e. they aren't created through the Content page, but through a special page (Content Type restricted to CMS service) that gives full access to create new content types.

6.3.1 Steps to Create a Basic New Page

1. **Create Content Type**

   If this is going to be a new type of CMS admin page (which it probably will be since most admin pages are unique, i.e. one content type for one page), first create a new record using the Content Type list online.

   Select the Vendor and set the name of the Bundle as well as the name of the Controller that will contain the code for this page. This controller name will also be the standard common name for form types, templates, etc. In all but the most simple bundles, the controller name should include a folder prefix to keep the code files for your content organized, e.g. `Content\contentSeoEdit` (the suffix "Controller" is assumed and will be added automatically in the code).

2. **Create Content**

   Create a page in the system with a friendly URL: /Admin/CMS/Edit

   Use the prefix '/Admin/' in the friendly URL, so that all admin pages are prefixed consistently, and follow other established patterns so that our URLs all match a predictable standard.

3. **Create Controller**

The controller will contain the code for the functionality of the page. If this is part of the core CMS, this will be located in a sub folder of **:namespace:'Sitetheory\CoreBundle\Controller'**, but if it's the admin controller for another bundle feature, it will go in whatever bundle where the related admin and public controllers and templates are located, e.g. **:namespace:'Sitetheory\CoreBundle\Controller\Content\contentSeoEditController'**

This controller should follow standard Symfony standards for controllers, and the indexAction "should" in pass in Request and InitController (the CMS core controller), e.g.

```
<?php
public function indexAction(Request $request, InitController
↪$initController)
```

If this page is going to be a list page it should probably extend the **:namespace:'Sitetheory\CoreBundle\Controller\Cms\ListControllerBase'** to utilize standard list, search and filtering features. See section about *How to Create List Pages* for details.

If this page is going to be an edit page it should probably extend the *SitetheoryCoreBundleController-CmsEditControllerBase* to utilize standard admin editing features. See section about *How to Create Editor Pages* for details.

4. **Create Template**

   Every page needs a template to provide the visual display for the controller. These are located in the standard Symfony locations, in the same Vendor and Bundle and the same naming convention and folder structure as the Controller, e.g. **:namespace:'Sitetheory\CoreBundle\Resources\contents\Content\contentSeoEdit.html.twig'**

   This template should extend the shell, e.g. `{% extends content.contentVersion.shell %}` (the selected for every view is set based on the design settings and applied to every view unless an alternative shell is specified for this page in the design layout settings).

   If this page is extending some standard functionality (e.g. List or Editor), then the template will extend the standard templates associated with that functionality which in turn extends the shell, e.g. `{% extends 'SitetheoryCoreBundle:Cms:EditBase.html.twig' %}`

6.3.2 How to Create List Pages

In order to utilize standard functionality for building lists, you should extend the standard List Controller and Templates.

[todo: add more details once we finalize this]

6.3.3 How to Create Editor Pages

In order to utilize standard functionality for building editing pages, you should extend the standard Editor Controller and Templates.

Editor Controller

If this is a generic editor for any entity, extend the standard edit controller **:namespace:'Sitetheory\CoreBundle\Controller\Cms\EditControllerBase.php'**.

If this is going to be a page that interacts with Content Types via the Content, extend the special version of this controller **:namespace:'Sitetheory\CoreBundle\Controller\Content\contentEditControllerBase.php'** which extends `EditControllerBase` with some additional functionality specific to Contents, e.g. publishing and versioning.

In both cases the base controller will load getForm() to return the path to the correct form type. By default this function will find the form based on the current page's controller (this works because everything follows the same common name of the controller).

Custom Editor Form

If you need an alternative form, you can write your own custom getForm() function to set your preferred form type.

```
<?php
public function getForm(InitController $initController) {
    return 'Sitetheory\CoreBundle\Form\Type\Content\contentSeoEditType';
}
```

See example code for reference of implementation in the file ‘‘ SitetheoryCoreBundleControllerContentcontentSeoEditController.php‘‘

Editor Templates

The template should extend the editor template (so that it has all the standard action buttons) and include it's own custom fields:

See example code for reference of implementation in the file **:namespace:‘Sitetheory\CoreBundle\Resources\views\Content\viewSeoEdit.html.twig‘**.

6.4 How To Create Content Type Entities

Content Types are a critical part of the CMS because they determine what Controller should be executed for each page (what that page should do and how that page should look). When a creating an *Entity Content Type* which will be a piece of content (e.g. an Article) the Content Type needs to specific the Entity name and the Entity must be created in a specific way.

6.4.1 Create Entity Class

An Entity class should be created for each Entity Content Type, whether or not you need unique fields for this Content Type (beyond what is included in the ContentVersion already). The reason for this is so that all Content Types follow the same predictable structure, i.e. we always know that there will be an entity at `$content->getContentVersion()->getContent()`. Most entities will need custom content fields, but either way we include it for consistency in case we need to add a custom field in the future and don't want to have to create new records for every existing record.

6.4.2 Register API Accessibility

The Entity class should register the entity properties (fields) that are readable, writable, and searchable by using SitetheoryApi annotation. This registration happens in the entity field declaration. See the **:namespace:‘SitetheoryCoreBundle:Content\ContentVersion‘** as an example. See *SitetheoryCoreBundleAnnotationsApi.php* for details.

- **Readable** All fields are readable by default. Set to false if you don't want them displaying. Set level="x" if you don't want the API traversing beyond a certain level. You can specify a sentinel of permissions to limit who can read, e.g. readable="false" or readable={"edit"}

- **Searchable** Fields are not searchable by default, you must enable them explicitly. You can specify a sentinel of permissions to limit who can search, e.g. searchable="true" or searchable={"edit"}
- **Writable** Fields are not writable by default, you must enable them explicitly. You can specify a sentinel of permissions to limit who can write, e.g. writable="true" or writable={"create","edit"}

You can add custom Require.js config for your Vendor or vhost, which will be compiled with the core config. For example, if you need to load your own directives or components.

JAVASCRIPT Create custom Require.js config, to tell Require.js (and Stratus) the location of your custom dependencies.

Example 1: Specify shim, paths, etc (full config structure) *AcmeFooBundle/Resources/public/js/boot/config.js*

```
boot.config({
    shim: {
        'angular-froala': { deps: ['angular', 'froala'] }
    },
    paths: {
        froala: boot.bundle + 'stratus/bower_components/froala-wysiwyg-editor/js/
↪froala_editor.min',
        'angular-froala': boot.bundle + 'stratus/bower_components/angular-froala/src/
↪angular-froala',
        'stratus.components.foo': 'acmefoo/js/foo'+boot.suffix
    }
});
```

Example 2: Specify only paths (shortcut) *AcmeFooBundle/Resources/public/js/boot/config.js*

```
boot.config({
    'stratus.components.foo': 'acmefoo/js/components/foo'+boot.suffix
});
```

Note: See *SitetheoryStratusBundle/Resources/public/stratus/boot/env.js* for available properties, e.g. *boot.suffix*. The Sitetheory config will load first and set the default values for these properties, which will be available for the custom config above.

If the path does not start with a slash, the config will automatically prefix the path provided with *boot.cdn* (which will be the path to the CDN if it's production or a relative '/' if it's dev) and the *boot.relative* (which is the path to the current version, e.g. 'assets/1/0/bundles'). So you only need to start with the vendor bundle name folder.

TWIG Then in your twig file, just load your custom config, BEFORE

```
{# Load Custom Vendor or Vhost Require Config #}
{% block scriptConfig %}

    {% javascripts '@AcmeFooBundle/Resources/public/js/boot/config.js' filter='?
↪uglifyjs2' %}
    <script src="{{ asset_url }}"></script>
    {% endjavascripts %}

    {# You MUST include the parent, so that it doesn't overwrite other instances of␣
↪custom config by other vendors #}
    {{ parent() }}

{% endblock scriptConfig %}
```

6.5 How To Customize Files

The Sitetheory framework allows you to easily customize any Controller, Template, CSS, Javascript, etc. The framework will find the "best" version of Controllers (PHP) and Templates (Twig) depending on a cascading order of which priority folders contain customized files, e.g. core templates can overwrite core files, vendors can overwrite core vendor, master sites can overwrite the vendor, and custom site files can overwrite templates. NOTE: public assets like CSS, Javascript or images are not able to be found dynamically (see section below).

The core platform files are located in the **:namespace:'Sitetheory\CoreBundle'** (and other bundles in the `Sitetheory` vendor directory). These can be customized for a specific **Client Site** or a **Template** by adding custom files to the right location.

6.5.1 Composer Autoloader for Controllers

By default Symfony uses Composer autoloader, which is setup in `app/autoload.php` and looks at registered standard paths for custom files in the src/{VENDOR}/{BUNDLE} or app directories. We can register additional paths that contain files for namespaces that start with a name. But since we need to point to a dynamic directory that we only discover inside the **:namespace:'Sitetheory\CoreBundle\Controller\InitController'** (looking in the Master Vendor, Vendor, Template, Client Site, dev User), we have to modify the $loader after the fact with a reference to the `$GLOBALS['loader']` (this works, although it is non-standard and not-recommended use of Globals). So in the `InitController` we register all namespaces that start with `Sitetheory` to point to the site's custom files with a priority the priority below:

-**Dev User**: for a developer testing new features (only accessible to this user). -**Site Template**: the site's custom template or customization of a vendor template located in site's folder, e.g. a template may customize the UserBundle. -**Site**: the custom version of any vendor and bundle file defined in generic site folder, e.g. customize the UserBundle's layout. -**Vendor Template**: any customizations to the core that were made by the vendor's template. -**Vendor**: any customizations in the vendor's folder structure -**Vendor Master Template**: any customizations in the vendor's master site's template (e.g. Vendor Gutensite has a master Vendor of Sitetheory) -**Vendor Master**: any customizations in the vendor's master site's folder structure (e.g. Vendor Gutensite has a master Vendor of Sitetheory) -**Sitetheory Core**: the core Sitetheory files (often the same as "vendor master")

This has a cascading priority that lets you customize files in a very targeted manner by creating files with the same vendor and bundle namespace directory structure and matching filename to easily overwrite core functionality (controllers) and design (templates). The example below assumes a site (id 100) which may be a child site of a master site (id 9) is assigned to a vendor called "Foo" for the template "Bar" which is a child site of it's master "Sitetheory". Each of these paths looks into a folder that emulates the main Sitetheory 'src' folder, which lets you customize any file by specifying the vendor and bundle name via the folder structure of their original locations. We also look in here specifically for templates that have been customized for the CMS Edit pages.

Namespace Path ——— ——

user /var/www/vhosts/100/user/1/ userVersioned /var/www/vhosts/100/user/1/v/1/0/ siteTemplate /var/www/vhosts/100/src/Foo/TemplateBarBundle/src/ siteTemplateVersioned /var/www/vhosts/100/v/1/0/src/Foo/TemplateBarBundle/src/ site /var/www/vhosts/100/src/ siteVersioned /var/www/vhosts/100/v/1/0/src/ siteMasterTemplate /var/www/vhosts/9/src/Foo/TemplateBarBundle/src/ siteMasterTemplateVersioned /var/www/vhosts/9/v/1/0/src/Foo/TemplateBarBundle/src/ siteMaster /var/www/vhosts/9/src/ siteMasterVersioned /var/www/vhosts/9/v/1/0/src/ siteVendorTemplate /var/www/core/v/1/0/src/Foo/TemplateBarBundle/src/ siteVendor /var/www/core/v/1/0/src/Foo/ siteVendorMasterTemplate /var/www/core/v/1/0/src/Sitetheory/TemplateBarBundle/src/ siteVendorMaster /var/www/core/v/1/0/src/Sitetheory/ Core /var/www/core/v/1/0/src/Sitetheory/

All of the vHost `src` folders can be versioned, if a particular Bundle needs to be specifically altered for each core version, but it isn't necessary in the case that you have some customizations that are universally accepted.

Note: Assets will only create a respective symlink to the first found bundle in the hierarchical algorithm. For instance, having the same Bundle in both a vHost `src` and a versioned `v/1/0/src` will only create a symlink to the former, while skipping an overwrite from the latter.

6.5.2 Twig Loader for Templates

By default Symfony looks for templates to override third party vendor bundles in the `src` or `app` directories. But in the `InitController` we tell Twig to look in other directories through the use of the Twig loader, e.g.

```
<?php
$this->container->get('twig.loader')->prependPath($templatePathContent,
↪$contentBundleNamespaceShortcut);
```

Then as long as we put the files in the right directory, they will override the core templates.

6.5.3 Assets

Standard Location of Assets

Assets are stored in the standard Symfony bundle locations below root, e.g. for a Foo bundle:

Since the website can't load these files below root, we have a script (see "Deployment of Files" below) which creates symlinks from the public web folder to each bundles public folder, e.g.

So anything you put in the public folder, will be publicly accessible on the webserver.

Loading an Image

To load an image from this folder, you would link to the file in this bundle:

But from Twig, we prefer to use an asset function that lets us dynamically request the correct version:

Loading CSS and Javascript

CSS and Javascript is loaded from the exact same structure, but we have a few extra functions to dynamically determine the best extension, to load the correct minified version on live sites or raw version when in development mode.

NOTE: we have Twig methods for compiling CSS and Javascript and adding the right extensions.

Twig Methods for CSS

- **styleExt(format)** -'css': In dev, it wil load ".css" and in live it will load ".min.css". -'less': In dev, we will have the ".less" extension, but stratus will dynamically compile the file into CSS so that it works (this requires rel="{{ styleRel('less') }}" to tell stratus to compile it). In live mode, it will append ".min.css" and load like normal. -'sass': this will append ".scss" in dev mode (but currently will break because there is no compiler). In Live mode it will load ".min.css" and work like normal.
- styleRel(format): this will add "css", "less", "sass" to the *rel* attribute, which in dev mode triggers the compiling (if necessary).

Twig Methods for Javascript

-scriptExt(format) -'coffeescript': On dev mode this will append ".coffee" and on live mode it will append ".min.js". -'typescript': On dev mode this will append ".ts" and on live mode it will append ".min.js". -'js': On dev mode this will append ".js" and on live mode it will append ".min.js".

-scriptType(format) -'coffeescript': On dev mode this add type="text/coffeescript" and on live type="text/javascript". -'typescript': On dev mode this add type="text/typescript" and on live type="text/javascript". -'js': ".js": On both dev and live mode this adds type="text/javascript"

Asset Management

Asset management is a little complex, because we allow designers and developers to use CSS helper languages like LESS and SASS, or javascript helper languages like CoffeeScript and TypeScript. So this requires compiling before deployment to the server. Plus we minify these for faster loading on the live server (but in keep non-minified in dev mode).

Right now we are using a customized configuration with Gulp to find files, pipe in a compiler and out web ready files before deploying to the server. NOTE: We anticipate that in the future we will use Symfony's Encore bundle on the backend and Webpack on the front end.

Supported Formats

- LESS 2: http://lesscss.org/
- SASS 3: https://sass-lang.com/
- CoffeeScript 2: http://coffeescript.org/
- TypeScript 2: https://www.typescriptlang.org/

Dev Mode

In dev mode only, we run Webpack on the front end to compile files dynamically (with minimal overhead), so that you can test your work in dev mode without constantly compiling and deploying compiled files.

Deployment of Assets

Compiling Files

Prior to deploying files to the production server, Gulp must be run to compile web ready versions of all the files. For example, this converts a LESS file into a CSS file that can be run from a browser, or a CoffeeScript into javascript, and minifies JS and CSS for optimized loading.

NOTE: Designers do not need to worry about using Gulp, since when testing in the dev mode the system can use the raw versions of the files. Eventually Gulp compiling will be done automatically on the server. But at the moment, we run gulp on a local git repository to compile the files, then we commit to git, and deploy the latest files to the server.

Deploying Files

Sitetheory has a Python Script that runs on a server cronjob (every 2 minutes) to ensure web access to assets. This script checks all the bundles in the core src and vendor and vhost, finds which have public assets in their Resources

folder and then creates symlinks from the public /web/ folder to the below root Resources folder where these are all stored. This is necessary so that these below root files can be loaded from the web.

For nested emulated bundles (where bundles customize another bundle) we make special symlinks via the following convention:

For vhosts with customized files, we must also make symlinks:

Customization of Assets

Unlike Controllers and Templates, currently the framework will not automatically find the "best" version public asset files (e.g. CSS, JS, Images).

We haven't found or created a method to instantly override custom CSS, images, etc. To do that, we would either need to create some fancy Apache rewrite to look in alternative folders if no file is found, or else make a custom asset loader function that checks if `file_exists()` on every single asset. That would not be very efficient. So for now, we just require that the a custom Twig template is created which points to the custom asset. That means right now, you can't just drop the images or css into a directory. The advantage with this method is that there is less "magic" and the CMS is more efficient on load. NOTE: The only time a website will automatically load a custom version of a file, is if a specific website has saved a file (in their vhost folder) in the exact same web folder location as the core files (in these cases Apache will load the custom version). But this isn't the recommended method of customizing files.

Templates load public assets like CSS, Javascript and images by pointing to hard coded source locations in their bundle's public web folder. So if you make a customized version of an asset, you have to manually update the template to point to the custom location. These assets could technically be located anywhere, but for consistency, we put them in the bundle's *src* folder, emmulating the Vendor and Bundle name of the file we are overwriting, e.g. if you are editing a template called "Foo" and you want to overwrite the some CSS, Javascript or Image sfile located in the core UserBundle, you would put them in nested emulated bundle structure (within the *FooBundle/src* folder), e.g. you would save these files in the following locations:

Customize CSS and Javascript

If you have a "Foo" bundle, and you want to overwrite the core CSS and Javascript assets of another bundle, you can place these new assets in the correct nested emulated folder structure. But since these are in a sub 'src' folder that emulates the nested bundle structure, you need to use the correct symlink, that was created for this non-standard location. We do that by just referencing the original bundle with a dash and then the second bundle, e.g. *sitetheoryfoo-sitetheorybar*

Customize Image Location

The template file would look like this:

```
1     <img src="{{ asset('bundles/sitetheoryfoo-sitetheorybar/images/jazz.jpg') }}">
```

Custom Assets for Client Sites

When you are customizing files from one bundle to overwrite another, you have to make a custom template that points to a special custom file location. But when you are customizing assets in a client's website, you can take advantage of a web server (Apache) feature that will load the "best" version of the file. The system looks first in the vhost folder before looking in the core framework folders. So if you just create and save files in an emulated src folder with vendor and bundle names. The framework system will load custom Controllers and Templates from these locations.

So to overwrite the FooBundle file from:

You would put a file here: .. code-block:: shell

/var/www/vhosts/100/v/1/0/src/Sitetheory/BarBundle/Resources/public/css/baz.css

6.5.4 Vendor Files

Vendors can customize their version of core files (so all their clients will get their customized version instead of the owning vendor's version). Vendors can also create their own custom Content Type Layouts (shared with any of their clients) or Content Types (shared via subscriptions).

Customized Vendor Layouts

All Vendor bundles are stored in the platform version `src` folder under their own namespace, e.g. `/var/www/core/v/1/0/src/Sitetheory` (Sitetheory is just one vendor among many). So if a vendor called "Foo" wants to customize the Sitetheory core Profile layout, they would add the following file

```
/var/www/core/v/1/0/src/Foo/Sitetheory/ProfileBundle/Resources/views/Profile.html.twig
```

Note: normally, inside the `Foo` namespace you would have bundles only. but if the vendor needs to overwrite another vendor, they can add the vendor's namespace directly to the bundle level.

And then the actual Twig template itself can extend the core version, by including an extends at the top. NOTE: this targets the Sitetheory vendor and the Profile bundle. Twig will look for the best version of this file according to namespace paths we've registered by priority in the InitController.

```
{% extends 'SitetheoryProfileBundle::Profile.html.twig' %}
```

Customized Vendor Edit Pages

Sometimes you want to customize the edit interface for a specific content type, this can be accomplished by just adding a custom file in any of the cascading priority paths, e.g. if your vendor is "Foo" and you want to customize the "Sitetheory" vendor's files

```
/var/www/core/v/1/0/src/Foo/Sitetheory/ProfileBundle/Resources/views/ProfileEdit.html.
↪twig
```

```
{% extends 'SitetheoryProfileBundle::ProfileEdit.html.twig' %}
```

Custom Vendor Content Type Edit Pages

At the moment, if you want to have a custom content type (e.g. an edit page for a new vendor Content Type) it requires a bit of work:

#1 Make a Content Type for the edit page, e.g. ComponentEventListEdit #2 Make a Controller and Template for this edit page. #3 Subscribe the Vendor's Admin site to this new Content Type #4 Create a new page on the Vendor's Admin site with a routing URL.

So for a lot of pages that don't require custom meta (e.g. a page to create an edit page, or a non-configurable content type usually in the admin) we allow you to create and edit generic pages at /Cms/Edit which is (ContentContentEdit) page.

But in many cases, we do need to have some custom template for the contentType edit page, but we don't want to go through the entire process above. So we need to be able to just create the template for the edit page and the system

should use that if it exists rather than the generic. Just add it to the vendor's folder with the name structure of the Content Type, e.g.

```
src/Foo/ComponentBundle/Controllers/ComponentEventListEditController.php
src/Foo/ComponentBundle/Resources/views/ComponentEventListEdit.html.twig
```

Custom Vendor ContentTypes

If the vendor creates their own ContentType, they would need to create a Bundle namespace, and then a Content Type namespace (assigned to that bundle), and put their files in that bundle, e.g. for a "Component" bundle with a Content Type called "VolunteerForm" create these files

```
src/Foo/ComponentBundle/Controllers/VolunteerFormController.php
src/Foo/ComponentBundle/Resources/views/VolunteerForm.html.twig
```

If this is a custom controller, then you will just either extend the base content, or the file directly

```
{% extends content.templates.shell %}
```

or

```
{% extends "SitetheoryCoreBundle:Core:ContentBase.html.twig" %}
```

If one of your vendor Content Type templates needs to extend another vendor template, then you need to target the vendor path in a slightly different manner to point Twig to the right vendor, by using the `@` notation to target the bundle name.

```
{% extends '@FooComponent/VolunteerForm.html.twig' %}
```

If you are customizing a site and need to customize the vendor's custom Content Type, you can use the following non-standard extending format (no @ symbol targetting):

6.5.5 Client Site Files

Client Site files are located in the relevant version directory `/var/www/vhosts/{ID}/v/1/0/src` which mimics the exact structure of the core Sitetheory framework directory. To customize controllers or templates, just add the exact same file to the client's site directory, e.g.

```
/var/www/vhosts/1/v/1/0/src/Sitetheory/MenuBundle/Controller/MenuPrimary.php
/var/www/vhosts/1/v/1/0/src/Sitetheory/MenuBundle/Resources/views/MenuPrimary.html.
↪twig
/var/www/vhosts/1/v/1/0/src/Sitetheory/MenuBundle/Resources/public/css/menu.css
```

Controllers must include the same namespace and object name as the original file as well. They literally are identical.

Customizing a Vendor Version

Whether the vendor has created a custom Content Type, or just customized a version of some other vendor's layout, the site can make their own custom version of the same file and the system will give preference to the Site's version. However, sometimes the site wants to use the Vendor's file, but just customize part of it. In this case, the site would create their own version of the template, but at the top "extend" the vendor's version. In order to do that, they must properly target the Twig template they are extending, by pointing to the vendor's version with the `@` notation. In this case it has the Vendor "Foo" and then the the vendor "Sitetheory" (which the Foo vendor is overwriting when it created it's version), and then the bundle name (without the word "Bundle").

```
{% extends '@FooSitetheoryStream/Profile.html.twig' %}
```

Customizing Unique Instances of a Page

If you need to customize a controller or template for a unique instance of a page, i.e. a specific `Content` ID (not just the generic controller or template for every instance of that content type), you can do that too! Just put the file in the same location as the generic file, but append the id to the end of the name, e.g.

```
/var/www/vhosts/1/v/1/0/src/Sitetheory/MenuBundle/Resources/views/MenuPrimary12345.
↪html.twig
```

For Controllers, since you append the contentId to the filename you will also need to append it to the classname, e.g.

```
/var/www/vhosts/1/v/1/0/src/Sitetheory/MenuBundle/Controller/MenuPrimary12345.php
<?php
class MenuPrimary12345 extends ContentController Base
{
    // rest of code here
}
```

6.5.6 Template Files

The same principle applies to Design Template files, but there is a slight alternative structure for where to put the files in the Design Template bundle.

Note: Templates are all located as bundles in their vendor's folder, e.g. the Sitetheory vendor has an "Admin" template, so it's located in `src\Sitetheory\TemplateAdminBundle`.

If you need to customize the Controller of another bundle (regardless of the vendor owner of that bundle) then you will simply put a file in the Template's src directory in subdirectories that mimic the core src directory, e.g.:

Templates will be located in the same cloned structure, e.g.:

Note: TODO: Assets

The framework should reference asset files in the same namespace as the original, e.g. `@SitetheoryCoreBundle/Resources/public/css/dash.css` should find files in `@SitetheoryTemplateAdminBundle/src/Sitetheory/CoreBundle/Resources/public/css/dash.css` if they are customized and exist in that location.

6.5.7 Custom Layout Controllers

In order to allow flexibility with executing custom functionality for each layer of design, we load 3 different types of controllers (if they exist) and execute their indexAction() (usually only the content type controller will exist). These can all load independently (they are not exclusive):

#1 Template: add an initController.php#indexAction() method in the template to execute on every page (e.g. to control template or entire site) #2 Layout: add an initController.php#indexAction() method in a Content Type layout, to give added functionality for every instance of when a particular layout is loaded. #3 Content Type: add an initController.php#indexAction() method in a ContentType controller for every instance of Content Type (regardless of

layout). #4 Unique Content ID: add an initController.php#indexAction to a specific contentId instance, e.g. Profile12345.php.

We only load one Template for the contentType, and that template extends other templates upward to the shell and base templates. But we need to find the best type of template, e.g. the contentType could be customized for:

#1 ContentType #2 Specific ID of page #3 Specific EditID of content being Edited

Each of these controllers and templates needs to look for the "Best" version in cascading location priority (See cascading priority list at top of page):

Templates

If a template requires a special customized controller, you can create that controller in the template bundle, e.g. *SitetheoryTemplateCustomBundleControllerTemplateController.php*. This will load and execute before the ContentType controller.

Layouts

Some layouts may require a custom controller. This can be accomplished by creating special files that the system looks for. If we look at the StreamBundle *Landing* contentType, the normal files will be: - Controller: *SitetheoryStreamBundleControllerLandingController.php* - Layout Template: *SitetheoryStreamBundleResourcesviewsLanding.html.twig*

Let's say we created a custom layout for the Landing ContentType and gave it the variable of *Candidate*. The system will then look for the specific Candidate layout controller and twig: - Controller: *SitetheoryStreamBundleControllerLandingCandidateController.php* - Layout Template: *SitetheoryStreamBundleResourcesviewsLandingCandidate.html.twig*

A Client may customize the layout controller as well by using the same naming convention in their vhost folder.

6.6 How to Customize Templates

6.6.1 Modify Body Tag

You can modify the template <*body*> tag by extending the block or adding attributes to the view.body.attributes object (required if you want to merge with existing attributes like class).

Add Attributes Directly

```
{% block bodyAttributes %}
    {{ parent() }}
    data-spy="scroll" data-target="#mainNavigationContainer
{% endblock bodyAttributes %}
```

Modify Existing Body Attributes without Overwriting, e.g. class

```
{% block bodyAttributes %}
    {% set bodyAttributes = bodyAttributes|merge({'class':'foobarstache'}) %}
    {{ parent() }}
{% endblock bodyAttributes %}
```

6.7 How to Make an Entity Versionable

Before you make an entity versionable, be sure to read the overview documentation for Versioning to understand how versioning works.

6.7.1 How to Create the Versionable Entity

Add Common Trait to Versionable Entity

The versionable entity registers the name of the parent entity and then the trait dynamically references the parent in a getParent() method. This same Interface and Trait is used on all versionable entities, regardless if the parent references one or multiple versionable entities.

```
<?php
class ContentVersion extends Base implements VersionEntityInterface {
    /**
     * Use Shared Versionable Traits
     */
    use Sitetheory\CoreBundle\Entity\VersionEntityTrait;

    /**
     * Register the name of this entity, as it's referenced in the parent.
     * @return string
    */
    public function getEntityName() {
        return 'contentVersion';
    }


    /**
     * Register Parent Entity Name
     */
    public function getParentName() {
        return 'content';
    }


    /**
     * @ORM\ManyToOne(targetEntity="\Sitetheory\CoreBundle\Entity\Content\content",␣
↪inversedBy="contentVersion")
     * @ORM\JoinColumn(name="contentId", referencedColumnName="id", nullable=true,␣
↪onDelete="SET NULL")
     */
    protected $content;

    /**
     * @ORM\Column(type="integer", nullable=true)
     */
    protected $contentId = NULL;
}
```

Add Clone Method to Versionable Entity

The entity will be cloned every time an version is iterated. So some standard functionality should be added.

```php
<?php
public function __clone() {
    if($this->id) {
        $this->setId(null);
        $this->setSiteId(null);
        $this->setLockVersion(1);
    }
}
```

6.7.2 How to Create a Parent Entity of a Versionable Entity

Add Common Trait to Parent

If this parent entity has only one versionable entity (e.g. Content with ContentVersion), then use the `VersionParentInterface` and `VersionParentTrait`. If this parent references more than one versionable entity, use `MultipleVersionParentInterface` and `MultipleVersionParentInterface`

```
<?php
class Content extends Base implements VersionParentInterface {
    /**
     * Use Shared Versionable Traits.
     */
    use Sitetheory\CoreBundle\Entity\VersionParentTrait;

    /**
     * Define the Container Manually
     */
    protected $contentVersion;

    /*
     * Manually define the versionable entities, required for generic␣
↪EditControllerBase
     */
    public function getEntityVersion() {
            return array(
                'contentVersion'
            );
        }


    /**
     * Manually define the getters/setters for container (required for symfony␣
↪functions that reference this, e.g. form type)
     */
    public function getContentVersion() {
            return $this->contentVersion;
        }
        public function setContentVersion($contentVersion) {
            $this->contentVersion = $contentVersion;
            return $this;
        }

    /**
     * @ORM\OneToMany(targetEntity=
↪"\Sitetheory\CoreBundle\Entity\Content\ContentVersion", mappedBy="content", cascade=
↪{"persist", "remove", "detach"}, orphanRemoval=true)
```

(continues on next page)

(continued from previous page)

```
     */
    protected $contentVersions;
    Versionable Entity Repository
    Implement Trait Interface & Custom Interface Methods
    class ContentVersionRepository extends EntityRepository implements␣
↪VersionRepositoryInterface
    {

    /**
     * Use Shared Version Trait Methods
     */
    use Sitetheory\CoreBundle\Entity\VersionRepositoryTrait;

    public function getPreview($id) {}
    public function getLive($id) {}
}
```

6.7.3 Registering Information for Dynamic Versions

If a parent entity has a "Fixed Version" relationship with a versionable entity, the parent must register the versionable entities via an array returned in `getEntityVersion()`. And the parent entity implements the VersionParentTrait that includes a getVersion() method. This lets you pass in the the name of the property where the dynamic version is stored (e.g. `getVersion('contentVersion')`) and that aliases to the `getContentVersion()` method.

But in some cases, you may not know what the versionable entity is offhand, but you just need to know whether it's published or not, e.g. in a generic edit template. So in those cases you can just call that method without an entity name and it will fetch the first versionable entity. For example, when editing a content (e.g. `ArticleEdit`), the editor will be set to edit the Content entity. The editControllerBase will put this Content entity into `$initController->content['entities']['editor']` which is accessible in the template as `{{ content.entities.editor }}`. So if you call `{{ content.entities.editor.version.timePublish }}`, it will get the timePublish for the `content.contentVersion` entity, since that is the first (and only) versionable entity.

Entities Associated with the Versionable Entity

The sub entities associated with the versionable entity (e.g. each content type, ContentSettings, etc), need to register their rootParent so that we can update the rootParent's mod time, e.g. update Content when Article is edited.

```
<?php
public function getParent() {
    return $this->getContentVersion();
}
public function getRootParent() {
    return $this->getParent()->getParent();
}
```

Find the Entity Version

The VersionParentTrait provides methods for interacting with the version, by specifying the dynamic entity name, e.g. `$content->getVersion('contentVersion')`. However, the parent entity MUST define the getters and setters for the associated versionable entity anyway, in order for symfony to function properly. So this dynamic

method should NOT be used (it's slower). It's ONLY needed for some dynamic internal reasons. You should always use the custom defined getter/setter, e.g. `$content->getContentVersion()`, `$site->getDesign()`, etc.

Associate the Correct Entity Version

Once you've set up an entity correctly, your controller can simply call the correct method on that entities repository. This will find the correct version based on the environment mode (live or preview). You can use the default repository methods to find the version by a specific id.

```
<?php
/**
 * VERSIONING
 * Get the Best ContentVersion of the Content based on the environment view mode
 */
$contentVersionRepo = $em->getRepository('SitetheoryCoreBundle:Content\ContentVersion
↪');
$contentVersionRepo->associateVersion('ContentVersion', $content, $this->env->
↪getMode());
```

Iterate Versions

Version iterations happen in the IsVersionableListener, which calls the onFlush event in the entity repository. We need to see if we can make a generic version of this.

TODO: * Parent entity clone needs to clone the associated entities. Child entities need a clone. * addVersion needs to be part of trait.

6.8 Angular Filters

6.8.1 Time

moment

Format a Unix timestamp into a human readable string. The default moment format is *MMM Do YYYY, H:mma* e.g. Jan 1st 2016, 2:30pm.

Arguments You can pass in arguments by specifying a JSON string, e.g. *moment:{format:"M/D/YY"}* - *format* (string): specify the [moment time format](http://momentjs.com/docs/#/displaying/). - *since* (boolean): specify that the date should be displayed as a time "since", e.g. "1 day ago". (default false) - *relative* (string): time that the *since* value should apply, after which it should go back to the format. Specify the time based a number and time identifier, e.g. *1y*, *4w*, *7d*, *8h*, *15m*, *30s*, or join multiple like *7d8h*. (default *1w*)

Example: {{ model.data.time | moment:{since:true,format:"MM/DD/YY, h:mm a"} }}

truncate

Shorten a string of text to specified value.

Arguments You can pass in arguments by specifying a JSON string, e.g. *truncate:{limit:25}* - *limit* (int): the max number of characters to display. - *suffix* (string): the suffix string that should be added to the end of the truncated string. Enter an empty string to not append any suffix. (default . . .)

Example:

```
{{ model.data.title | truncate:{limit:25, suffix:""} }}
```

6.9 How to Use Stratus Components

You can add a lot of really powerful Stratus components to your HTML to add functionality to your design. Components can be UI functionality intended for Editing in the admin (e.g. Theme Selector) or for the public websites like (e.g. Carousel for slideshows). Other components are built to be logic on the page (e.g. Lazy Loading which loads the right size image for the container, only when it's needed).

See our Stratus-Components documentation for available components and how to use them.

6.10 How To Utilize Special Conditions in the Body Class

The Stratus.js adds a lot of useful CSS classes to the HTML body, which can assist you with styling under certain conditions.

```
<body class="SitetheoryArticleBundle-Article contentId-12345 layout-Main mac chrome␣
↪version47 loaded">
```

6.10.1 Browser

The OS, and Browser and version is specified, e.g. mac chrome version47. This lets you customize special styling rules for specific browsers (usually as a fallback if a browser doesn't support some desired styling).

6.10.2 Load Status

-loading: when the page is in the process of loading (the HTML structure is there, but the image resources are not). This is useful if you want to make an element look a certain way while the page is loaded, e.g. a page loader animation.

-loaded: when the DOM is finished loading and the images are fully loaded. This is useful if you don't want an animation to start until after all the images are loaded.

-unloaded: when the DOM is unloaded (e.g. link out or reload). This is useful if you want to trigger an animation when the page is unloading.

6.10.3 Content Type

The Content Type of this page, e.g. SitetheoryArticleBundle-Article. This lets you style some common element differently on specific types of pages, e.g. make the page title look differently on Articles.

6.10.4 ID

The id of the page, e.g. contentId-12345. This is useful if you need to style something differently for one specific page.

CHAPTER 7

Security

7.1 Overview

We have a set of fuzzy logic that brings varied levels of granularity.

7.1.1 Restricting Access

If you create a permission (or role with permissions) for a specific Asset (Site, User, Bundle, Entity, content record, etc), then that entity is immediately made private and can only be accessed by other Identities with the same permissions (or role with permissions).

For example, if you create a page on your site called "Member Dashboard" and you want to restrict access to this page, then you would create a role called "Member" and give it permission to view this new page (as well as other pages that you want restricted to members). As soon as you give one identity (Vendor, Site, or User) access to a specific Asset, that asset is no longer public anymore.

7.1.2 Admin Restrictions

If you have a website and someone creates an account on that site, most likely they will be assigned to a predefined "member" role. You would have created that role, and given that role view access to a specific group of pages that only "members" could access. But the user will not have any "edit" permissions. So if they sign in to the admin control panel (e.g. admin.sitetheory.io) the system will not recognize that they have access to edit any sites, the sites dropdown menu at the top of the page will not show any sites to switch to, and going to ?siteEditId=x will not work because they do not have edit permissions for any site. They can create their own site and edit that, but simply having "member" role on the site doesn't give them editing permissions on any website.

7.1.3 Entity Restrictions

By default all Assets (Vendor, Site, User, Bundle, Entity or Content Records) are viewable by the public. So if you have an entity for BillingBundleBillingMethod, and someone went to /Api/BillingMethod then they would be able to

see all billing method records for the current site. But as soon as we create a permission (or role with permissions) restricting the BillingBundle, then this content is restricted. The same is true of /Api/User, but fortunately whenever a new user is created, we create a permission that makes them the owner of their own user, which renders all their information private.

That means we have to be very careful to always create permissions for anything that could be private. Fortunately we also have field based restrictions defined/hardcoded on the entity itself.

7.1.4 Field Restrictions

When an entity is defined, the individual fields have annotations that specify whether each individual field is readable, searchable, writable,

7.2 Permissions

7.2.1 How Permissions Work

[TODO] Provide a one paragraph overview.

Permissions on Sitetheory are extremely granular and powerful. They fall in line with a basic structure: * Scope * Identity * Asset * Sentinel

7.2.2 Entities

It's important to understand the entities and their relationships involved in controlling permission. Permissions are controled in the Core universal database (versus the Nest which is limited to a cluster, e.g. site content).

User

Users (*core.user*) are universal to the entire sitetheory system (NOTE: this may change). They can be granted permission to "assets" (site, bundle, entity), by defining their "UserPermissions". This allows a User, once authenticated, to assume other identities in the form of Roles as well as gain permissions for any shared or owned content.

UserPermissions

Permissions (*core.user_permissions*) can be granted for any "identity" (user or role) to access any "asset" (site, bundle, entity), and confined to a specific "scope" (vendor, site).

Standard Fields for Permissions

NOTE: normally siteId, vendorId & userId are the standard fields defining what site and user created the record the first time. But in the case of user permissions, that is not the case, since permissions are universal. When a user is first created by the system, it will define the site as the one where the user was created, and the user as itself. FYI, The normal "audit trail" fields are *editUserId* (if the permissions are changed by a user at some point, we record who made the change, but that isn't related to the permission level.

Control Fields on UserPermissions

Control fields create "Locking Realms" for permissions. They contain:

- scope: Site or Vendor
- siteId, vendorId: these are the who "owns" the record when it was first created, they determine the scope.

master: This should be NULL for most permissions and only set to 1 if you want to elevate a user to have "root" access to all assets (SECURITY RISK!). Only some people on our internal team will have this ability, so that our support team can access sites without being granted specific access.

- Identity Fields - Authentication (either user or role, not both)
- identityUserid: the user being granted permissions.
- identityRoleId - the role being granted permissions.

Assets Fields - Correlate to Entity

- asset: a string defining a bundle, or an entity that an identity is being granted permissions for. If this is a bundle, all entities in the bundle are granted access. If this is an entity, all entity records are granted permissions, unless a specific assetId is set in the other field.
- assetId: the specific asset record id being granted access.

Access Control - Correlate to permissions

- permission: a byte value (bitwise inclusive "OR")
- scopes: Site or Vendor

Role

Roles (core.user_role) are an "identity" (like User) that can be granted permissions. Role is a list of all the roles created for specific sites. Roles allow you to create premade permissions that can assign standard roles to users (and change permissions of entire groups of people assigned to that role, e.g. moderator, writer, editor, publisher, etc). The role then must have one or more permissions (user_permissions) assigned to it in order to define the access of that role. AssignedRole (core.user_assigned_role): *AssignedRole* is a simple entity that associates a user with a specific role.

Settings

Settings (*core.user_settings*) is not currently used because most user settings are need to be unique for each site, so they are stored in the *nest.user_profile*.

UserProfile

Every user stores profile and preferences uniquely for each site (*nest.user_profile*), these can vary from site to site, since users are global on our entire system.

7.2.3 AuthHelper Sentinel

The authHelper constructs a "Sentinel" for every asset that determines what a given user can do with it (e.g. create, read, update, delete, etc). The Sentinel access flags are based on the "Asset" (e.g. Site, Content, Media, etc) for the current "Scope" (e.g. Site or Vendor) and specifies what access is available to the "Identity" (User Role) based on the permissions for that Identity.

The Sentinel is requested "on demand" when the system needs to know, e.g. EnvironmentHelper needs to know the Sentinel for the current site, the APIController needs to know the Sentinel for the current asset/record being edited, etc.

The Sentinel Entity is very simple, it just contains true or false flags for the basic CRUD permissions (plus a few extra) of create, edit, delete, publish, design, dev. These can then be checked on a case by case basis, to know if a user has access to the asset.

Example: Site Permissions

A user has permissions defined which give access level to a specific site. In the EnvironmentHelper we create a Sentinel for the Site by calling AuthHelper#getSentinel()

AuthHelper#getSentinel() uses Fuzzy Logic (https://en.wikipedia.org/wiki/Fuzzy_logic) (learn more: https://plato.stanford.edu/entries/logic-fuzzy/#FuzzLogiVagu) to determine if someone should be allowed access based on their defined permissions. This requires fuzzy logic, because they may have permission to the site, but not specifically to the article. But if the article hasn't been locked (e.g. a specific permission defined for that article) then their permission level to the site is used to give them access to the article. We just look up the tree to see the closest permissions they have. The priority order is: - Site - Bundle - Entity - Record

The AuthHelper#getSentinel() fetches permissions on the current asset that you passed. Which means it checks to see if there are any "permissions" set for the current asset. If there are no permissions defined, then there are no restrictions, which is why when we first create a site, we have to assign permissions to someone.

The **Sentinel** then contains true or false for each CRUD level, so it can easily be used to determine what kind of access is granted.

The EnvironmentHelper then checks to see if you have access to the Site, or the Page, or the APIController checks if you have access to view or edit a specific bundle, asset, record or field.

Debugging Permissions Issues

The AuthHelper should never need "debugging", it's just a tool that will always work the same way. But if you need to debug the authHelper, give Yourself "Auditor" (but be careful because it's a TON of backtrace information that will be dumped). To give yourself "auditor" flag the "auditor" field in your "user" record (temporarily).

In most cases you will debug the use case where the getSentinel() is called, and then dump the values going into that, e.g. the Asset, Scope, and Identity, and then look at the sentinel that is returned. Then debug back up to figure why those values are wrong, e.g. why you don't have "edit" permissions (or the permissions you expect).

Check the Permissions table to confirm what permissions you have.

7.2.4 Permission Levels

Permission Byte Values

00000001 (001) - View 00000010 (002) - Create 00000100 (004) - Edit 00001000 (008) - Delete 00010000 (016) - Publish 00100000 (032) - Designer 01000000 (064) - Dev 10000000 (128) - Master (can do anything on the current asset)

00000001 (001) Authenticated - View 00000011 (003) Writer - View, Create 00000111 (015) Editor - View, Create, Edit, Delete 00001111 (031) Publisher - View, Create, Edit, Publish, Delete 00010101 (029) Moderator - View, Edit, Publish, Delete 10000000 (128) Admin - Full

7.2.5 Example Use Cases for Permissions

New User

Ownership of Self

When a new user is created, they are granted *Permissions* to themselves (so they can edit their user account):

- vendorId & siteId: NULL (because ownership of your own user asset is not restricted to a specific site or vendor scope)

userId & identityUserId: equals the user.id

-asset: “SitetheoryUserBundle:User”

- assetId: equals their user.id
- permissions: 128 (full master ownership)

Association with a Site

The user is also associated with the site where they created their user account, by granting basic View access to this site. This allows the site to have permission to know which users were created on their site. If the same user creates an account on another site, a new user is not created, instead they just get another view permission on that site. The site will only have permission to view the username, email, and phone of the user (plus any other public information or information the user agreed to share).

- siteId: site.id (where user was created)
- userId: NULL (user scope not restricted)

identityUserId: equals their user.id (the owning identity) - asset: “SitetheoryHostingBundle:Site” - assetId: equals the site.id - permissions: 1 (view only)

Site Permissions (created or invited to edit)

Create New Site

When you create a new site you are granted full ownership permissions on the site.

- siteId: equals site.id just created
- userId: NULL (user scope not restricted)
- identityUserId: equals their user.id (the owning identity)
- asset: “SitetheoryHostingBundle:Site”
- assetId: equals site.id just created
- permissions: 128 (full master ownership)

Invite to Edit Site

When you are invited to edit a site that you didn't create, your permissions will be restricted to whatever access you were given by the administrator. See Proposal System for more info on workflow.

- siteId: equals site.id being invited to edit (where invitation was sent from)
- userId: NULL (user scope not restricted)
- identityUserId: equals their user.id (the owning identity)
- asset: "SitetheoryHostingBundle:Site"
- assetId: equals site.id being invited to edit
- permissions: byte value signifying permissions being granted
- Roles: Roles can be created (core.user_role) and granted to existing users who have accepted permission (e.g. have a permission record where the asset is for the current site), or users can be invited by granting a role. The process is the same as specifying permissions, it's just that permissions are assigned to the role first (roleIdentityId versus userIdentityId), and then the user is granted that role in AssignedRoles (core.user_assigned_role).
- siteId: equals site.id being edited when the role is created
- userId: NULL (user scope not restricted)
- identityUserId: NULL (permissions for entire role, not an individual user)
- identityRoleId: equals their role.id (the owning identity)
- asset: "SitetheoryHostingBundle:Site"
- assetId: equals site.id being edited
- permissions: byte value signifying permissions being granted for the entire role.

CHAPTER 8

Entities

8.1 Overview

Sitetheory Entities use a mixture of MongoDB and Doctrine ORM's architecture to maintain a scalable endpoint for any algorithmic needs.

8.1.1 Structure

Entities are associated with one of two databases (Core and Nest) in order to easier facilitate scaling. A bundle will be defined as belonging to one of these database, based on whether it's a universal entity or the data can be restricted to specific sites, e.g. the lookup for Site information must be universal but Content is site specific. There is one Core database, but clusters of Nest databases, that serve a group of sites.

Core Database

We have one Core database that contains entities that need to be accessible from all websites. We want to limit what goes in the Core database, because a universal database will be a bottle neck for scaling. For example, the Site, User, Template entities are in the Core, because every website needs to reference information from these entities.

Nest Database

The Nest database contains entities that are created by and for specific websites. For example, Content, Article, Routing, Media, etc. Every website is assigned to a specific Nest, which is a cluster of web and database servers. The entities on the Nest database, contain data specific to the websites associated with that nest. So one Nest may serve 1000 websites, and contain 5 web servers and 5 database servers accessed in round robin. The web servers and database read servers can scale horizontally to handle increased traffic, but the nest is prevented from getting too many sites on it, because the database write server cannot scale horizontally (unless you want to implement complex sharding). So we have one write server with as many read slaves as necessary.

8.2 Readable/Writable Properties

When defining an entity you should define which properties are readable and writable. You can specify different values for each definition beyond just the name, e.g. ['attribute' => 'view', 'alias' => 'SitetheoryCoreBundle:Contentcontent']

attribute: the property name

alias: The path to an associated entity. This is usually not necessary if the property is a valid associated entity because the tree builder will find this based on the target's repo->classMetaData()->getAssociation Mapping(). But in cases like content.contentVersion where there is no doctrine association, but we want to tree build the contentVersion readable properties, we need to include this.

searchable: true|false determines if the field is searchable

joinable: true|false determines if the field should be auto-joined when the field is searched. If it's not searchable, it will never be joinable. You don't need to mark joinable=>false if searchable=>false already.

level: integer determines whether the field is searchable or readable (API finalizer) if it's found as a readable property on nested entities, beyond a certain level. With a value of 1, it will be searchable/readable only on the first level, e.g. routing.timeEdit is set as direct=>1 because we don't want to see the timeEdit of routing when we fetch the content.

sentinel: an array that contains the sentinel required in order to interact with the field, e.g. contentVersionNotes has sentinel=>['edit']

CHAPTER 9

Internals

9.1 Overview

9.1.1 Entity & Event Listener Timing

We regularly use Entity and Event listeners to execute specific actions. It is critical that we create those actions in the right type of listener, so that they are able to modify the entity at the right sequence. Failure to execute your code could result in your changes not being persisted, or other dirty states that will create errors.

NOTE: we have customized Doctrine's UnitOfWork file to add custom listeners that were lacking, and this has not been accepted by Doctrine's team (yet).

Below is a summary of the timing and order of operations for the UnitOfWork.php (not real PHP code) to help you understand what order methods are executed for the Listeners:

```
em->persist() {
    prePersist()
}

em->remove() {
    preRemove()
    ->scheduleForDelete
}

em->flush() {
    preFlush()

    ->computeChangeSets() || ->computeSingleEntityChangeSet

    onFlush()

    ->MySQL Connection

    if (new) {
```

(continues on next page)

(continued from previous page)

```
        ->executeInserts()
        postPersist()
    }

    if (update) {
        preUpdate()
        ->executeUpdates()
        postUpdate()
    }

    ->associationDelete()
    ->associationUpdate()

    if (remove) {
        ->executeDeletions()
        postRemove()
    }

    PostFlush()
}
```

CHAPTER 10

API

10.1 Overview

Aside from handling regular website page requests, Sitetheory also handles all API requests through the /Api url on any domain. Sitetheory is a **REST**ful API using the standard methods of (GET, PUT, POST, DELETE, etc) for interaction with all entities and standard actions for viewing, creating, editing, and deleting content. Everything is accessible through the API and can be requested in JSON or XML. Access is controlled through a robust permissions system (based on your user account), with granular role or user based access to assets (site, bundles, content type, specific records or even specific fields). Additional SOAP style requests are available in the API for advanced filtering, e.g. you can pass in options to a custom Entity API Controller, e.g. ContentApiController accepts "options" *?options[showContentInfo]=true.*

10.1.1 SECURITY

IMPORTANT: All API calls should be to the HTTPS (SSL enabled) version of the URL (so you do not send information in plain text)!

The API will only allow you to interact with the data, based on your permissions. Some entities and records may be viewable by the public but not editable (e.g. Content) while others will be completely private (e.g. Billing). Others will be a mix, based on the annotation of the fields (e.g. User will expose the public username and avatar, but email and other private fields will be visible only to the user or a site administrator).

10.1.2 Basics

Methods

Use the appropriate HTTP method to determine what kind of action you want to take. These RESTful methods are mapped to internal API methods that correlate with the default behavior.

GET: select one or more records, using our internal *getAction()* method. *PUT*: edit one or more records, using our internal *setAction()* method. *POST*: create one or more new entities, using our internal *newAction()* method. *DELETE*: delete one or more entities, using our internal *delAction()* method.

Targeting a Specific Entity

You can target **any entity** by the specifying the entity name, e.g. Profile can be found at /Api/Profile, Users at /Api/User, etc.

For example, get all articles by doing a request for: *GET /Api/Article* Or get a a specific article by appending the ID of the article: *Get /Api/Article/12345*

10.1.3 API Structure

Data is transferred to and from the API in a **payload**. When you receive a response from the API, you will receive a full **convoy** which will contain an outer object that will contain a *route*, *meta*, and *payload*. When you send to data to the API, the full convoy structure is optional (e.g. you don't need a *route* and *meta* because the essentials are specified in the Method and URI structure), so you could just send the payload as the top level (without a *payload* wrapper).

Route

The *route* contains information for where the payload needs to go, whether it be a specific *controller*, *user*, or some other destination. If this is filled out by using a *RESTful* request like */Api/User* then we've already specified the controller in the URL and that is why it's not necessary to include a *route* when sending a request to the API.

Each API request is sent to a Controller based on the name of the Entity, e.g. if you want to edit the

Meta

The *meta* always contains a *method* and *status*, at the very least. The methods are *get*, *set*, *new*, and *del*. These can also be set by using a *RESTful* interface of *GET*, *PUT*, *POST*, and *DELETE*, respectively (which is why the Meta is not required for sending to the API). There is also a *PATCH* option, but it is synonymous with *PUT*, as they are both designed to be a patchable request. The *status* should not be set unless we encounter an error. If nothing fails at the time of serialization, the system will automatically place a *SUCCESS* status into the correlating array. There may be other bits of information inside the *meta* as required by the destination.

The *Meta* also be used to send data to the API, e.g. if an custom Entity API Controller needs extra *options* these can be put into the meta (instead of sending through the URL).

Listing 1: foo.js

```
{"meta":{"options":{"showContentInfo":true,"allRouting":true}}}
```

Payload

The *payload* is the entity data that should be created or edited on the database. The entity may be a nested object of associated entities. The *payload* can also be a collection of entities.

Example of Full Convoy

This could be a typical response from the API.

Listing 2: foo.js

```
{
    "route": {
        "controller": "User"
    },
    "meta": {
        "method": "get",
        "status": [
            {
                "code": "SUCCESS",
                "message": "Successfully executed request."
            }
        ]
    },
    "payload": [
        {
            "id": 1,
            "username": "Plato",
            "email": "plato@epistemology.edu"
        },
        {
            "id": 2,
            "username": "Aristotle",
            "email": "aristotle@metaphysics.edu"
        },
        {
            "id": 3,
            "username": "Socrates",
            "email": "socrates@maieutics.edu"
        },
        {
            "id": 4,
            "username": "Nietzsche",
            "email": "friedrich@nihilism.org"
        },
        {
            "id": 5,
            "username": "Kierkegaard",
            "email": "søren@existence.net"
        }
    ]
}
```

Example of Simple Convoy

This could be a typical PUT to the API to edit one field on the record. Note that there is no *payload* wrapper, it's just the single field (not even a full object). The API request would specify the *Route* Controller (User) and the ID being edited (so you don't need to include that in the payload):

PUT /Api/User/1

Listing 3: foo.js

```
{
    "email": "plato@epistemology.edu"
}
```

10.1.4 API Request Lifecycle

1. The Request

Send a request to the /Api and specify the HTTP **method** (required), **controller** (required), **ID** (optional), and **convoy** (optional).

2. The APIController

All requests to /Api are routed to the **ApiController** (*SitetheoryCoreBundleControllerApiController.php*) instead of the InitController that is normally executed for regular page loads. Like the InitController it controllers the high level routing and environment. It will detect the method being used (e.g. GET, PUT), as well as what entity you are targeting so that it loads the correct entity API controller. It passes this information to the custom entity API controller. It also interprets the convoy being requested or assembles it to send back to the requestor.

Initialize()

This runs the *initialize()* method on the API controller which does the initial setup of the API (extended from the shared EntityApiController, see #4 below).

[METHOD]Action()

Based on your requested method (e.g. GET) this will run the related action on the the Content Entity API Controller, e.g. *getAction()* (extended from the shared EntityApiController, see #4 below).

Finalize()

This runs the *finalizer()* method on the API controller which does the initial setup of the API (extended from the shared EntityApiController, see #4 below).

3. Custom Entity API Controllers

Every entity that is accessible in the API will have a controller, e.g. Article has a custom API controller found at **ArticleApiController** (*SitetheoryArticleBundleControllerArticleApiController.php*). This controller may just be a stub, because not every entity needs special API functionality (the default behavior is sufficient). But in this case the articles are a ContentType that function as a routable page on the site (e.g. like Profile, Event, Stream, etc), so this controller actually extends the shared **ContentApiController** (*SitetheoryCoreBundleControllerContentContentApiController.php*) because it shares a lot of similar functionality with all other page related Content.

All API controllers also extend the **EntityApiController** (*SitetheoryComponentBundleControllerEntityApiController.php*), which does the heavy lifting for managing the lifecycle of an API request for selecting, editing, creating, and deleting records, e.g. standard searching/filtering, permissions control, etc.

We often need to customize the data for specific entities, e.g. if a Profile is requested (or any Content), by default we also want to fetch the Route, the best version, and the related meta data for profiles. So in each entity's custom API controller we extend methods from the *EntityApiController* to modify the database lookup (e.g. join additional tables). So each Entity API Controller has full control over the lifecycle of the request.

Custom Actions

Be aware that the bulk of the code referenced below are actually in related "default" methods, e.g. *initialize()* calls *initializeDefault()*. The default versions of these methods are used **most of the time**, but you can create custom actions, by telling the API to use a custom API action, e.g. *?options[action]=fancy* or *{"meta":{"options":{"action":"fancy"}}}*. This would make initialize() execute *initializeFancy()* which would also execute *getActionFancy()* instead of *getActionDefault()*. Then you can define these custom methods in your custom entity API controller.

4. The EntityApiController

This is a very high level overview of the lifecycle of the *EntityApiController*. We don't want to document this here in case there are changes. Instead, the code is heavily documented so you can read what it does there.

initialize()

-Merge Default Options from Custom Controller (if exist) -Get Options from Request URL, and Convoy Meta -Manage Access Control (allowed actions for this Entity)

If no custom *action* is specified, the default version *initializeDefault()* method is run. This default method is often extended to instruct the API fetch additional associated entities. See *ContentApiController* for example.

Method Specific Actions

Depending on the type of Method requested, the relevant method will be used. Each action will verify that you have the right CRUD permissions to act on the entities, based on your permissions and the **Sentinel** (See /1.0/Security/Overview for more details about security and permissions).

- **GET** *getAction()*: This gets the requested records and return frames, which are then set in the convoy payload.
- **PUT** *setAction()*: This fetches the records being edited and then executes the *persist()* method to apply the changes to the records it just fetched and persist the changes to the database.
- **DELETE** *delAction()*: This deletes the requested records.
- **POST** *newAction()*: This creates new records.

persist()

This persists changes to the entities (e.g. for PUT, POST and DELETE methods). This is smart enough to persist cross entity managers! It also references the Entity Annotations to determine CRUD access level on a per field basis.

This is where the crazy starts. You will have to step through this method line by line (and really it's the persister() that does the recursive "Tree Building").

- Uses "Tree Building" to recurse through nested entities.
- Hydrates Associated Entities (when an ID changes, e.g. Site.SiteVersion.theme changes to a new template).

- Validates CRUD permissions to edit on every nested entity and field.
- Merges in Changes for Persisting
- Handles AutoVersioning of Versionable Entities

Many problems with the API are likely caused by issues in the complex *persister()* with permissions that result in changes to entities (or fields) to be discarded.

finalize()

Finalize Structures the entity data that you send back from the API to the requesting script. It is called for all methods (e.g. GET, PUT, POST, etc). The *finalizeDefault()* is often customized to manipulate data before the request is returned. (see ContentApiController for example.)

manifest()

This is a special functionality to "Manifest" an empty new entity and it's associated parents and/or children. This should be added entity API controllers that have manual associations that need to be manifested, e.g. Content Integration (see ContentApiController).

10.1.5 Admin Lists

For the purpose of editing (e.g. on List Pages) in the admin context, the API adds the editUrl in the meta data it returns, so that you can know where entities should be edited. This is based on the entity's controller, but sometimes you need to specify an alternative URL. That can be easily customized for an entire entity by editing the entity's custom ApiController, e.g. for the Site entity, you edit the SiteApiController and add options like this:

```
protected $options = [
        'altEditUrl' => [
            'bundle' => 'Hosting',
            'controller' => 'SiteSettingsEdit'
        ]
    ];
```

Or if you just want an alternative editUrl in specific widgets, just add it to the data attribute like this:

Listing 4: GET Variable

```
data-api='{"options":{"altEditUrl":{"bundle":"Hosting", "controller":"SiteSettingsEdit
↪"}}}'
```

10.1.6 Advanced API Options

Limits and Paging

The *meta* object of the response contains pagination information that describes how the total records, current records on this page, and total pages.

Listing 5: Pagination

```
{
    "pagination": {
        "countCurrent": 25,
        "countTotal": 100,
        "pageCurrent": 2,
        "pageTotal": 4
    }
}
```

You can modify the how many records are returned and which page you want to view by passing variables to the API either through the URL or through the meta.

Listing 6: Meta

```
{"meta":{"options":{"page":2,"limit":10}}}
```

Listing 7: GET Variables

```
/Api/Content?page=2&limit=10
```

Paging

By default the API loads the first page (if more records than one page exist), so you can pass in a variable to specify the page you wish to receive.

Variable: *page* or *p* Type: integer Example: */Api/{ENTITY}/?p=2*

Paging Type

By default all content will be paged after a specific max limit. TODO: this may not be implemented yet (or relevant since infinite scroll is really just the front end UI making paging requests as you scroll. Variable: *pagingType* Values: *pager* (default), *infiniteScroll* Example: */Api/{ENTITY}/?pagingType=infiniteScroll*

Limit

By default the API returns a fixed number of results (e.g. 25). If you wish to modify the number, you can pass in a limit.

Variable: *limit* or *ql* ("query limit") Value: integer Example: ‘/Api/{ENTITY}/?ql=10

Offset

By default the API returns a fixed number of results (e.g. 25). If you wish to modify the number, you can pass in a limit.

Variable: *offset* or *qlo* ("query limit offset") Value: integer Example: ‘/Api/{ENTITY}/?qlo=5

Sort

By default the API sorts by timeEdit DESC (most recent).

Variable: *sort* or *qs* ("query sort") Value: string of valid field name, which are visible in the meta.searchable fields list in the API meta object. Example: '/Api/{ENTITY}/?qs=version.title

If you need to sort by more than one field, you can pass a comma separated list of sort options. Example: */Api/{ENTITY}/?qs=version.title ASC, version.pullout DESC But the recommended method is to pass an array: Example: '/Api/{ENTITY}/?sort[version.title]=ASC&sort[name]=DESC*

NOTE: there are cases where we can pass in special conditions into the sort order and we'll parse that command. Example: */Api/{ENTITY}/?sort[version.title]=ASC:LASTWORD*

NOTE: there are special sort options that we parse to find the best way to search, by specifying the property as 'SITETHEORY:CUSTOM' and the property as the method we want to execute. Example: */Api/{ENTITY}/?sort[SITETHEORY:CUSTOM]=bestTime*

Sort Order

By default the API sorts by DESC. If you don't want to modify the field that is sorting and only want to modify the order, you can pass in just the sortOrder.

Variable: *sortOrder* or *qso* ("query sort order") Value: *ASC*, *DESC* Example: '/Api/{ENTITY}/?qso=ASC

Output Format

By default all content will be returned in JSON format, but if you prefer XML, RSS, ICS, or other relevant formats you can specify the output format Variable: *output* Values: json (default), xml, rss, ics Example: */Api/{ENTITY}/?output=xml*

Keyword Search Queries

The query parameter lets you search all the entity records, on all fields annotated as "searchable". This allows you to pass a string from a user search field exactly as formatted (giving the user more power to do complex searches). (NOTE: if you want to do searches on the API from a programmatic perspective, you should use the *filter* format specified later in this document.)

Variable: *query* or *q* Values: string Example: */Api/{ENTITY}/?q=foobar*

TODO: specify the format for limiting search to specific fields

Advanced Keyword Search Filtering

You can pass in specific fields through the query field, e.g. "title=my title". This removes the filters that were found, so other parsing will not reference them. To search for strings for all searchable fields, in addition to value for a specific field, put the general string at the front of the search and put the field searches at the end

Comparison Values

NOTE: when hard coding searches of the $options['filter']for ApiRepositoryTrait#findByFilter() in code, you should use the standard comparison values used by the database, but alternative aliases are available for use in query parameters where you don't want to URL Encode reserved characters like '='.

[=] or [!=] - comparison means the values exactly equal or do not exactly equal each other. e.g. title="my title" (query alias: [EQ] and [NEQ]) [>] or [<] or [>=] or [<=] - comparison means the values are greater than or equal. e.g. timeEdit>2014-10-14 (query alias: [GT] [GTE] [LT] [LTE]) [LIKE] or [NOT LIKE] - comparison means "contains" instead of '=' which means "exactly equal". [REGEXP] or [NOT REGEXP] - comparison means the following is a regular expression. (not implemented at this time) [IN] or [NOT IN] - comparison means the following is an IN comparison and the value should be separated by commas.

You can do advanced searches on one or more specific fields by using a special field syntax `FIELD[=]VALUE`. `FIELD` is the field name in format `entity.field` (if there is no entity it will be assumed to be the main entity 'e'), e.g. *version.title*. `VALUE` is the value (one or more words). The comparison can be:

- **exactly equals: `[=]` or `[!=]`** Example: `title[=]foo bar stache` *(the title is exactly "foo bar stache")*
- **contains: `[LIKE]` or `[NOT LIKE]`** Example: `title[LIKE]foo` *(the title contains "foo" anywhere, e.g. "foobar" or "barfoodo")*
- **greater or less than: `[>]` or `[<]` `[>=]` or `[<=]`** Note: if searching a time field, the human readable formats will be converted to a unix time stamp. Example: `time[>]2015-05-01`
- **regular expression: `[REGEXP]` or `[NOT REGEXP]`** Note: reserved Regular Expression special characters need to be commented out with a backslash "". Examples:

  `title[REGEXP]^foo[a-z]+ar` (the title starts with "foo" followed by any character a-z followed by "ar", e.g. "foobar" or "foojar") `title[NOT REGEXP]\(copy\)$` (anything with a title that doesn't end in "(copy)")
- **in list: `[IN]` or `[NOT IN]` *(the value is in the list of options)*** Note: the value should be a comma separated list. Example: `id[IN]1,2,3` *(id equals 1,2 or 3)*

Multi Part Filters

**title[LIKE]foo bar time[>]2014-10-14** - finds where title contains "foo bar" **and** time is greater than the date _**baz shazam title[LIKE]foo bar**_ - finds where content includes baz and shazam in any field **and** "foo bar" only in the title field.

Target Nested Fields

Many fields you want to search are on nested entities, so you must specify the field name in dot notation, e.g. when searching the /Api/Content the main Content entity has very few fields of interest, most of what you search is the contentVersion, so your search would be on the nested version entity, e.g. to search the title:

Listing 8: GET Variables

```
/Api/Content?q=version.title[LIKE]foo
```

Get Multiple Records by Id

Normally you would get a single record in RESTful format /Api/Media/1234, but if you need to get multiple ids you can do one of the following:

Listing 9: GET Variables

```
/Api/Content?q=id[=]1234
/Api/Content?q=id[IN]1234,5678
/Api/Content?options[id][]=1234&options[id][]=5678
/Api/Content?filter={"field":"id", "value":[1234,5678], "comparison": "IN"}
```

Select and Unselect

NOTE: this shouldn't hurt UPDATES, since the API just updates the fields you provide, and if you are missing specific fields it won't modify them.

By default all readable fields will be returned in the API. If you only want to return specific fields, you can select which fields are returned.

Variable: *select* or *unselect* Values: array of field names

Examples:

- */Api/{ENTITY}/?select[]=foo&select[]=bar*
- */Api/{ENTITY}/?unselect[]=baz&select[]=fuzz*

Filter

The simple string query parameter (above) can allow you to search all searchable fields. But if you want to search one or more fields specifically, you can pass in a filter as a single JSON array/object, or as key value pairs.

Variable: *filter* Values: array with field name and value (for exact match) or JSON string as an array with *field*, *value* and *comparison*

Examples:

- */Api/{ENTITY}/?filter[title]=foo&filter[price]=1000* (Exactly Equals)
- */Api/{ENTITY}/?filter=[{"field":"foo","value":"bar", "comparison":"LIKE"},{"field":"extension","value":"jpg"}]* (Multiple Fields)
- */Api/{ENTITY}/?filter={"field":"mime","value":"image", "comparison":"LIKE"}* (Single Field)

NOTE: the EntityApiController will compile these filters and confirm that you have permissions to search each requested field.

Flatten

Variable: *flatten* TODO: Explain what this is for.

Alternative Edit URL

If you need to lookup the URL for a content other than the current controller's corresponding Edit page, just pass in a bundle and controller. Variable: *altEditUrl* Value: Array of bundle and controller names. Example: /Api/Content?altEditUrl[bundle]=Article&altEditUrl[controller]=Article

Special API Action

Specify a special API action to run, e.g. "duplicate". Variable: *apiSpecialAction*

Options:

- "duplicate" - triggers duplication of an entity
- "iterateVersion - iterates a versionable entity

Example: /Api/Content/12345?apiSpecialAction=duplicate

API Action

Specify an alternative action (besides the default API action). This is an advanced feature if you have created a custom API controller that needs to do unique SOAP style actions that don't use the normal REST methods.

Variable: *action*

Show Assets

Specify whether to show assets or not. By default assets are shown on the main entity if they exist, but in some contexts they may not be. Variable: *showAssets* Value: boolean (default: true, but depends on context)

Manifest a Version Parent

When manifesting a new entity that is versionable, it will twiddle the entity and manifest a version parent by default. But if you need to return the version entity directly, set this to false.

Variable: *manifestVersionParent* Value: boolean (default: false)

10.2 Article

10.2.1 Overview

The Article entity is a *routable* Content Type (i.e. "page"). This API lets you fetch all Article content specifically (without other types of content mixed in). It inherits all the standard functionality and features of a Content entity and normally we would use the Content API for fetching all content or specific content based on ID. See Content API for more details /1.0/API/Content.

10.2.2 API

/Api/Article

10.3 Content

10.3.1 Overview

Content is **routable** Content Type (e.g. pages with an attached URL like "/Team") and a special "meta" (e.g. Articles, Profiles, Events, Streams, etc). These related Content Types inherit all the standard functionality and features of the

Content entity. Content is automatically versionable. This API fetches all content (NOTE: in fact, not all, just most, e.g. not Menu, Streams, etc). If you want to only fetch a specific Content Type like Article, you would use that API.

10.3.2 API

/Api/Content

10.3.3 ADVANCED OPTIONS

Include Only Specific Content Types

Limit by one or more content types Example: *options[contentType][]=Article*

Exclude Content Types

Include all content types except the ones listed to exclude. Example: *options[contentTypeExclude][]=Stream*

Show Usable Content Types

Include a list of content types that are available to be created. This would be used on a content list page, in order to generate a list of "add" buttons. Example: *options[showUsableContentTypes]=true*

Show Used Content Types

Include a list of all content types that are used by this site. This would be used on a content list page, to create filters. Example: *options[showUsedContentTypes]=true*

Show Routing

The routes are always included in the query, so that we can get the primary. But this will make sure that the full list of routes is included in the API results. Example: *options[showRouting]=true*

Show Content Info

Include the extra information about the Content, including the main image and icon (based on content type). This is used on list pages in the admin. Example: *options[showContentInfo]=true*

Show Edit URL

Fetch and show the correct edit URL (this is used in the admin list pages and live edit pages so we know where to go to edit the record). Example: *options[showEditUrl]=true*

Show Menu Links

Show the Menu Links for the current page. Example: *options[showMenuLinks]=true*

Hide Pages in Menu

Exclude pages that are already in the menu. Defaults to false, so it shows all pages. NOTE: This is a very unique use case where we are joining menus on content in a special way and we have to limit this for some reason. Example: *options[flagNotInMenu]=true*

Limit Menu to Specific ID

If set to showMenuLinks or flagNotInMenu, the default "main" menu will be used unless an alternative menuId is specified. Example: *options[menuId]=true*

Limit Fields Selected

Only select the fields listed in this array. Example: *options[select][]=title*

Avoid Selecting Specific Fields

Select all fields except the fields listed in this array. Example: *options[unselect][]=name&options[unselect][]=title*

10.4 Domain

10.4.1 Overview

The Domain entity (i.e. example.com) are associated with a particular site, and can be either the primary domain, or a secondary domain that is set to go to a landing page, or redirect to the main page.

10.4.2 API

/Api/Domain

10.5 Event

10.5.1 Overview

The Event entity is a *routable* Content Type (i.e. "page"). This API lets you fetch all Event content specifically (not other types of content). It inherits all the standard functionality and features of a Content entity and normally we would use the Content API for fetching all content or specific content based on ID. See Content API for more details /1.0/API/Content.

10.5.2 API

/Api/Event

10.6 Media

10.6.1 Overview

Media are files that are files that are used on a site, which can be uploaded and hosted by Sitetheory (e.g. image, MP3, PDF, etc) or linked externally (e.g. Youtube embed code or links, or third party file server URL.

- Image: multiple size versions stored on our file servers.
- Video: stored in database as a URL to the third party streaming service (e.g. Youtube, Vimeo).
- Audio: stored on our file servers or a URL to a third party file server
- Document: stored on our file servers or a URL to a third party file server

10.6.2 API SAMPLES

Create Media

Creating new Media is slightly more complicated than other content types because we need to upload a file first, then create a new Media record and associate that file with the Media record. Fortunately, we have a Media App server that handles the upload of files and it then it sends the POST command to the API from the server to create the Media record and associates the file with that record. Then after it successfully uploads the file and creates the media record, it will return a result. So your app should wait for an Asynchronous reply, which will include a standard Media object in JSON. You can use that information to followup with subsequent API calls to edit the record or add the media to some other entity association.

If you are using HTML, this is as simple as a form that posts like this:

Listing 10: API POST

```
<form action="http://app.sitetheory.io:3000/?session=SESSIONID" method="post" enctype=
↪"multipart/form-data">
    <input type="file" name="file" id="file">
    <input type="submit" value="Upload" name="submit">
</form>
```

Required: - *session* - You must include the session ID in the post URL - The body of the POST must be encoded as "multipart/form-data", with the fields and files split up with boundary separators.

URL: POST https://app.sitetheory.io:3000/?session={SESSIONID}

Listing 11: API POST

```
------WebKitFormBoundaryyNWzAjJ1ALa1ZByI
Content-Disposition: form-data; name="key"

Avocado.jpg
------WebKitFormBoundaryyNWzAjJ1ALa1ZByI
Content-Disposition: form-data; name="acl"

private
------WebKitFormBoundaryyNWzAjJ1ALa1ZByI
Content-Disposition: form-data; name="Content-Type"

image/jpeg
```

(continues on next page)

(continued from previous page)

```
------WebKitFormBoundaryyNWzAjJ1ALa1ZByI
Content-Disposition: form-data; name="filename"

Avocado.jpg
------WebKitFormBoundaryyNWzAjJ1ALa1ZByI
Content-Disposition: form-data; name="file"; filename="Avocado.jpg"
Content-Type: image/jpeg


------WebKitFormBoundaryyNWzAjJ1ALa1ZByI--
XXXXXXX --- FILE DATA HERE --- XXXXXXXX
```

Sample Response

Listing 12: API POST


```
{
    "tags":[],
    "storageService":null,
    "priority":null,
    "authorId":null,
    "storageServiceId":null,
    "duplicateId":null,
    "name":"austin-schmid-134030-unsplash",
    "description":null,
    "embed":null,
    "prefix":"cdn.sitetheory.io/nest001/1000/foo-bar-baz",
    "url":"//cdn.sitetheory.io/nest001/1000/foo-bar-baz-xs.jpg?v=1550596025",
    "file":"sitetheorynest001.s3.us-west-2.amazonaws.com/nest001/1000/foo-bar-baz",
    "filename":"foo-bar-baz",
    "extension":"jpg",
    "mime":"image/jpeg",
    "bytes":977258,
    "bytesHuman":"954.35 KB",
    "ratio":"3:2",
    "dimensions":"2700,1800",
    "meta":[],
    "id":1000,
    "siteId":100,
    "time":1550596023,
    "timeEdit":1550596025,
    "status":1
}
```


Fetch All Media (with paging)

URL: GET /Api/Media

Fetch Specific Media by ID

GET /Api/Media/2

Edit Media

Note: You can just send only the fields you want to update Required: - *id* - you must specify the ID you are editing (best practice is to include it in the PUT URL).

URL: PUT /Api/Media/2

Listing 13: API PUT

```
{
   "title": "New Foo Title for Image"
}
```

Filter

-Filter Media by Mime Type, e.g. /Api/Media?q=mime[:]video (mime field contains "video")

10.7 Menu

10.7.1 Overview

Menus are technically a non-routable Content Type (they don't have a URL but they are based on the versionable Content entity). The reason for this is so that they can be utilized in many contexts (in the future). At the moment they are primarily used for site navigation. The main menu is marked as the primary menu and is editable in the Admin Control Panel. Menu's have associated entities MenuLinks.

By default the Menu is not auto-versioned, but it can be manually versioned if desired by passing the *?option[apiSpecialAction]=iterateVersion*.

10.7.2 API

/Api/Menu

10.8 MenuLink

10.8.1 Overview

Menu Links are simply child entities that are defined by a particular Menu. If you fetch only the MenuLink, you will have to specify the the MenuId that they are associated with, otherwise you will get all versions of links for all menus (which is not helpful). Normally you will want to fetch the Menu content entity, in order to get the Best Version of the menu and menu links.

10.8.2 API

/Api/MenuLink

10.9 Permission

10.9.1 Overview

The Permission entity contains the information about a Permission, e.g. Identity (User/Role), Scope (Site/Vendor), Asset (Site, Bundle, ContentType, Record ID), etc.

10.9.2 Advanced Options

[TODO] Include advanced options used for Permission page or edit page, e.g. to show info about the permissions.

API

/Api/Permission

10.10 Product

10.10.1 Overview

The Product entity is a *routable* Content Type (i.e. "page"). This API lets you fetch all Product content specifically (without other types of content mixed in), which is useful in many cases where you only want products. It inherits all the standard functionality and features of a Content entity. See Content API for more details /1.0/API/Content.

10.10.2 API SAMPLES

Create Product and Associate with Media

Note: You can just send only the fields you want to set, the rest will set to default values.

Required: - *contentType* - contentType is required (in order to specify what type of content you are creating, e.g. Product, Article, etc). Product is contentType.id = 181. - *version* - the version object is required because the review has to be associated with some content that is being reviewed. - *version.meta* - the meta object is associated with the version. It is not actually required, but it includes specific fields that make the Product unique from other content, e.g. price. So it's basically required if you are creating a product.

Products, like all content must be published in order to be publicly visible. You can set the *version.timePublish* manually or set it to the current time with the value "API::NOW"

URL: POST /Api/Review

Listing 14: API POST

```
{
    "contentType":{
        "id": 181
    }
    "version": {
        "meta":{
            "price": 100.00
            "isOrganic": true
```

(continues on next page)

(continued from previous page)

```
        }
        "title":"Foo Bar Title",
        "subtitle":"Lorem ipsum dolor",
        "text":"Some text describing the product.",
        "timePublish":"API::NOW",
        "timeCustom":null,
        "images": [{"id":100},{"id":101},{"id":102},{"id":103}]
    }
}
```

Fetch All Products (with paging)

URL: GET /Api/Product

Fetch Specific Product by ID

GET /Api/Product/2

Edit Product

Note: You can just send only the fields you want to update Required: - *id* - you must specify the ID you are editing (best practice is to include it in the PUT URL).

URL: PUT /Api/Product/2

Listing 15: API PUT

```
{
    "version": {
        "title":"Update Title",
        "meta": {
            "price": 99.99
        }
}
```

10.11 Profile

10.11.1 Overview

The Profile entity is a *routable* Content Type (i.e. "page"). This API lets you fetch all Profile content specifically (not other types of content). It inherits all the standard functionality and features of a Content entity and normally we would use the Content API for fetching all content or specific content based on ID. See Content API for more details /1.0/API/Content.

10.11.2 API

/Api/Profile

10.12 Review

10.12.1 Overview

Review entity contains the information about a Review that is associated with a Content record (e.g. Product, Profile, Article, etc).

API SAMPLES

Create Review and Associate with Media

Note: You can just send only the fields you want to set, the rest will set to default values.

Required: - *content* - content is required because the review has to be associated with some content that is being reviewed. - *rating* - rating is required so the review has some numerical value.

URL: POST /Api/Review

Listing 16: API POST

```
{
    "content": {
        "id":36518
    },
    "title":"Sample Title",
    "text":"Sample Description",
    "images":[{"id":100},{"id":101},{"id":102},{"id":103}],
    "documents":[{"id":104}],
    "latitude":"100.0000",
    "longitude":"50.000",
    "analysis":"{\"status\":\"good\"}",
    "device":"{\"device\":\"iphone\"}",
    "ip":"198.168.1.1",
    "rating":4,
    "isPurchased":true,
    "enableMarketing":false,
    "isReference":true,
    "barcode":"1234567890",
    "meta":"{\"customKey\":\"customValue123\"}"
}
```

Fetch All Reviews (with paging)

URL: GET /Api/Review

Fetch Specific Review by ID

GET /Api/Review/2

Edit Review

Note: You can just send only the fields you want to update Required: - *id* - you must specify the ID you are editing in the PUT URL.

URL: PUT /Api/Review/2

Listing 17: API POST

```
{
    "title":"Update Title",
    "isPurchased":true
}
```

Add Media

Note: You can just send only the fields you want to update Required: - *id* - you must specify the ID you are editing in the PUT URL.

URL: PUT /Api/Review/2

Listing 18: API POST

```
{
    "images":[{"id":100},{"id":101},{"id":102},{"id":103}],
    "documents":[{"id":104}],
}
```

10.13 Role

10.13.1 Overview

Role entity contains the information about a Role.

10.13.2 Advanced Options

[TODO] Include advanced options used for Role page or edit page, e.g. to show a count of users in the role, etc.

API

/Api/Role

10.14 Site

10.14.1 Overview

Site entity contains the basic information about the website, along with associated settings, primary domain, etc.

10.14.2 API

/Api/Site

10.15 Site Version

10.15.1 Overview

NOTE: this is not enabled at this time.

SiteVersion contains related Site settings that are versionable, e.g. Theme, Style, etc. Normally when you fetch the Site information, the version will be joined so you don't need to fetch the SiteVersion independently in most cases.

10.15.2 API

/Api/SiteVersion

A generic call to this API will return ALL versions for ALL sites that you have access to edit. If you want to limit by a specific site, you will need to pass in the site ID, e.g. */Api/Site/100/SiteVersion*

10.16 Stream

10.16.1 Overview

Streams are a Content Type that displays content associated with specific tags. Streams come in many different layouts to fit different needs, e.g. a landing page with a big slideshow and other dynamic modules below, a blog style page with images and words flowing down the page, a grid of photos, or a simple compact text list, etc.

Streams are technically just another content type like Article, since they are essentially just a page on the website with a URL. But since they have a very specific functionality to display other content types (rather than being content themselves), it makes more sense to separate them in the normal workflow of creating content. So in the admin they exist in a separate section and when we fetch all content we generally exclude Streams (and other unique content like Menu).

10.16.2 API

List of Stream Pages

/Api/Stream - Get All Streams (pages that are Content Type "Stream" with contentType.collection=1) Example: https://dev.sitetheory.io/Api/Stream

Get Content for a specific Stream

/Api/[contentType]/[contentId]/Asset/Content - Get all Content that is tagged with the Asset for a specific stream content type (e.g. Collection, Landing, etc). This finds all the tags that a Stream is set to display, and then all content that is associated with those tags.

Example: https://dev.sitetheory.io/Api/Collection/12345/Asset/Content

Limit Content by Tag

/Api/[contentType]/Content?tags=100 - Limit the content for a stream by specific tags that may be associated with that content (unrelated to the main tag that links it to the current stream). For example, you may have 10 articles tagged "foo" and a stream that is set to display the "foo" articles. But these articles may also contain secondary descriptive tags like "bar", "baz". This lets us filter the 10 articles on the stream, by additional unrelated tags.

Example: https://dev.sitetheory.io/Api/Content?tags=100 Or Limit Only Articles by Tag: Example: https://dev.sitetheory.io/Api/Article?tags=100

NOTE: you can also further limit an existing stream by only some of the tags, e.g. if you want to have filters on the side. This would find a subset of the stream's content, based on whether any of that content ALSO had the requested overlapping or extra tags, e.g. https://dev.sitetheory.io/Api/Collection/12345/Asset/Content?tags={[}100,101]

NOTE: to filter more than one tag, pass in a comma separated value for the "tags" variable, e.g. ?tags=[100,200]

/Api/Tag/537/Content /Api/Content?tags=537 /Api/Content?tags=[537] /Api/Content?tags=[535,536,537,538]

Note: the Variable is plural ("tags")

10.17 Tag

10.17.1 Overview

Tags are a way to organize entities (usually content like Articles, Products, Profiles, etc). Content can be tagged, and then it's displayed on Streams that accept content based on tags.

Functional (Singular & Multiple):

/Api/Tag/537/Content /Api/Content?tags=537 /Api/Content?tags=[537] /Api/Content?tags=[535,536,537,538]

Note: the Variable is plural ("tags")

10.17.2 API

/Api/Tag

10.18 User

10.18.1 Overview

To see all fields that you have access to read and edit, do a generic call to the API.

SECURITY: All API calls should be to the HTTPS (SSL enabled) version of the URL (so you do not send information in plain text).

10.18.2 API

/Api/User

10.18.3 Examples

Update Password (or any other field)

Method: PUT URL: /Api/User/{ID}

Listing 19: API PUT Body

```
{"password":"abcd1234"}
```

Note: Only the user has permission to reset their own password.

Create User

Method: POST URL: /Api/User

Required:

- *email* (string) - You must provide an email (for account verification email)
- *phone* (string) - Phone is not technically required, but one must be provide for successful password resets.

Listing 20: API Post Body

```
{
    "email":"foo@bar.com",
    "phone":"1112223333"
}
```

NOTE: when the user is created, the system will send a verification email for the user to click a link to verify the email and also set a password.

Other Fields: -facebookId, googleId, twitterId: these are IDs obtained by doing single-signon verification with these users approving your app and being given a login id by Facebook, Google, Twitter, etc.

Listing 21: API Post Body

```
{
    "email":"foo@bar.com",
    "phone":"1112223333",
    "password":"Abcd1234",
    "username":"foobarly",
    "facebookId":null,
    "googleId":null,
    "twitterId":null,
    "settings": {
        "privateLegalName": "Mr. Foobert Barly"
    },
    "profile": {
        "publicName":"The Boss",
        "position":null
        "birthday":277171200,
        "gender":2,
        "relationshipStatus":null,
        "ageGroup":4,
        "zip":"44444",
        "mailLists":[{"id":100}],
        "meta":[],
```

(continues on next page)

(continued from previous page)


```
        "units":null,
        "timezone":null,
        "lat":null,
        "lng":null,
        "tracking":[],
        "tosAccepted":[],
        "device":[],
        "ip":null,
        "id":4260,
        "dates":[{"type":"date","name":"Hired","value":"1234567890"}],
        "phones":[{"type":"phone","name":"Mobile","value":"9251231234"}],
        "emails":[{"type":"email","value":"foo@bar.com","name":"Work"}],
        "locations":[{"type":"location","name":"Office","value":"100 HQ Drive"}],
        "urls":[{"type":"url","name":"Website","value":"https://sitetheory.io"}],
        "socialUrls":[{"type":"social","name":"Instagram","value":"instagram.com/
↪testing"}],
    }
}
```


Get Roles

Method: GET URL: /Api/Role

Find the role ID you want and add to a User.

Add User to Role

Method: PUT URL: /Api/User/1 NOTE: You must be signed in to edit a user, and have permissions to edit this user and assign this user to site roles.

Listing 22: API Post Body


```
{
    "Role": 214
}
```


Update Password

Method: PUT URL: /Api/User/1 NOTE: You must be signed in to edit your user.

Listing 23: API Post Body


```
{
    "password":"xxxyyyzzz3"
}
```


Update User Info

Method: PUT URL: /Api/User/1 NOTE: You must be signed in to edit your user.

Listing 24: API Post Body

```
{
    "settings": {
        "privateLegalName":"Mr. Foobert Barly"
    },
    "profile": {
        "phones":[{"type":"phone","name":"Mobile","value":"1112223333"}]
    }
}
```

10.18.4 Utilities

Login

Method: POST URL: /Api/Login

Listing 25: API POST Body

```
{"email":"foo@bar.com","password":"abcd1234"}
```

Logout

URL: /Api/Login?options[action]=logout

Just go to this URL to logout.

Request Password Reset

URL: /Api/Login?options[action]=requestPassword

Listing 26: API POST Body

```
{"email":"foo@bar.com","phone":"1234567890"}
```

NOTE: if you want to suppress email because your app prefers to handle the verification process itself, then you can pass in "disableEmail": true. This will cause the User account to become unverified (locked) to preventing logins. This also creates a "proposal" for the user, with a token that must be sent to the Verify Account API request in order to verify and unlock the account. So if you do this, you will need to handle verification manually by calling the "Request Proposal" API to get the

Listing 27: API POST Body

```
{
    "email":"foo@bar.com",
    "phone":"1234567890",
    "disableEmail": true
}
```

Request Proposal

URL: /Api/Login?options[action]=requestVerifyProposal

From an authenticated User account that has "dev" (64) permissions for a site, you can request a proposal for a specific user. This is used if your app needs to handle verification internally, i.e. send a custom verification email or handle verification in some other way.

@SECURITY: This MUST ONLY be run from a secure server (not in the app) which is able to send an verification email to the user. If you do not protect this authentication token, anyone can use it to hack a User account by sending a Verify request with this token, which will sign them in and allow them to reset a password (or change any user data).

You would send the request for a UserId, which can be found by calling the /Api/User?q={email} to lookup a user account by their email.

Listing 28: API POST Body

```
{"userId": 1}
```

Verify Account

URL: /Api/Login?options[action]=verify The token will have been returned in the Body of the "Request Password Reset" call (and also sent to the user via email). If your app prefers to handle this, you can send this token to the API to verify. If this token is valid, it will verify the account and log the user in.

@SECURITY: the user will be signed in, and can make requests to change data (e.g. update password) through the normal /Api/User controller.

Listing 29: API POST Body

```
{
        "token": "5cee063a533467d244d1be7c56238bb93807de602f3aa06ea1352a20d1a86b61"
}
```

=== Stratus ===

10.19 Stratus

Stratus is our own javascript library for managing the front end of websites. It allows us to use require.js to load the files we need when we need them. We use an Angular model to create Components, Services, Controllers, Filters, etc. See our Components documentation for generic information about how we use components on the site.

10.19.1 Workflow

env.js - The site loads the Environment to set key variables about the environment.

config.js - Stratus has it's own core config.js that defines core stratus paths to components, services, filters, etc. And Sitetheory has a custom config.js that defines custom components for Sitetheory, or Stratus "extra" components that are being enabled for Sitetheory.

init.js - loads the Stratus boot.js and the merged config files.

See how we implement this loading of files in the CoreBundle:base.html.twig

10.19.2 Components

See our Stratus Components documentation for an overview of Component Architecture.

10.19.3 Filters

See our Stratus Filters documentation for an overview of Filters Architecture.

10.19.4 Directives

See our Stratus Directives documentation for an overview of Directives Architecture.

NOTE: See our Components documentation for an overview of Component Architecture if you want to build your own.

Stratus components are available on any page by adding the stratus component name. A few basic Stratus components are defined in the Stratus.js library. And Sitetheory has created custom Stratus Components which are specific to our platform. These are located in the most relevant related bundle's Resources/public/js/stratus/ folder and defined in Sitetheory's stratus config file (CoreBundle/Resources/public/js/boot/config.js, e.g. *stratus-carousel*.

10.20 Components For Functionality

10.20.1 Lazy Load Correct Sized Images

stratus-src

This stratus internal component allows you to load the best sized image based on the size of the container (XS, S, M, L, XL, HQ) so that it fills that area (which means it doesn't load images larger than mobile devices need).

Example

Load a default small image, and then use the src path to find the best version of image.

```
        <img stratus-src src="foo-xs.jpg">


Do not load a default image, use stratus-src to find the best version of the␣
↪image.
```

```
        <img stratus-src="foo.jpg">


If you want a placeholder image to appear on the page, you can just enter that as the␣
↪regular image src. It is usually recommended to specify the smallest version of the␣
↪image, so that the image's native ratio will be available to the CSS so that the␣
↪height is correctly proportional to the width (which means when the real image␣
↪loads the page isn't going to shift as element heights change).

NOTE: If you use the lazy loading on images in your hard coded design template assets␣
↪(not created by the CMS system so they don't automatically have the different size␣
↪options, e.g. XS, S, M, L, XL, HQ), you will need to create these versions of your␣
↪images that the component can load. Your sizes should be the standard sizes, since␣
↪we check the container and load the best size based on the expected size of the␣
↪images.

XS: 200px
S:  400px
M:  600px
L:  800px
```

(continues on next page)

(continued from previous page)

```
XL: 1000px
HQ: 1200px
```

Classes

- placeholder: When the image is first collected for lazy-loading a 'placeholder' class will be added to it, so that you can style default look of an image that isn't loaded, e.g. gray background with a loading icon.
- loading: when the image is on screen and is in the process of loading, a 'loading' class will be added.
- loaded: when the image is loaded, the 'loading' class will be replaced by 'loaded'.

Attribute Options:

- stratus-src: the stratus-src should point to the image that you want to lazy-load. If you have specified a regular img src as a placeholder image (e.g. a small version), and you want to lazy load the best size of that image, than you can avoid typing out the path a second time and just specify data-src="lazy" and the system will load the best version of the current image src.
- data-spy: By default the image will load when it is "on screen". But in some cases (like a Carousel) you need to specify a CSS selector for an alternative element on the screen that should trigger the loading, e.g. the container div.
- data-ignore-visibility: normally it will look for the size of the container and load the correct image that will fill the container (assuming a 100% width is set on CSS). But if the container is invisible, it will try to go up the element tree to the first parent that is visible. This is often desirable because the parent is collapsed. However, in some cases, like a Carousel, if you have the parent width set explicitly on a containing element, you want to use that (not the outer carousel width). So you set data-ignoreVisibility="true" and it will use the parent container width.
- data-disable-fadein: All images will fade in from opacity 0 to 1, when the placeholder class is replaced with the loaded class. If you have specified a src because you want a default placeholder image to show up, then obviously you don't want the placeholder image to go invisible. So you should add a "disable-fadein" class to the image.

10.20.2 OnScreen

The OnScreen component will detect when an element is visible on the screen and add classes that can be styled in CSS.

Initiate the onScreen component by adding *stratus-on-screen* to any element.

```
        <div stratus-on-screen>Fancy Area</div>

This component will add classes to the element depending on the user's actions: 'on-
↪screen' or 'off-screen' as well as 'scroll-up' or 'scroll-dDown'. You can then␣
↪target any combination of these two options, to do some fancy things like make a␣
↪secondary header appear when the main header is 'offscreen' but you are scrolling␣
↪up. Or make CSS animations start only when you scroll them into view.
```

```
        <div stratus-on-screen class="on-screen scroll-down">Fancy Area</div>
```

Additional Options

- data-target: the CSS selector of an alternative element that should have the classes added (instead of itself), e.g. a parent element. Defaults to the current element.

- data-spy: the CSS selector of an an alternative element that should be watched to check if it's on or off screen. Defaults to the current element.
- data-offset: an integer (positive or negative) that determines where the spy element begins on the page. So if you set this to 200, the element onScreen class would be added to the target after the spy element was 200 pixels onto the screen.
- data-event: one or more events names that can trigger actions. The only option at the moment is "reset" which allows the classes to be reset if the page is scrolled to the very top, or if the data-reset value is set when the page is scrolled to that position.
- data-reset: an integer representing a vertical (y) pixel position on the page that should trigger a reset when the page is scrolled to that point (defaults to 0).

10.20.3 Embed

Because of the way Angular controls the DOM, it is not possible to just paste third party Javascript on the page (e.g. a Twitter widget), if it is inside any Angular controlled areas, e.g. any part of the page that is inside an *ng-if*, *ng-repeat*, *ng-controller*, etc, essentially almost anywhere. If you do, there will be timing issues because Angular removes elements from the DOM unpredictably and controls when they appear. So we have to use a component to load the third party code at the right time.

NOTE: this may not be implemented yet (10/14/2019)

```
    <stratus-embed scripts="['foo.js', 'bar.js']">
        <!-- code here -->
    </stratus-embed>
```

So looking at a third party embed code like Twitter (NOTE: we have a specific twitter component already):

```
<a href="https://twitter.com/intent/tweet?button_hashtag=gutensite&ref_src=twsrc%5Etfw
↪" class="twitter-hashtag-button" data-show-count="false">Tweet #gutensite</a>
↪<script async src="https://platform.twitter.com/widgets.js" charset="utf-8"></
↪script>
```

Instead you would do this:

```
    <stratus-embed scripts="['https://platform.twitter.com/widgets.js']">
        <a href="https://twitter.com/intent/tweet?button_hashtag=gutensite&ref_
↪src=twsrc%5Etfw" class="twitter-hashtag-button" data-show-count="false">Tweet
↪#gutensite</a>
    </stratus-embed>
```

10.20.4 Twitter

Becuase of the way that Angular controls the DOM, it's not possible to copy/paste third party embed code, so we have created a component that passes through the settings at the right time. It accepts the standard Twitter parameters for Twitter Component (https://developer.twitter.com/en/docs/twitter-for-websites/timelines/guides/parameter-reference).

```
<stratus-twitter-feed screen-name="gutensite"></stratus-twitter-feed>
```

10.20.5 Carousel

NOTE: for most cases we are using a simple CSS and basic Angular implementation of carousels, rather than this third party Swiper carousel, because it's easier to control the design and much more lightweight (no extra libraries). See examples in our streams that load modules.

This carousel component uses Swiper .

Swiper Natively Supports: -lazy loading -autoplays (has transition times to set if needed) -loop -mouse/finger swiping (or keyboard), -swipe up/down -pagination/counter -transition types/effects -html frames

Implementation

The component is rendered as a dedicated element using *stratus-carousel* element.

```
<stratus-carousel></stratus-carousel>
```

Options

In addition to the standard Swiper options (listed in their documentation) we have additional options for our stratus component implementation.

Slide Object Reference: Each slide will either be an image, video, or html. So a slide can be defined as an ARRAY with simple image URLs, or as an array with multiple OBJECT elements. If it is defined as an Object (in different contexts) the slide object has standard elements no matter where it's defined (e.g. directly in the *data-slides* settings or pulled from an API in *data-model*).

- src: URL to image
- link: URL to load on click
- target: standard browser target (where to open the link), defaults to "_self".
- title: Unused currently
- description: Unused currently
- **data-model (JSON Array)**
  - TODO: this is not yet implemented. This should be a Collection object that will contain information for the slides. These variables should resemble the standard Slide Object defined above.
- **data-slides (JSON Array)**
  - TODO: consider changing this to 'data-slides' and allow passing in String as URL or HTML, and Object can specify 'src' or 'html' so that we can put anything we want in each slide.
  - This may be either a JSON Array of Strings or Objects.
  - Strings: Array of URLS to an image, e.g. ["https://domain.com/image1.jpg", "https://domain.com/image2.jpg"]
  - Objects: Array of standard Slide Objects (see reference above), e.g. [{src: "https://domain.com/image1", link:"http://domain.com/foo"}]
- **data-loop (boolean - default: true)**
  - During pagination, allows a the last slide to return to the very first slide
- **data-autoplay (boolean or JSON object - defaults to false)**
  - Automatically changes the slide at set time intervals.

    - **JSON:**
        - TODO: specify format for options of object, time (seconds or milliseconds?)
- **data-transition-effect (string - default: 'slide')**
    - Options: 'slide','fade,'cube,'coverflow','flip'
    - TODO: Some transitions seem to have trouble with lazyLoad that we'll need to work on
- **data-pagination (boolean or JSON Object - default: false)**
    - See Swiper Documentation
    - **JSON: TODO**
        - clickable (boolean - default: false)
        - dynamicBullets (boolean - default: false)
        - dynamicMainBullets (integer - default: 1)
        - render (String): 'fraction', 'customFaction' (not complete), 'progressbar', 'progressbarOpposite', 'numberBullet', 'bullet'
- **data-init-now (Javascript variable)**
    - Specify a variable to watch. Delays initialization until provided variable exists/if not empty.
- **data-slides-link-target (string - default: "_self")**
    - If data-slides doesn't have a browser target, uses this option as it's default instead of "_self".
- **data-direction (String - default: 'horizontal')**
    - Determine direction of slide movement.
    - Options: 'horizontal', 'vertical'
- **data-round-lengths (boolean - default: true)**
    - Set to true to round values of slides width and height to prevent blurry texts on usual resolution screens (if you have such)
- **data-scale-height (boolean - default: true)**
    - Scales an image 'out' if it is too big for a the containing element to match to fit. Also centers all images that don't fit perfectly
- **data-allow-zoom (boolean - default: false)**
    - Allow Zooming into an image by double clicking or pinching on Mobile. Requires and force enabled scaleHeight
- **data-stretch-width (oolean - default: false)**
    - Allow image to stretch wider than the image provided to fill the element. May cause expected blurriness.
- **data-auto-height (boolean - default: false)**
    - Resizes the entire element to match the height of the current slide. WARNING: May cause resizing of this part of the page every time slide changes!
- **data-allow-touch-move (boolean - default: true)**
    - Allow moving the slides using a finger of mouse
- **data-lazy-load (boolean or JSON Object - default: true)**

  - Enable Lazy Loading to prevent everything from being fetched at once. This will lazy-load images only for the next and previous images to give a buffer.

TODO: Determine if Alex's other stratus lazyloading conflicts

- **data-navigation (boolean - default: true)**
  - TODO: implement
- **data-scrollbar (boolean - default: true)**
  - TODO: implement
- **data-slides-per-group (boolean - default: false)**
  - TODO: implement
- **data-autoplay-delay**
  - TODO: no longer an option?

Multi-Colunns

The Swiper Carousel has many advanced api options, including to control grouped/multiple slides in view (See section for "Slides Grid").

Demos: Multiple Slides Per View Slide Multiple Per Group

init-now="model.completed"

Examples

Display Image Slides

```
        <stratus-carousel
            data-slides='["https://foo.com/image1","https://foo.com/image2","https://foo.com/image3"]'
        ></stratus-carousel>
```

Display HTML Slides NOTE: HTML must be escaped for JSON.

```
    <stratus-carousel
        data-slides='["<h1>Foo</h1><img src=\"https://foo.com/images1\">", "<h1>Bar</h1><img src=\"https://foo.com/images2\">"]'
    ></stratus-carousel>
```

Display Images with Links

```
        <stratus-carousel
           data-slides='[{"src":"https://foo.com/image1", "link":"https://foo.com/", "target":"_blank"}]'
        ></stratus-carousel>
```

Modify Default Settings

```
    <stratus-carousel
       data-slides='["https://foo.com/image1"]'
       data-autoplay="true"
       data-transition-effect="fade"
```

(continues on next page)

(continued from previous page)

```
        data-pagination='{"clickable":true, "render":"bullet"}'
        data-direction="vertical"
    ></stratus-carousel>
```

10.20.6 HOW TO USE STANDARD ANGULAR TO DO COMMON COMPONENT-LIKE FEATURES

We do not need specific components to do common design template features anymore, instead we just use standard Angular. And we have a core components.css that applies basic styles to the examples below.

In 95% of cases we can use a simple CSS and Angular version of a carousel, instead of a third party library (e.g. stratus-carousel). This is easier to style, and doesn't require loading any extra files. This requires a bit of HTML/Angular, so we have a Twig component that injects the necessary code onto a page which displays images or HTML slides.

Example See the Landing.html.twig Stream for an implementation of the *ComponentsAdminList.html.twig carousel* macro.

```
{# Arguements:
    model - (default: 'model.data')
    ratio ('portrait', 'square', 'landscape', 'cinema')
    carouselType ('images', 'HTML')
    controlSize ('standard-controls', 'small-controls')
#}
{{ streamComponents.carousel('model.data', 'square', 'images') }}
```

ng-class

Use ng-class to add a class based on a conditions, e.g. *ng-class="{'my-class': myVariable}"* will add "my-class" if "myVariable" is true.

ng-click or ng-hover

Use ng-click or ng-hover to modify variables that can be used on other elements that conditionally add a class with ng-class

Example

Add a "More Box"

We use a "More Box" for various popups on the site, e.g. immersive popups that dim the screen and show a popup in the middle of the site, or local popup that just covers a button locally.

Examples

For the **Local Popup** the button can contain the popup inside it.

Add a Drawer

It is often necessary to make a drawer slide in and out of the side of the website (e.g. a toolbar, or a responsive mobile menu drawer). This works basically exactly like a More Box, but with slightly different CSS. The core plugins.css has

basic styling that makes the drawer and the app container slide in together, but you can customize specifics in your own CSS.

```
        <!--Place button anywhere-->
        <a ng-click="myVariable=true">
            Toggle Drawer
        </a>

        <!--Place popup directly above closing body tag-->
        <div ng-class="{'show':myVariable}" class="drawer-position">
            <div class="drawer">
                <!--Optional close button-->
                <button type="button" class="btn-close" ng-click="myVariable=false">
                    <md-icon ng-class="{'show':myVariable}" md-svg-src="/Api/Resource?
↪path=@SitetheoryCoreBundle:images/icons/actionButtons/close.svg" aria-hidden="true"␣
↪role="img"></md-icon>
                <button>
                <!--Drawer content here-->
            </div>
        </div>
```

10.21 Components For Editing

TODO: this needs to be updated since it was written in 2017 because we moved to Angular and most editing is done with Angular Material widgets

Generally we use Angular Material components for enabling editing on a page. But Sitetheory has created custom Editing Components that are specifically intended to allow a designer to render controller elements on the page to give editing functionality. A component can be a simple display field to show the value of an entity, a text field that allows editing the value of an entity property, or it can be like a complex media selector that shows you all the elements you have selected and allows you to upload or select new media. Components render a template and add functionality to the page so the designer can control the user experience. Most components are set to auto-save changes, so the experience is much more responsive than traditional forms. Components are used extensively throughout the CMS admin and Live Edit mode.

See the Javascript documentation for detailed specs of each widget. TODO: update link to documentation once we move these to Sitetheory http://js.sitetheory.io/1/0/stratus.html

** TODO: update these to Stratus Components and document**

data-property (string): This is the model property that is being edited.

data-label (string): The label for the information being edited.

data-help (string): Additional information to help users, which will appear as a popover on a help icon.

data-template (string): This would be a full web path to a template file or a template key from config.js.

data-templates (JSON): This is a JSON object with names of the templates and a key or web path to the template that should be used for each part of the widget. Usually most components have only one template, but in cases like the Collection widget, there may be a list, container, and entity template, and this allows you to customize all of them, e.g. {“list”: “/path-to-list”, “container”: “/path-to-container”, “entity”: “/path-to-entity”

TODO: determine if this is used

TODO: determine if this is used

TODO: add documenation for our Editor

```
<stratus-redactor></stratus-redactor>
```

Add pagination for a specific collection. TODO: Explain how stratus knows which Collection to paginate (does this need to be inside the parent element?) .. code-block:: html

linenos <stratus-pagination></stratus-pagination>

TODO: is this still valid? data-meta: this allows you to pass in data to the collection widget so that it will be accessible in the template, e.g. when defining the widget on the DOM, add an attribute for data-meta='{"foo":"bar"}' will pass in values to the template to be accessed as {{ globals.meta.foo }}

This adds a save button to the page to save current version. ** TODO: update these to Stratus Components and document**

This adds a publish button to the page to publish the current version. ** TODO: update these to Stratus Components and document**

This adds a delete button to the page to delete current record. ** TODO: update these to Stratus Components and document**

Add a "Help" icon that reveals more information on hover.

```
<stratus-help flex="5">This field allows you to explain how awesome you are.</
↪stratus-help>
```

Add different types of dynamic fields that allow you to enter a value and select a label to describe what kind of infromation this is, e.g. an email field, that lets you select "Main", "Work", "Personal" or enter your own custom label.

- **data-type** (*string*): a string of one of the valid field types. A valid field type will add special styling, functionality, and validation relevant to that type of data. Valid options include: "phone", "email", "url", "location", "date". If no valid type is specified it will just be a simple field.
- **data-options** (*array: required*) an array of labels to choose from for this field e.g. ["Main", "Mobile", "Work", "Personal"]
- **data-custom** (*boolean*): specify *true* if you want users to be able to enter a custom value for the label. (*default: true*)
- **data-multiple** (*boolean*): specify *true* if you want users to be able to add more than one version of this type of field, e.g. multiple phone numbers. (*default: true*)
- location: when saved, a location will attempt to do a geolocation lookup and store the latitude/longitude of the address.

```
<stratus-option-value flex="95" ng-show="model.completed"
    ng-model="model.data.contentVersion.meta.phones"
    data-options='["Main", "Mobile", "Work", "Personal"]'
    data-type="phone"
    data-custom="true"
    data-multiple="true">
</stratus-option-value>
```

The label/value pairs are stored in the AssetManager, which allows for multiple dynamic fields to be attached to any entity.

NOTE: See our Stratus documentation for an overview of how Stratus works.

10.22 Stratus Filters

Like Stratus Components, you can create Angular style Directives for implementing functionality that isn't a Component or a Filter.

10.22.1 Available Directives

trigger

TODO: NOTE - this does not work at the moment.

There are cases where we need to set a variable for use in other parts of the page (and ng-init is deprecated and/or doesn't have the right timing). So we can use a directive to trigger a variable to be set on *ng-model* basaed on the *stratus-trigger* expression.

Usage

```
<span ng-model="foo"
      stratus-trigger="model.data.version.tags.length > 0 ? 'tags' : 'manual'"
      style="display:none">
</span>
<md-select ng-model="foo" flex>
    <md-option value="manual">Curated Content</md-option>
    <md-option value="tags">Tags</md-option>
</md-select>
```

NOTE: See our Stratus documentation for an overview of how Stratus works.

10.23 Stratus Filters

Like Stratus Components, you can create Angular style filters for processing data on a page.

10.23.1 Available Filters

assetPath

This allows you to specify a relative bundle bath for a local image, and automatically add the correct full path to the local web asset for the current version, e.g. providing input of *sitetheorybildtemplate/images/placeholder-square.png* would output */assets/1/0/bundles/sitetheorybildtemplate/images/placeholder-square.png*.

NOTE: This is intended for use with images, SVG, and other local files that are not minified (e.g. not css or javascript, those should be loaded through components which already have a system for finding the best version based on environment.

Usage

```
<img ng-src="sitetheorybildtemplate/images/placeholder-square.png | assetPath">
```

Will output on the page:

```
<img src="/assets/1/0/bundles/sitetheorytemplate/images/foo.png?v=12249595">
```

Options

// NOTE: if we wanted to make this more fancy, we should make a service that this references. // OPTIONS: // options.disableCacheBusting - (boolean - default: false) by default we add cache busting to the end of files. // options.enableMin - (boolean - default: false) by default we do not add min

CHAPTER 11

Vendors

11.1 Vendor Overview

11.1.1 How to Create a Vendor Account

We will release a vendor sign-up form, but if it's not visible on the Sitetheory.io website then your vendor account will be created manually. The vendor account can be accessed and managed on the admin.sitetheory.io website.

11.1.2 Types of Vendor Accounts

Most vendors will have a syndication relationship with the Sitetheory vendor. This means that they will inherit all the functionality and content types of the Sitetheory vendor. This prevents them from having to recreate everything from scratch. But if a vendor wants to use the Sitetheory platform for something entirely different than a traditional website builder and CMS, they can become an independent vendor and create their own content types and functionality from scratch.

11.1.3 Vendor Account Functionality and Purpose

The Vendor account has several purposes, but generally allows a vendor to customize the service they offer their clients.

Manage Site Genres

Based on the Vendor's target audience, the Vendor will create Genres that are selected by clients when they sign up. These genres will be linked to the Vendor's Master sites, so that when a client creates a new site, it will be deployed with content that is duplicated to provide a fully functional basic site.

Typical Genres may include vertical market categories like "Blog", "Artist", "Small Business", "Realtor", "Church", etc.

Manage Subscription Products

Vendors are required to create their own subscription packages so they can set prices and service levels that are appropriate for their business model. It's simple to create subscriptions and specify what content types (pages and functionality) each package has the ability to access.

Vendors can also view all the clients that are using their subscriptions, add, edit or remove subscriptions for their clients.

Automated Billing and Invoice Management

Vendors will provide their merchant account (credit card processing) account information and the Sitetheory system will manage the billing process for all their clients (using their bank information).

Whenever a product is purchased (or a subscription is billed on a regular basis) a new invoice is added to the Billing History. Billing History will show the products (including subscriptions) that were purchased and indicate the billing method that was used. If there was an error with billing, the invoice will be flagged as unpaid and the client will be required to update billing information to retry. Vendors can manage these invoices, edit or cancel.

11.1.4 Vendor's Custom Admin Site

When a vendor creates an account, they will also have a separate "Admin Site" created for them on their vendor's Admin site (e.g. admin.sitetheory.io). By default this will be duplicated (syndicated) from the Sitetheory Admin site, which means it's a fully functional Admin optimized for providing your clients all the functionality they need for building and managing their website.

The vendor's custom Admin site will be white-labeled with the vendor's logo/name and contact information. It can also be customized (like any website) by editing the theme template files to match any design you want. It can also create custom pages and links to add, remove or customize the workflow to meet your business needs, e.g. you may want to create another main section for "marketing" or any other service you offer, and then provide custom forms, functionality, workflows for your clients there.

11.1.5 Vendor Master Sites

Most vendors are going to want to create Master Sites for each of their different genres. The master sites will provide default content and menu structure that will be duplicated whenever a new client creates a website and selects the genre associated with that master site.

Master Sites are created in the Vendor's Admin site like any other website, and choose the genre that they belong to. Any user with "vendor" permissions will be given the ability to flag a site as "Master Site" and also set whether sites that are deployed from the master site either "duplicate" the content and then remain independent, or if they "syndicate" the content (and are linked to the master site, to receive updates and new content as it's modified on the master site).

CHAPTER 12

Other References

- Framework Docs
- Stratus Docs