
SiteSupra Documentation

Release 9.0.1

SiteSupra

February 12, 2016

1 About SiteSupra:	1
2 Installation:	7
3 Quick Start:	11
4 Developer's Guide:	13
5 Cookbook Articles:	35
6 CMS:	39
7 Reference:	59
8 Indices and Tables	61

About SiteSupra:

1.1 Why Another CMS

SiteSupra combines a powerful web framework and an elegant CMS to allow developing and managing websites easily. Developers will enjoy SiteSupra while building simple websites or large, feature-rich web apps. Website editors will love SiteSupra because of advanced CMS features packed into unique elegant and simple interface. While there are many great web frameworks and good CMS products separately there are few products that offer those qualities together. SiteSupra is on the mission to bring the best of both worlds in a single product.

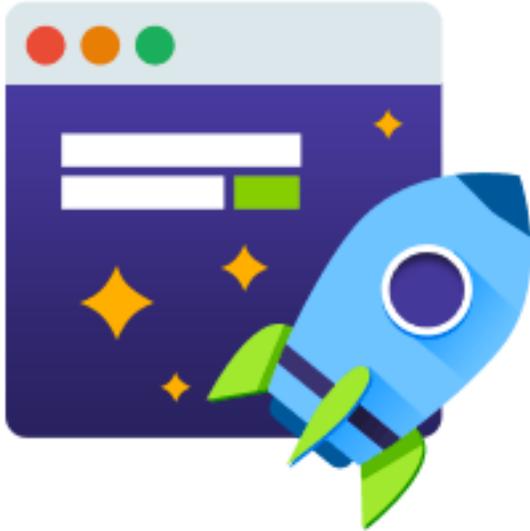
1.1.1 Is SiteSupra the right tool for my website?

SiteSupra is based on many years of research and development into web frameworks, CMS usability, and now is able to offer solutions for almost any task:



Corporate and enterprise websites

Build full-scale multilingual, multi-country websites with ability to maintain a single set of features and localized content, control page localization status, setup content editors access rights and approval workflow.



Promo websites

Develop and manage landing pages or one-pagers easily by setting up HTML into a page template and dragging and dropping required page widgets quickly.



Web applications

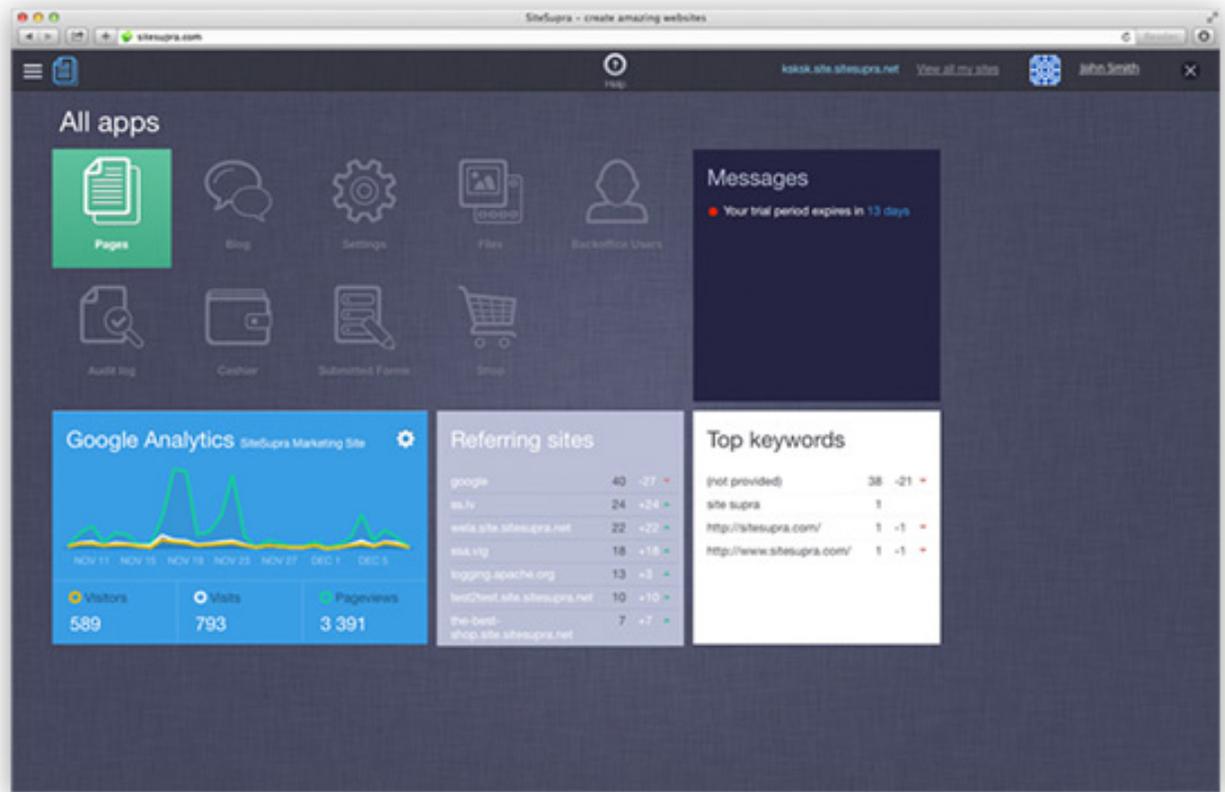
SiteSupra offers a powerful development framework and an array of internal managers that will allow you developing even very sophisticated web applications easily.

1.1.2 What will be my benefits if I will use SiteSupra?

Beautiful CMS

SiteSupra will add a great value to your website by making website editors fall in love with the CMS. Intuitive, elegant, beautifully-designed, it will allow your clients doing sophisticated website maintenance tasks easily thanks to Word-

like visual interface that covers not only text editing, but website structure, page widget functionality and back-office applications management.



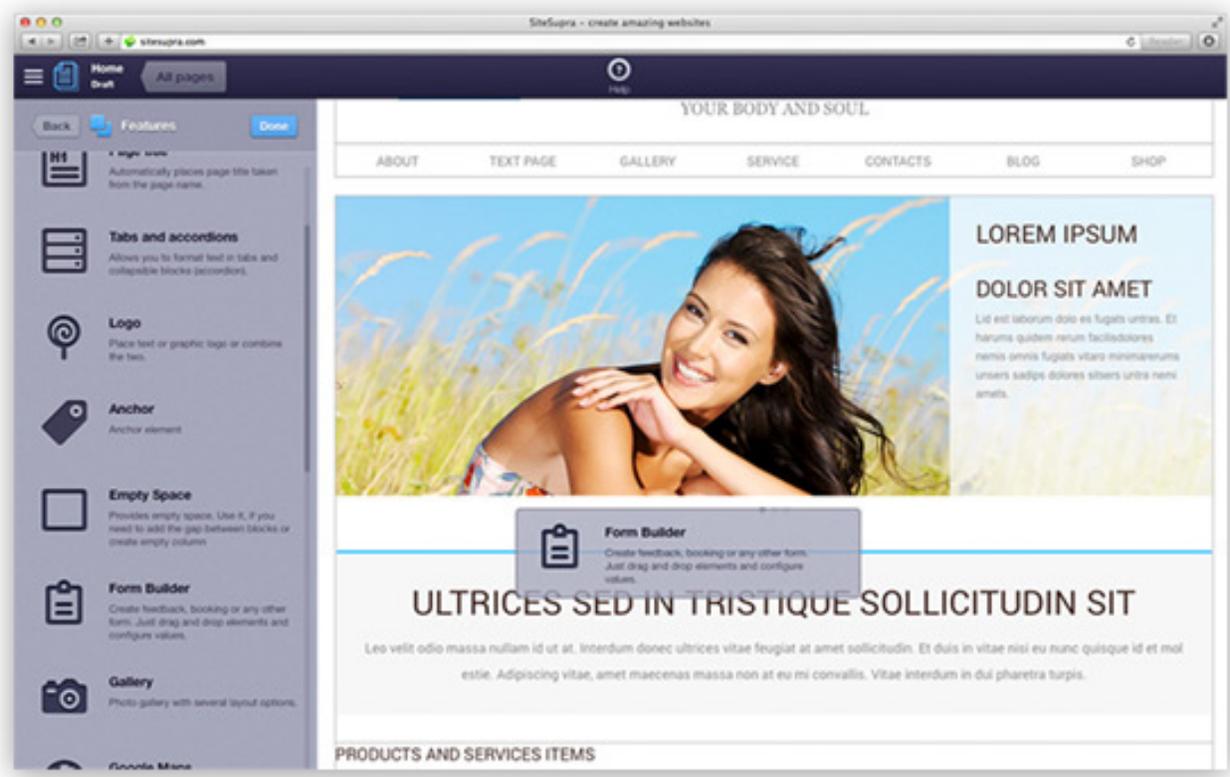
Industry-standard development approach

There's no learning curve developing with SiteSupra if you are familiar with Symfony and Doctrine components. SiteSupra goal is to allow you developing advanced application using well-known PHP components instantly.



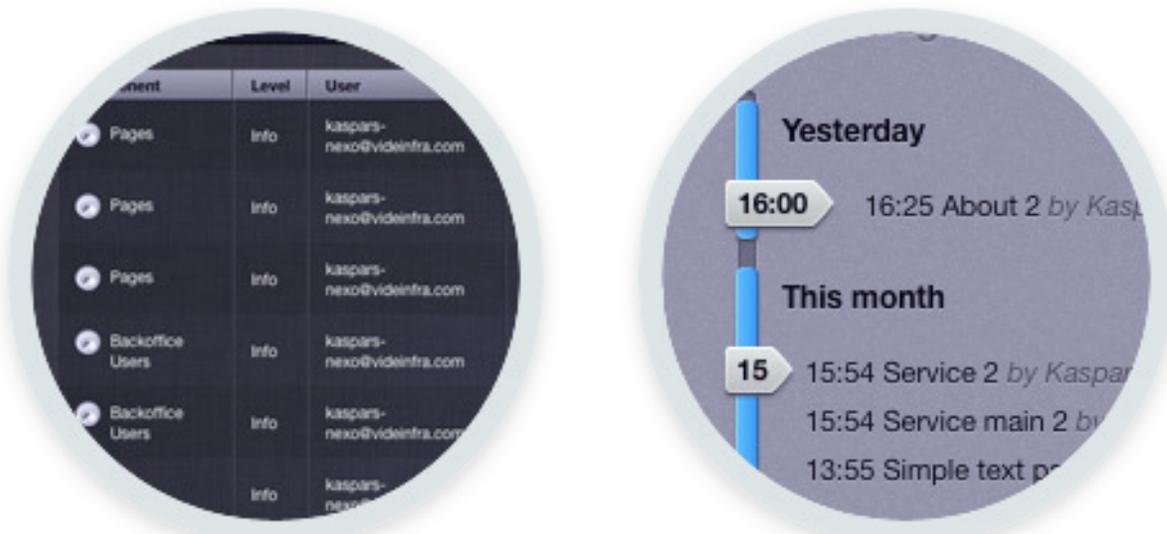
Reusable drag-and-drop page widgets

Code a page block once and drag-and-drop it anywhere on the site and reuse later in other websites. SiteSupra ships with expanding selection of widgets to allow you creating and managing website handily.



Rich CMS and back-office features

Enjoy advanced CMS features as page versions, scheduled publish and undo history all packed in a beautiful UI accessible without any learning by regular business users. In case of an enterprise environment set up access rights to internal applications or website pages and maintain role-based approval workflow.



Extensive modification capacity

Implement your own logic where needed through custom routes and controllers. SiteSupra has no limitations whether you decide to build a responsive themes or not - it's only a matter of how you code HTML.



1.1.3 What else will I get?

Customisation

We will be glad to help you developing custom SiteSupra extensions, blocks and modules to enhance your website and bring it to your requirements in the most effective way.

Integration

Shall you require help developing SiteSupra-based website or web application to your unique requirements we will be happy to assist through all development cycle starting with front-end development to SiteSupra blocks and modules.

Support

We offer help and assistance for your SiteSupra development tasks as well for system installation and maintenance.

1.2 SiteSupra at a Glance

Installation:

2.1 Software Requirements

2.1.1 Web Server

SiteSupra can run under any web server that supports PHP. We prefer Apache or nginx. Although, SiteSupra is not fully tested on Windows platform yet. However, from technical perspective we see no reasons why it may not run well on a WAMP-powered server.

Additional Modules

As many other CMS and frameworks SiteSupra rewrites URLs to hide reference to `index.php` file and make URLs human readable. To enable URL rewriting Apache requires `mod_rewrite` and nginx requires `ngx_http_rewrite_module`.

PHP

PHP version 5.4 or higher is required.

2.1.2 Database

Communications with the database layer is handled by Doctrine ORM. However, only MySQL database server is supported at the moment. Install MySQL version 5.1 or later.

2.2 Installing SiteSupra

SiteSupra is based on Symfony components and manages dependencies with composer. The installation process is pretty straightforward. However, you may consider checking [Software Requirements](#) first.

For first time users we would recommend to start with cloning SiteSupra Demo site. If you would like to contribute feel free to clone SiteSupra core.

2.3 Installing and Configuring SiteSupra Demo Site

2.3.1 Cloning SiteSupra Demo Site

SiteSupra source code is hosted at [github](#). Clone or checkout SiteSupra into your work folder.

```
$ git clone https://github.com/sitesupra/sitesupra-demo.git sitesupra
```

2.3.2 Configuring SiteSupra Demo Site

To configure SiteSupra on your computer follow the next steps :

1. Run `composer update` to update dependencies;
2. Create an empty database;
3. Copy `supra/config.yml.example` to `supra/config.yml` and configure database connection;
4. Create tables by running `php supra/cli.php doctrine:schema:create`
5. Setup assets `php supra/cli.php assets:publish`;
6. Load fixtures `php supra/cli.php sample:fixtures:load storage/fixtures`;

That's all! Now configure web server of your choice (see chapter [Configuring Web Server](#) below) and enjoy SiteSupra.

2.4 Installing and Configuring SiteSupra Core

If you would like to contribute to SiteSupra project consider to checkout SiteSupra core. Please note SiteSupra core doesn't contain any web site or blocks. You may need to add and configure them by yourself.

2.4.1 Cloning SiteSupra Core

SiteSupra source code is hosted at [github](#). Clone or checkout SiteSupra into your work folder.

```
$ git clone https://github.com/sitesupra/sitesupra.git sitesupra
```

2.4.2 Configuring SiteSupra

To configure SiteSupra on your computer follow the next steps :

1. Run `composer update` to update dependencies;
2. Set up web server permissions for `storage` folder (you can stick to plain old `chmod 777` or use ACL approach as [Symfony](#) does);
3. Create an empty database;
4. Copy `supra/config.yml.example` to `supra/config.yml` and provide database credentials;
5. Create tables by running `php supra/cli.php doctrine:schema:create`;
6. Load initial fixtures by running `php supra/cli.php supra:bootstrap`;
7. Publish assets with `php supra/cli.php assets:publish`.

All done! Now just point your web server of choice to web directory in SiteSupra project's root.

2.5 Configuring Web Server

2.5.1 Apache

Point `DocumentRoot` to the web directory in SiteSupra project's root. Allow to follow symlinks and configure rewrite rules as listed below.

```
Options +FollowSymlinks

RewriteEngine On

RewriteCond %{DOCUMENT_ROOT}%{REQUEST_FILENAME} -f

RewriteRule ^ - [L,NS]
RewriteRule ^.*$ /index.php$0 [L,NS]
```

Rewrite rules for `.htaccess` are provided in `.htaccess` file that comes along with SiteSupra source code.

2.5.2 nginx

Point `root` to the web directory in SiteSupra project's root. Configure rewrite rules as shown below:

```
location / {
    try_files $uri $uri/ /index.php;
}
```

Quick Start:

SiteSupra is open source PHP framework and extremely powerful web site builder bundled with user-friendly CMS.

3.1 Where to Start

- SiteSupra at a glance
- Installation
- Building your first SiteSupra web site

Tip: no tips yet.

3.2 SiteSupra Concepts

Please document what is an application (in controller context)

Developer's Guide:

4.1 Standard Packages

4.1.1 Core

Not much of a package but SiteSupra itself. It contains core classes and definitions of SiteSupra framework.

4.1.2 Cms

SiteSupra CMS by itself. This package contains controllers, routing, entities and frontend assets that are used throughout CMS backend.

4.1.3 CmsAuthentication

SiteSupra authentication layer is separated in CmsAuthentication bundle. It sets up `SecurityContext` and handles backend user authentication. It is not suitable for front-end user authentication.

4.1.4 DebugBar

Integrates [PHP DebugBar](#) into SiteSupra allowing you to monitor requests, their time lines, SQL queries, events, and much more. This package is active only when SiteSupra runs in debug mode.

4.1.5 Framework

This package combines and integrates everything together. It sets up Doctrine, EntityAudit, Twig, and all other components required to run SiteSupra. It also registers most of the commands that you can access from [Command Line Interface](#).

4.2 HTTP Kernel and Bootstrap Process

Note: SiteSupra does not use [HttpKernel](#). We had eight versions of SiteSupra before moving to Symfony. Unfortunately, not all of the Symfony concepts do suit our needs well. Our implementation is still a bit incomplete, especially `RequestStack` and forwarding, so expect refactoring soon.

SiteSupra uses plain `HttpFoundation` component (see [documentation of HttpFoundation](#) and [HTTP requests in symfony](#) for more information).

SiteSupra mimics Symfony behaviour as close as possible - a `Request` object is created, every controller returns `Response` (read more on controllers [here](#)). Basically, request processing happens in the following order:

- Web server hits entry point `webroot/index.php`;
- SiteSupra builds `container`, `buildContainer()` is called;
- SiteSupra boots, `boot()` is called. Two events are fired at that moment - `Supra::EVENT_BOOT_START` and `Supra::EVENT_BOOT_END`. Method `boot()` is called for every registered package, allowing early initialization;
- Request handling starts by calling `handleRequest($request)`. This method loads `Supra\Core\Kernel\HttpKernel` and calls `handle()`. Request handling by `HttpKernel` traverses through stages below:
 1. `KernelEvent::REQUEST` is thrown. Request processing stops when there is at least one event configured for response to `RequestResponseEvent`. Request object is returned;
 2. Kernel tries to resolve controller and action by checking `_controller` and `_action` parameters of `requestAttributeBag`; if found, controller is instantiated, action is called and `HttpKernel` expects `Controller` to return a `Response`. This is used when forwarding requests or instantiating requests without a route;
 3. If controller is not resolved yet kernel loads routing and tries to find current route. `AttributeBag` is overwritten by one created from route configuration and controller is actually executed. `KernelEvent::CONTROLLER_START` and `KernelEvent::CONTROLLER_END` events are fired, and any listener can override response during `KernelEvent::CONTROLLER_END` event. If any exception (generic or `ResourceNotFoundException`) is thrown request processing moves to exceptions processing (see stage 5 below);
 4. Kernel fires final `KernelEvent::RESPONSE` event and returns resulting `Response` object. Exception is thrown when the object is not an instance of `Response`;
 5. If any exception is caught during request processing, then the exception is processed in the following way:
 1. Kernel fires `KernelEvent::EXCEPTION` event (again, if any listener provides `Response` inside this exception event, then the response is returned);
 2. If exception is instance of `ResourceNotFoundException`, a special event is fired - `KernelEvent::ERROR404`, which allows, for example, on-the-fly compilation of assets. Finally, if no response is available after the event is processed, the exception is re-thrown (in debug mode) or `exception404Action` of default exception controller (stored in container under `exception.controller` key) is called;
 3. If there's still an exception and no response, an exception is re-thrown (in debug mode) or `exception500Action` of default exception controller (stored in container under `exception.controller` key) is called.
- Resulting response is sent to the browser (by calling `send()` method). Any unhandled exception is caught by `Debug` component;
- SiteSupra shuts down firing `Supra::EVENT_SHUTDOWN_START` and `Supra::EVENT_SHUTDOWN_END` events and calls `shutdown()` method of all registered packages, thus allowing some late cleanup.

As you can see, this process is pretty simple and transparent. Last thing to note must be a `SupraJsonResponse` class that is used throughout CMS backend for passing messages, warnings, and errors to frontend in a common way. See the class source code to learn more on messaging process.

4.3 Dependency Injection

4.3.1 Core Concepts

SiteSupra Dependency Injection layer or DI in short is based on [Pimple](#). If you're curious about what DI is you can read the following Wiki articles:

- [Inversion of control](#);
- [Dependency injection](#).

We also recommend read [Symfony's documentation](#) about basic DI principles.

SiteSupra main container class is `Supra\Core\DependencyInjection\Container`. It extends [Pimple's Container](#), implements SiteSupra's `Supra\Core\DependencyInjection\ContainerInterface`, implements some hard-coded methods (remember, we're a CMS, and not a full-stack framework, some items like Doctrine or Cache are always present), and provides parameters handling.

Note: It's almost certain that we will drop hard-coded getters later and build container definition on-the-fly in the same way Symfony does.

Everything is simple, right? Last but not least to note would be that any object implementing `Supra\Core\DependencyInjection\Container\ContainerAware` will be provided with `Container` on instantiation (calling `setContainer`).

4.3.2 Container Building Process

Note: The code will be refactored soon.

SiteSupra core class (`Supra\Core\Supra`, extending `Supra\Core\DependencyInjection\ContainerBuilder`) builds and returns `Container` object during call to `buildContainer`. This is done in the following steps:

- Pre-setting some basic variables and objects (like directories, [HttpFoundation](#) objects, [Command Line Interface](#), and so on);
- Injecting packages (allowing to expose their basic configuration);
- Building configuration (there the configuration is being validated, default values set, container parameters are substituted, and so on);
- Finishing configuration when packages can override or extend config values of other packages;
- Firing `Supra::EVENT_CONTAINER_BUILD_COMPLETE` event.

4.3.3 Package Integration and Two-pass Container Building

First of all, a package needs to be registered. This is done by overriding `registerPackages` in `SupraApplication` class (located in `supra/SupraApplication.php`). This method simply returns array

of package instances, like in the example below:

```

1 <?php
2
3 use Supra\Core\Supra;
4
5 class SupraApplication extends Supra
6 {
7     protected function registerPackages()
8     {
9         return array(
10            new \Supra\Package\Framework\SupraPackageFramework(),
11            new \Supra\Package\Cms\SupraPackageCms(),
12            new \Supra\Package\CmsAuthentication\SupraPackageCmsAuthentication(),
13            new \Supra\Package\DebugBar\SupraPackageDebugBar(),
14
15            new \Sample\SamplePackage()
16        );
17    }
18 }

```

Each package must extend `Supra\Core\Package\AbstractSupraPackage`. You can override any of the following methods to alter SiteSupra behavior:

- `boot()` method will be called during SiteSupra boot, see [HTTP Kernel and Bootstrap Process](#);
- `inject(ContainerInterface $container)` method will be called during Container building in package injection phase (see above);
- `finish(ContainerInterface $container)` method will be called finishing Container build after the configuration is processed;
- `shutdown()` method will be called during SiteSupra shutdown, see [HTTP Kernel and Bootstrap Process](#).

4.3.4 Package Configuration

As mentioned above package configuration may occur in two phases - injection phase and finishing phase. Let's look at both of them starting from `inject()`:

```

1 <?php
2
3 public function inject(ContainerInterface $container)
4 {
5     $this->loadConfiguration($container);
6
7     $container->getConsole()->add(new DoFooBarCommand());
8
9     $container[$this->name.'.some_service_name'] = function (ContainerInterface $container) {
10         return new SomeService();
11     };
12
13     if ($container->getParameter('debug')) {
14         //prepare some extended logging, for example
15     }
16 }

```

The most important call would be `$this->loadConfiguration()` (line 5). This method loads configuration file (by default it is `Resources/config/config.yml`). To load your own configuration pass the file name to the method as a second parameter.

This call parses config file, processes the configuration using package configuration definition (more on that on [Symfony configuration component article](#)), and stores the values for further processing.

Later you can access already defined services (see line line 7, which though is not a very good approach since it instantiates the service), add your own service definitions (lines 9-11), and access container parameters (line 13).

Each package has it's own configuration definition. Concrete configuration object is created during call to `getConfiguration()` method. By default, if there is a package named `SupraPackageFooBar` in namespace `Com\Package\FooBar`, then the method will search for configuration definition `SupraPackageFooBarConfiguration` in namespace `Com\Package\FooBar\Configuration`. Of course, you can always override you package's method `getConfiguration()` and implement your own logic.

The configuration class should extend `Supra\Core\Configuration\AbstractPackageConfiguration` and implement `ConfigurationInterface`. This forces you to implement function `getConfigTreeBuilder()`, returning instance of `Symfony\Component\Config\Definition\Builder\TreeBuilder`. If you're curious about what is a `TreeBuilder` and how exactly the configuration is being defined, please read [Defining a Hierarchy of Configuration Values Using the TreeBuilder](#) on official Symfony documentation web site. Let's take configuration of `SupraPackageFrameworkConfiguration` as an example:

```

1  <?php
2
3  class SupraPackageFrameworkConfiguration extends AbstractPackageConfiguration implements ConfigurationInterface
4  {
5      /**
6       * Generates the configuration tree builder.
7       *
8       * @return \Symfony\Component\Config\Definition\Builder\TreeBuilder The tree builder
9       */
10     public function getConfigTreeBuilder()
11     {
12         $treeBuilder = new TreeBuilder();
13
14         $treeBuilder->root('framework')
15             ->children()
16                 ->append($this->getAuditDefinition())
17                 //some other definitions are skipped for illustrative purposes
18                 ->append($this->getServicesDefinition())
19             ->end();
20
21         return $treeBuilder;
22     }
23
24     public function getAuditDefinition()
25     {
26         $definition = new ArrayNodeDefinition('doctrine_audit');
27
28         $definition->children()
29             ->arrayNode('entities')
30                 ->prototype('scalar')->end()
31             ->end()
32             ->arrayNode('ignore_columns')
33                 ->prototype('scalar')->end()
34             ->end()
35         ->end();
36
37         return $definition;
38     }
39 }

```

Root node (line 14) must match your package name. The rest of configuration definition is standard for Symfony-based applications (lines 24–38), except for call of `->append($this->getServicesDefinition())`, which is inherited from `AbstractPackageConfiguration` and enables parsing of services section of your configuration file.

Package configuration files are simple yml files as shown below:

```
1 services:
2   supra.framework.session_storage_native:
3     class: \Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage
4     parameters: [[], "@supra.framework.session_handler_doctrine"]
5   supra.framework.session_handler_doctrine:
6     class: \Supra\Package\Framework\Session\DoctrineSessionHandler
7 #some config parts are skipped for illustrative purposes
8 doctrine:
9   #some config parts are skipped for illustrative purposes
10  credentials:
11    hostname: localhost
12    username: root
13    password: ~
14    charset: utf8
15    database: supra9
16  connections:
17    default:
18      host: %framework.doctrine.credentials.hostname%
19      user: %framework.doctrine.credentials.username%
20      password: %framework.doctrine.credentials.password%
21      dbname: %framework.doctrine.credentials.database%
22      charset: %framework.doctrine.credentials.charset%
23      driver: mysql
24      event_manager: public
25  entity_managers:
26    public:
27      connection: default
28      event_manager: public
29  default_entity_manager: public
30  default_connection: default
31 doctrine_audit:
32  entities: []
33  ignore_columns:
34    - created_at
35    - updated_at
36    - lock
```

Lines 1–6 define services. Key is service ID, ‘class’ defines class name and ‘parameters’ section enables setter injection (note that you can inject other services referenced with ‘@’ as shown in line 4). Setter injection is not yet supported.

First level keys will become container parameters prefixed with package name. In the example above, container parameters are ‘framework.doctrine’ and ‘framework.doctrine_audit’, and you can call something like `$container->getParameter('framework.doctrine_audit')['entities']` later in your code.

You may also reference any parameter using percent notation (`%parameter.name%`). In the example above, line 18 references value from line 11, possibly overridden by another package or main SiteSupra’s `config.yml`.

After calling `inject()` method of all packages, container builder merges configuration values (also replacing / referencing parameters), and starts calling `finish()` method of all packages, in load order. You `finish()` method can look like so:

```

1 <?php
2
3 public function finish(ContainerInterface $container)
4 {
5     //extend some other package service
6     $container->extend('some.other.service', function ($originalService, $container) {
7         $originalService->callSomeMethod();
8
9         return new SomeWrapper($originalService);
10    });
11
12    $doctrineConfig = $container->getParameter('framework.doctrine');
13
14    //processed configuration from example above. with merged parameters and optionally overridden by
15    $connectionDetails = $doctrineConfig['connections']['default'];
16 }

```

So, summing up:

1. You define your configuration in `inject()` method;
2. Container processes your configuration and merges it;
3. You retrieve processed values from container in `finish()` method and define your services;
4. Resulting container is available throughout SiteSupra.

4.3.5 Main SiteSupra Configuration File (config.yml)

Default SiteSupra configuration file `supra/config.yml`. example:

```

1 cms:
2   active_theme: default
3 framework:
4   doctrine:
5     credentials:
6       hostname: localhost
7       username: root
8       password: ~
9       charset: utf8
10      database: supra9
11 cms_authentication:
12   users:
13     shared_connection: null
14     user_providers:
15       doctrine:
16         supra.authentication.user_provider.public:
17           em: public
18           entity: CmsAuthentication\User
19     provider_chain: [ doctrine.entity_managers.public ]

```

Top-level keys correspond to package names, corresponding values are deep-merged with default values resolved in injection phase. Here you can see how default 'doctrine.configuration' values are merged with defaults from Supra-PackageFramework; any part of configuration can be overridden.

4.3.6 Container Parameter Handling, Parameter Substitution

Parameters are SiteSupra-specific extension to Pimple. Basically they represent simple key-value storage (with all the getters and setters. Refer to `Supra\Core\DependencyInjection\Container` for more information. However, some of the methods are worth to be noted separately:

- `replaceParameters` searches array of data and replaces all parameters enclosed in percent signs (like `%foo.bar%`) to their respective values;
- `replaceParametersScalar` replaces all parameters enclosed in percent signs (like `%foo.bar%`) to their respective values in a scalar variable (string);
- `getParameter` threads dots inside parameter name as internal array keys (thus allowing you to call `$container->getParameter('foo.bar.buz.example')` instead of `$container->getParameter('foo.bar')['buz']['example']`).

4.3.7 Standard Container Parameters

Standard container parameters that can help you in development process are listed below.

Directories

There is a number of container parameters reflecting SiteSupra directory structure:

- `directories.project_root` for project root folder (with `composer.json` and other core files);
- `directories.supra_root` for directory where `Supra.php` and `config.yml` reside;
- `directories.storage` for storage folder;
- `directories.cache` for cache folder (inside storage root);
- `directories.web` for webroot (this is where SiteSupra entry point, `index.php`, is);
- `directories.public` for asset root, `Resources\public` folders of every package are symlinked there.

Environments and Debugging

Some parameters are affected by current [development settings](#):

- `environment` shows current environment - currently on of `cli`, `prod`, or `dev`;
- `debug` shows current debug state - either `true` or `false`.

4.3.8 Service Definition

Adding `->addServiceDefinition()` to package configuration will allow that package to define services. Service definition has to reside under section `services` in configuration file.

A simple service definition contains service id and class name:

```
1 services:
2     locale.manager:
3         class: \Supra\Core\Locale\LocaleManager
```

you can provide constructor arguments as an array:

```

1 services:
2   supra.doctrine.event_subscriber.table_name_prefixer:
3     class: \Supra\Core\Doctrine\Subscriber\TableNamePrefixer
4     parameters: ['su_', '']

```

or even use container parameters as arguments:

```

1 services:
2   supra.framework.session_storage_native:
3     class: \Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage
4     parameters: [[], "@supra.framework.session_handler_doctrine"]

```

Unfortunately, caller injections are not possible with SiteSupra yet. But still you can use common Pimple's approach during `inject()` or `finish()`:

```

1 <?php
2
3 $container['some.service'] = function ($container) use ($dependency1, $dependency2) {
4     $service = new SomeService($dependency1);
5
6     $service->setDependency2($dependency2);
7
8     $service->intialize();
9
10    return $service;
11 };

```

4.4 Command Line Interface

4.4.1 General Concepts

SiteSupra uses [Symfony console component](#) for console operations. [Boris](#) is used for REPL shell.

The main console executable is `supra/cli.php`. Running it without parameters (like `php supra/cli.php`) will give you a list of all available commands.

By default CLI loads with `cli` environment and debugging enabled.

CLI Events

You can use events in your console application; you can use `Symfony\Component\Console\Event\ConsoleEvent` or `Supra\Core\Event\ConsoleEvent` (providing methods `getData/setData`) for such cases. Currently, there's only one CLI-only event in SiteSupra, `Supra\Package\Framework\Event\FrameworkConsoleEvent::ASSETS_PUBLISH`, provided by `SupraPackageFramework`, that is fired running `supra/cli.php assets:publish`. This event can be used to copy some non-standard assets to your webroot (to simplify things, you can implement `Supra\Core\Event\ConsoleEventListenerInterface` for your listener and use `webRoot` and `webRootPublic` keys of `$assetsPublishEvent->getData()` to determine copy locations).

4.4.2 Core Commands

SiteSupra does not provide any built-in commands itself; on a plain installation you have `help` and `list` commands, that are standard for `Symfony Console`. Only two arguments (`--env`, or `-e`, for environment, defaulting to `cli`, and `--debug`, defaulting to `true`), are present.

4.4.3 Package Commands

SupraPackageCms

This package does not provide any CLI commands.

SupraPackageCmsAuthentication

This package provides the following commands:

- groups:add
- groups:list
- groups:remove
- groups:update
- users:add
- users:list
- users:remove
- users:update

these are self-explanatory and documented inline (try, for example, `supra/cli.php help groups:update` - you'll like it!).

SupraPackageDebugBar

This package does not provide any CLI commands.

SupraPackageFramework

assets:publish

Each package can contain assets (in `Resources\public` directory). *Publishing* assets means that this directory will be symlinked (sorry, no hard copy option yet) into `web/public/PACKAGE_NAME`. Supra cleans up package name, so assets from `SupraPackageCms` will be symlinked into `web/public/cms`, enabling you to access them from the frontend.

cache:clear

Clears SiteSupra cache. Cleans up all cache entries, if no argument is provided, or a particular segment. For more information SiteSupra cache see [Cache](#) chapter of this manual.

cache:list

Shows info about SiteSupra cache and segments:

Directory	Size	Files
assets_404	0.13M	14
cms_assets	1.80M	2
combo	1.06M	27
config	0.04M	8
doctrine	0.04M	5
frontend_cache	0.02M	1
twig	0.02M	7

container:dump

Dumps information about `container` parameters and services:

```
Container parameters:
```

Parameter	Value
directories.supra_root	/home/developer/www/composer-test/supra
directories.project_root	/home/developer/www/composer-test
directories.storage	/home/developer/www/composer-test/storage
directories.cache	/home/developer/www/composer-test/storage/cache
directories.web	/home/developer/www/composer-test/web
directories.public	/home/developer/www/composer-test/web/public
environment	cli
debug	TRUE
cms.media_library_known_file_extensions	array
results truncated...	

```
Container services:
```

ID
application
config.universal_loader
routing.router
kernel.kernel
exception.controller
http.request
cache.driver
cache.cache
results truncated...

container:packages:list

Lists enabled Packages (showing both package name and class).

Doctrine-specific commands

The following commands are directly mapped to their Doctrine counterparts:

- doctrine:cache-clear:metadata

- doctrine:cache-clear:query
- doctrine:cache-clear:result
- doctrine:convert-encodings
- doctrine:generate:proxies
- doctrine:schema:create
- doctrine:schema:drop
- doctrine:schema:update

Please refer to [Doctrine documentation](#) should you need help on that.

framework:routing:list

Displays all registered routes, patterns, resulting controller, and whether the route is exported to frontend:

```
Defined routes:
+-----+-----+-----+
| Name                | Pattern                                     | Controller |
+-----+-----+-----+
| framework_combo    | /_framework_internal/combo/{paths}        | Framework  |
| framework_routes   | /_framework_internal/routes              | Framework  |
| cms_dashboard      | /backend                                  | Cms        |
| cms_dashboard_applications_list | /backend/applications-list              | Cms        |
| results truncated... |                                           |           |
+-----+-----+-----+
```

supra:bootstrap

Creates default user (username admin, password admin) and loads some initial templates so you can access backend and create new pages.

supra:shell

Launches REPL shell, with pre-set `$container` and `$application` variables. You can play around with some SiteSupra code without having debug controllers:

```
[1] supra> $container->getRouter()->generate('cms_dashboard');
// '/backend'
[2] supra>
```

supra:nested_set:check

Warning: Warning! There is a risk of losing your data. Please don't forget to backup your database prior to running the command.

SiteSupra uses custom NestedSet implementation. It's quite stable and almost bulletproof, although may need in repair from time to time.

4.4.4 Writing your own Command

See [Writing Your Own Command](#) for a complete reference.

4.5 Database (Doctrine 2) and EntityAudit

SiteSupra uses [Doctrine ORM](#) for database operations (a nice introduction to Doctrine is available at [Symfony docs](#)).

4.5.1 Doctrine Configuration

In most cases you do not need to configure anything but set up database credentials in `supra/config.yml`. Commonly, you have to override `framework.doctrine` parameter only:

```

1 framework:
2   doctrine:
3     credentials:
4       hostname: localhost
5       username: root
6       password: ~
7       charset: utf8
8       database: supra9

```

If you need auditing for your project issues, you will have to add them to `framework.doctrine_audit.entities`, along with default values, like shown below:

```

1 framework:
2   doctrine_audit:
3     entities:
4       Supra\Package\Cms\Entity\Abstraction\Localization
5       Supra\Package\Cms\Entity\Abstraction\AbstractPage
6       Supra\Package\Cms\Entity\Page
7       Supra\Package\Cms\Entity\GroupPage
8       Supra\Package\Cms\Entity\Template
9       Supra\Package\Cms\Entity\PageLocalization
10      Supra\Package\Cms\Entity\PageLocalizationPath
11      Supra\Package\Cms\Entity\TemplateLocalization
12      Supra\Package\Cms\Entity\Abstraction\Block
13      Supra\Package\Cms\Entity\Abstraction\Placeholder
14      Supra\Package\Cms\Entity\PagePlaceholder
15      Supra\Package\Cms\Entity\TemplatePlaceholder
16      Supra\Package\Cms\Entity\PageBlock
17      Supra\Package\Cms\Entity\TemplateBlock
18      Supra\Package\Cms\Entity\BlockProperty
19      Supra\Package\Cms\Entity\BlockPropertyMetadata
20      Supra\Package\Cms\Entity\ReferencedElement\LinkReferencedElement
21      Supra\Package\Cms\Entity\ReferencedElement\ImageReferencedElement
22      Supra\Package\Cms\Entity\ReferencedElement\ReferencedElementAbstract
23      Package\MyCustomPackage\Entity\CustomAuditedEntity

```

However, this will override previous values and possibly mess other packages. Thus, a better approach would be adding them during package injection phase:

```

1 <?php
2
3 public function inject(ContainerInterface $container)

```

```

4 {
5     $frameworkConfiguration = $container->getApplication()->getConfigurationSection('framework');
6
7     //add audited entities
8     $frameworkConfiguration['doctrine_audit']['entities'] = array_merge(
9         $frameworkConfiguration['doctrine_audit']['entities'],
10        array(
11            'Package\MyCustomPackage\Entity\CustomAuditedEntity',
12        )
13    );
14
15     $container->getApplication()->setConfigurationSection('framework', $frameworkConfiguration);
16 }

```

Basically, you can override any package configuration by using `getConfigurationSection()` and `setConfigurationSection()`.

4.5.2 CLI Commands

Please refer to SiteSupra [Command Line Interface](#) for more information. All Doctrine commands known by Symfony are available via SiteSupra CLI.

4.5.3 Standard event listeners

By default `SupraPackageFramework` defines and initializes Doctrine using its own `config.yml`:

```

1 doctrine:
2     event_managers:
3         public:
4             subscribers:
5                 - supra.doctrine.event_subscriber.table_name_prefixer
6                 - supra.doctrine.event_subscriber.detached_discriminator_handler
7                 - supra.doctrine.event_subscriber.nested_set_listener
8                 - supra.doctrine.event_subscriber.timestampable

```

`subscribers` array references the following classes, also defined in `config.yml`, `services` section:

```

1 services:
2     supra.doctrine.event_subscriber.table_name_prefixer:
3         class: \Supra\Core\Doctrine\Subscriber\TableNamePrefixer
4         parameters: ['su_', '']
5     supra.doctrine.event_subscriber.detached_discriminator_handler:
6         class: \Supra\Core\Doctrine\Subscriber\DetachedDiscriminatorHandler
7     supra.doctrine.event_subscriber.timestampable:
8         class: \Supra\Package\Framework\Doctrine\Subscriber\TimestampableListener
9     supra.doctrine.event_subscriber.nested_set_listener:
10        class: \Supra\Core\NestedSet\Listener\NestedSetListener

```

They serve for the following purposes:

- `TableNamePrefixer` adds prefixes to SiteSupra database tables (currently not-changeable, default `su_`);
- `DetachedDiscriminatorHandler` is internal SiteSupra feature. Quite probably we'll tune it up and document later;

- `TimestampableListener` listens to changes in entities implementing `Supra\Package\Cms\Entity\Abstraction\TimestampableInterface`, calls `setCreationTime()` and `setModificationTime` if needed;
- `NestedSetListener` handles changes in SiteSupra's `NestedSet` implementation.

If some other package must add other event subscribers, this can be done by overriding `SupraPackageFramework` configuration like it is done in `SupraPackageCms`:

```

1 <?php
2
3 public function inject(ContainerInterface $container)
4 {
5     //setting up doctrine
6     $frameworkConfiguration = $container->getApplication()->getConfigurationSection('framework');
7
8     $frameworkConfiguration['doctrine']['event_managers']['public'] = array_merge_recursive(
9         $frameworkConfiguration['doctrine']['event_managers']['public'],
10        array(
11            'subscribers' => array(
12                'supra.cms.file_storage.event_subscriber.file_path_change_listener',
13                'supra.cms.pages.event_subscriber.page_path_generator',
14                'supra.cms.pages.event_subscriber.image_size_creator_listener',
15            )
16        )
17    );
18
19    $container->getApplication()->setConfigurationSection('framework', $frameworkConfiguration);
20 }

```

You can freely alter any configurations during package injection phase (since actual entity managers and subscribers are set up only in finishing phase).

4.5.4 Internal Entities and Suprald

Doctrine, by itself, is a very sensitive system. For example, it does not like when you are trying to persist entity that already has id or restore entities with pre-set foreign keys. However, SiteSupra's versioning, based on `EntityAudit`, does exactly that! Therefore, we are using:

- A custom type, called `supraId20` (use `@Column(type="supraId20")`). That's currently just a 20 characters length string;
- A custom base entity `Supra\Package\Cms\Entity\Abstraction\Entity`, which is a `@MappedSuperclass`, and provides base methods like `regenerateId`, `__clone` etc.

SiteSupra Id contains twenty symbols and looks like "018dusx9903wosockckg", where:

- First 9 symbols are reserved for timestamp converted to base36. To be honest, we do not use standard unix timestamps. Our base date is 16 Dec 2011, 11:33:05. That's the day when `supraId` was introduced;
- Next two symbols are reserved for internal counter of entities persisted in current session;
- Trailing 9 symbols are just a randomly generated suffix.

Note: This is expected to be refactored to `@GeneratedValue(strategy="CUSTOM")` and `@CustomIdGenerator(class="...")` soon

4.5.5 EntityAudit and Versioning

SiteSupra's versioning is almost completely based on EntityAudit library. For more information refer to [respective documentation](#). We do not override anything there, so this should be enough if you need to implement auditing of your project entities.

4.6 Routing

4.6.1 Loading Routes

SiteSupra routing is heavily based on Symfony's routing component and uses very similar syntax. However, there are some minor differences. For example, you have to load all your routing files manually in your package's `inject()` method:

```
1 <?php
2
3 $container->getRouter()->loadConfiguration(
4     $container->getApplication()->locateConfigFile($this, 'routes.yml')
5 );
```

Function `locateConfigFile` searches `routes.yml` in your package's `Resources\config` directory.

4.6.2 Common Example

Let's take a look at some routing definition examples. The most simple would be `SupraPackageFramework` main routing file:

```
1 configuration:
2   prefix:           ~
3 routes:
4   framework_combo:
5     pattern:        /_framework_internal/combo/{paths}
6     controller:     Framework:Combo:combo
7     requirements:
8       paths:       .+
9     defaults:
10      paths:       ~
11   framework_routes:
12     pattern:        /_framework_internal/routes
13     controller:     Framework:Routing:export
```

`configuration` section at line 1 defines global `prefix` and `defaults` (default parameter values) keys. `prefix` must be explicitly defined even with default `~` value.

`routes` section, starting from line 3, defines actual routes. Each route may contain the following fields:

- `pattern` defines actual URI that will trigger the route;
- `controller` specifies the controller (in the example above, `Framework:Combo:combo` resolves into `SupraPackageFramework` → `ComboController` → `comboAction` (just like Symfony does!));
- `filters` defines Symfony route filters;
- `requirements` here you can specify per-parameter regex requirements;
- `defaults` provides default parameter values;

- options at the moment supports frontend key only.

where only pattern and controller are required.

Tip: Due to the fact SiteSupra's routing is based on [Symfony Routing component](#), everything written in Symfony documentation applies to SiteSupra as well - we did not reinvent the wheel here.

4.6.3 Container Parameters and JavaScript

Let's see a bit more complicated example from SupraPackageCms:

```

1 configuration:
2   prefix: ~
3 routes:
4   cms_dashboard:
5     pattern:          %cms.prefix%
6     controller:      Cms:Dashboard:index
7     options:
8       frontend:     true

```

First of all, you can use container parameters (in `%container.parameter.name%` form) in your route pattern. Secondly, you can provide `frontend: true` option and use in Javascript like this:

```

1 Supra.Url.generate('cms_dashboard');

```

4.7 Controllers

4.7.1 What are Controllers?

SiteSupra, following [Symfony](#), is a request-response framework. It boots up in a request context, applies business logic to the data provided, creates a response object, and returns result to the client. In general, requests are mapped through the [routing](#) engine to Controllers, which are PHP classes implementing the logic behind SiteSupra.

Each [Package](#) has its own set of controllers and routing rules. There is no limit on number of controllers and actions your web application may have. You may build and group your logic in the way you like.

4.7.2 Controller, Naming, and Routes

Controllers are package - specific classes, extending `Supra\Core\Controller\Controller`. They must reside in `Controller` namespace (like `Supra\Package\Framework\Controller`). Each controller method should accept `Request` and return a `Response` (more on these classes in [Symfony HttpFoundation documentation](#)).

SiteSupra expect `Controller` suffix in each Controller name (like `FooBarController` or `UserController`) and `Action` suffix in each action method (like `deleteAction` or `listAction`).

Routes use short syntax (`Package:Controller:action`). For example, `Framework:Routing:export` resolves to `SupraPackageFramework` (namespace `Supra\Package\Framework\Controller`), class `RoutingController`, and method `exportAction` (thus, `Supra\Package\Framework\Controller\RoutingController::exportAction`).

Each action is expected to return a `Symfony\Component\HttpFoundation\Response` object. Returning scalar value or not returning any value at all (which is equivalent to return `NULL`) will cause `HttpKernel` to throw an exception.

```
1 <?php
2
3 public function doStuffAction()
4 {
5     // doing stuff here
6
7     return new Response('<html><head><title>Hello!</title></head><body>Hello!</body></html>');
8 }
```

Of course, all types of Symfony responses are supported (like `JsonResponse`, or `RedirectResponse`). For example, a redirect to another URL could be called in the following way:

```
1 <?php
2
3 public function doStuffWithRedirectAction()
4 {
5     // doing stuff here
6
7     return new RedirectResponse('http://example.com/');
8 }
```

4.7.3 Base Controller Class

SiteSupra provides base class for your controllers, which is `Supra\Core\Controller\Controller`. First of all, it is `ContainerAware`, so you can always access DI Container via `$this->container` (container instance is set by `Http kernel` when controller is instantiated). It's provided with `$package` and `$application` properties, which are set to current package class name (like `Supra\Package\Framework\SupraPackageFramework`), and frontend application name (like `cms_authentication`).

Other handy methods of `Controller` class are listed below:

- `renderResponse` renders `twig` template and returns a `Response` object;
- `render` renders `twig` template and returns result as a string;
- `setApplication` overrides current application for `ApplicationManager` (see [SiteSupra Concepts](#) for more details);
- `getUser` returns current user or returns null if there's no security context, or if the security context does not contain valid token, or if the token does not contain valid user. See [Security](#) for more information;
- `getPackage` returns current package name (without namespace prefix, like `Framework`);
- `checkActionPermission` is a security-oriented stub that is not yet ported from legacy SiteSupra code to Symfony's ACL.

4.7.4 Exceptions

Controllers do not provide any custom exception handling. Instead, any exception is caught by `Http kernel`. Depending on current `debug settings` either trace is written or a special controller is being called (invoking `exception500Action`).

A special case is `Symfony\Component\Routing\Exception\ResourceNotFoundException`, which is forwarded to `exception404Action` of exception controller, thus allowing you to show pretty 404 page in production mode.

4.8 Page routing

4.9 Cache

SiteSupra does not provide any public caching interfaces. However, it has an internal cache system that you can use and benefit on.

4.9.1 Cache Class

Cache class (Supra\Core\Cache\Cache, accessible by `cache.cache` service key or `$container->getCache()` method) exposes the following methods:

- `fetch($prefix, $key, $default, $timestamp, $ttl, $respectDebug)` fetches and probably stores value in the cache (if current value was not found). The parameters are explained below:
 - `$prefix` and `$key` parameters are quite self-explanatory;
 - `$default` value can be a scalar or a callable (checked by `is_callable`), which is being called only on cache miss;
 - `$timestamp` is a last modification timestamp allowing you to refresh the cache. It's very handy for storing and combining assets;
 - `$ttl` is a time to live value;
 - `$respectDebug` turns cache off **development environment** when set to `true`. Basically, the cache will still work, but the values in there will be overwritten with every page request.
- `store($prefix, $key, $value, $timestamp, $ttl)` directly proxies data to storage driver. If `$value` is callable, it is invoked;
- `delete($prefix, $key)` deletes value from driver;
- `clear($prefix)` deletes all values for particular prefix;

We've tried to make the cache as simple as possible, so the common usage pattern with callback looks like the following:

```

1 <?php
2
3 $result = $container->getCache()->fetch('internal', 'value1', function() use ($container) {
4     //this code will be called only on cache miss
5     $value = $container['some.service']->doSomeHeavyOperation();
6
7     return $value;
8 }, filemtime('file.txt'), 3600);

```

4.9.2 Doctrine Wrapper

Some libraries can use **Doctrine Cache** as a cache layer. SiteSupra provides wrapper over its native cache, `DoctrineCacheWrapper`, which you can use in the following way:

```

1 <?php
2
3 use Supra\Core\Cache\DoctrineCacheWrapper;
4
5 $wrapper = new DoctrineCacheWrapper();

```

```
6 $wrapper->setContainer($container);
7
8 $wrapper->setPrefix('supra_native_prefix');
9 $wrapper->setSuffix('doctrine_cache_suffix');
10
11 // now you can use $wrapper anywhere in your code
12 // where instance of CacheProvider is required.
13 // Supra will respect ``prefix`` that you have set,
14 // and Doctrine will use ``suffix``
```

4.9.3 Cache Drivers and Current Implementation

SiteSupra has only one cache driver implemented at the moment. The driver called `File` stores cached data under `storage/cache` folder. The driver creates separate sub-folders for each `prefix` you define. There are some cache-specific `CLI` commands available for cache data management.

More cache drivers to come soon. You can always write your own driver just by implementing `Supra\Core\Cache\Driver\DriverInterface` interface.

4.10 Blocks and Editables

4.11 Templating

SiteSupra uses `Twig` template engine. Nothing unusual, just few things worth to mention:

- There's one `Twig` extension, called `CmsExtension`, that provides functions for preparing CMS JS/CSS assets, and another one, called `PageExtension`, that manages CMS-specific functions and tags (more on that later);
- Views reside in package's `Resource\view` folders, just like in `Symfony`;
- You can reference template inside a package and use shorthand syntax like `{% extends 'SamplePackage:layouts/base.html.twig' %}`.

To reference and render a template you can always access `Twig` environment by calling `$container->getTemplating()` or call `renderResponse` from your controller (package defaults to current package here):

```
1 <?php
2
3 public function indexAction()
4 {
5     return $this->renderResponse('index.html.twig');
6 }
```

You can register a custom extension during package injection;

```
1 <?php
2
3 public function inject(ContainerInterface $container)
4 {
5     $container->getTemplating()->addExtension(new PageExtension());
6 }
```

4.11.1 PageExtension

To explain `PageExtension`, we need to discuss two things: `PageExecutionContext` and `BlockExecutionContext` (make sure you've read [SiteSupra Concepts](#) first).

Both of these objects are simple container classes for objects defining current block or page being executed. Both of them contain a `Request` object (plain `Request` for `Blocks` and `PageRequest` for `Pages`) and a `Controller` object (`BlockController` and `PageController` accordingly).

Note: As other SiteSupra internal features, this is likely to change in the future.

This extension defines one filter, called `decorate` that works with internal `HtmlTag` instances, and a few functions listed below:

- `collection()`, and `list()` resolve property to a collection, example would be writing `{% for item in collection(property('image', 'image')) %}`;
- `set()` resolves property to a set;
- `property()` fetches single property from a `Block` or `Page`;
- `isPropertyEmpty()` checks if property value is empty;
- `placeholder()` defines a placeholder (see [Blocks and Editables](#) and [SiteSupra Concepts](#) for more information).

Every function in `PageExtension` is based on a custom `node_class`. This facilitates the process of dynamic creation of block properties when template is parsed. `BlockPropertyNodeVisitor` creates block properties on-the-fly.

4.12 Development and Production

4.13 Security

SiteSupra does not provide any kind of authentication for user part of CMS; it only provides authentication and user management layer for CMS part, decoupled in separate `SupraPackageCmsAuthentication` (more on standard packages in [corresponding section](#)). So, the documentation below applies only to CMS part, but you can always add authentication to your website following this [cookbook article](#).

SiteSupra security layer is based on [Symfony security component](#).

4.13.1 Security Concepts and Configuration

Security is blindly bound to `cms.prefix` container parameter and secures all URLs beginning that. URL mapping happens in `CmsAuthenticationRequestListener`. When visitor is not authorized yet, then the visitor is being redirected to CMS login page.

Note: We are likely to extend security layer to both backend and frontend - stay tuned!

The second listener, `CmsAuthenticationResponseListener`, ensures that current `Token` is stored in user session under the key defined by `cms_authentication.session.storage_key` parameter.

SiteSupra dispatches `AuthController::TOKEN_CHANGE_EVENT` every time a new token is stored in the session. Voters and ACL's are enabled, but not used yet.

Default security configuration is stored in `Supra\Package\CmsAuthentication\Resources\config\config.yml`. Apart from paths and services, it defines a shared user source (explained below), sets up user providers (bound to `CmsAuthentication:User` entity), both combined into provider chain, and sets `SupraBlowfishEncoder` as a default password encoder.

4.13.2 CLI Commands

SiteSupra provides some basic user management commands (for adding and removing backend user groups) allowing you to manage users event if the database is empty. refer to [Command Line Interface](#) for more details.

4.13.3 User Source and User Provider

By default SiteSupra uses `Supra\Package\CmsAuthentication\Entity\User` as base user entity and corresponding repository (which already implements `Symfony\Component\Security\Core\User\UserProviderInterface`) as a user source. Again, by default it is bound to current connection (please refer to [Database \(Doctrine 2\)](#) and [EntityAudit](#) if you want to learn more on SiteSupra database layer).

4.13.4 Shared User Provider

While developing web project it is good to have a shared user database with some default user accounts in there or share users between SiteSupra installations in production. This is possible by defining a new database connection in main configuration file (`supra\config.yml`) under `cms_authentication` → `users` → `shared_connection` as shown in example below:

```
1 cms:
2   active_theme: default
3 framework:
4   doctrine:
5     credentials:
6       hostname: localhost
7       username: root
8       password: ~
9       charset: utf8
10      database: supra9
11 cms_authentication:
12   users:
13     shared_connection:
14       host: localhost
15       user: root
16       password: ~
17       charset: utf8
18       dbname: supra9_shared_users
19       driver: mysql
20       event_manager: public
21     user_providers:
22       doctrine:
23         supra.authentication.user_provider.public:
24           em: public
25           entity: CmsAuthentication:User
26       provider_chain: [ doctrine.entity_managers.public ]
```

Cookbook Articles:

5.1 Creating Custom Block

This tutorial will help you to create a simple block with manageable HTML content.

Note: This tutorial assumes that you've already read the section about [Blocks and Editables](#) and already have sample [CMS Package](#) configured.

5.1.1 Block Configuration

Create a class, that would represent your block configuration. It should extend abstract `BlockConfig` class:

```
1 <?php
2
3 namespace MySamplePackage\Blocks;
4
5 use Supra\Package\Cms\Pages\Block\Config\BlockConfig;
6
7 class MyTextBlock extends BlockConfig
8 {
9 }
```

Override `configureAttributes` method to define block title and description.

```
1 <?php
2
3 namespace MySamplePackage\Blocks;
4
5 use Supra\Package\Cms\Pages\Block\Config\BlockConfig;
6
7 class MyTextBlock extends BlockConfig
8 {
9     public function configureAttributes(AttributeMapper $mapper)
10    {
11        $mapper->title('My Text Block')
12        ->description('This block provides you WYSIWYG editor.');
```

5.1.2 Create Block Template

Create Twig file named `my_text_block.html.twig` in `Resources/view/blocks/` directory with the following code in there:

```
1 <div>{{ property('my_content', 'html') }}</div>
```

This will dynamically create block property named `'my_content'` and link CMS WYSIWYG editor to that property.

Note: You may override template name by calling `$mapper->template('MyPackage:path/to/file.html.twig')` inside `BlockConfig::configureAttributes()` method.

5.1.3 Register Your Block in CMS

The last but not least step is register the block with CMS. If your package extends `AbstractSupraCmsPackage`, then just override `getBlocks` method:

```
1 <?php
2
3 namespace MySamplePackage;
4
5 use Supra\Package\Cms\AbstractSupraCmsPackage;
6
7 class MySamplePackage extends AbstractSupraCmsPackage
8 {
9     ...
10
11     public function getBlocks()
12     {
13         return array(new Blocks\MyTextBlock());
14     }
15 }
```

Otherwise, this can be done by calling `BlockCollection::addConfig()` on package initialization finish.

```
1 <?php
2
3 namespace MySamplePackage;
4
5 use Supra\Core\Package\AbstractSupraPackage;
6
7 class MySamplePackage extends AbstractSupraPackage
8 {
9     ...
10
11     public function finish(ContainerInterface $container)
12     {
13         $blockCollection = $container['cms.pages.blocks.collection'];
14         /* @var $blockCollection \Supra\Package\Cms\Pages\Block\BlockCollection */
15
16         $blockCollection->addConfig(new MyTextBlock(), $this);
17     }
18 }
```

That's all. Your block is now registered and should appear in site block list.

5.2 Creating a CRUD

5.3 Creating Custom Controller

5.4 Writing Your Own Command

5.4.1 General Considerations

We would recommend to read [Symfony's guide to creating basic command](#) prior to proceed further. SiteSupra mostly follows the same approach with a few minor differences:

- A command must extend `Supra\Core\Console\AbstractCommand` (or implement `Supra\Core\DependencyInjection\ContainerAware`), read more about that in [Dependency Injection](#) chapter;
- There is no limit on where you can store your commands or what namespace are you using. SiteSupra does not autoload commands, so you are free to choose your class structure.

Basic console command can look like shown below:

```
<?php

namespace Some\Your\Namespace\Command;

use Supra\Core\Console\AbstractCommand;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class FoobarCommand extends AbstractCommand
{
    protected function configure()
    {
        $this->setName('foobar:do')
            ->setDescription('Does some foobar');
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        //foobar is happening here
    }
}
```

Of course, you can use any of the [helpers](#) bundled in Symfony Console - tables, dialogs, and so on.

5.4.2 Registering New Command from Your Package

Let's assume that you have a Package created (refer to [Creating Sample CMS Package](#) for package creating instructions). Now you can register a new command instance in package's `inject()` method:

```
<?php

namespace Vendor\Package;

use Supra\Core\DependencyInjection\ContainerInterface;
use Some\Your\Namespace\Command\FoobarCommand;
use Supra\Core\Package\AbstractSupraPackage;
```

```
class FoobarPackage extends AbstractSupraPackage
{
    public function inject (ContainerInterface $container)
    {
        //some magic here...

        $container->getConsole()->add(new FoobarCommand());

        //even more magic here...
    }
}
```

After that, you can run your command with `supra/cli.php foobar:do` (shortcuts like `supra/cli.php f:d` are working also).

5.5 Creating Sample CMS Package

6.1 SiteSupra User Manual

Welcome to the SiteSupra CMS guide. This guide provides comprehensive information on how to organize your SiteSupra site's content and manage it easily. Free and open source, SiteSupra offers fast and easy website development and makes website editing and administration stress-free thanks to intuitive visual CMS.

6.1.1 Log in

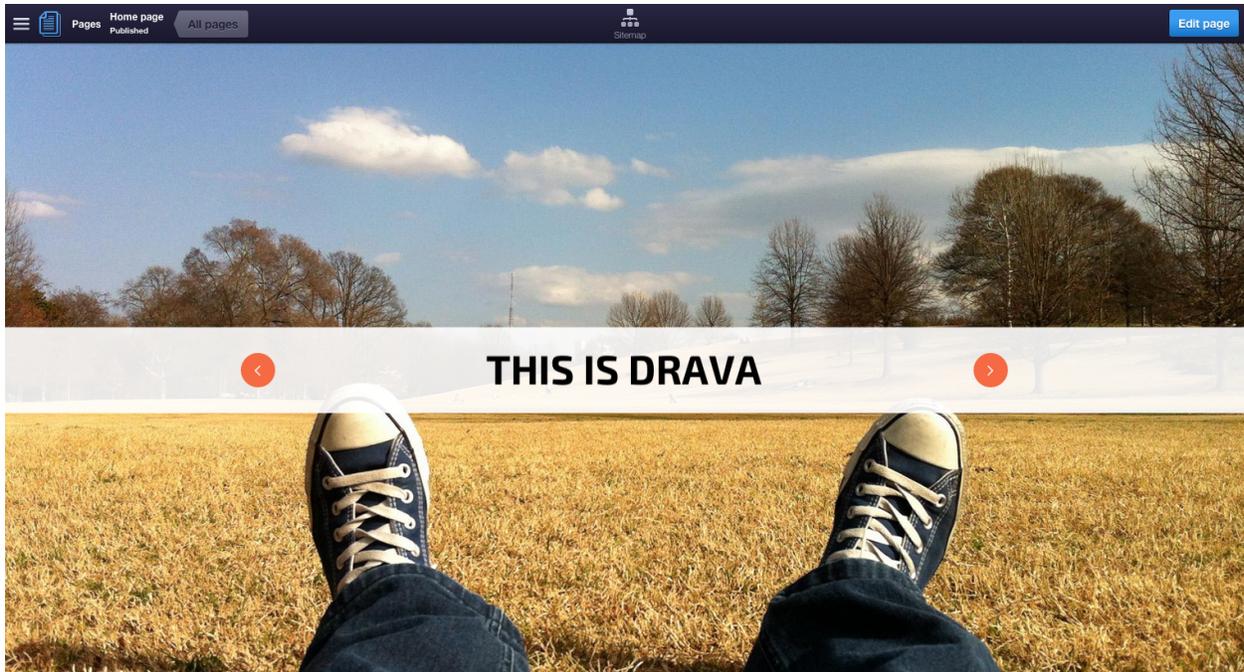
Start by logging into the admin panel of your website. Visit your site's login page by placing “/backend” after your domain name e.g. <http://example.com/backend>. Default login name is `admin@sitesupra.org` and password is `admin`.

Initial view

After you logged in, you are on the site's main page view. This is the Home page of your website. Here your creativity can break out and let you build as lovely website as possible.

On the upper menu you can do following:

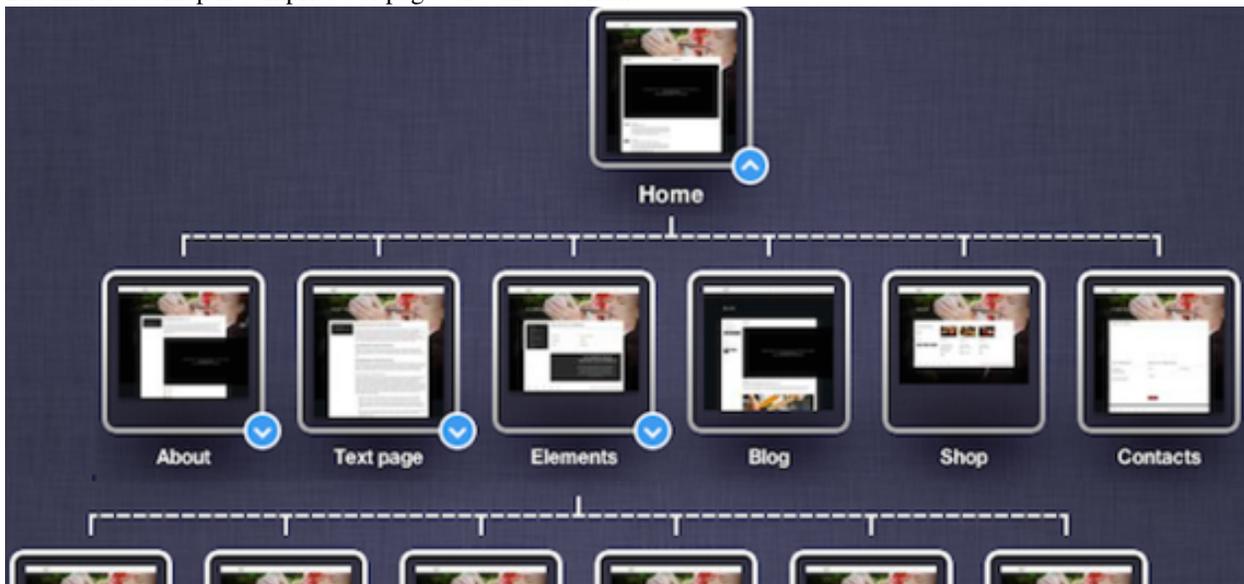
- Open the Dashboard;
- Open website's structure or “**Sitemap**”;
- Open Page for editing



We recommend you to check the demo of Home page and the Blocks to see most of the SiteSupra page and text management features. However, ahead of jumping in and start editing, take a look at the website's structure by clicking **Sitemap** icon.

6.1.2 Site Structure

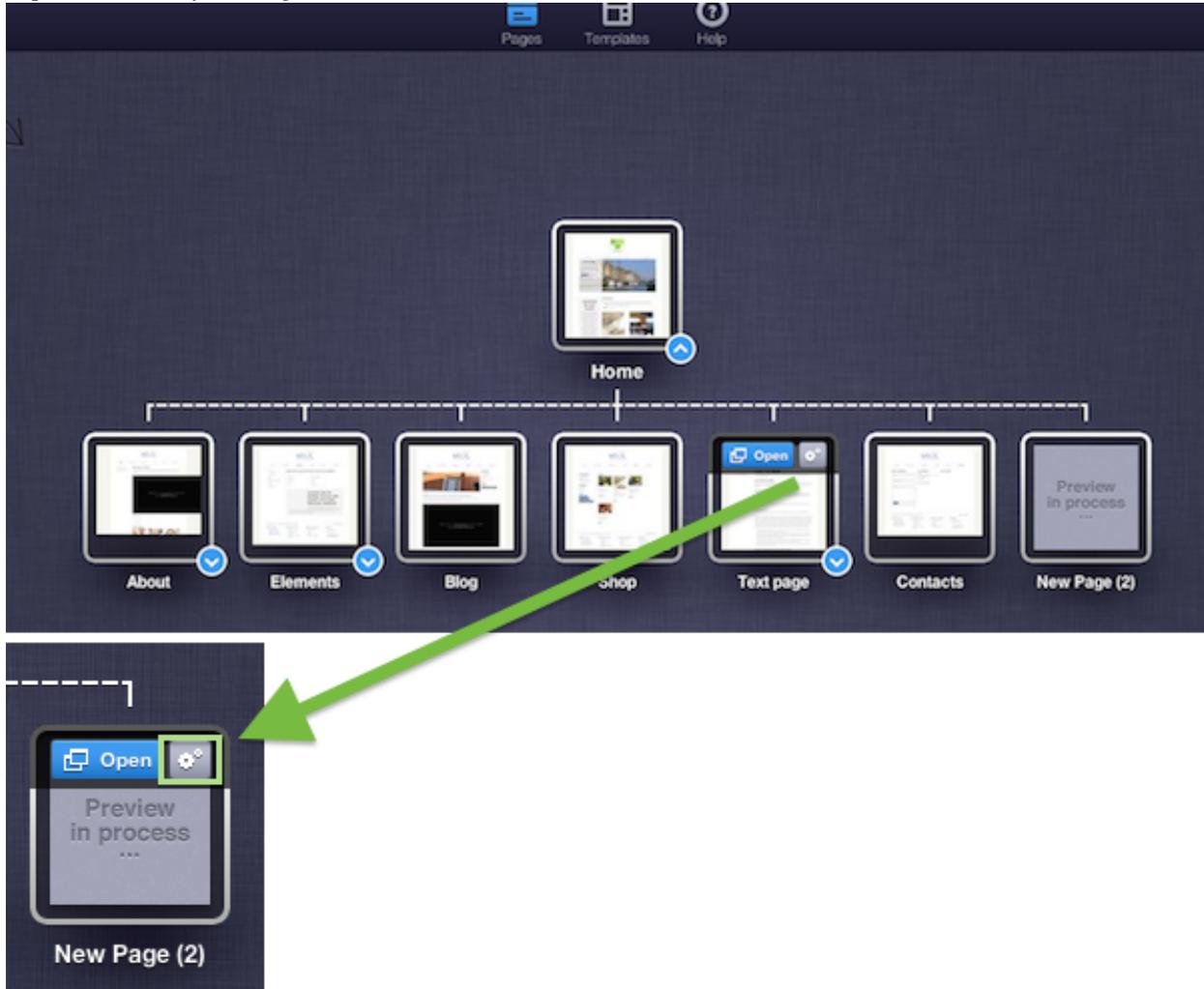
The initial site map is comprised of pages that show content.

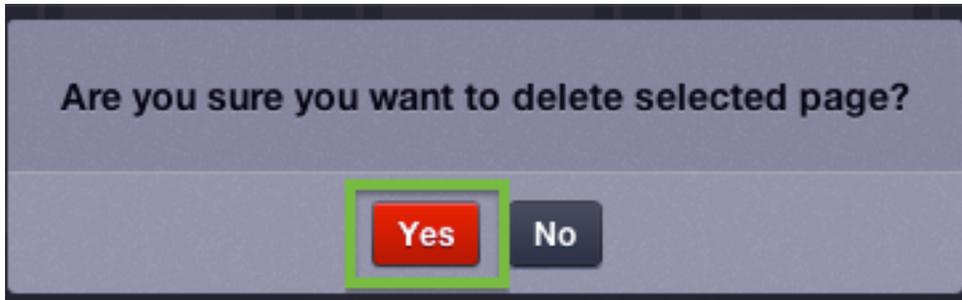
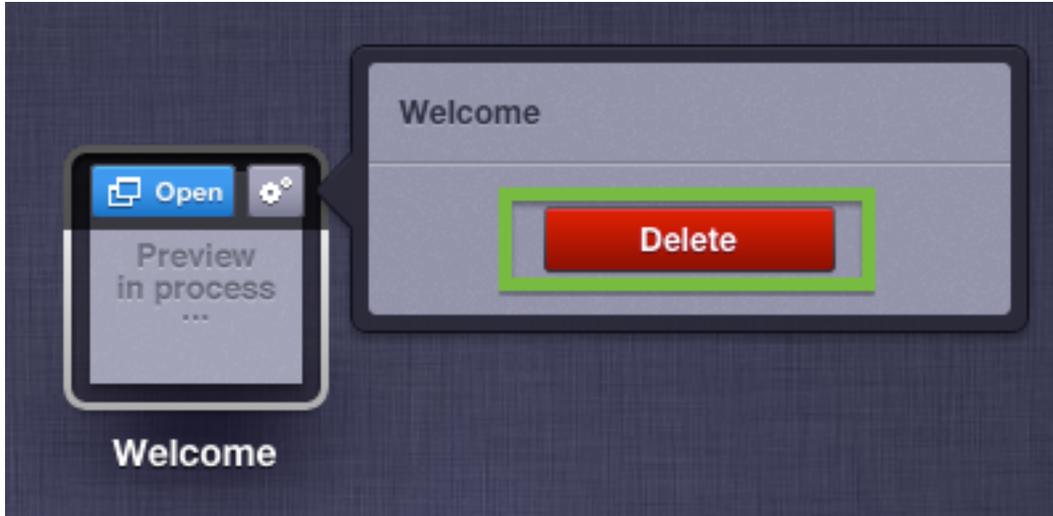


6.1.3 Site Structure Management

To delete a page

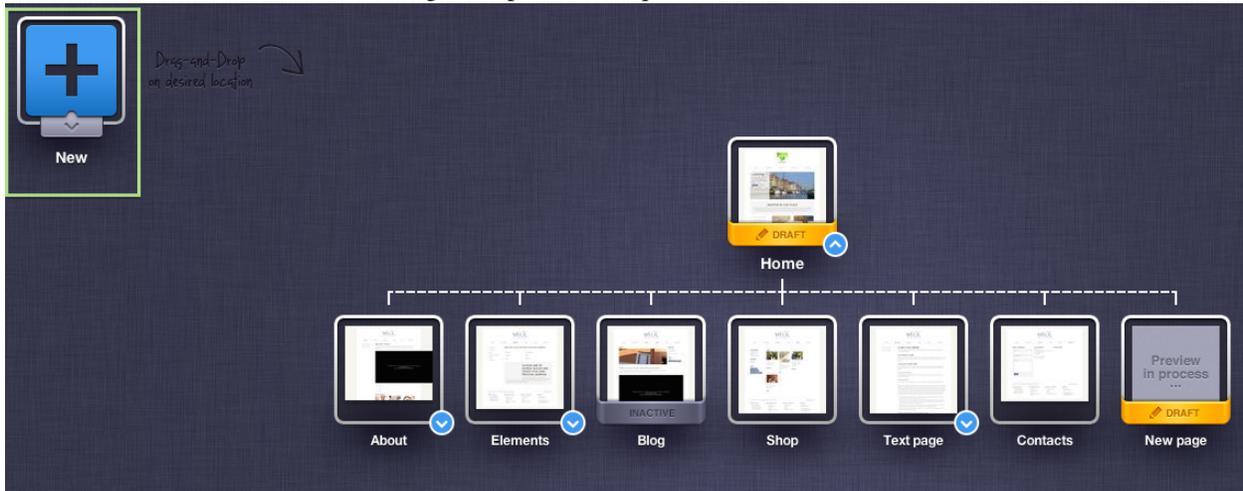
Choose the page you have decided to delete and click the **Gears** icon. To delete a page click **Delete** and confirm requested action by clicking **Yes** button.





To add a new page

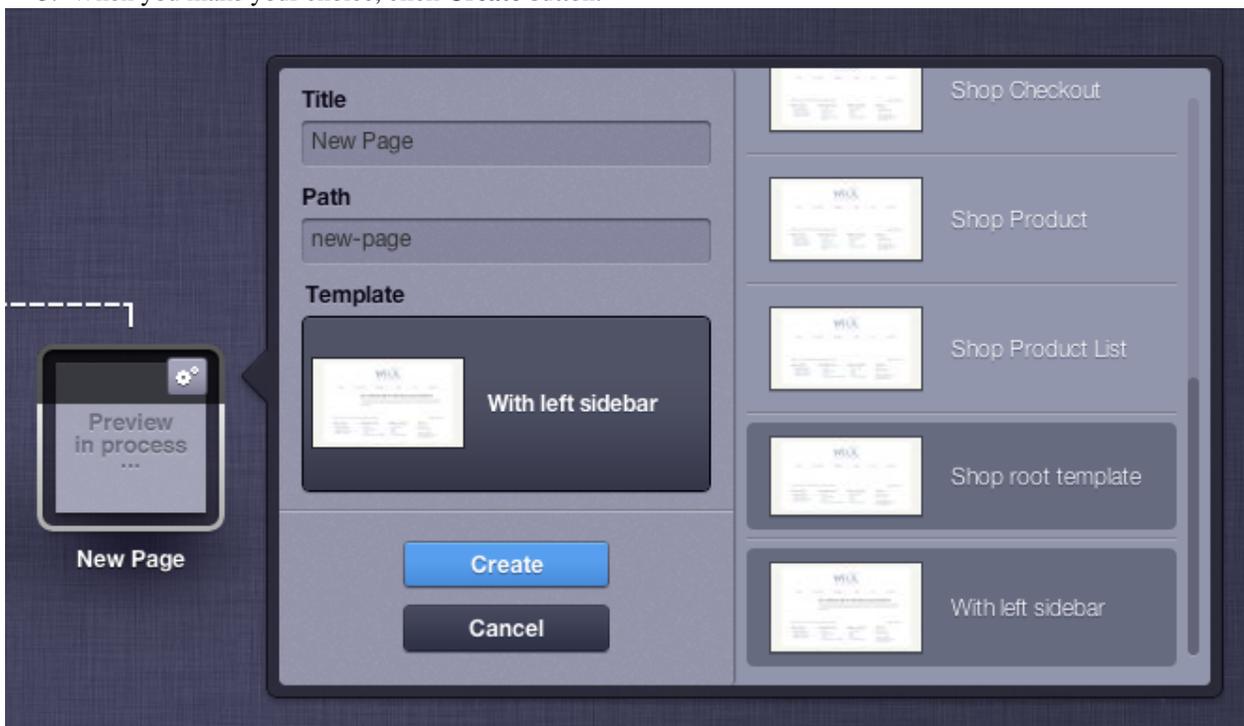
1. Click and hold ”+“ icon and drag & drop it to the required location.



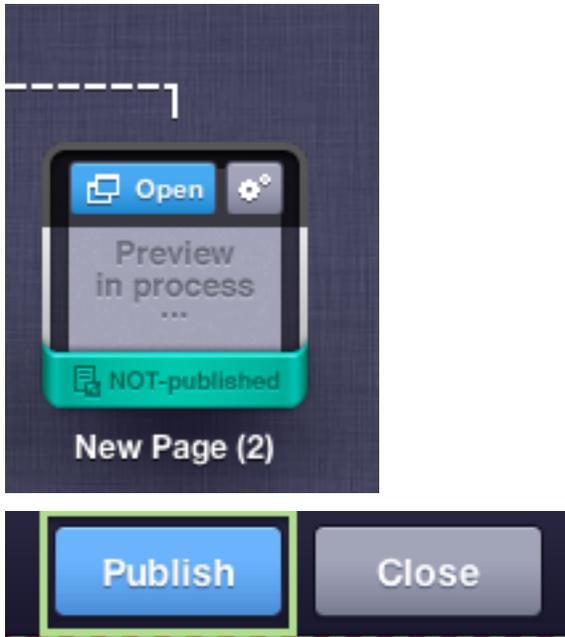
2. When you hold and move the page around the **Sitemap** the drop location will be marked in blue.



3. After you drop the page on the desired location a page properties window appears. Specify page **Title**. You may define Path which is a page name that appears in the page address.
4. Click **Template** to choose a page template you want. You can go through a wide array of templates.
5. When you make your choice, click **Create** button.



6. Open created new page and specify required content for page blocks. Then click **Publish** to the upper right of the window.



NOTE

If by an accident you have dropped your new page in a wrong place, you can easily change page order in the menu. Simply drag & drop new page to the required place in Sitemap view.



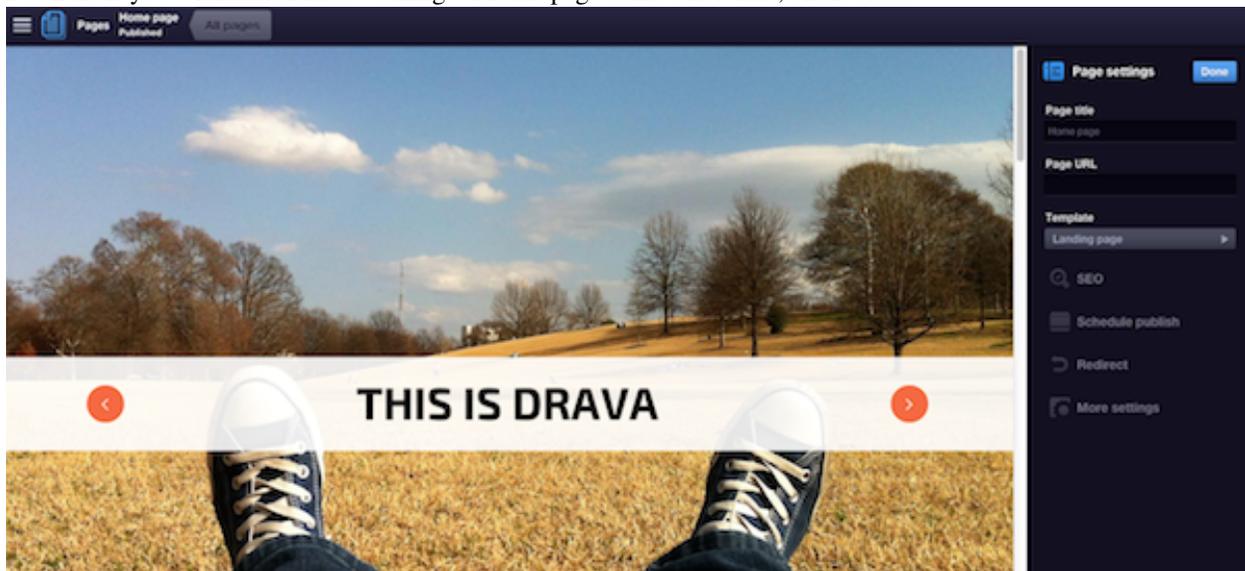
Page Settings

Page Settings contains the page editing menu in the upper right pane and here you can specify following:

- Page Title - Page Heading

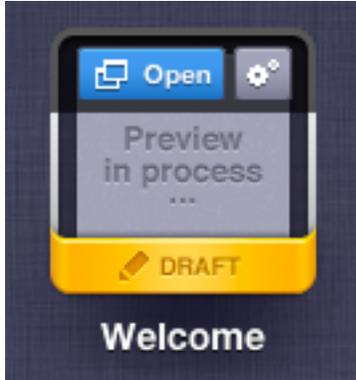
- Page URL - Page address
- SEO - additional parameters for website optimisation to help perform better in search results.
- Schedule publish - you can schedule a page to publish automatically in the future.
- Redirect - you can redirect your page visitors to other pages or external resources.
- More settings - additional page settings:
 - Page status management (active or inactive)
 - Page appearance in search results (Yes or No)
 - Page appearance in Menu
 - Page appearance in Sitemap
 - Admission to translate the page

After you have made all the necessary changes, click **Save** to save changes. To publish final version of the page, click **Publish**. If you wish to continue working with this page some other time, click **Close**.



Page status

While in a **Sitemap** view, above the page icons or folders can appear inscription **Draft**, which means that page was changed, but the changes were not published. This inscription appears whenever you save the changes and close the page with the **Close** button without **Publishing**.



6.1.4 Main Toolbar options

General editing options are displayed at the top of the menu.



- **Insert block** - to insert new blocks on the page (see full description here [link to blocks])
- **Page blocks** - displays a list of elements available for editing;
- **Page settings** - to access and manage page properties
 - Page Title
 - Page URL
 - Template
 - SEO
 - Schedule publish
 - Redirect
 - More settings

6.1.5 Main Menu Management

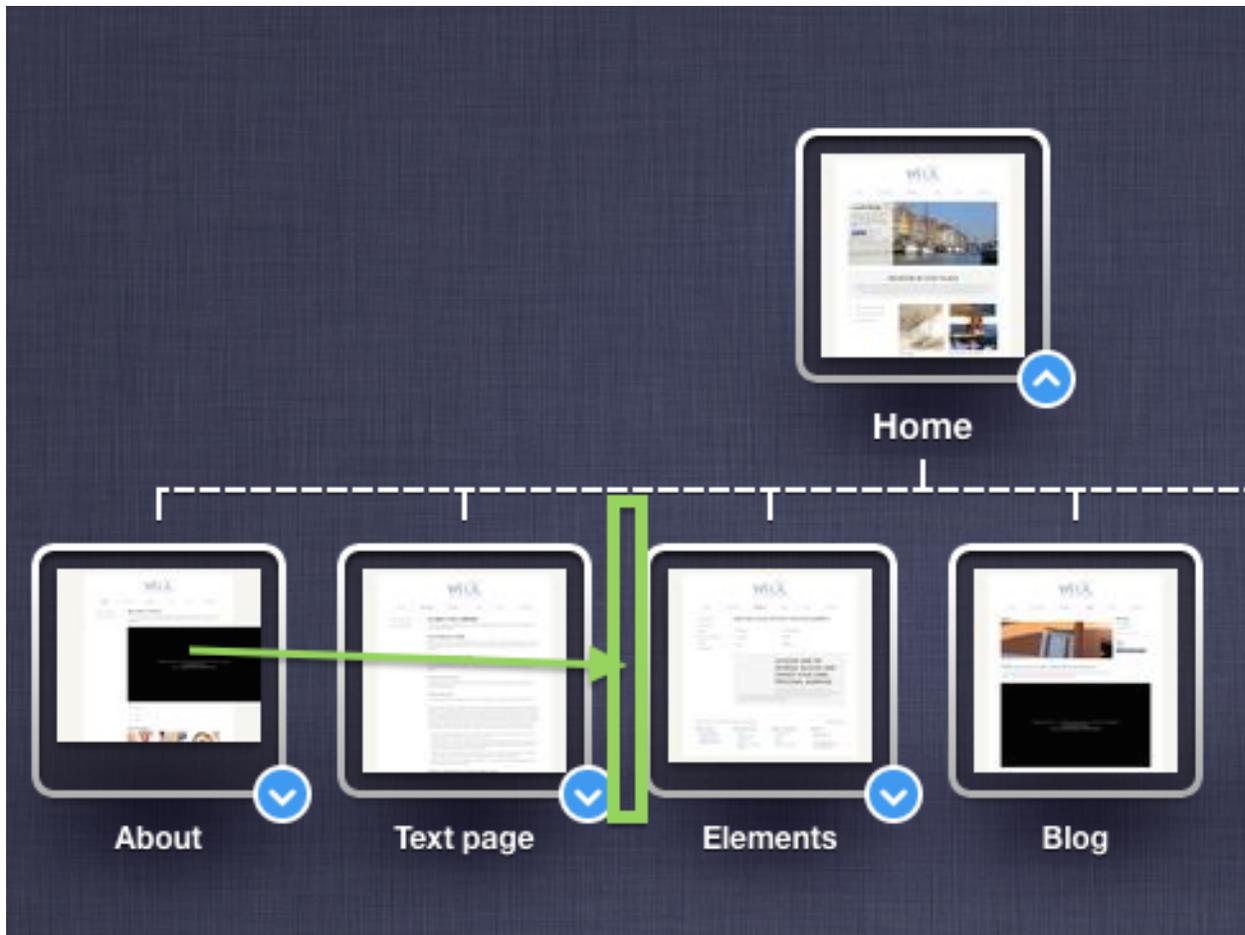
The main menu block automatically picks up pages located under the Home page and builds menu according to the web site styles. If you want your main menu to have a page simply place it under the Home page in the **SiteMap**. You can drag and drop the pages into this level or rename existing pages. Here's how:

To change page order in the menu

1. You can access Site map by clicking **Sitemap** or **All pages** buttons on the page toolbar.



2. In the Site map, click and hold required page and drag it to the new location.

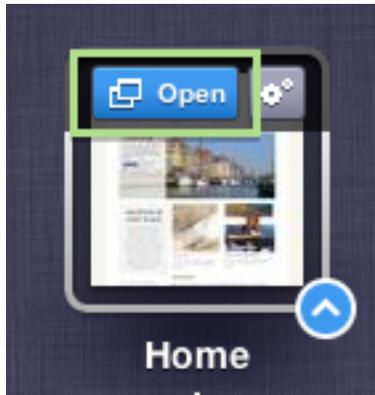


6.1.6 Blocks

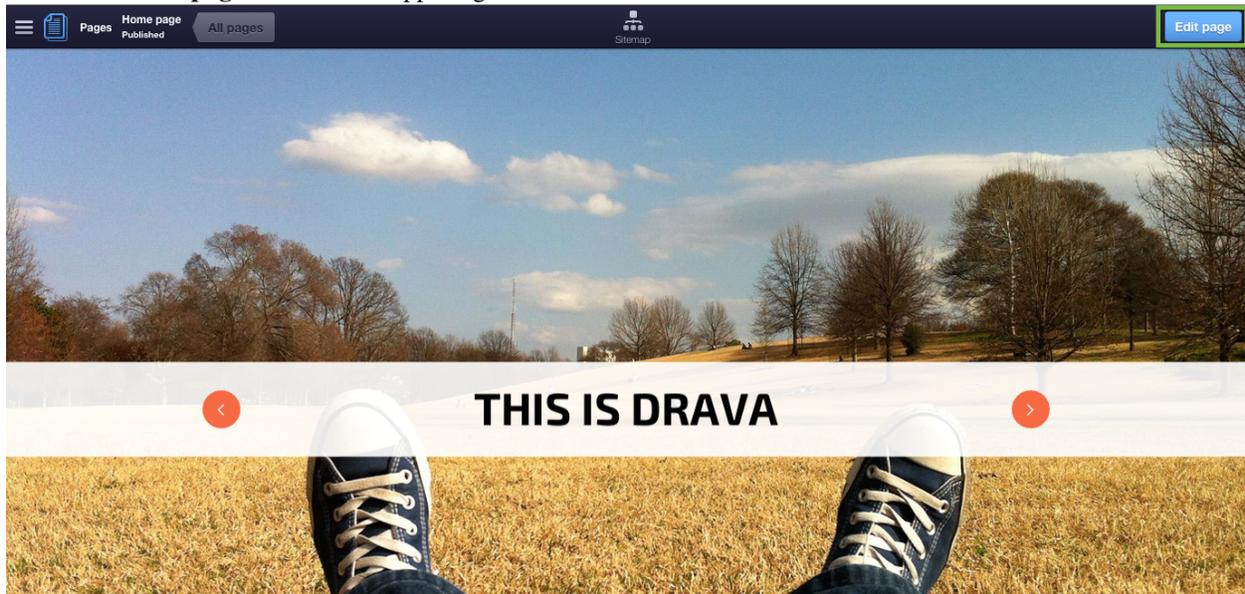
Each page consists of several functional widgets, the blocks, which add functions and events to your website. You can edit them by clicking and managing customisations in the block properties which opens on the right. For each block there are different properties, for example, you can add images to the gallery block and set the links for social media follow block.

To add new block

1. While in a **Sitemap** view, select the page you want to edit and click **Open**.



2. Click **Edit page** button to the upper right of the window.



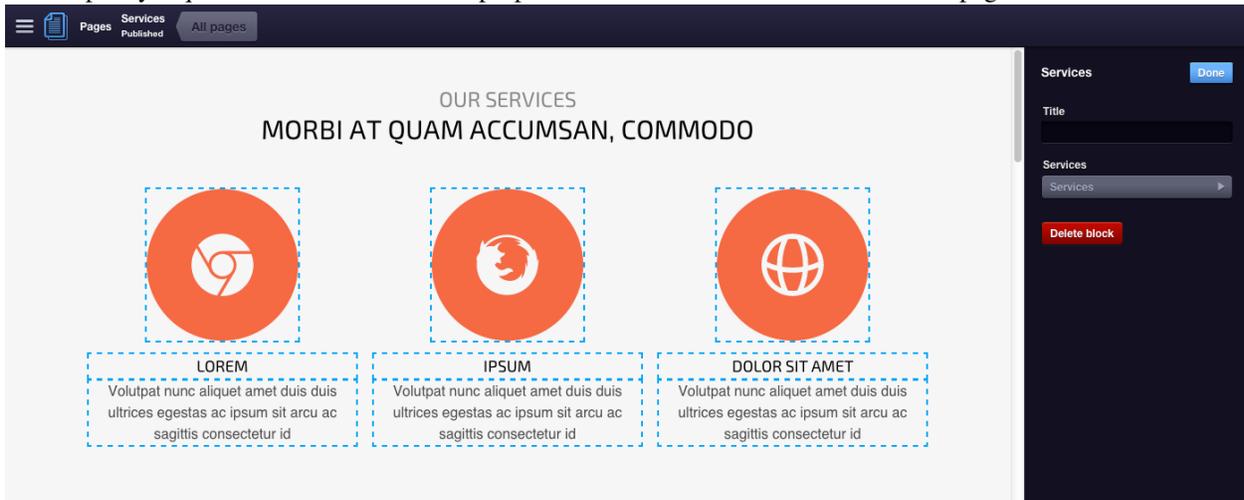
3. Click **Insert block** in the main toolbar, which appears on top of the page. The **Insert block** panel opens on the left. Then select **Features**.



4. Click the block you need and drag & drop it to where you want it in the page.



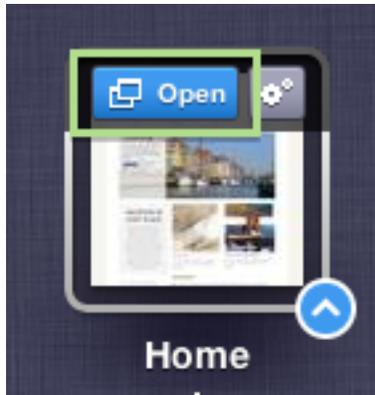
5. Specify required content for the block properties. Then click **Done** and **Publish** the page.



To delete block

If you don't need a certain block it's easy to remove it from the page:

1. While in a **Sitemap** view, select the page you want to edit and click **Open**.

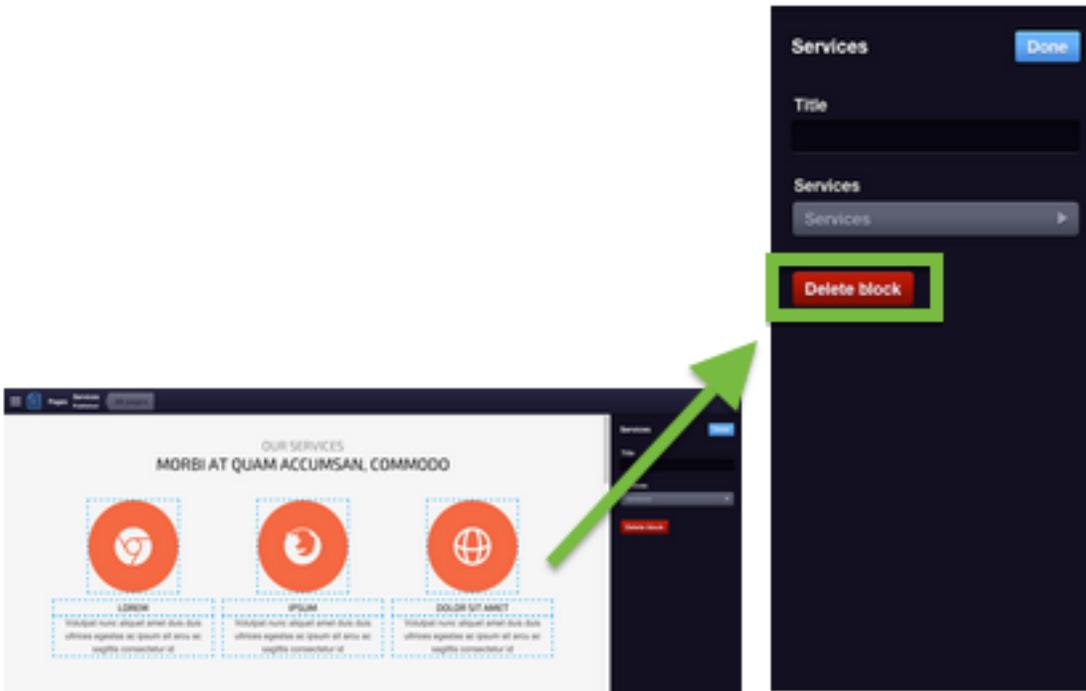


2. Click **Edit** page button to the upper right of the window to start page customisation.

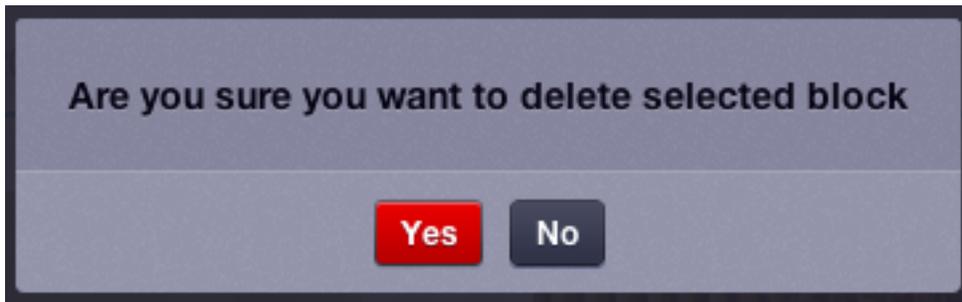


3. Click the block you want to remove. **Block properties** panel opens on the right side of window.

4. To remove a block click **Delete** block button.



5. Confirm your choice and click Yes.

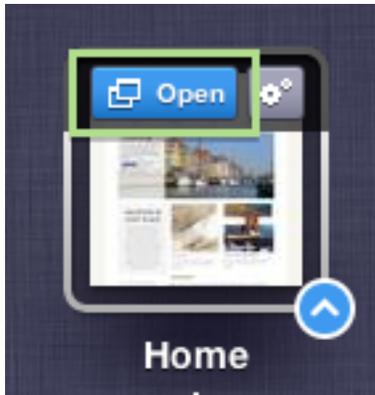


6. To finish work and save changes, click **Publish** button to the upper right of the window.

To move blocks within a page

It's easy to change location of the block on a page, here's how:

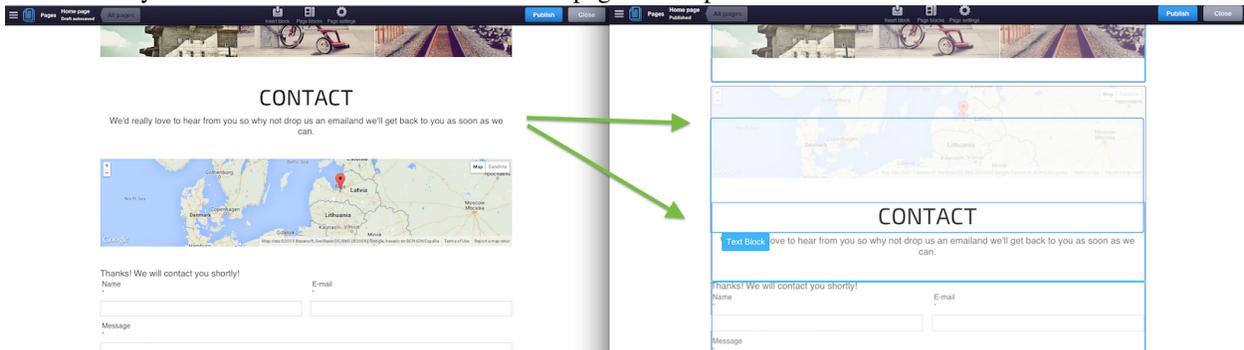
1. While in the **Sitemap** view, select the page you want to edit and click Open.



2. Click **Edit** page button to the upper right of the window.



3. Select the block you want to move by Clicking and holding it with the mouse and move it to the new location. Make sure not to click the block as you will enter in a block editing mode where moving blocks isn't possible. When you hold and move the block around the page the drop location will be marked in blue.



4. To finish and save block repositioning changes within a page, click **Publish** button on right-top of the window.

IMPORTANT: Blocks are divided into two main categories:

- **Global;**
- **Non-Global.**

If a block is Non-Global, it will appear only on pages where it is added manually, but if a block is Global, it will appear on all pages from selected template. These settings you can specify while creating or customising Templates.

6.1.7 Templates

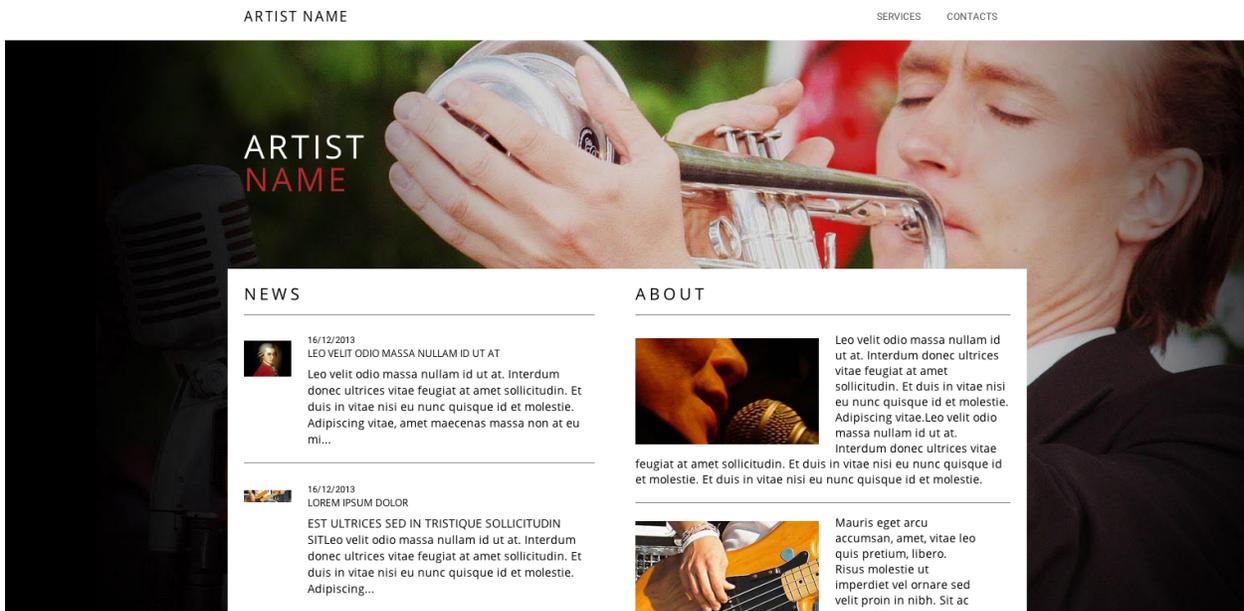
Templates control how your website appears. SiteSupra template provides a method of integration between content and blocks in a specific, controlled view. Site is created by first placing one or more blocks on a template and then creating pages based on those templates. While each template can be configured separately, when adding new pages and selecting template, page will consist of template's specified design, layout and blocks within placeholders, so you can significantly save the time by creating new pages and content.

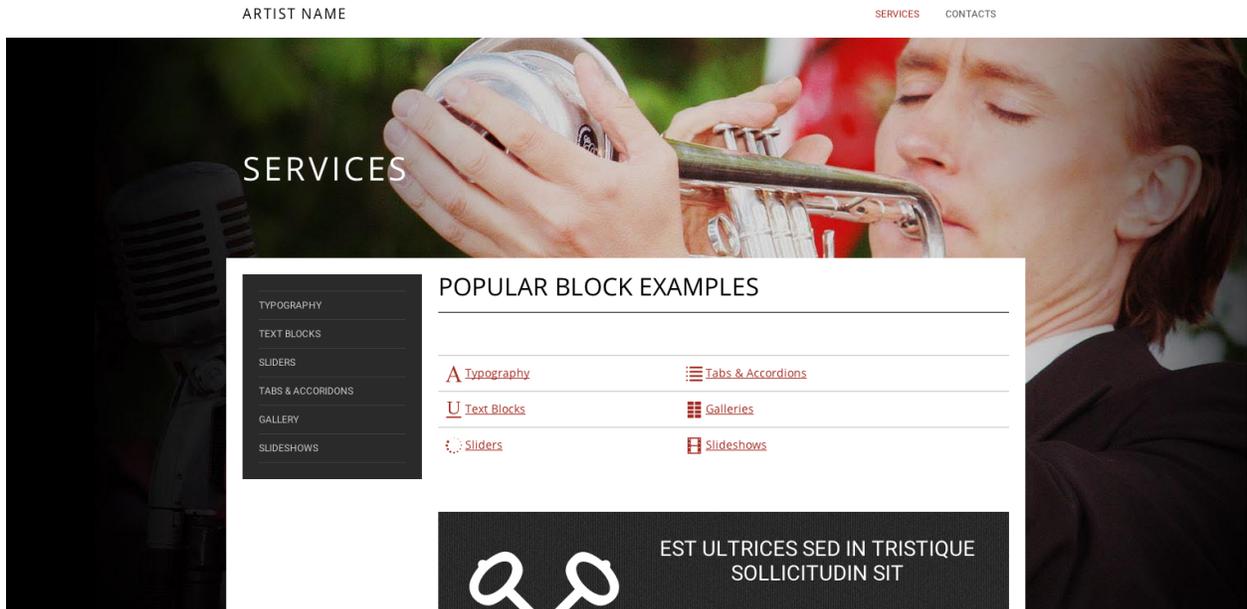
While in a **Sitemap** view, you can switch to **Templates** editing mode:



If you want to create a page with unique set of blocks to implement original page design, you would need to create a separate page template first so that other pages are not affected. Creating page template is easy, you can just duplicate a page template you like.

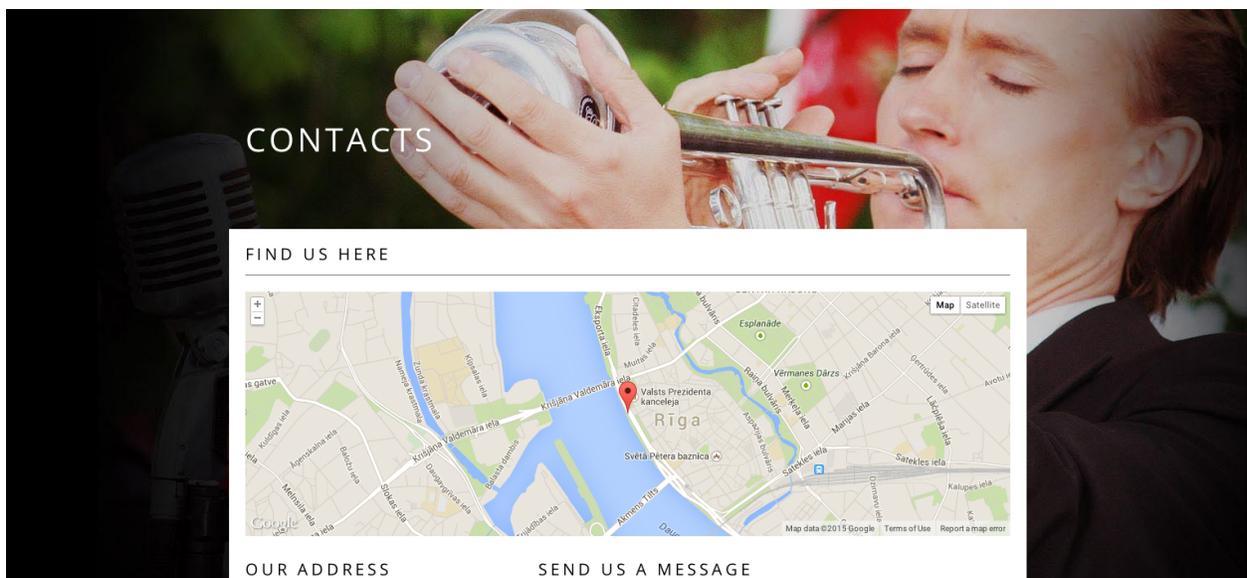
For example, website consists of three simple pages: Home, Services and Contacts.





ARTIST NAME

SERVICES CONTACTS

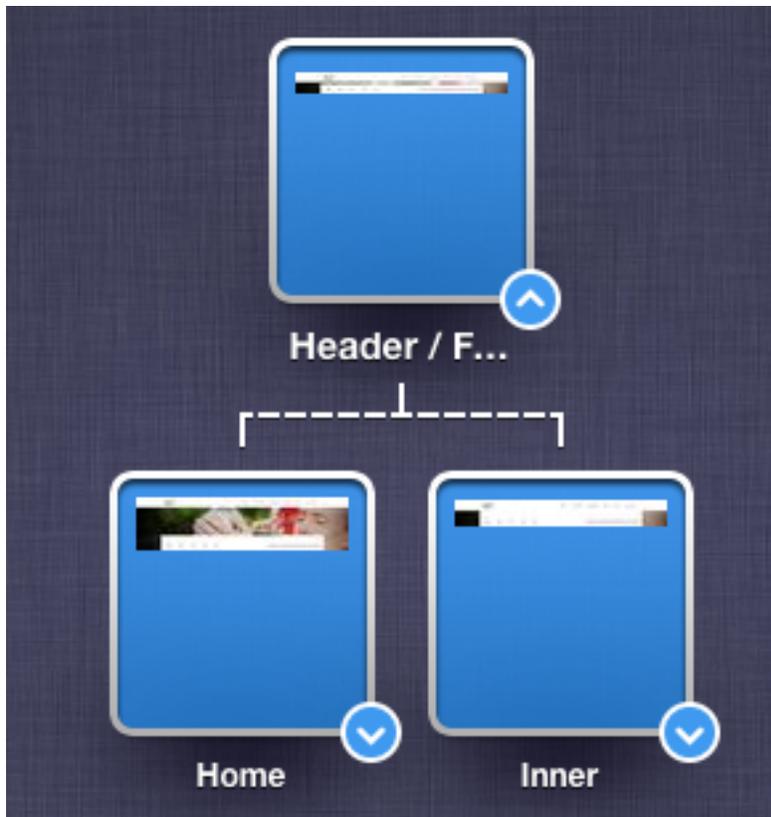


FIND US HERE

OUR ADDRESS

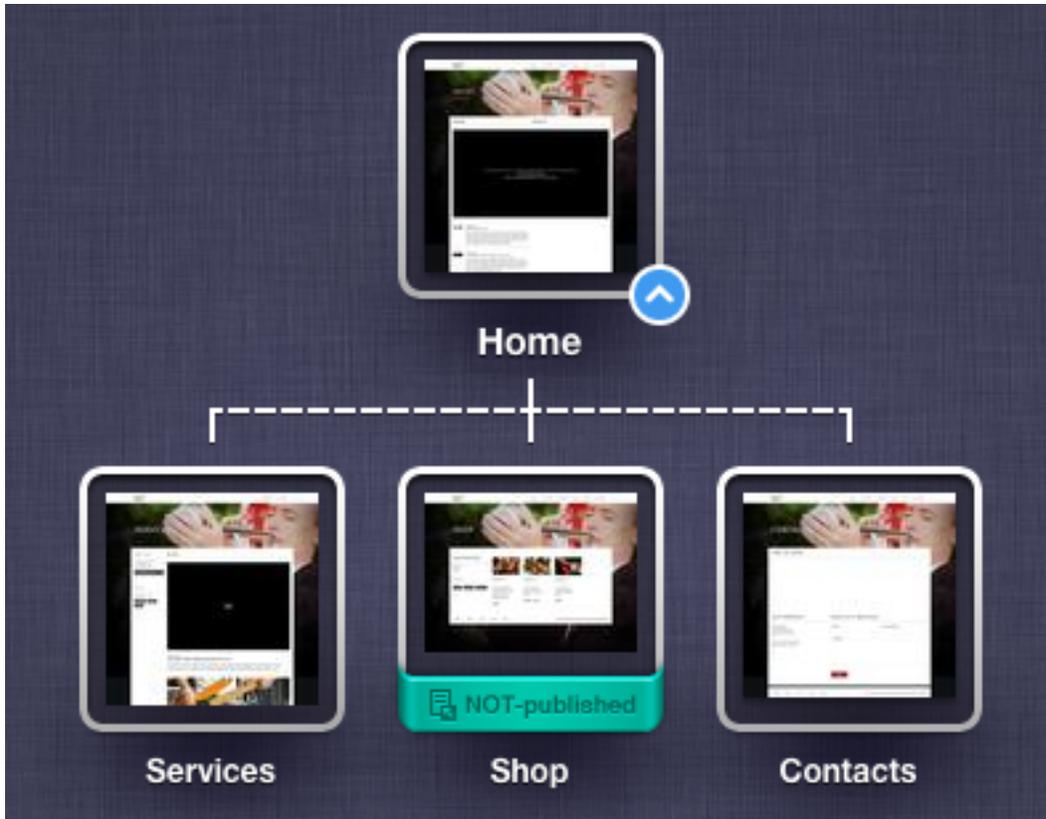
SEND US A MESSAGE

Which in templates view will look like this:



Where pages are built on basis of:

- Page Home is created from template Home, where Home page content is inherit Header/Footer and Home template content.
- Page Services is created from template Inner, where Services page content is inherit from Header/Footer and Inner templates.
- Page Contacts is created from template Inner, where Contacts page content is inherit from Header/Footer and Inner - the same as for Services.



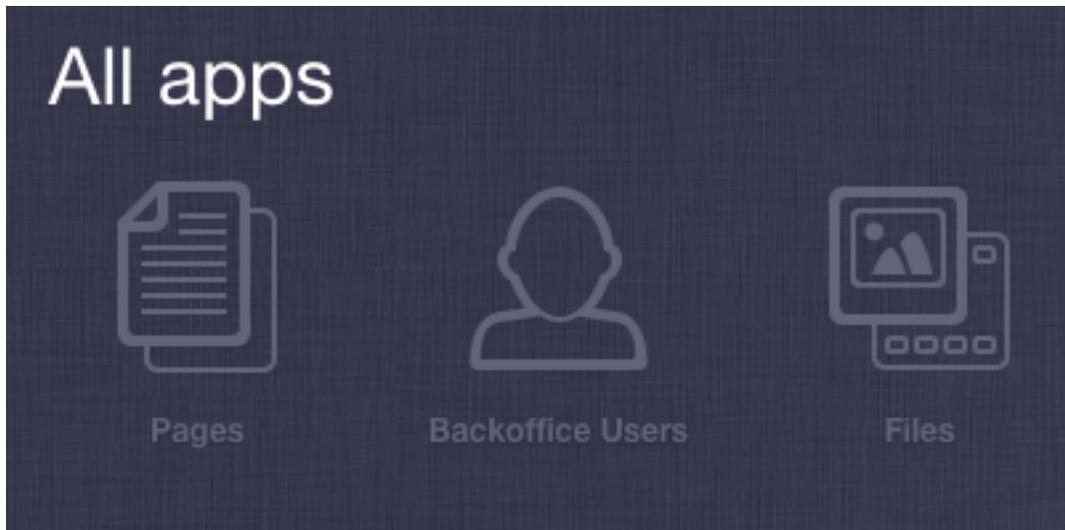
While creating pages, you can customise templates separately by adding necessary blocks to the placeholders.

6.1.8 Dashboard

You can open the **Dashboard** by clicking the icon on the upper left of the page.



The admin panel is very simplistic and from here you can manage *Back-office Users, Files* or return to *Site Structure Management*.

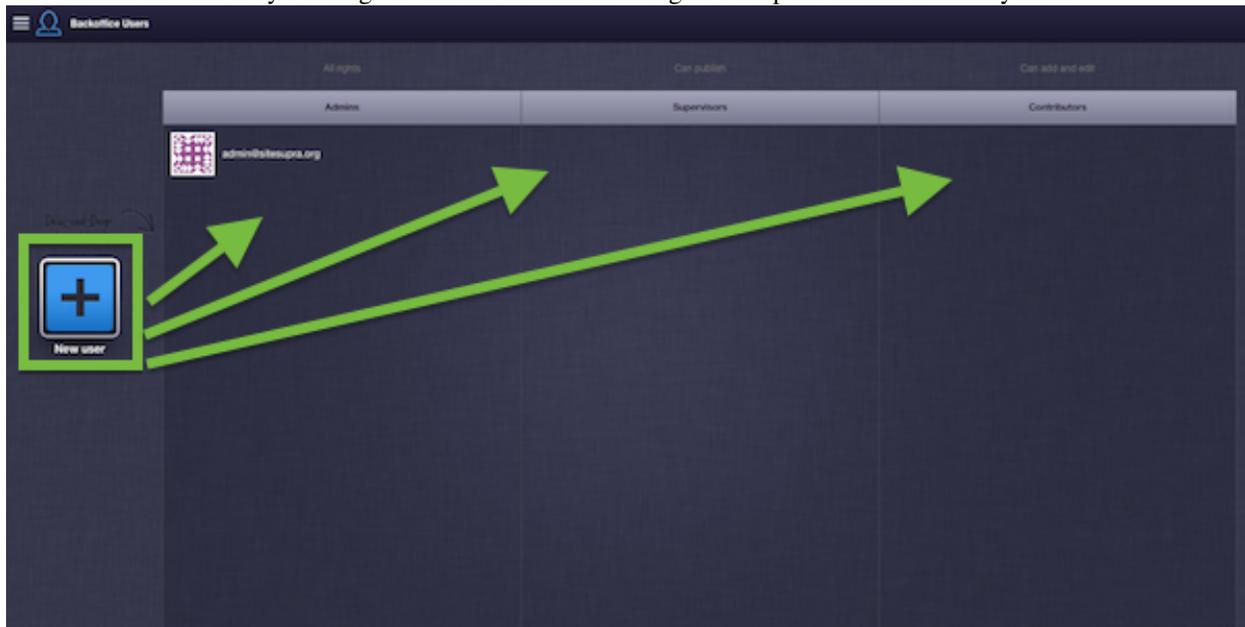


Back-office Users

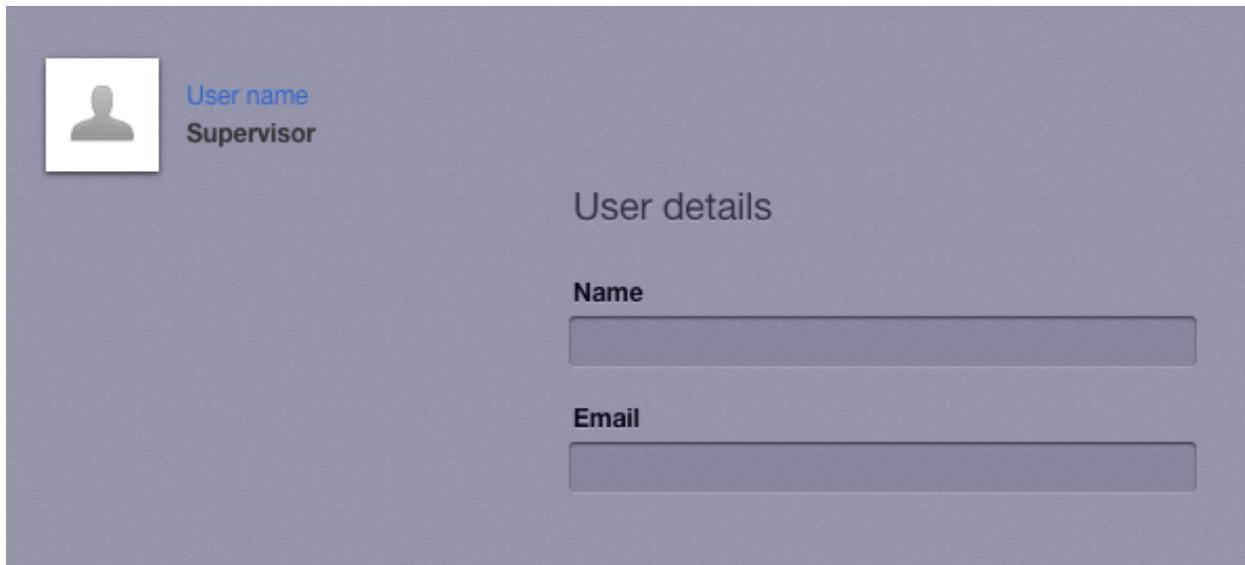
You can setup a user account for your website contributors and assign certain privileges. These privileges are known as roles. The following roles exist in the SiteSupra:

- **Admins:** You can do everything including creating new users and assigning them access rights.
- **Supervisors:** As a supervisor, you can publish (and unpublish) documents on the website, and approve or disapprove Contributor's requests for publication.
- **Contributors:** As a contributor, you can create and edit content in the CMS. When your changes are ready to be published on the website, they have to be approved by a Supervisor.

You can add new users by clicking “+” on the left side and drag-and-drop the icon to necessary role.



Then add User Name and E-mail, click **Done**. New user will receive invitation to become an admin, supervisor or contributor of your site.



The image shows a user profile interface. On the left, there is a square placeholder for a user profile picture. To its right, the text "User name" is displayed in blue, and "Supervisor" is displayed in black below it. On the right side of the interface, the heading "User details" is shown. Below this heading are two input fields: the first is labeled "Name" and the second is labeled "Email". Both input fields are currently empty.

Files

Program files (Files) app purpose is to gather all uploaded visual information files for further use on the website. This app also allows you to create the directory tree in order to improve your work with files.

General options are displayed at the top of the menu:

- Upload - possibility to upload necessary files
- New Folder - allows to create new directory
- Delete - possibility to delete unnecessary files or directories

Also it is possible to add new files by dragging and dropping them to required folders. To view image details, click on an image icon. From here you can also Download or Replace the file.

Reference:

7.1 Standard Blocks

Indices and Tables

- genindex
- search

B

- Block, 32
 - Creating new block, 35

C

- Cache, 31
- CLI, 21
- Command
 - Writing your own command, 37
- Controllers, 29
- Cookbook
 - Creating a CRUD, 36
 - Creating custom Controller, 37
 - Creating new block, 35
 - Creating sample CMS package, 38
 - Writing your own command, 37

CRUD, 36

D

- Dependency injection, 15
- Development and Production, 33
- Doctrine, 25

E

- Editable, 32
- EntityAudit, 25

H

HTTP kernel, 13

I

- Index, 1
- Installation, 7
 - Apache, 9
 - nginx, 9
 - Requirements, 7
- Internal: Page routing, 30
- Internals
 - Cache, 31
 - Concept, 11

- Database, 25
- Dependency injection, 15
- HTTP kernel, 13
- Standard packages, 13

P

- Package
 - Creating sample CMS package, 38
- Page Routing, 30
- Promo, 5

Q

Quickstart, 11

R

- Reference
 - Standard blocks, 59
- Routing, 28

S

- Security, 33
- SiteSupra Concepts, 11
- Software Requirements, 7

T

- Templating, 32
- Tools
 - CLI, 21