chill-doc Documentation

Release 1.0

Champs-Libres

November 29, 2016

Contents

Chill is a free software for social workers.

Chill rely on the php framework Symfony.

Contents of this documentation:

CHAPTER 1

Installation

1.1 Canonical installation

1.1.1 Install chill

Basic installation

Chill is written in PHP and use the symfony framework. We take advantages of all the framework's feature, and installation should be as simple as installing symfony.

We are going to describe a basic installation on Unix systems (Unix, Mac OS). Windows installation has not been tested.

Requirements

Client requirements Chill is accessible through a web browser. Currently, we focus our support on Firefox, because firefox is open source, cross-platform, and very well active. The software should work with other browser (Chromium, Opera, ...) but some functionalities might break.

Server requirements

- a postgresql database, with the *unaccent* extension enabled. The minimum version is postgresql 9.4. You can use the docker image provided. Using the docker image is also a solution for production site. Alternatively you can install a PosgresSql server see *Install PostgresSql server*.
- PHP version >= 5.5. Check that extensions php5-intl and php5-pgsql are installed and that '*date.timezone*' is correctly defined in your php.ini.
- Composer.
- If you run Chill in production mode, you should also install a web server (apache, ngnix, ...) see *Install production webserver*. For this basic installation meant for testing and/or development, we will make it simplier using the php built-in server.

Let's start by installing composer as it is needed to create and update our Chill project.

Install composer

Note: If you do not know composer, it is a good idea to have a glance at the composer documentation

Install composer on your system :

```
curl -sS https://getcomposer.org/installer | php
```

Install composer.phar globally

Note: This part is not mandatory, if you do not want to install composer globally, you will have to replace in the commands of this tutorial *composer* by *php composer.phar*.

Install composer globally on you system will made the installation process easier. To do this, simply run

```
sudo mv composer.phar /usr/local/bin/composer
sudo chmod +x /usr/local/bin/composer
```

You can test the installation by running *which composer* or *composer*: those command should not raise any error.

Note: See the composer introduction to learn how to install composer on Mac OS X and Windows

The docker database Let's continue now by installing and configuring the docker database. You will find all details concerning the installation of docker on their official site looking for your OS into the menu Install/Docker Engine.

Once docker is installed, run :

```
sudo docker run -P --name=chill_db chill/database
```

This will download the chill/database image and start a new docker instance with the name *chill_db* and export the postgresql port *5432* on another random local port.

The db will start in your terminal. In another terminal, you can check if the docker database is running and showing the exposed port with the following command:

```
sudo docker ps

>>>> CONTAINER ID   IMAGE           COMMAND                  CREATED        STATUS       PORTS
>>>> 08bbb62bd5e8   chill/database  "/docker-entrypoint.   6 days ago     Up 5 hours   0.0.0.0:3276
```

You can start the container it via:

```
sudo docker start chill_db
>>>> chill_db
```

Note: The commande to stop the docker container is:

```
sudo docker stop chill_db
>>>> chill_db
```

Installation

Chill is installed with composer.

Preparation Before creating your project, make sure that you know the following information :

- how to access to your database: host, port, database name, and your credentials (username and password) ;
- a random string, which will be used to improve entropy in security. Choose anything you want (random character, your father's birthplace, ...).

Note: If you have installed the docker database:

Open a terminal and run

```
sudo docker port chill_db 5432
```

This command will show on which port the docker container is listening, on your localhost. This is the value to be used to fill the field 'database_port' hereafter.

Your information should be:

- database_host: 127.0.0.1
- database_port: result of the command hereabove.
- database_name: postgres
- database_user: postgres
- database_password: postgres
- locale: fr

Create your project Create your Chill project using composer:

```
composer create-project chill-project/standard path/to/your/directory ~1.0
```

Composer will download the standard architecture and ask you a few question about how to configure your project.

- *database_host* : your postgresql server's address
- *database_port* : the port to reach your postgresql server
- *database_name* : the name of your database
- *database_user* : the username to reach your database
- *database_password* : your username's password
- *locale*: the language, as iso code. Until now, only fr is supported
- *secret*: the secret string you prepared (see *Preparation*)

You may accept the default parameters of *debug_toolbar*, *debug_redirects* and *use_assetic_controller* for a demonstration installation. For production, set them all to *false*.

Note: If composer ask you the following question :

```
Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]?
```

You may answer *Y* (Yes), or simply press the *return* button.

Note: At the end of the installation, composer will warn you to execute database migration script, with this message :

```
Some migration files have been imported. You should run
`php app/console doctrine:migrations:status` and/or
`php app/console doctrine:migrations:migrate` to apply them to your DB.
```

We will proceed this step a bit later. See *Create your database schema*.

Check your requirements Move your cursor to the new directory

```
cd path/to/your/directory
```

You should check your installation by running

```
php app/check.php
```

Which will give you information about how your installation fullfill the requirements to running Chill, and give you advices to optimize your installation.

Create your database schema This step will create your table and minimum information into your database. Simply run

```
php app/console doctrine:migrations:migrate
```

SQL queries will be printed into your console.

Populate your database with basic information Once your database schema is ready, if you want to test the application you have the opportunity to populate your database with some basic data. Those are provided through a script and might depends from the bundle you choose to install (see *Install additional bundles*). **This script has not to be launched for a production server** and will erase any existing data. It is meant only for testing the application.

```
php app/console doctrine:fixtures:load
```

Preparing assets You have to dump assets into the web directory. Even if the command should be run by Composer, you may run it manually :

```
php app/console assetic:dump
php app/console assets:install
```

Launch your server If everything was fine, you are able to launch your built-in server :

```
php app/console server:run
```

Your server should now be available at *http://localhost:8000*. Type this address on your browser and you should see the homepage. The default login is 'center a_social' with password 'password'.

Have fun exploring Chill.

1.2 Very quick install with docker

1.2.1 Very quick start with docker

We have created a docker container to let you test *Chill* easily.

Note: We assume docker is already installed on your machine. If Docker is not installed, have a look at the install page in the docker documentation.

Starting the containers

Mac OS X & docker

To use docker on Mac OS X you need to use *boot2docker* or *docker-machine*

Configuration of *boot2docker*

```
$ boot2docker start
$ boot2docker shellinit
```

Configuration of *docker-machine*

```
$ docker-machine start default
$ eval "$(docker-machine env default)"
```

Getting the last version of the docker images

```
docker pull chill/database
docker pull chill/demo-flavor
```

Prepare a database

```
docker run --rm --name=chill_database chill/database
```

The first time you will run this command, the image will be downloaded from docker registry. Please be patient.

Run the container containing the code and attach it to the database

```
docker run --rm --link chill_database:db -p 8989:8000 --name=chill_php chill/demo-flavor
```

The image will also be downloaded from docker registry on first run.

You can then browse on http://localhost:8989 and login with the created users, like *center a_social* (the complete list is below). Password is always 'password'. For Mac OS X, replace *localhost* by the IP of the docker VM (*boot2docker ip* or *docker-machine ip default*).

Stopping the containers

```
docker stop chill_php
docker stop chill_database
```

Limitations

- Those container should not be used in production
- The database should not be persisted or store persistant information: at each container startup, the container's data will be erased and replaced by new (partially) random fixtures

Users created

The passwords are always *password* :

The user's login created are :

- center a_social
- center a_administrative
- center a_direction
- center b_social

- center b_administrative
- center b_direction
- multi_center
- admin

1.3 Usage in production

1.3.1 Install production webserver

Todo

the section “Install production webserver” must be written. Help appreciated :-)

Warning: Some sensitive data (like the person data, ...) might be logged in a special channel, called `chill`. This channel will log events like items removed by a user, what where the details of this item, who removed it, ...
You should take care of encrypting or discarding those data if required.
For an how-to of how to encrypt those data, you may consult the appropriate section of the symfony documentation

1.4 Update Chill and maintenance

1.4.1 Update Chill

Updating the application is very simple, thanks to the use of composer.

```
cd path/to/your/directory

composer update
```

Composer allow to control strictly the versions to be used in the application. This allow to manage a production server, a staging one and a developpement one.

For the production one, define very precisely the versions of each component in the *composer.json* file at the root of the application and require only *stable* versions. In this situation even if an update of composer is launched, the sources will not be modified.

For the staging and development one, let open choices to the versions or accept development versions and using *composer update* will automatically update the selected modules. Once validated at staging level they can be sent in production re-precising the latest accepted versions.

Note: It is advisable to keep the version of composer up to date. To do that, just launch the following command in a terminal.

```
composer self-update
```

If this update create troubles (which doesn’t happen often) you can always undo this update running:

```
composer self-update --rollback
```

See the the composer documentation for all details.

1.4.2 Install additional bundles

A basic installation of Chill include four bundles:

- *Main bundle*
- *Person bundle*
- *Report bundle*
- *Custom fields bundle*

but you can add as many as needed by your project, and if the bundle does not exists yet, you can create a new one, see *Create a new bundle* .

In Chill you are free to do what is most suitable for your activity, so let's go into details on how to add an existing bundle.

We will add the bundle 'icpc2' that set icpc code available as custom field.

Todo

Add description of the bundle

Open your terminal let composer do the magic for you:

```
cd path/to/your/directory
composer require chill-main/icpc2
```

As composer ends its task, it could notify you that *Some migration files have been imported. In this case You should run 'php app/console doctrine:migrations:status* and/or *php app/console doctrine:migrations:migrate* to apply them to your DB.'

So just do it:

```
php app/console doctrine:migrations:migrate
```

Note: The following has to be automated:

Finally we should modify the AppKernel.php file adding *new Chill\Icpc2Bundle\ChillIcpc2Bundle(),* in the $bundle array as described in the symfony documentation.

1.4.3 Uninstall Chill

Todo

the section "Uninstall Chill" must be written. Help appreciated :-)

Uninstall the docker database

Todo

the section "Uninstall the docker database" must be written. Help appreciated :-)

Uninstall the application

Todo

the section "Uninstall the application" must be written. Help appreciated :-)

1.5 Database preparation

1.5.1 Install PostgresSql server

On a linux environment, installing Postgresql server is very easy. Here follows the instructions for a debian based distribution (as Ubuntu) using the distribution repositories. To have the latest version follow the instructions of the Postgresql wiki.

```
sudo apt-get update
sudo apt-get install postgresql
```

Having a look at the install messages, you will guess quickly which is the version installed. Anyway you can check this with the following code:

```
psql --version
>>> psql (PostgreSQL) 9.4.5
```

To be able to add the unaccent extension to your database, you will have to install the following package where x.x are the first 2 parts of the version, 9.4 in this example.

```
sudo apt-get install postgresql-contrib-x.x
```

You are ready to play with Postgressql.

Note: To avoid installation and configuration of a postgresql server, you may use our docker image to start and configure a database as decribed in the basic installation chapter. This solution can be used also in a production environment.

1.5.2 Install PostresSql database

Here follows as an example the instructions that has been used on Ubuntu 14.04 distribution to install the Chill database. Feel free to customize it following your preferences, but do not forget to enable the *unaccent* extension on your database.

```
sudo su
su postgres
# At the prompt of the following instruction, I have typed 'my_terrible_secret' as password
createuser --pwprompt chill_user
createdb -O chill_user chill_db
psql -d chill_db -c "CREATE EXTENSION unaccent;"
```

When you will use composer to install Chill, you will have to provide some database information. If you follow this tutorial these will be:

- database_host: localhost
- database_port: 5432
- database_name: chill_db
- database_user: chill_user

- database_password: my_terrible_secret
- locale: en

Todo

We should write a section on "how to update your installation".

Todo

We should write a section on "what are the concepts of chill" and explain what is a bundle, why we should install it, how to find them, ...

CHAPTER 2

Development

As Chill rely on the symfony framework, reading the framework's documentation should answer most of your questions. We are explaining here some tips to work with Chill, and things we provide to encounter our needs.

2.1 Installation for development

Installation for development should allow:

- access to the source code,
- upload the code to our CVS (i.e. git) and
- work with composer.

As Chill is divided into bundles (the Symfony name for 'modules'), each bundle has his own repository.

2.1.1 Installation and big picture

First, you should install Chill as described in the *Basic installation* section.

Two things must be modified :

First, add the *–prefer-source* argument when you create project.

```
composer create-project chill-project/standard path/to/your/directory --stability=dev --prefer-sou
```

Second, when composer ask you the following question :

```
Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]?
```

You should answer 'n' (no).

Once Chill is installed, all the downloaded bundles will be stored in the */vendor* directories. In those directories, you will have access to the git commands.

```
$ cd vendor/chill-project/main/
$ git remote -v
composer    git://github.com/Chill-project/Standard.git (fetch)
composer    git://github.com/Chill-project/Standard.git (push)
origin      git://github.com/Chill-project/Standard.git (fetch)
origin      git@github.com:Chill-project/Standard.git (push)
```

Files cleaning after installation

Composer will delete unrequired files, and add some. This is perfectly normal and will appears in your git index. But you should NOT delete those files.

This is the expected 'git status' result:

```
$git status
  #(...)

  Modifications qui ne seront pas validées :
    (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
    (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de trava

          modifié:         app/SymfonyRequirements.php
          supprimé:        app/SymfonyStandard/Composer.php
          supprimé:        app/SymfonyStandard/RootPackageInstallSubscriber.php
          modifié:         app/check.php
```

You can ignore the local changes using the *git update-index –assume-unchanged* command.

```
$ git update-index --assume-unchanged app/check.php
$ git update-index --assume-unchanged app/SymfonyRequirements.php
$ git update-index --assume-unchanged app/SymfonyStandard/Composer.php
$ git update-index --assume-unchanged app/SymfonyStandard/RootPackageInstallSuscriber.php
```

Working with your own fork

Ideally, you will work on a fork of the main github repository. To ensure that composer will download the code from **your** repository, you will have to adapt the *composer.json* file accordingly, using your own repositories.

For each Chill module that you have forked, add an indexed array into the "repositories" key of the composer.json file at the root of the chill installation directory if you want to force composer to download from your own forked repositories:

```
"repositories": [
   {
     "type": "git",
     "url": "git://github.com/your-github-username/ChillMain.git"
   },
   {
     "type": "git",
     "url": "git://github.com/your-github-username/Chill-Person.git"
   }
]
```

Then run composer update to load your forked code. If it does not happen, delete the content of the chill/vendor/chill-project/my_forked_bundle and relaunch composer update and the code will be downloaded from your fork.

```
composer update
```

You may also use aliases to define versions.

2.1.2 Editing the code and commiting

You may edit code in the *vendor/path/to/the/bundle* directory.

Once satisfied with your changes, you should commit as usually :

```
$ cd vendor/path/to/bundle
$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

rien à valider, la copie de travail est propre
```

Tips

The command *composer status* (see composer documentation) will give you and idea of which bundle has been edited :

```
$ cd ./../../ #back to the root project directory
$ composer status
You have changes in the following dependencies:
/path/to/your/project/install/vendor/chill-project/main
Use --verbose (-v) to see modified files
```

2.2 Create a new bundle

Create your own bundle is not a trivial task.

The easiest way to achieve this is seems to be :

1. Prepare a fresh installation of the chill project, in a new directory
2. Create a new bundle in this project, in the src directory
3. Initialize a git repository **at the root bundle**, and create your initial commit.
4. Register the bundle with composer/packagist. If you do not plan to distribute your bundle with packagist, you may use a custom repository for achieve this [1]
5. Move to a development installation, made as described in the *Installation for development* section, and add your new repository to the composer.json file
6. Work as *usual*

Warning: This part of the doc is not yet tested

TODO

2.3 Routing

Our goal is to ease the installation of the different bundle. Users should not have to dive into complicated config files to install bundles.

2.3.1 A routing loader available for all bundles

A Chill bundle may rely on the Routing Loader defined in ChillMain.

The loader will load *yml* or *xml* files. You simply have to add them into *chill_main* config

```
chill_main:
# ... other stuff here
   routing:
      resources:
         - @ChillMyBundle/Resources/config/routing.yml
```

[1] Be aware that we use the Affero GPL Licence, which ensure that all users must have access to derivative works done with this software.

Load routes automatically

But this force users to modify config files. To avoid this, you may prepend config implementing the *PrependExtensionInterface* in the *YourBundleExtension* class. This is an example from **chill main** bundle :

```
namespace Chill\MainBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;

class ChillMainExtension extends Extension implements PrependExtensionInterface
{

    public function load(array $configs, ContainerBuilder $container)
    {
        // ...
    }

    public function prepend(ContainerBuilder $container)
    {

        //add current route to chill main
        //this is where the resource is added automatically in the config
        $container->prependExtensionConfig('chill_main', array(
           'routing' => array(
              'resources' => array(
                 '@ChillMainBundle/Resources/config/routing.yml'
              )

           )
        ));
    }
}
```

2.4 Menus

Chill has created his own menu system

See also:

Routes dans Chill [specification] The issue wich discussed the implementation of routes.

2.4.1 Concepts

Warning: to be written

2.4.2 Add a menu in a template

In your twig template, use the *chill_menu* function :

```
{{ chill_menu('person', {
          'layout': 'ChillPersonBundle::menu.html.twig',
          'args' : {'id': person.id },
          'activeRouteKey': 'chill_person_view'
     }) }}
```

The available arguments are:

- *layout* : a custom layout. Default to *ChillMainBundle:Menu:defaultMenu.html.twig*
- *args* : those arguments will be passed through the url generator.
- *activeRouteKey* must be the route key name.

Note: The argument *activeRouteKey* may be a twig variable, defined elsewhere in your template, even in child templates.

2.4.3 Create an entry in an existing menu

If a route belongs to a menu, you simply add this to his definition in routing.yml :

```
chill_person_history_list:
    pattern: /person/{person_id}/history
    defaults: { _controller: ChillPersonBundle:History:list }
    options:
        #declare menus
        menus:
            # the route should be in 'person' menu :
            person:
                #and have those arguments :
                order: 100
                label: menu.person.history
```

- *order* (mandatory) : the order in the menu. It is preferrable to increment by far more than 1.
- *label* (mandatory) : a translatable string.
- *helper* (optional) : a text to help people to understand what does the menu do. Not used in default implementation.
- *condition* (optional) : an ‘Expression Language <http://symfony.com/doc/current/components/expression_language/index.htm ‘_ which will make the menu appears or not. Typically, it may be used to say “show this menu only if the person concerned is more than 18”. **Not implemented yet**.
- *access* (optional) : an Expression Language to evalute the possibility, for the user, to show this menu according to Access Control Model. **Not implemented yet.**

You may add additional keys, but should not use the keys described above.

You may add the same route to multiple menus :

```
chill_person_history_list:
    pattern: /person/{person_id}/history
    defaults: { _controller: ChillPersonBundle:History:list }
    options:
        menus:
            menu1:
                order: 100
                label: menu.person.history
            menu2:
                order: 100
                label: another.label
```

2.4.4 Customize menu rendering

You may customize menu rendering by using the *layout* option.

Warning: TODO : this part should be written.

2.4.5 Caveats

Currently, you may pass arguments globally to each menu, and they will be all passed to route url. This means that :

- the argument name in the route entry must match the argument key in menu declaration in twig template
- if an argument is missing to generate an url, the url generator will throw a *SymfonyComponentRoutingExceptionMissingMandatoryParametersException*
- if the argument name is not declared in route entry, it will be added to the url, (example: */my/route?additional=foo*)

2.5 Access controle model

Table of content

2.5.1 Concepts

Every time an entity is created, viewed or updated, the software check if the user has the permission to make this action. The decision is made with three parameters :

- the type of entity ;
- the entity's center ;
- the entity'scope

The user must be granted access to the action on this particular entity, with this scope and center.

From an user point of view

The software is design to allow fine tuned access rights for complicated installation and team structure. The administrators may also decide that every user has the right to see all resources, where team have a more simple structure.

Here is an overview of the model.

Chill can be multi-center

Chill is designed to be installed once for social center who work with multiple teams separated, or for social services's federation who would like to share the same installation of the software for all their members.

This was required for cost reduction, but also to ease the generation of statistics agregated across federation's members, or from the central unit of the social center with multiple teams.

Otherwise, it is not required to create multiple center: Chill can also work for one center.

Obviously, users working in the different centers are not allowed to see the entities (_persons_, _reports_, _activities_) of other centers. But users may be attached to multiple centers: consequently they will be able to see the entities of the multiple centers they are attached to.

Inside center, scope divide team

Users are attached to one or more center and, inside to those center, there may exists differents scopes. The aim of those _scopes_ is to divide the whole team of social worker amongst different departement, for instance: the social team, the psychologist team, the nurse team, the administrative team, ... Each team is granted of different rights amongst scope. For instance, the social team may not see the _activities_ of the psychologist team. The administrative team may see the date & time's activities, but is not allowed to see the detail of those entities (the personal notes, ...).

The administrator is responsible of creating those scopes and team. *He may also decide to not define a division inside his team*: he creates only one scope and all entities will belong to this scope, all users will be able to see all entities.

As entities have only one scopes, if some entities must be shared across two different teams, the administrator will have to create a scope *shared* by two different team, and add the ability to create, view, or update this scope to those team.

Example: if some activities must be seen and updated between nurses and psychologists, the administrator will create a scope "nurse and psy" and add the ability for both team "nurse" and "psychologist" to "create", "see", and "update" the activities belonging to scope "nurse and psy".

The concepts translated into code

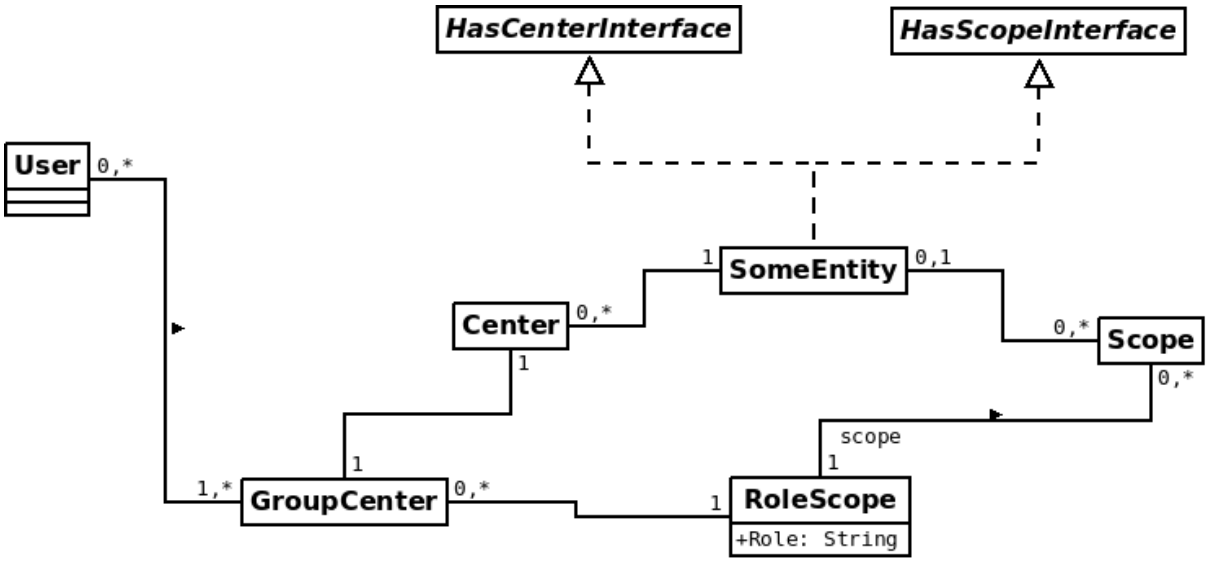


Fig. 2.1: Schema of the access control model

Chill handle **entities**, like *persons*, *reports* (associated to *persons*), *activities* (also associated to *_persons*), ... On creation, those entities are linked to one center and, eventually, to one scope. They implements the interface *HasCenterInterface*.

Note: Somes entities are linked to a center through the entity they are associated with. For instance, *activities* or *reports* are associated to a *person*, and the person is associated to a *center*. The *report*'s *center* is always the *person*'s *center*.

Entities may be associated with a scope. In this case, they implement the *HasScopeInterface*.

Note: Currently, only the *person* entity is not associated with a scope.

At each step of his lifetime (creation, view of the entity and eventually of his details, update and, eventually, deletion), the right of the user are checked. To decide wether the user is granted right to execute the action, the software must decide with those elements :

- the entity's type;
- the entity's center ;
- the entity's scope, if it exists,
- and, obviously, a string representing the action

All those action are executed through symfony voters and helpers.

How to check authorization ?

Just use the symfony way-of-doing, but do not forget to associate the entity you want to check access. For instance, in controller :

```
class MyController extends Controller
{

   public function viewAction($entity)
   {
      $this->denyAccessUnlessGranted('CHILL_ENTITY_SEE', $entity);

      //... go on with this action
   }
}
```

And in template :

```
{{ if is_granted('CHILL_ENTITY_SEE', entity) %}print something{% endif %}
```

Retrieving reachable scopes and centers

The class `AuthorizationHelper` helps you to get centers and scope reachable by a user.

Those methods are intentionnaly build to give information about user rights:

- getReachableCenters: to get reachable centers for a user
- getReachableScopes : to get reachable scopes for a user

Note: The service is reachable through the Depedency injection with the key *chill.main.security.authorization.helper*. Example :

```
$helper = $container->get('chill.main.security.authorization.helper');
```

Todo

Waiting for a link between our api and this doc, we invite you to read the method signatures here

2.5.2 Adding your own roles

Extending Chill will requires you to define your own roles and rules for your entities. You will have to define your own voter to do so.

To create your own roles, you should:

- implement your own voter. This voter will have to extends the `AbstractChillVoter`. As defined by Symfony, this voter must be declared as a service and tagged with *security.voter*;
- declare the role through implementing a service tagged with *chill.role* and implementing `ProvideRoleInterface`.

Note: Both operation may be done through a simple class: you can implements `ProvideRoleInterface` and `AbstractChillVoter` on the same class. See live example: `ActivityVoter`, and similar examples in the *PersonBundle* and *ReportBundle*.

See also:

How to Use Voters to Check User Permissions

From the symfony cookbook

New in Symfony 2.6: Simpler Security Voters

From the symfony blog

Declare your role

To declare new role, implement the class `ProvideRoleInterface`.

```
interface ProvideRoleInterface
{
    /**
     * return an array of role provided by the object
     *
     * @return string[] array of roles (as string)
     */
    public function getRoles();

    /**
     * return roles which doesn't need
     *
     * @return string[] array of roles without scopes
     */
    public function getRolesWithoutScope();
}
```

Then declare your service with a tag *chill.role*. Example :

```
your_service:
    class: Chill\YourBundle\Security\Authorization\YourVoter
    tags:
        - { name: chill.role }
```

Example of an implementation of `ProvideRoleInterface`:

```
namespace Chill\PersonBundle\Security\Authorization;

use Chill\MainBundle\Security\ProvideRoleInterface;

class PersonVoter implements ProvideRoleInterface
{
```

```
    const CREATE = 'CHILL_PERSON_CREATE';
    const UPDATE = 'CHILL_PERSON_UPDATE';
    const SEE    = 'CHILL_PERSON_SEE';

    public function getRoles()
    {
        return array(self::CREATE, self::UPDATE, self::SEE);
    }

    public function getRolesWithoutScope()
    {
        return array(self::CREATE, self::UPDATE, self::SEE);
    }

}
```

Implement your voter

Inside this class, you might use the `AuthorizationHelper` to check permission (do not re-invent the wheel). This is a real-world example:

```
namespace Chill\ReportBundle\Security\Authorization;
use Chill\MainBundle\Security\Authorization\AbstractChillVoter;
use Chill\MainBundle\Security\Authorization\AuthorizationHelper;


class ReportVoter extends AbstractChillVoter
{
    const CREATE = 'CHILL_REPORT_CREATE';
    const SEE    = 'CHILL_REPORT_SEE';
    const UPDATE = 'CHILL_REPORT_UPDATE';

    /**
     *
     * @var AuthorizationHelper
     */
    protected $helper;

    public function __construct(AuthorizationHelper $helper)
    {
        $this->helper = $helper;
    }

    protected function getSupportedAttributes()
    {
        return array(self::CREATE, self::SEE, self::UPDATE);
    }
    protected function getSupportedClasses()
    {
        return array('Chill\ReportBundle\Entity\Report');
    }
    protected function isGranted($attribute, $report, $user = null)
    {
        if (! $user instanceof \Chill\MainBundle\Entity\User){

            return false;
        }

        return $this->helper->userHasAccess($user, $report, $attribute);
    }
}
```

Then, you will have to declare the service and tag it as a voter :

```
services:
    chill.report.security.authorization.report_voter:
        class: Chill\ReportBundle\Security\Authorization\ReportVoter
        arguments:
            - "@chill.main.security.authorization.helper"
        tags:
         - { name: security.voter }
```

Adding role hierarchy

You should prepend Symfony's security component directly from your code.

```
namespace Chill\ReportBundle\DependencyInjection;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
use Chill\MainBundle\DependencyInjection\MissingBundleException;

/**
 * This is the class that loads and manages your bundle configuration
 *
 * To learn more see {@link http://symfony.com/doc/current/cookbook/bundles/extension.html}
 */
class ChillReportExtension extends Extension implements PrependExtensionInterface
{
    public function prepend(ContainerBuilder $container)
    {
        $this->prependRoleHierarchy($container);
    }

    protected function prependRoleHierarchy(ContainerBuilder $container)
    {
        $container->prependExtensionConfig('security', array(
            'role_hierarchy' => array(
                'CHILL_REPORT_UPDATE' => array('CHILL_REPORT_SEE'),
                'CHILL_REPORT_CREATE' => array('CHILL_REPORT_SEE')
            )
        ));
    }
}
```

2.6 Messages to users

2.6.1 Flashbags

The four following levels are defined :

Key	Intent
alert	A message not linked with the user action, but which should require an action or a correction.
success	The user action succeeds.
notice	A simple message to give information to the user. The message may be linked or not linked with the user action.
warning	A message linked with an action, the user should correct.
error	The user's action failed: he must correct something to process the action.

See also:

Flash Messages on Symfony documentation Learn how to use flash messages in controller.

2.6.2 Buttons

Four actions are available to decorate *a* links and *buttons*.

To add the action on button, use them as class along with *sc-button* :

```
<a class="sc-button bt-create">Create an entity</a>

<button class="sc-button bt-submit" type="submit">Submit</button>
```

Action	Class	Description
Submit	*bt-submit*	Submit a form.
Create	*bt-create*	Link to a form to create an entity
Reset	*bt-reset*	Reset a form
Delete	*bt-delete*	Link to a form to delete an entity
Edit	*bt-edit*	Link to a form to edit an entity
Update	*bt-update*	Submitting this form will update the entity
Action	*bt-action*	Generic link to an action

2.7 Pagination

The Bundle `Chill\MainBundle` provides a **Pagination** api which allow you to easily divide results list on different pages.

2.7.1 A simple example

In the controller, get the `ChillMainPaginationPaginatorFactory` from the *Container* and use this `PaginatorFactory` to create a `Paginator` instance.

```
<?php

namespace Chill\MyBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;


class ItemController extends Controller {

    public function yourAction()
    {
        $em = $this->getDoctrine()->getManager();
        // first, get the number of total item are available
        $total = $em
```

```
            ->createQuery("COUNT (item.id) FROM ChillMyBundle:Item item")
            ->getSingleScalarResult();

        // get the PaginatorFactory
        $paginatorFactory = $this->get('chill_main.paginator_factory');

        // create a pagination instance. This instance is only valid for
        // the current route and parameters
        $paginator = $paginatorFactory->create($total);

        // launch your query on item. Limit the query to the results
        // for the current page using the paginator
        $items = $em->createQuery("SELECT item FROM ChillMyBundle:Item item WHERE <your clause>")
            // use the paginator to get the first item number
            ->setFirstResult($paginator->getCurrentPage()->getFirstItemNumber())
            // use the paginator to get the number of items to display
            ->setMaxResults($paginator->getItemsPerPage());

        return $this->render('ChillMyBundle:Item:list.html.twig', array(
                'items' => $items,
                'paginator' => $paginator
                );

    }

}
```

Then, render the pagination using the dedicated twig function.

```
{% extends "ChillPersonBundle::layout.html.twig"  %}

{% block title 'Item list'|trans %}

{% block personcontent %}

<table>

{# ... your items here... #}

</table>

{% if items|length > paginator.getTotalItems %}
{{ chill_pagination(paginator) }}
{% endif %}
```

The function `chill_pagination` will, by default, render a link to the 10 previous page (if they exists) and the 10 next pages (if they exists). Assuming that we are on page 5, the function will render a list to

```
Previous 1 2 3 4 **5** 6 7 8 9 10 11 12 13 14 Next
```

Understanding the magic

Where does the `$paginator` get the page number ?

Internally, the `$paginator` object has a link to the `Request` object, and it reads the `page` parameter which contains the current page number. If this parameter is not present, the `$paginator` assumes that we are on page 1.

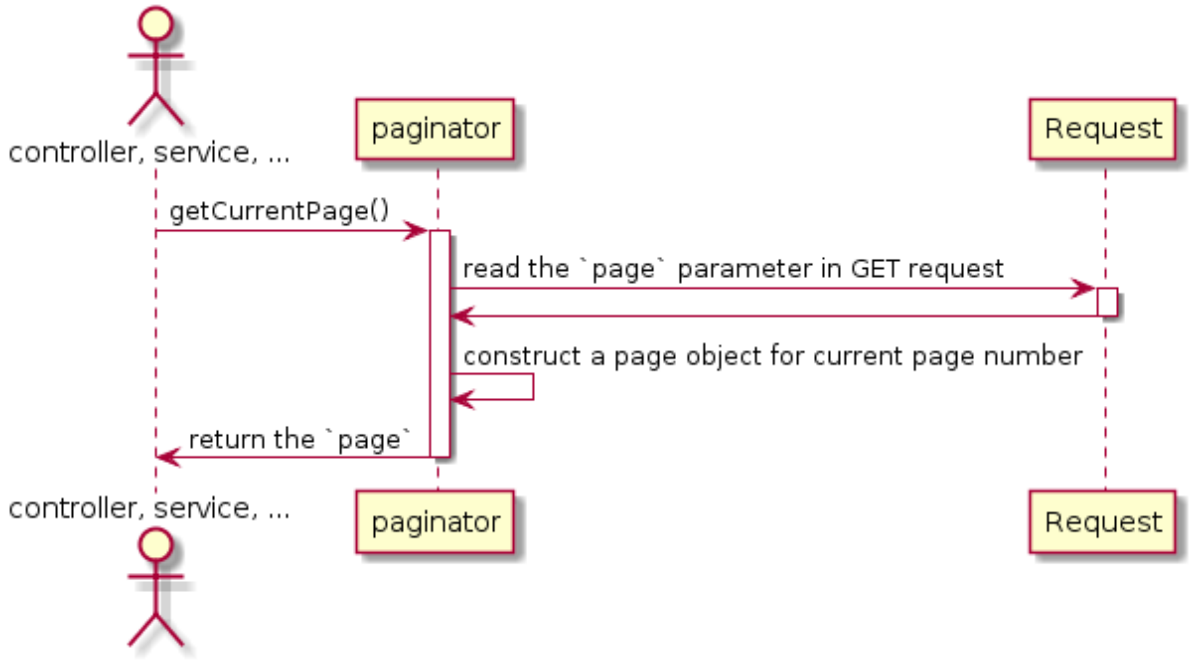


Fig. 2.2: The `$paginator` get the current page from the request.

Where does the `$paginator` get the number of items per page ?

As above, the `$paginator` can get the number of items per page from the `Request`. If none is provided, this is given by the configuration which is, by default, 50 items per page.

2.7.2 `PaginatorFactory`, `Paginator` and `Page`

`PaginatorFactory`

The `PaginatorFactory` may create more than one `Paginator` in a single action. Those `Paginator` instance may redirect to different routes and/or routes parameters.

```
// create a paginator for the route 'my_route' with some parameters (arg1 and arg2)
$paginatorMyRoute = $paginatorFactory->create($total, 'my_route', array('arg1' => 'foo', 'arg2' =
```

Those parameters will override the current parameters.

The `PaginatorFactory` has also some useful shortcuts :

```
// get current page number
$paginatorFactory->getCurrentPageNumber( )
// get the number of items per page **for the current request**
$paginatorFactory->getCurrentItemsPerPage( )
// get the number of the first item **for the current page**
$paginatorFactory->getCurrentPageFirstItemNumber( )
```

Working with `Paginator` and `Page`

The paginator has some function to give the number of pages are required to displayed all the results, and give some information about the number of items per page :

```
// how many page count this paginator ?
$paginator->countPages(); // return 20 in our example
```

```
// we may get the number of items per page
$paginator->getItemsPerPage(); // return 20 in our example
```

A `Paginator` instance create instance of `Page`, each `Page`, which is responsible for generating the URL to the page number it represents. Here are some possibilities using `Page` and `Paginator` :

```
// get the current page
$page = $paginator->getCurrentPage();
// on which page are we ?
$page->getNumber(); // return 5 in this example (we are on page 5)
// generate the url for page 5
$page->generateUrl(); // return '/<your route here>?page=5
// what is the first item number on this page ?
$page->getFistItemNumber(); // return 101 in our example (20 items per page)
// what is the last item number on this page ?
$page->getLastItemNumber(); // return 120 in our example

// we can access directly the next and current page
if ($paginator->hasNextPage()) {
    $next = $paginator->getNextPage();
}
if ($paginator->hasPreviousPage()) {
    $previous = $paginator->getPreviousPage();
}

// we can access directly to a given page number
if ($paginator->hasPage(10)) {
   $page10 = $paginator->getPage(10);
}

// we can iterate over our pages through a generator
foreach ($paginator->getPagesGenerator() as $page) {
   $page->getNumber();
}

// check that a page object is the current page
$paginator->isCurrentPage($page); // return false
```

Warning: When calling a page which does not exists, the `Paginator` will throw a *RuntimeException*. Example :

```
// our last page is 10
$paginator->getPage(99); // out of range => throw `RuntimeException`

// our current page is 1 (the first page)
$paginator->getPreviousPage; // does not exists (the fist page is always 1) => throw `RuntimeExc
```

Note: When you create a `Paginator` for the current route and route parameters, the `Page` instances will keep the same parameters and routes :

```
// assuming our route is 'my_route', for the pattern '/my/{foo}/route',
// and the current route is '/my/value/route?arg2=bar'

// create a paginator for the current route and route parameters :
$paginator = $paginatorFactory->create($total);

// get the next page
if ($paginator->hasNext()) {
   $next = $paginator->getNextPage();

   // get the route to the page
```

```
    $page->generateUrl(); // will print 'my/value/route?arg2=bar&page=2'
}
```

Having a look to the full classes documentation may provide some useful information.

2.7.3 Customizing the rendering of twig's `chill_pagination`

You can provide your own layout for rendering the pagination: provides your twig template as a second argument :

```
{{ chill_pagination(paginator, 'MyBundle:Pagination:MyTemplate.html.twig') }}
```

The template will receive the `$paginator` as `paginator` variable. Let's have a look at the current template.

2.8 Localisation

2.8.1 Language in url

Language should be present in URL, conventionnaly as first argument.

```
/fr/your/url/here
```

This allow users to change from one language to another one on each page, which may be useful in multilanguages teams. If the installation is single-language, the language switcher will not appears.

This is an example of routing defined in yaml :

```
chill_person_general_edit:
    pattern: /{_locale}/person/{person_id}/general/edit
    defaults: {_controller: ChillPersonBundle:Person:edit }
```

2.8.2 Date and time

The Intl extension is enabled on the Chill application.

You may format date and time using the *localizeddate* function :

```
date|localizeddate('long', 'none')
```

By default, we prefer using the *long* format for date formatting.

See also:

Documentation for Intl Extension Read the complete doc for the Intl extension.

2.9 Logging

See also:

Symfony documentation: **How to user Monolog to write logs** The symfony cookbook page about logging.

A channel for custom logging has been created to store sensitive data.

The channel is named `chill`.

The installer of chill should be aware that this channel may contains sensitive data and encrypted during backup.

2.9.1 Logging to channel *chill*

You should use the service named `chill.main.logger`, as this :

```
$logger = $this->get('chill.main.logger');
```

You should store data into context, not in the log himself, which should remains the same for the action.

Example of usage :

```
$logger->info("An action has been performed about a person", array(
   'person_lastname' => $person->getLastName(),
   'person_firstname' => $person->getFirstName(),
   'person_id' => $person->getId(),
   'by_user' => $user->getUsername()
));
```

For further processing, it is a good idea to separate all fields (like firstname, lastname, ...) into different context keys.

By convention, you should store the username of the user performing the action under the `by_user` key.

2.10 Database Migrations

Every bundle potentially brings his own database operations : to persist entities, developers have to create database schema, creating indexes,...

Those schema might be changed (the less is the better) from time to time.

Consequence: each bundle should bring his own migration files, which will bring the database consistent with the operation you will run in your code. They will be gathered into the app installation, ready to be executed by the chill's installer.

Currently, we use doctrine migration to manage those migration files. A composer script located in the **chill standard** component will copy the migrations from your bundle to the doctrne migration's excepted directory after each install and/or update operation.

See also:

The doctrine migration documentation Learn concepts about migrations files and scripts and the doctrine ORM

The doctrine migration bundle documentation Learn about doctrine migration integration with Symfony framework

2.10.1 Shipping migration files

Migrations files should be shipped under the Resource/migrations directory. You could customize the migration directory by adding extra information in your composer.json:

```
"extra": {
        "migration-source": "path/to/my/dir"
    }
```

The class namespace should be *ApplicationMigrations*, as expected by doctrine migration. Only the files which will be executed by doctrine migration will be moved: they must have the pattern *VersionYYYYMMDDHHMMSS.php* where YYYY is the year, MM the month, DD the day, HH the hour, MM the month and SS the second of creation.

They will be moved automatically by composer when you install or update a bundle.

2.10.2 Executing migration files

The installers will have to execute migrations files manually, running

```
php app/console doctrine:migrations:status #will give the current status of the database
php app/console doctrine:migrations:migrate #process the update
```

2.10.3 Updating migration files

Warning: After an installation, migration files will be executed and registered as executed in the database (the version timestamp is recorded into the *migrations_versions* table). If you update your migration file code, the file will still be considered as "executed" by doctrine migration, which will not offers the possibility to run the migration again.
Consequently, updating migration file should only be considered during development phase, and not published on public git branches. If you want to edit your database schema, you should create a new migration file, with a new timestamp, which will proceed to your schema adaptations.

Every time a migration file is discovered, the composer'script will check if the migration exists in the local migration directory. If yes, the script will compare two file for changes (using a md5 hash). If migrations are discovered, the script will ask the installer to know if he must replace the file or ignore it.

Note: You can manually run composer script by launching *composer run-script post-update-cmd* from your root chill installation's directory.

2.10.4 Tips for development

Migration and data

Each time you create a migration script, you should ensure that it will not lead to data losing. Eventually, feel free to use intermediate steps.

Generation

You can generate migration file from the command line, using those commands:

- *php app/console doctrine:migrations:diff* to generate a migration file by comparing your current database to your mapping information
- *php app/console doctrine:migrations:generate* to generate a blank migration file.

Those files will be located into *app/DoctrineMigrations* directory. You will have to copy those file to your the directory *Resource/migrations* into your bundle directory.

Comments and documentation

As files are copied from your bundle to the *app/DoctrineMigrations* directory, the link between your bundle and the copied file will be unclear. Please add all relevant documentation which will allow future developers to make a link between your file and your bundle.

Inside the script

The script which move the migrations files to app directory might be found here.

2.11 Searching

Chill should provide information needed by users when they need it. Searching within bundle, entities,... is an important feature to achieve this goal.

The Main Bundle provide interfaces to ease the developer work. It will also attempt that search will work in the same way accross bundles.

Table of content

See also:

Our blog post about searching (in French) This blog post give some information for end-users about searching.

The issue about search behaviour Where the search behaviour is defined.

2.11.1 Searching in a glance for developers

Chill suggests to use an easy-to-learn language search.

Note: We are planning to provide a form to create automatically search pattern according to this language. Watch the issue regarding this feature.

The language is an association of search terms. Search terms may contains :

- **a domain**: this is "the domain you want to search" : it may some entities like people, reports, ... Example : *@person* to search accross people, *@report* to browse reports, ... The search pattern may have **a maximum of one** domain by search, providing more should throw an error, and trigger a warning for users.
- **arguments and their values** : This is "what you search". Arguments narrow the search to specific fields : username, date of birth, nationality, ... The syntax is *argument:value*. I.e.: ' birthdate:2014-12-15', *firstname:Depardieu*, ... **Arguments are optional**. If the value of an argument contains spaces or characters like punctuation, quotes ("), the value should be provided between parenthesis : *firstname:(Van de snoeck)*, *firstname:(M'bola)*, ...
- **default value** : this the "rest" of the search, not linked with any arguments or domain. Example : *@person dep* (*dep* is the "default value"), or simply *dep* if any domain is provided (which is perfectly acceptable). If a string is not idenfied as argument or domain, it will be present in the "default" term.

If a search pattern (provided by the user) does not contains any domain, the search must be run across default domain/search modules.

A domain may be supported by different search modules. For instance, if you provide the domain *@person*, the end-user may receive results of exact firstname/lastname, but also result with spelling suggestion, ... **But** if results

do not fit into the first page (if you have 75 results and the screen show only 50 results), the next page should contains only the results from the required module.

For instance : a user search across people by firstname/lastname, the exact spelling contains 10 results, the "spelling suggestion" results contains 75 names, but show only the first 50. If the user want to see the last 25, the next screen should not contains the results by firstname/lastname.

2.11.2 Allowed characters as arguments

In order to execute regular expression, the allowed chararcters in arguments are a-z characters, numbers, and the sign '-'. Spaces and special characters like accents are note allowed (the accents are removed during parsing).

2.11.3 Special characters and uppercase

The search should not care about lowercase/uppercase and accentued characters. Currently, they are removed automatically by the *chill.main.search_provider*.

2.11.4 Implementing search module for dev

To implement a search module, you should :

- create a class which implements the *ChillMainBundleSearchSearchInterface* class. An abstract class *ChillMainBundleSearchAbstractSearch* will provide useful assertions for parsing date string to *DateTime* objects, ...
- register the class as a service, and tag the service with *chill.search* and an appropriate alias

The search logic is provided under the */search* route.

See also:

The implementation of a search module in Person bundle An example of implementationhttps://github.com/Chill-project/Main/blob/master/DependencyInjection/SearchableServicesCompilerPass.php

---

Note: Internals explained : the services tagged with *chill.search* are gathered into the *chill.main.search_provider* service during compilation (see the compiler pass).

The *chill.main.search_provider* service allow to :

- retrieve all results (as html string) for all search module concerned by the search (according to the domain provided or modules marked as default)
- retrieve result for one search module

The SearchInterface class

```
namespace Chill\PersonBundle\Search;

use Chill\MainBundle\Search\AbstractSearch;
use Doctrine\ORM\EntityManagerInterface;
use Chill\PersonBundle\Entity\Person;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\DependencyInjection\ContainerAware;
use Symfony\Component\DependencyInjection\ContainerAwareTrait;
use Chill\MainBundle\Search\ParsingException;

class PersonSearch extends AbstractSearch
{
```

```
    // indicate which domain you support
    // you may respond TRUE to multiple domain, according to your logic
    public function supports($domain)
    {
        return 'person' === $domain;
    }

    // if your domain must be called when no domain is provided, should return true
    public function isActiveByDefault()
    {
        return true;
    }

    // if multiple module respond to the same domain, indicate an order for your search.
    public function getOrder()
    {
        return 100;
    }


    // This is where your search logic should be executed.
    // This method must return an HTML string (a string with HTML tags)
    // see below about the structure of the $term array
    public function renderResult(array $terms, $start = 0, $limit = 50, array $options = array())
    {
        return $this->container->get('templating')->render('ChillPersonBundle:Person:list.html.twi
                array(
                    // you should implements the `search` function somewhere :-)
                    'persons' => $this->search($terms, $start, $limit, $options),
                    // recomposePattern is available in AbstractSearch class
                    'pattern' => $this->recomposePattern($terms, array('nationality',
                        'firstname', 'lastname', 'birthdate', 'gender'), $terms['_domain']),
                    // you should implement the `count` function somewhere :-)
                    'total' => $this->count($terms)
                ));
    }
}
```

Structure of array *$term*

The array term is parsed automatically by the *main.chill.search_provider* service.

Note: If you need to parse a search pattern, you may use the function *parse($pattern)* provided by the service.

The array *$term* have the following structure after parsing :

```
array(
   '_domain' => 'person', //the domain, without the '@'
   'argument1' => 'value', //the argument1, with his value
   'argument2' => 'my value with spaces', //the argument2
   '_default' => 'abcde ef' // the default term
);
```

The original search would have been : *@person argument1:value argument2:(my value with spaces) abcde ef*

Warning: The search values are always unaccented.

Register the service

You should add your service in the configuration, and add a *chill.search* tag and an alias.

Example :

```
services:
   chill.person.search_person:
      class: Chill\PersonBundle\Search\PersonSearch
      #your logic here
      tags:
         - { name: chill.search, alias: 'person_regular' }
```

The alias will be used to get the results narrowed to this search module, in case of pagination (see above).

2.11.5 Parsing date

The class *ChillMainBundleSearchAbstractSearch* provides a method to parse date :

```
//from subclasses
$date = $this->parseDate($string);
```

$date will be an instance of DateTime.

See also:

The possibility to add periods instead of date Which may be a future improvement for search with date.

2.11.6 Exceptions

The logic of the search is handled by the controller for the */search* path.

You should throw those Exception from your instance of *SearchInterface* if needed :

ChillMainBundleSearchParsingException If the terms does not fit your search logic (for instance, conflicting terms)

2.11.7 Expected behaviour

Operators between multiple terms

Multiple terms should be considered are "AND" instructions :

@person nationality:RU firstname:dep the people having the Russian nationality AND having DEP in their name

@person birthdate:2015-12-12 charles the people having 'charles' in their name or firstname AND born on December 12 2015

Spaces in default

Spaces in default terms should be considered as "AND" instruction

@person charle dep people having "dep" AND "charles" in their firstname or lastname. Match "Charles Depardieu" but not "Gérard Depardieu" ('charle' is not present)

Rendering

The rendering should contains :

- the total number of results ;
- the search pattern in the search language. The aim of this is to let users learn the search language easily.
- a title

2.11.8 Frequently Asked Questions (FAQ)

Why renderResults returns an HTML string and not structured array ? It seems that the form of results may vary (according to access-right logic, ...) and is not easily structurable

2.12 Timelines

Table of content

2.12.1 Concept

From an user point of view

Chill has two objectives :

- make the administrative tasks more lightweight ;
- help social workers to have all information they need to work

To reach this second objective, Chill provides a special view: **timeline**. On a timeline view, information is gathered and shown on a single page, from the most recent event to the oldest one.

The information gathered is linked to a *context*. This *context* may be, for instance :

- a person : events linked to this person are shown on the page ;
- a center: events linked to a center are shown. They may concern different peoples ;
- ...

In other word, the *context* is the kind of argument that will be used in the event's query.

Let us recall that only the data the user has allowed to see should be shown.

See also:

The issue where the subject was first discussed

For developers

The *Main* bundle provides interfaces and services to help to build timelines.

If a bundle wants to *push* information in a timeline, it should be create a service which implements *ChillMainBundleTimelineTimelineProviderInterface*, and tag is with *chill.timeline* and arguments defining the supported context (you may use multiple *chill.timeline* tags in order to support multiple context with a single service/class).

If a bundle wants to provide a new context for a timeline, the service *chill.main.timeline_builder* will helps to gather timeline's services supporting the defined context, and run queries across the models.

Understanding queries

Due to the fact that timelines should show only the X last events from Y differents tables, queries for a timeline may consume a lot of resources: at first on the database, and then on the ORM part, which will have to deserialize DB data to PHP classes, which may not be used if they are not part of the "last X events".

To avoid such load on database, the objects are queried in two steps :

1. An UNION request which gather the last X events, ordered by date. The data retrieved are the ID, the date, and a string key: a type. This type discriminates the data type.
2. The PHP objects are queried by ID, the type helps the program to link id with the kind of objects.

Those methods should ensure that only X PHP objects will be gathered and build by the ORM.

What does the master timeline builder service ?

When the service *chill.main.timeline_builder* is instanciated, the service is informed of each service taggued with *chill.timeline* tags. Then,

1. The service build an UNION query by assembling column and tables names provided by the *fetchQuery* result ;
2. The UNION query is run, the result contains an id and a type for each row (see *above*)
3. The master service gather all id with the same type. Then he searches for the *chill.timeline*'s service which will be able to get the entities. Then, the entities will be fetched using the *fetchEntities* function. All entities are gathered in one query ;
4. The information to render entities in HTML is gathered by passing entity, one by one, on *getEntityTemplate* function.

2.12.2 Pushing events to a timeline

To push events on a timeline :

1. Create a class which implements *ChillMainBundleTimelineTimelineProviderInterface* ;
2. Define the class as a service, and tag the service with *chill.timeline*, and define the context associated with this timeline (you may add multiple tags for different contexts).

Implementing the TimelineProviderInterface

The has the following signature :

```
namespace Chill\MainBundle\Timeline;

interface TimelineProviderInterface
{
```

```php
    /**
     *
     * @param string $context
     * @param mixed[] $args the argument to the context.
     * @return string[]
     * @throw  \LogicException if the context is not supported
     */
    public function fetchQuery($context, array $args);

    /**
     * Indicate if the result type may be handled by the service
     *
     * @param string $type the key present in the SELECT query
     * @return boolean
     */
    public function supportsType($type);

    /**
     * fetch entities from db into an associative array. The keys **MUST BE**
     * the id
     *
     * All ids returned by all SELECT queries
     * (@see TimeLineProviderInterface::fetchQuery) and with the type
     * supported by the provider (@see TimelineProviderInterface::supportsType)
     * will be passed as argument.
     *
     * @param array $ids an array of id
     * @return mixed[] an associative array of entities, with id as key
     */
    public function getEntities(array $ids);

    /**
     * return an associative array with argument to render the entity
     * in an html template, which will be included in the timeline page
     *
     * The result must have the following key :
     *
     * - `template` : the template FQDN
     * - `template_data`: the data required by the template
     *
     *
     * Example:
     *
     * ```
     * array(
     *    'template'      => 'ChillMyBundle:timeline:template.html.twig',
     *    'template_data' => array(
     *              'accompanyingPeriod' => $entity,
     *               'person' => $args['person']
     *         )
     *    );
     * ```
     *
     * `$context` and `$args` are defined by the bundle which will call the timeline
     * rendering.
     *
     * @param type $entity
     * @param type $context
     * @param array $args
     * @return mixed[]
     * @throws \LogicException if the context is not supported
     */
    public function getEntityTemplate($entity, $context, array $args);
```

```
}
```

The *fetchQuery* function

The fetchQuery function help to build the UNION query to gather events. This function should return an associative array MUST have the following key : * *id* : the name of the id column * *type*: a string to indicate the type * *date*: the name of the datetime column, used to order entities by date * *FROM* (in capital) : the FROM clause. May contains JOIN instructions

Those key are optional: * *WHERE* (in capital) : the WHERE clause.

> Where relevant, the data must be quoted to avoid SQL injection.

$context and *$args* are defined by the bundle which will call the timeline rendering. You may use them to build a different query depending on this context.

For instance, if the context is *'person'*, the args will be this array :

```
array(
    'person' => $person //a \Chill\PersonBundle\Entity\Person entity
);
```

You should find in the bundle documentation which contexts are arguments the bundle defines.

Note: We encourage to use *ClassMetaData* to define column names arguments. If you change your column names, changes will be reflected automatically during the execution of your code.

Example of an implementation :

```
namespace Chill\ReportBundle\Timeline;

use Chill\MainBundle\Timeline\TimelineProviderInterface;
use Doctrine\ORM\EntityManager;

/**
 * Provide report for inclusion in timeline
 *
 * @author Julien Fastré <julien.fastre@champs-libres.coop>
 * @author Champs Libres <info@champs-libres.coop>
 */
class TimelineReportProvider implements TimelineProviderInterface
{

    /**
     *
     * @var EntityManager
     */
    protected $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    public function fetchQuery($context, array $args)
        {
            $this->checkContext($context);

            $metadata = $this->em->getClassMetadata('ChillReportBundle:Report');

            return array(
```

```
                'id' => $metadata->getColumnName('id'),
                'type' => 'report',
                'date' => $metadata->getColumnName('date'),
                'FROM' => $metadata->getTableName(),
                'WHERE' => sprintf('%s = %d',
                      $metadata
                         ->getAssociationMapping('person')['joinColumns'][0]['name'],
                      $args['person']->getId())
            );
        }

    //....


}
```

The *supportsType* function

This function indicate to the master *chill.main.timeline_builder* service (which orchestrate the build of UNION queries) that the service supports the type indicated in the result's array of the *fetchQuery* function.

The implementation of our previous example will be :

```
namespace Chill\ReportBundle\Timeline;

use Chill\MainBundle\Timeline\TimelineProviderInterface;
use Doctrine\ORM\EntityManager;

class TimelineReportProvider implements TimelineProviderInterface
{

    //...

    /**
     *
     * {@inheritDoc}
     */
    public function supportsType($type)
    {
        return $type === 'report';
    }

    //...
}
```

The *getEntities* function

This is where the service must fetch entities from database and return them to the master service.

The results **must be** an array where the id given by the UNION query (remember *fetchQuery*).

```
namespace Chill\ReportBundle\Timeline;

use Chill\MainBundle\Timeline\TimelineProviderInterface;
use Doctrine\ORM\EntityManager;

class TimelineReportProvider implements TimelineProviderInterface
{

    public function getEntities(array $ids)
    {
```

```
        $reports = $this->em->getRepository('ChillReportBundle:Report')
              ->findBy(array('id' => $ids));

        $result = array();
        foreach($reports as $report) {
            $result[$report->getId()] = $report;
        }

        return $result;
    }

}
```

The *getEntityTemplate* function

This is where the master service will collect information to render the entity.

The result must be an associative array with :

- **template** is the FQDN of the template ;
- **template_data** is an associative array where keys are the variables'names for this template, and values are the values.

Example :

```
array(
    'template' => 'ChillMyBundle:timeline:template.html.twig',
    'template_data' => array(
        'period' => $entity,
        'person' => $args['person']
        )
    );
```

The template must, obviously, exists. Example :

```
<p><i class="fa fa-folder-open"></i> {{ 'An accompanying period is opened for %person% on %da
```

2.12.3 Create a timeline with his own context

You have to create a Controller which will execute the service *chill.main.timeline_builder*. Using the *ChillMainBundleTimelineTimelineBuilder::getTimelineHTML* function, you will get an HTML representation of the timeline, which you may include with twig *raw* filter.

Example :

```
namespace Chill\PersonBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TimelinePersonController extends Controller
{

    public function personAction(Request $request, $person_id)
    {
        $person = $this->getDoctrine()
                ->getRepository('ChillPersonBundle:Person')
                ->find($person_id);

        if ($person === NULL) {
```

```
            throw $this->createNotFoundException();
        }

        return $this->render('ChillPersonBundle:Timeline:index.html.twig', array
            (
                'timeline' => $this->get('chill.main.timeline_builder')
                        ->getTimelineHTML('person', array('person' => $person)),
                'person' => $person
            )
        );
    }

}
```

2.13 Exports

Export is an important issue for the Chill software : users should be able to :

- compute statistics about their activity ;
- list "things" which make part of their activities.

The main bundle provides a powerful framework to build custom queries with re-usable parts across differents bundles.

Table of content

See also:

The issue where this framework was discussed Provides some information about the pursued features and architecture.

2.13.1 Concepts

Some vocabulary: 3 "Export elements"

Three terms are used for this framework :

exports provides some basic operation on the date. Two kind of exports are available :

- computed data : it may be "the number of people", "the number of activities", "the duration of activities", ...
- list data : it may be "the list of people", "the list of activity", ...

filters The filters make a filter on the date: it removes some information the user doesn't want to introduce in the computation done by export. In other word, filters make a filter...

Example of filter: "people under 18 years olds", "activities between the 1st of June and the 31st December", ...

aggregators The aggregator aggregates the data into some group (some software use the term 'bucket').

Example of aggregator : "group people by gender", "group people by nationality", "group activity by type", ...

Anatomy of an export

An export may be explained as a sentence, where each part of this sentence refers to one or multiple exports element. Examples :

Example 1: Count the number of people having at least one activity in the last 12 month, and group them by nationality and gender

Here :

- *count the number of people* is the export part
- *having at least one activity* is the filter part
- *group them by nationality* is the aggregator part
- *group them by gender* is a second aggregator part

Note that :

- aggregators, filters and exports are cross-bundle. Here the bundle activity provides a filter which apply on an export provided by the person bundle ;
- there may exists multiple aggregator or filter for one export. Currently, only one export is allowed.

The result might be :

Nationality	Gender	Number of people
Russian	Male	12
Russian	Female	24
France	Male	110
France	Female	150

Example 2: Count the average duration of an activity with type “meeting”, which occurs between the 1st of June and the 31st of December, and group them by week.

Here :

- *count the average duration of an activity* is the export part
- *activity with type meeting* is a filter part
- *activity which occurs between the 1st of June and the 31st of December* is a filter
- *group them by week* is the aggregator part

The result might be :

Week	Number of activities
2015-10	10
2015-11	12
2015-12	10
2015-13	9

Authorization and exports

Exports, filters and aggregators should not make see data the user is not allowed to see.

In other words, developers are required to take care of user authorization for each export.

It should exists a special role that should be granted to users which are allowed to build exports. For more simplicity, this role should apply on center, and should not requires special circles.

2.13.2 How the magic works

To build an export, we rely on the capacity of the database to execute queries with aggregate (i.e. GROUP BY) and filter (i.e. WHERE) instructions.

An export is an SQL query which is initiated by an export, and modified by aggregators and filters.

Note: **Example 1**: Count the number of people having at least one activity in the last 12 month, and group them by nationality and gender

1. The report initiate the query `SELECT count(people.*) FROM people`
2. The filter add a where and join clause : `SELECT count(people.*) FROM people RIGHT JOIN activity WHERE activity.date IS BETWEEN now AND 6 month ago`
3. The aggregator "nationality" add a GROUP BY clause and a column in the SELECT statement: `SELECT people.nationality, count(people.*) FROM people RIGHT JOIN activity WHERE activity.date IS BETWEEN now AND 6 month ago GROUP BY nationality`
4. The aggregator "gender" do the same job as the nationality aggregator : it adds a GROUP BY clause and a column in the SELECT statement : `SELECT people.nationality, people.gender, count(people.*) FROM people RIGHT JOIN activity WHERE activity.date IS BETWEEN now AND 6 month ago GROUP BY nationality, gender`

Todo

Continue to explain the export framework

2.14 Make tests working

Unit and functional tests are important to ensure that bundle may be deployed securely.

In reason of the Chill architecture, test should be runnable from the bundle's directory and works correctly: this will allow continuous integration tools to run tests automatically.

Note: Integration tools (i.e. travis-ci) works like this :

- they clone the bundle repository in a virtual machine, using git
- they optionnaly run *composer* to download and install depedencies
- they optionnaly run other command to prepare a database, insert fixtures, ...
- they run test

On the developer's machine test should be runnable with two or three commands **runned from the bundle directory** :

```
$ composer install --dev
$ // command to insert fixtures, ...
$ phpunit
```

This chapter has been inspired by this useful blog post.

2.14.1 Bootstrap phpunit for a standalone bundle

Unit tests should run after achieving this step.

phpunit.xml

A *phpunit.xml.dist* file should be present at the bundle root.

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="./Tests/bootstrap.php" colors="true">
     <!-- the file "./Tests/boostrap.php" will be created on the next step -->
    <testsuites>
        <testsuite name="ChillMain test suite">
            <directory suffix="Test.php">./Tests</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist>
            <directory>./</directory>
            <exclude>
                <directory>./Resources</directory>
                <directory>./Tests</directory>
                <directory>./vendor</directory>
            </exclude>
       </whitelist>
   </filter>
</phpunit>
```

bootstrap.php

A file *boostrap.php*, located in the *Tests* directory, will allow phpunit to resolve class autoloading :

```
<?php

if (!is_file($autoloadFile = __DIR__.'/../vendor/autoload.php')) {
    throw new \LogicException('Could not find autoload.php in vendor/. Did you run "composer insta
}

require $autoloadFile;
```

composer.json

The *composer.json* file **located at the bundle's root** should contains all depencies needed to run test (and to execute bundle functions).

Ensure that all dependencies are included in the *require* and *require-dev* sections.

2.14.2 Functional tests

If you want to access services, database, and run functional tests, you will have to bootstrap a symfony app, with the minimal configuration. Three files are required :

- a *config_test.yml* file (eventually with a *config.yml*);
- a *routing.yml* file
- an *AppKernel.php* file

Adapt phpunit.xml

You should add reference to the new application within *phpunit.xml.dist*. The directive <*php*> should be added like this, if your *AppKernel.php* file is located in *Tests/Fixtures/App* directory:

```
<?xml version="1.0" encoding="UTF-8"?>

<phpunit bootstrap="./Tests/bootstrap.php" colors="true">
    <testsuites>
        <testsuite name="ChillMain test suite">
            <directory suffix="Test.php">./Tests</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist>
            <directory>./</directory>
            <exclude>
                <directory>./Resources</directory>
                <directory>./Tests</directory>
                <directory>./vendor</directory>
            </exclude>
        </whitelist>
    </filter>
    <!-- the lines we added -->
   <php>
        <server name="KERNEL_DIR" value="./Tests/Fixtures/App/" />
    </php>
</phpunit>
```

AppKernel.php

This file boostrap the app. It contains three functions. This is the file used in the ChillMain bundle :

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        return array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new Chill\MainBundle\ChillMainBundle(),
            new Symfony\Bundle\SecurityBundle\SecurityBundle(),
            new Symfony\Bundle\TwigBundle\TwigBundle(),
            new \Symfony\Bundle\AsseticBundle\AsseticBundle(),
            #add here all the required bundle (some bundle are not required)
        );
    }

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
    }

    /**
     * @return string
     */
    public function getCacheDir()
    {
        return sys_get_temp_dir().'/ChillMainBundle/cache';
    }

    /**
     * @return string
```

```
     */
    public function getLogDir()
    {
        return sys_get_temp_dir().'/ChillMainBundle/logs';
    }
}
```

config_test.yml

There are only few parameters required for the config file. This is a basic version for ChillMain :

```
# config/config_test.yml
imports:
    - { resource: config.yml } #here we import a config.yml file, this is not required

framework:
    test: ~
    session:
        storage_id: session.storage.filesystem
```

```
# config/config.yml
framework:
 secret:          Not very secret
 router:          { resource: "%kernel.root_dir%/config/routing.yml" }
 form:            true
 csrf_protection: true
 session:         ~
 default_locale:  fr
 translator:      { fallback: fr }
 profiler:        { only_exceptions: false }
 templating: #required for assetic. Remove if not needed
     engines: ['twig']
```

Note: You must adapt config.yml file according to your required bundle. Some options will be missing, other may be removed...

Note: If you would like to tests different environments, with differents configuration, you could create differents config_XXX.yml files.

routing.yml

You should add there all routing information needed for your bundle.

That's it. Tests should pass.

2.15 Developer manual

2.15.1 Routing and menus

The *Chill*'s architecture allows to choose bundle on each installation. This may lead to a huge diversity of installations, and a the developper challenge is to make his code working with all those possibles installations.

Chill uses menus to let users access easily to the most used functionalities. For instance, when you land on a "Person" page, you may access directly to his activities, notes, documents, ... in a single click on a side menu.

For a developer, it is easy to extend this menu with his own entries.

See also:

Symfony documentation about routing This documentation should be read before diving into those lines

Routes dans Chill (FR) The issue where we discussed routes. In French.

Create routes

Note: We recommand using *yaml* to define routes. We have not tested the existing other ways to create routes (annotations, ...). Help wanted.

The first step is as easy as create a route in symfony, and add some options in his description :

```
chill_main_dummy_0:
    pattern: /dummy/{personId}
    defaults: { _controller: CLChillMainBundle:Default:index }
    options:
        #we begin menu information here :
        menus:
            foo: #must appears in menu named 'foo'
                order: 500  #the order will be '500'
                label: foolabel #the label shown on menu. Will be translated
                otherkey: othervalue #you may add other informations, as needed by your layout
            bar: #must also appears in menu named 'bar'
                order: 500
                label: barlabel
```

The mandatory parameters under the *menus* definition are :

- *name*: the menu's name, defined as an key for the following entries
- *order*. Note: if we have duplicate order's values, the order will be incremented. We recommand using big intervals within orders and publishing the orders in your documentation
- *label*: the text which will be rendered inside the <*a*> tag. The label should be processed trough the *trans* filter (*{{ route.label|trans }}*)

You *may* also add other keys, which will be used optionally in the way the menu is rendered. See

Warning: Although all keys will be kept from your *yaml* definition to your menu template, we recommend not using those keys, which are reserved for a future implementations of Chill :

- *helper*, a text to help user or add more informations to him
- *access* : which will run a test with Expression Langage to determine if the user has the ACL to show the menu entry ;
- *condition*, which will test with the menu context if the entry must appears

Show menu in twig templates

To show our previous menu in the twig template, we invoke the *chill_menu* function. This will render the *foo* menu :

```
{{ chill_menu('foo') }}
```

Passing variables

If your routes need arguments, i.e. an entity id, you should pass the as argument to the chill_menu function. If your route's pattern is */person/{personId}*, your code become :

```
{{ chill_menu('foo', { 'args' : { 'personId' : person.id } } ) }}
```

Of course, *person* is a variable you must define in your code, which should have an *id* accessible property (i.e. : *$person->getId()*).

Note: Be aware that your arguments will be passed to all routes in a menu. If a route does not require *personId* in his pattern, the route will become */pattern?personId=XYZ*. This should not cause problem in your application.

Warning: It is a good idea to reuse the same parameter's name in your pattern, to avoid collision. Prefer */person/{personId}* to */person/{id}*.
If you don't do that and another developer create a bundle with *person/{personId}/{id}* where *{id}* is the key for something else, this will cause a lot of trouble...

Rendering active entry

Now, you want to render differently the *active* route of the menu [1]. You should, in your controller or template, add the active route in your menu :

```
{{ chill_menu('foo', { 'activeRouteKey' : 'chill_main_dummy_0' } ) }}
```

On menu creation, the route wich has the key *chill_main_dummy_0* will be rendered on a different manner.

Define your own template By default, the menu is rendered with the default template, which is a simple *ul* list. You may create your own templates :

```
#MyBundle/Resources/views/Menu/MyMenu.html.twig
<ul class="myMenu">
{% for route in routes %}
    <li><a href="{{ path(route.key, args ) }}" class="{%- if activeRouteKey == route.key -%}active
{% endfor %}
</ul>
```

Arguments available in your template :

- The *args* value are the value passed in the 'args' arguments requested by the *chill_menu* function.
- *activeRouteKey* is the key of the currently active route.
- *routes* is an array of routes. The array has this structure: *routes[order] = { 'key' : 'the_route_key', 'label' : 'the route label' }* The order is *resolved*: in case of collision (two routes from different bundles having the same order), the order will be incremented. You may find in the array your own keys (*{ 'otherkey' : 'othervalue'}* in the example above).

Then, you will call your own template with the *layout* argument :

```
{{ chill_menu('foo', { 'layout' : 'MyBundle:Menu:MyMenu.html.twig' } ) }}
```

Note: Take care of specifying the absolute path to layout in the function.

[1] In the default template, the currently active entry will be rendered with an "active" class : *<li class="active"> ... *

2.16 Layout and UI

2.16.1 Layout / Template usage

We recommand the use of the existing layouts to ensure the consistency of the design. This section explains the different templates and how to use it.

The layouts are twig templates.

Organisation of the layouts

ChillMainBundle::layout.html.twig

This is the base layout. It includes the most import css / js files. It display a page with

- a horizontal navigation menu
- a place for content
- a footer

The layout containts blocks, that are :

- title
  - to display title
- css
  - where to add some custom css
- navigation_section_menu
  - place where to insert the section menu in the navigation menu (by default the navigation menu is inserted)
- navigation_search_bar
  - place where to insert a search bar in the navigation menu (by default the search bar is inserted)
- top_banner
  - place where to display a banner below the navigation menu (this place is use to display the details of the person)
- sublayout_containter
  - place between the header and the footer that can be used to create a new layout (with vertical menu for example)
- content
  - place where to display the content (flash message are included outside of this block)
- js
  - where to add some custom javascript

ChillMainBundle::layoutWithVerticalMenu.html.twig

This layout extends *ChillMainBundle::layout.html.twig*. It replaces the block *layout_content* and divides this block for displaying a vertical menu and some content.

It proposes 2 new blocks :

- layout_wvm_content

    - where to display the page content
- vertical_menu_content
    - where to place the vertical menu

ChillMainBundle::Admin/layout.html.twig

This layout extends *ChillMainBundle::layout.html.twig*. It hides the search bar, remplaces the *section menu* with the *admin section menu*.

It proposes a new block :

- admin_content
    - where to display the admin content

ChillMainBundle::Admin/layoutWithVerticalMenu.html.twig

This layout extends *ChillMainBundle::layoutWithVerticalMenu.html.twig*. It do the same changes than *ChillMainBundle::Admin/layout.html.twig* : hiding the search bar, remplacing the *section menu* with the *admin section menu*.

It proposes a new block :

- admin_content
    - where to display the admin content

ChillPersonBundle::layout.html.twig

This layout extend *ChillMainBundle::layoutWithVerticalMenu.html.twig* add the person details in the block *top_banner*, set the menu *person* as the vertical menu.

It proposes 1 new block :

- personcontent
    - where to display the information of the person

ChillMainBundle::Export/layout.html.twig

This layout extends *ChillMainBundle::layoutWithVerticalMenu.html.twig* and set the menu *export* as the vertical menu.

It proposes 1 new block :

- export_content
    - where to display the content of the export

Useful template and helpers

Macros

Every bundle may bring their own macro to print resources with uniformized styles.

See :

- *Macros in person bundle* ;
- *Macros in activity bundle* ;

- *Macros in group bundle* ;
- *Macros in main bundle* ;

Templates

ChillMainBundle::Util:confirmation_template.html.twig This template show a confirmation template before making dangerous things. You can add your own message and title, or define those message by yourself in another template.

The accepted parameters are :

- *title* (string) a title for the page. Not mandatory (it won't be rendered if not defined)
- *confirm_question* (string) a confirmation question. This question will not be translated into the template, and may be printed as raw. Not mandatory (it won't be rendered if not defined)
- *form* : (`SymfonyComponentFormFormView`) a form wich **must** contains an input named *submit*, which must be a `SymfonyComponentFormExtensionCoreTypeSubmitType`. Mandatory
- *cancel_route* : (string) the name of a route if the user want to cancel the action
- *cancel_parameters* (array) the parameters for the route defined in *cancel_route*

Usage :

```
{{ include('ChillMainBundle:Util:confirmation_template.html.twig',
    {
        # a title, not mandatory
        'title'             : 'Remove membership'|trans,
        # a confirmation question, not mandatory
        'confirm_question'  : 'Are you sure you want to remove membership ?'|trans
        # a route for "cancel" button (mandatory)
        'cancel_route'      : 'chill_group_membership_by_person',
        # the parameters for 'cancel' route (default to {} )
        'cancel_parameters' : { 'person_id' : membership.person.id },
        # the form which will send the deletion. This form
        # **must** contains a SubmitType
        'form'              : form
    } ) }}
```

2.16.2 CSS classes and mixins

The stylesheet are based on the framework ScratchCSS.

We added some useful classes and mixins for the Chill usage.

CSS Classes

Statement "empty data"

CSS Selector `.chill-no-data-statement`

In which case will you use this selector ? When a list is empty, and a message fill the list to inform that the data is empty

Example usage

```
    <span class="chill-no-data-statement">{{ 'No reason associated'|trans }}</span>
```

Quotation of user text

CSS Selector `blockquote.chill-user-quote`

In which case will you use this selector ? When you quote text that were filled by the user in a form.

Example usage

```
    <blockquote class="chill-user-quote">{{ entity.remark|nl2br }}</blockquote>
```

Mixins

Entity decorator

Mixin `@mixin entity($background-color, $color:  white)`

In which case including this mixin ? When you create a *sticker*, a sort of label to represent a text in a way that the user can associate immediatly with a certain type of class / entity.

Example usage

```
    span.entity.entity-activity.activity-reason {
        @include entity($chill-pink, white);
    }
```

2.16.3 Widgets

Rationale

Widgets are useful if you want to publish content on a page provided by another bundle.

Examples :

- you want to publish a list of people on the homepage ;
- you may want to show the group belonging (see *Group bundle*) below of the vertical menu, only if the bundle is installed.

The administrator of the chill instance may configure the presence of widget. Although, some widget are defined by default (see *Declaring a widget by default*).

Concepts

A bundle may define *place(s)* where a widget may be rendered.

In a single *place*, zero, one or more *widget* may be displayed.

Some *widget* may require some *configuration*, and some does not require any configuration.

Example:

Use case	place	place defined by...	widget provided by...
Publishing a list of people on the homepage	home-page	defined by *Main bundle*	widget provided by *Person bundle*

Creating a widget without configuration

To add a widget, you should :

- define your widget, implementing `ChillMainBundleTemplatingWidgetWidgetInterface` ;
- declare your widget with tag *chill_widget*.

Define the widget class

Define your widget class by implemeting `ChillMainBundleTemplatingWidgetWidgetInterface`.

Example :

```
namespace Chill\PersonBundle\Widget;

use Chill\MainBundle\Templating\Widget\WidgetInterface;


/**
* Add a button "add a person"
*
*/
class AddAPersonWidget implements WidgetInterface
{
   public function render(
      \Twig_Environment $env,
      $place,
      array $context,
      array $config
   ) {
       // this will render a link to the page "add a person"
       return $env->render("ChillPersonBundle:Widget:homepage_add_a_person.html.twig");
   }
}
```

Arguments are :

- `$env` the `Twig_Environment`, which you can use to render your widget ;
- `$place` a string representing the place where the widget is rendered ;
- `$context` the context given by the template ;
- `$config` the configuration which is, in this case, always an empty array (see *Creating a widget with configuration*).

Note: The html returned by the `render` function will be considered as html safe. You should strip html before returning it. See also How to escape output in template.

Declare your widget

Declare your widget as a service and add it the tag `chill_widget`:

```
service:
    chill_person.widget.add_person:
        class: Chill\PersonBundle\Widget\AddAPersonWidget
        tags:
            - { name: chill_widget, alias: add_person, place: homepage }
```

The tag must contains those arguments :

- `alias`: an alias, which will be used to reference the widget into the config
- `place`: a place where this widget is authorized

If you want your widget to be available on multiple places, you should add one tag with each place.

Conclusion

Once your widget is correctly declared, your widget should be available in configuration.

```
$ php app/console config:dump-reference chill_main
# Default configuration for extension with alias: "chill_main"
chill_main:
        [...]
        # register widgets on place "homepage"
        homepage:

            # the ordering of the widget. May be a number with decimal
            order:                ~ # Required, Example: 10.58

            # the widget alias (see your installed bundles config). Possible values are (maybe in
            widget_alias:         ~ # Required
```

If you want to add your widget by default, see *Declaring a widget by default*.

Creating a widget with configuration

You can declare some configuration with your widget, which allow administrators to add their own configuration.

To add some configuration, you will :

- declare a widget as defined above ;
- optionnaly declare it as a service ;
- add a widget factory, which will add configuration to the bundle which provide the place.

Declare your widget class

Declare your widget. You can use some configuration elements in your process, as used here :

```
<?php
# Chill/PersonBundle/Widget/PersonListWidget.php

namespace Chill\PersonBundle\Widget;

use Chill\MainBundle\Templating\Widget\WidgetInterface;
use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\Query\Expr;
use Doctrine\DBAL\Types\Type;
use Chill\MainBundle\Security\Authorization\AuthorizationHelper;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorage;
use Chill\PersonBundle\Security\Authorization\PersonVoter;
use Symfony\Component\Security\Core\Role\Role;
use Doctrine\ORM\EntityManager;

/**
 * add a widget with person list.
 *
 * The configuration is defined by `PersonListWidgetFactory`
 */
class PersonListWidget implements WidgetInterface
{

    /**
     * Repository for persons
     *
```

```
     * @var EntityRepository
     */
    protected $personRepository;

    /**
     * The entity manager
     *
     * @var EntityManager
     */
    protected $entityManager;

    /**
     * the authorization helper
     *
     * @var AuthorizationHelper;
     */
    protected $authorizationHelper;

    /**
     *
     * @var TokenStorage
     */
    protected $tokenStorage;

    /**
     *
     * @var UserInterface
     */
    protected $user;

    public function __construct(
            EntityRepository $personRepostory,
            EntityManager $em,
            AuthorizationHelper $authorizationHelper,
            TokenStorage $tokenStorage
            ) {
        $this->personRepository = $personRepostory;
        $this->authorizationHelper = $authorizationHelper;
        $this->tokenStorage = $tokenStorage;
        $this->entityManager = $em;
    }

    /**
     *
     * @param type $place
     * @param array $context
     * @param array $config
     * @return string
     */
    public function render(\Twig_Environment $env, $place, array $context, array $config)
    {
        $qb = $this->personRepository
                ->createQueryBuilder('person');

        // show only the person from the authorized centers
        $and = $qb->expr()->andX();
        $centers = $this->authorizationHelper
                ->getReachableCenters($this->getUser(), new Role(PersonVoter::SEE));
        $and->add($qb->expr()->in('person.center', ':centers'));
        $qb->setParameter('centers', $centers);


        // add the "only active" where clause
```

```
        if ($config['only_active'] === true) {
            $qb->join('person.accompanyingPeriods', 'ap');
            $or = new Expr\Orx();
            // add the case where closingDate IS NULL
            $andWhenClosingDateIsNull = new Expr\Andx();
            $andWhenClosingDateIsNull->add((new Expr())->isNull('ap.closingDate'));
            $andWhenClosingDateIsNull->add((new Expr())->gte(':now', 'ap.openingDate'));
            $or->add($andWhenClosingDateIsNull);
            // add the case when now is between opening date and closing date
            $or->add(
                    (new Expr())->between(':now', 'ap.openingDate', 'ap.closingDate')
                );
            $and->add($or);
            $qb->setParameter('now', new \DateTime(), Type::DATE);
        }

        // adding the where clause to the query
        $qb->where($and);

        $qb->setFirstResult(0)->setMaxResults($config['number_of_items']);

        $persons = $qb->getQuery()->getResult();

        return $env->render(
            'ChillPersonBundle:Widget:homepage_person_list.html.twig',
            array('persons' => $persons)
            );
    }

    /**
     *
     * @return UserInterface
     */
    private function getUser()
    {
        // return a user
    }

}
```

Declare your widget as a service

You can declare your widget as a service. Not tag is required, as the service will be defined by the `Factory` during next step.

```
services:
   chill_person.widget.person_list:
       class: Chill\PersonBundle\Widget\PersonListWidget
       arguments:
           - "@chill.person.repository.person"
           - "@doctrine.orm.entity_manager"
           - "@chill.main.security.authorization.helper"
           - "@security.token_storage"
   # this widget is defined by the PersonListWidgetFactory
```

You can eventually skip this step and declare your service into the container through the factory (see above).

Declare your widget factory

The widget factory must implements *ChillMainBundleDependencyInjectionWidgetFactoryWidgetFactoryInterface*. For your convenience, an `ChillMainBundleDependencyInjectionWidgetFactoryAbstractWidgetFa` will already implements some easy method.

```
<?php

# Chill/PersonBundle/Widget/PersonListWidgetFactory

namespace Chill\PersonBundle\Widget;

use Chill\MainBundle\DependencyInjection\Widget\Factory\AbstractWidgetFactory;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\Definition\Builder\NodeBuilder;

/**
 * add configuration for the person_list widget.
 */
class PersonListWidgetFactory extends AbstractWidgetFactory
{
   /*
    * append the option to the configuration
    * see http://symfony.com/doc/current/components/config/definition.html
    *
    */
    public function configureOptions($place, NodeBuilder $node)
    {
        $node->booleanNode('only_active')
                ->defaultTrue()
                ->end();
        $node->integerNode('number_of_items')
            ->defaultValue(50)
            ->end();
        $node->scalarNode('filtering_class')
                ->defaultNull()
                ->end();

    }

   /*
    * return an array with the allowed places where the widget can be rendered
    *
    * @return string[]
    */
    public function getAllowedPlaces()
    {
        return array('homepage');
    }

   /*
    * return the widget alias
    *
    * @return string
    */
    public function getWidgetAlias()
    {
        return 'person_list';
    }

    /*
     * return the service id for the service which will render the widget.
     *
```

```
     * this service must implements `Chill\MainBundle\Templating\Widget\WidgetInterface`
     *
     * the service must exists in the container, and it is not required that the service
     * has the `chill_main` tag.
     */
    public function getServiceId(ContainerBuilder $containerBuilder, $place, $order, array $confi
    {
        return 'chill_person.widget.person_list';
    }

}
```

Note: You can declare your widget into the container by overriding the *createDefinition* method. By default, this method will return the already existing service definition with the id given by `getServiceId`. But you can create or adapt programmatically the definition. See the symfony doc on how to do it.

```
public function createDefinition(ContainerBuilder $containerBuilder, $place, $order, array $confi
{
    $definition = new \Symfony\Component\DependencyInjection\Definition('my\Class');
    // create or adapt your definition here

    return $definition;
}
```

You must then register your factory into the `Extension` class which provide the place. This is done in the :code: *Bundle* class.

```
# Chill/PersonBundle/ChillPersonBundle.php

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Chill\PersonBundle\Widget\PersonListWidgetFactory;

class ChillPersonBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        $container->getExtension('chill_main')
            ->addWidgetFactory(new PersonListWidgetFactory());
    }
}
```

Declaring a widget by default

Use the ability to prepend configuration of other bundle. A living example here :

```
<?php

# Chill/PersonBundle/DependencyInjection/ChillPersonExtension.php

namespace Chill\PersonBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
use Chill\MainBundle\DependencyInjection\MissingBundleException;
```

```
use Chill\PersonBundle\Security\Authorization\PersonVoter;

/**
 * This is the class that loads and manages your bundle configuration
 *
 * To learn more see {@link http://symfony.com/doc/current/cookbook/bundles/extension.html}
 */
class ChillPersonExtension extends Extension implements PrependExtensionInterface
{
    /**
     * {@inheritDoc}
     */
    public function load(array $configs, ContainerBuilder $container)
    {
      // ...
    }


    /**
     *
     * Add a widget "add a person" on the homepage, automatically
     *
     * @param \Chill\PersonBundle\DependencyInjection\containerBuilder $container
     */
    public function prepend(ContainerBuilder $container)
    {
        $container->prependExtensionConfig('chill_main', array(
            'widgets' => array(
                'homepage' => array(
                    array(
                        'widget_alias' => 'add_person',
                        'order' => 2
                    )
                )
            )
        ));
    }
}
```

Defining a place

Add your place in template

A place should be defined by using the `chill_widget` function, which take as argument :

- `place` (string) a string defining the place ;
- `context` (array) an array defining the context.

The context should be documented by the bundle. It will give some information about the context of the page. Example: if the page concerns a people, the `ChillPersonBundleEntityPerson` class will be in the context.

Example :

```
{# an empty context on homepage #}
{{ chill_widget('homepage', {} }}
```

```
{# defining a place 'right column' with the person currently viewed
{{ chill_widget('right_column', { 'person' : person } }}
```

Declare configuration for you place

In order to let other bundle, or user, to define the widgets inside the given place, you should open a configuration. You can use the Trait `ChillMainBundleDependencyInjectionWidgetAddWidgetConfigurationTrait`, which provide the method *addWidgetConfiguration($place, ContainerBuilder $container)*.

Example :

```php
<?php
   # Chill\MainBundle\DependencyInjection\Configuration.php

   namespace Chill\MainBundle\DependencyInjection;

   use Symfony\Component\Config\Definition\Builder\TreeBuilder;
   use Symfony\Component\Config\Definition\ConfigurationInterface;
   use Chill\MainBundle\DependencyInjection\Widget\AddWidgetConfigurationTrait;
   use Symfony\Component\DependencyInjection\ContainerBuilder;

   /**
    * Configure the main bundle
    */
   class Configuration implements ConfigurationInterface
   {

       use AddWidgetConfigurationTrait;

       /**
        *
        * @var ContainerBuilder
        */
       private $containerBuilder;


       public function __construct(array $widgetFactories = array(),
               ContainerBuilder $containerBuilder)
       {
           // we register here widget factories (see below)
           $this->setWidgetFactories($widgetFactories);
           // we will need the container builder later...
           $this->containerBuilder = $containerBuilder;
       }

       /**
        * {@inheritDoc}
        */
       public function getConfigTreeBuilder()
       {
           $treeBuilder = new TreeBuilder();
           $rootNode = $treeBuilder->root('chill_main');

           $rootNode
               ->children()

                   // ...

                   ->arrayNode('widgets')
                       ->canBeDisabled()
                       ->children()
                           // we declare here all configuration for homepage place
                           ->append($this->addWidgetsConfiguration('homepage', $this->containerBu
                       ->end() // end of widgets/children
                   ->end() // end of widgets
```

```
                    ->end() // end of root/children
                ->end() // end of root
            ;


            return $treeBuilder;
        }
    }
```

You should also adapt the `DependencyInjection*Extension` class to add ContainerBuilder and WidgetFactories :

```
<?php

#Chill\MainBundle\DependencyInjection\ChillMainExtension.php

namespace Chill\MainBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
use Chill\MainBundle\DependencyInjection\Widget\Factory\WidgetFactoryInterface;
use Chill\MainBundle\DependencyInjection\Configuration;

/**
 * This class load config for chillMainExtension.
 */
class ChillMainExtension extends Extension implements Widget\HasWidgetFactoriesExtensionInterface
{
    /**
     * widget factory
     *
     * @var WidgetFactoryInterface[]
     */
    protected $widgetFactories = array();

    public function addWidgetFactory(WidgetFactoryInterface $factory)
    {
        $this->widgetFactories[] = $factory;
    }

    /**
     *
     * @return WidgetFactoryInterface[]
     */
    public function getWidgetFactories()
    {
        return $this->widgetFactories;
    }

    public function load(array $configs, ContainerBuilder $container)
    {
        // configuration for main bundle
        $configuration = $this->getConfiguration($configs, $container);
        $config = $this->processConfiguration($configuration, $configs);

        // add the key 'widget' without the key 'enable'
        $container->setParameter('chill_main.widgets',
            array('homepage' => $config['widgets']['homepage']));

        // ...
```

```
52      }
53
54      public function getConfiguration(array $config, ContainerBuilder $container)
55      {
56          return new Configuration($this->widgetFactories, $container);
57      }
58
59  }
```

- line 25-39: we implements the method required by `ChillMainBundleDependencyInjectionWidgetHasWidgetE` ;
- line 48-49: we record the configuration of widget into container's parameter ;
- line 56 : we create an instance of `Configuration` (declared above)

Compile the possible widget using Compiler pass

For your convenience, simply extends `ChillMainBundleDependencyInjectionWidgetAbstractWidgetsCompile` This class provides a *doProcess(ContainerBuildere $container, $extension, $parameterName)* method which will do the job for you:

- `$container` is the container builder
- `$extension` is the extension name
- `$parameterName` is the name of the parameter which contains the configuration for widgets (see *the example with ChillMain above*.

```
namespace Chill\MainBundle\DependencyInjection\CompilerPass;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Chill\MainBundle\DependencyInjection\Widget\AbstractWidgetsCompilerPass;

/**
* Compile the service definition to register widgets.
*
*/
class WidgetsCompilerPass extends AbstractWidgetsCompilerPass {

    public function process(ContainerBuilder $container)
    {
        $this->doProcess($container, 'chill_main', 'chill_main.widgets');
    }
}
```

As explained in the symfony docs, you should register your Compiler Pass into your bundle :

```
namespace Chill\MainBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Chill\MainBundle\DependencyInjection\CompilerPass\WidgetsCompilerPass;


class ChillMainBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);
        $container->addCompilerPass(new WidgetsCompilerPass());
    }
}
```

2.16.4 Javascript functions

Some function may be useful to manipulate elements on the page.

Show-hide elements according to checkbox

The function `chill.listenerDisplayCheckbox` will make appears / disappears elements according to a checkbox (if the checkbox is checked, the elements will appears).

Usage

The checkbox must have the data *data-display-target* with an id, and the parts to show/hide must have the data *data-display-show-hide* with the same value.

On the same page, you should run the function `chill.listenerDisplayCheckbox`.

Example :

```
<input data-display-target="export_abc" value="1" type="checkbox">

<div data-display-show-hide="export_abc">
<!-- your content here will be hidden / shown according to checked state -->
</div>

<!-- you could use the block "js" to render this script at the bottom of the page -->
<!-- {{ block js }} -->
<script type="text/javascript">
   window.addEventListener("DOMContentLoaded", chill.listenerDisplayCheckbox);
</script>
<!-- {{ endblock js }} -->
```

Note: Hint

For forms in symfony, you could use the *id* of the form element, accessible through *{{ form.vars.id }}*. This id should be unique.

2.17 Help, I am lost !

Write an email at info@champs-libres.coop, and we will help you !

CHAPTER 3

Bundles documentation

You will find here documentation about bundles working with Chill.

3.1 Main bundle

This bundle is **required** for running Chill.

This bundle provide :

- Access control model (users, groups, and all concepts)
- ...

Warning: this section is incomplete.

3.1.1 Macros

Address sticker

Macro file *ChillMainBundle:Address:macro.html.twig*

Macro name `_render`

Macro envelope `address`, instance of `ChillMainBundleEntityAddress`

When to use this macro ? When you want to represent an address.

Example usage :

```
{% import 'ChillMainBundle:Address:macro.html.twig' as m %}

{{ m._render(address) }}
```

3.2 Custom fields bundle

This bundle provides the ability to add custom fields to existing entities.

Those custom fields contains extra data and will be stored in the DB, along with other data's entities. It may be string, text, date, ... or a link to an existing entities.

In the database, custom fields are stored in json format.

See also:

The full specification discussed here.

JSON Type on postgresql documentation The documentation of json type, which is used to store data in the database.

Table of contents

3.2.1 Custom Fields concepts

Custom fields are extra data which may be added to entities by user. If a developer implements custom fields on a entity, users will be able to add more fields on this entity.

Example: the *person bundle* allows to record *firstname*, *lastname*, *date of birth* fields. But users need to store information about the kind of house he has (if he owns his house, if he rents it, ...). Custom fields allows to create those fields.

Automatically, those fields are added at the person form. They are also printed in the person view and in exports.

Custom fields and custom fields group

Custom fields are associated to a custom fields group. When a user want to add custom fields on an entity, he must first create a custom fields group and associate this field with an entity. Then he may add add custom fields to this groups.

Some entities needs a **default** custom fields group. For instance, the default custom fields group will be printed on the main form for person, and will be appended on the main person view. Some bundle does not use this feature (i.e. the *report* bundle).

Note: In the future of the *person bundle*, other custom fields group will be added in forms accessible from the menu, allowing users to completely customize and separate their entities.

3.2.2 Allow custom fields on an entity

As a developer, you must allow your users to add custom fields on your entities.

Warning: For having custom fields, the class of the entity must contain a variable for storing the custom data. **By convention this variable must be called $cFData**

Create a json field on your entity

Declare a json field in your database :

```
Chill\CustomFieldsBundle\Entity\BlopEntity:
   type: entity
   # ...
   fields:
      cFData:
         type: json_array
```

Create the field accordingly in the class logic :

```php
namespace Chill\CustomFieldsBundle\Entity;

/**
* BlopEntity
*/
class BlopEntity
{

/**
* @var array
*/
private $cFData;

/**
* You must set a setter in order to save automatically custom
* fields from forms, using Form Component
*
* @param array $cFData
* @return BlopEntity
*/
public function setCFData(array $cFData)
{
   $this->cFData = $cFData;
   return $this;
}

/**
* You also must create a getter in order to let Form
* component populate form fields
*
* @return array
*/
public function getCFData()
{
   return $this->cFData;
}
```

Declare your customizable entity in configuration

This step is necessary to allow user to create custom fields group associated with this entity.

Two methods are available.

The recommended method is to do it in DependencyInjection/Extension class. It is recommended as your bundle will be well set up (for custom field) in every chill installation that use it.

The discouraged method is to declare via the app/config.yml file. This is very quick to set up for a given chill installation but it is not done automatically : in every chill installation that use your bundle, this step has to be performed.

In app/config.yml file (discouraged)

This method is discouraged but explained first as it helps to undersand the recommended method.

Add those file under *chill_custom_fields* section :

```
chill_custom_fields:
   customizables_entities:
      - { class: Chill\YourBundleBundle\Entity\BlopEntity, name: blop_entity }
```

- The *name* allow you to define a string which is translatable. This string will appears when chill's admin will add/retrieve new customFieldsGroup.
- The class, which is a full FQDN class path

Automatically, in DependencyInjection/Extension class (recommended)

This is the recommended way for declaring customizable classes.

You can prepend configuration of *custom fields bundle* from the class *YourBundleDependencyInjectionYourBundleExtension*. **Note** that you also have to implements *SymfonyComponentDependencyInjectionExtensionPrependExtensionInterface* on this class to make the *prepend* function being taken into account.

Example here :

```
class ChillYourBundleExtension extends Extension implements PrependExtensionInterface
{
    /**
     * @param ContainerBuilder $container
     */
    public function prepend(ContainerBuilder $container)
    {
        $bundles = $container->getParameter('kernel.bundles');
        if (!isset($bundles['ChillCustomFieldsBundle'])) {
            throw new MissingBundleException('ChillCustomFieldsBundle');
        }

        $container->prependExtensionConfig('chill_custom_fields',
            array('customizables_entities' =>
                array(
                    array(
                        'class' => 'Chill\YourBundleBundle\Entity\BlopEntity',
                        'name' => 'blop_entity',)
                )
            )
        );
    }
}
```

- The *name* allow you to define a string which is translatable. This string will appears when chill's admin will add/retrieve new customFieldsGroup.
- The class, which is a full FQDN class path

See also:

How to simplify configuration of multiple bundles A cookbook page about prepending configuration.

Adding options to your custom fields groups

You may add options to the groups associated with an entity.

In *config.yml* the declaration should be :

```
chill_custom_fields:
    customizables_entities:
        -
            class: Chill\YourBundleBundle\Entity\BlopEntity
            name: BlopEntity
            options:
                # this will create a "myFieldKey" field as text, with a maxlength attribute to 15
                myFieldKey: {form_type: text, form_options: {attr: [maxlength: 150]}}
```

In the *PrependExtensionInterface::prepend* function, the options key will be added in the configuration definition :

```
class ChillYourBundleExtension extends Extension implements PrependExtensionInterface
{
    /**
     * @param ContainerBuilder $container
     */
    public function prepend(ContainerBuilder $container)
    {
        $bundles = $container->getParameter('kernel.bundles');
        if (!isset($bundles['ChillCustomFieldsBundle'])) {
            throw new MissingBundleException('ChillCustomFieldsBundle');
        }

        $container->prependExtensionConfig('chill_custom_fields',
            array('customizables_entities' =>
                array(
                    array(
                        'class' => 'Chill\YourBundleBundle\Entity\BlopEntity',
                        'name' => 'BlopEntity',
                        'options' => array(
                              'myFieldKey' => [ 'form_type' => 'text', 'form_options' => [ 'attr' =
                        ))
                )
            )
        );
    }
}
```

Example : the entity *Report* from **ReportBundle** has to pick some custom fields belonging to a group to print them in *summaries* the timeline page. The definition will use the special type *custom_fields_group_linked_custom_field* which will add a select input with all fields associated with the current custom fields group :

```
class ChillReportExtension extends Extension implements PrependExtensionInterface
{
    /**
     *
     *
     * @param ContainerBuilder $container
     */
    public function prepend(ContainerBuilder $container)
    {
        $bundles = $container->getParameter('kernel.bundles');
        if (!isset($bundles['ChillCustomFieldsBundle'])) {
            throw new MissingBundleException('ChillCustomFieldsBundle');
        }

        $container->prependExtensionConfig('chill_custom_fields',
            array('customizables_entities' =>
                array(
                    array(
                        'class' => 'Chill\ReportBundle\Entity\Report',
                        'name' => 'ReportEntity',
```

```
                        'options' => array(
                            'summary_fields' => array(
                                'form_type' => 'custom_fields_group_linked_custom_fields',
                                'form_options' =>
                                    [
                                        'multiple' => true,
                                        'expanded' => false
                                    ]
                            )
                        ))
                )
            )
        );
    }
}
```

Note that *custom_fields_group_linked_custom_fields* does not create any input on *CustomFieldsGroup* creation : there aren't any fields associated with the custom fields just after the group creation... You have to add custom fields and associate them with the newly created group to see them appears.

3.2.3 Rendering custom fields and custom fields group in a template

Warning: Each custom field can be *active* or not. Only *active* custom fields has to be dislayed.

For rendering custom fields, two function are available :

- *chill_custom_field_widget* to render the widget. This function is defined on a customFieldType basis.
- *chill_custom_field_label* to render the label. You can customize the label rendering by choosing the layout you would like to use.
- *chill_custom_field_is_empty* indicates if the content of the custom fields is empty (return a boolean)

For rendering custom fields group, a function is available :

- *chill_custom_fields_group_widget* to render the widget. It will display the custom fields of the group in a dd / dt structure.

chill_custom_field_label

The signature is :

- *CustomField* **$customField** a customField instance
- *array* **params** the parameters for rendering. Currently, 'label_layout' allow to choose a different label. Default is 'ChillCustomFieldsBundle:CustomField:render_label.html.twig'

Examples

```
{{ chill_custom_field_label(customField) }}
```

chill_custom_field_widget

The signature is :

- array **$fields** the array raw, as stored in the db
- CustomField **$customField** a customField instance
- string **$documentType** the type of document. Default to *html*.

Examples:

```
{{ chill_custom_field_widget(entity.customFields, customField) }}
```

chill_custom_field_is_empty

The signature is :

- array **$fields** the array raw, as stored in the db
- CustomField **$customField** a customField instance

Examples :

```
{%- if chill_custom_field_is_empty(cFData, customField) == false -%}
```

chill_custom_fields_group_widget

This function only display custom fields that are *active*.

The signature is :

- array **$fields** the array raw, as stored in the db
- CustomFieldsGroup **$customFieldsGroup** the custom field group to render

```
{{ chill_custom_fields_group_widget(entity.cFData, entity.customFieldsGroup) }}
```

3.2.4 Custom Fields's form

You should simply use the 'custom_field' type in a template, with the group you would like to render in the *group* option's type.

Example :

```
namespace Chill\ReportBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ReportType extends AbstractType
{
    /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $entityManager = $options['em'];

        $builder
            ->add('user')
            ->add('date', 'date',
                array('required' => true, 'widget' => 'single_text', 'format' => 'dd-MM-yyyy'))
            #add the custom fields :
            ->add('cFData', 'custom_field',
                array('attr' => array('class' => 'cf-fields'), 'group' => $options['cFGroup']))
        ;
    }

    /**
     * @param OptionsResolverInterface $resolver
```

```
     */
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Chill\ReportBundle\Entity\Report'
        ));

        $resolver->setRequired(array(
            'em',
            'cFGroup',
        ));

        $resolver->setAllowedTypes(array(
            'em' => 'Doctrine\Common\Persistence\ObjectManager',
            'cFGroup' => 'Chill\CustomFieldsBundle\Entity\CustomFieldsGroup'
        ));
    }

    /**
     * @return string
     */
    public function getName()
    {
        return 'chill_reportbundle_report';
    }
}
```

3.2.5 Available configuration

Those options are available in the configuration, under the *chill_custom_field* key.

Example :

```
chill_custom_field:
   show_empty_values_in_views:  false
```

show_empty_values_in_views *boolean*: Allow to hide / show empty values in views. The aim of this configuration parameter is to hide (or show) empty values when *custom fields group* are rendered.

Default value : *true*

3.2.6 Glossary

custom fields group A group of custom fields

3.3 Person bundle

This bundle provides the ability to record people in the software. This bundle is required by other bundle.

Table of content

3.3.1 Entities provided

Todo

describe entities provided by person bundle

3.3.2 Search terms

The class *ChillPersonBundleSearchPersonSearch* provide the search module.

Domain

The search upon "person" is provided by default. The *@person* domain search may be omitted.

- *@person* is the domain search for people.

Arguments

- *firstname* : provide the search on firstname. Example : *firstname:Depardieu*. May match part of the firstname (*firsname:dep* will match Depardieu)
- *lastname* : provide the search on lastname. May match part of the lastname.
- *birthdate* : provide the search on the birthdate. Example : *birthdate:1996-01-19*
- *gender*: performs search on man/woman. The accepted values are *man* or *woman*.
- *nationality* : performs search on nationality. Value must be a country code as described in ISO 3166. Example : *nationality:FR*.

Default

The default search is performed on firstname and/or lastname. Both are concatened before search. If values are separated by spaces, the clause *AND* is used : the search *dep ge* will match 'Gérard Depardieu' or 'Jean Depagelles', but not 'Charline Depardieu' (missing 'Ge' in word).

3.3.3 Configuration options

Those options are available under *chill_person* key.

Example of configuration:

```
chill_person:
   validation:
      birthdate_not_after: P15Y
   person_fields:
      # note: visible is the default config. This key may be omitted if visible is chosen.
      nationality: hidden
      email: hidden
      place_of_birth: visible
      phonenumber: hidden
      country_of_birth: hidden
      marital_status: visible
      spoken_languages: hidden
      address: visible
```

birthdate_not_after ***string*** The period duration before today during which encoding birthdate is not possible. The period is a string matching the format of *ISO_8601*, which is also use to build DateInterval classes.

Example: *P1D*, *P18Y*

Default value: *P1D* which means that birthdate before the current day (= yesterday) are allowed.

person_fields ***array*** This define the visibility of some fields. By default, all fields are visible, but you can choose to hide some of them. Available keys are :

- *nationality*
- *country_of_birth*
- *place_of_birth*
- *phonenumber*
- *email*
- *marital_status*
- *spoken_languages*
- *address*

Possibles values: *hidden* or *visible* (all other value will raise an Exception).

Default value : *visible*, which means that all fields are visible.

Example:

```
chill_person:
   person_fields:
      nationality: hidden
      email: hidden
      phonenumber: hidden
```

Note: If all the field of a "box" are hidden, the whole box does not appears. Example: if the fields *phonenumber* and *email* are hidden, the title *Contact information* will be hidden in the UI.

Note: If you hide multiple fields, for a better integration you may want to override the template, for a better appeareance. See the symfony documentation about this feature.

3.3.4 Macros

Sticker for a person

Macro file *ChillPersonBundle:Person:macro.html.twig*

Macro envelope `render(p, withLink=false)`

`p` is an instance of `ChillPersonBundleEntityPerson`

`withLink` `boolean`

When to use this macro ? When you want to represent a person.

Example usage :

```
{% import "ChillPersonBundle:Person:macro.html.twig" as person_ %}

{{ person_.render(person, true) }}
```

3.3.5 Layout events and delegated blocks

`chill_block.person_post_vertical_menu` event

This event is available to add content below of the vertical menu (on the right).

The context is :

- `person` : the current person which is rendered. Instance of `ChillPersonBundleEntityPerson`

3.4 Report bundle

This bundle provides the ability to record report about people. We use custom fields to let user add fields to reports.

Table of content

Todo

The documentation about report is not writtend

3.4.1 Concepts

3.4.2 Search

Domain

- *@report* is the domain search for reports.

Arguments

- *date* : The date of the report

Default

The report's date is the default value.

An error is thrown if an argument *date* and a default is used.

3.5 Activity bundle

This bundle provides the ability to record people in the software. This bundle is required by other bundle.

Table of content

3.5.1 Entities provided

Todo

Describe the entities provided.

3.5.2 Configuration options

Those options are available under *chill_activity* key.

Example of configuration:

```
chill_activity:
   form:
      time_duration:
         - { label: '12 minutes', seconds: 720 }
         - { label: '30 minutes', seconds: 1800 }
```

form.time_duration ***array*** The duration which might be suggested when the user create or update an activity. The value must be an array of object, where each object must have a `label` and a `seconds` key. The label provide which is shown to user (the label will be translated, if possible) and the seconds the duration.

Example: see the example above

Default value: the values available are 5, 10, 15, 20, 25, 30, 45 minutes, and 1 hour, 1 hour 15, 1 hour 30, 1 hour 45 and 2 hours.

3.5.3 Macros

Activity reason sticker

Macro file *ChillActivityBundle:ActivityReason:macro.html.twig*

Macro envelope `reason(r)`

`p` is an instance of `ChillActivityBundleEntityActivityReason`

When to use this macro ? When you want to represent an activity reason.

Example usage :

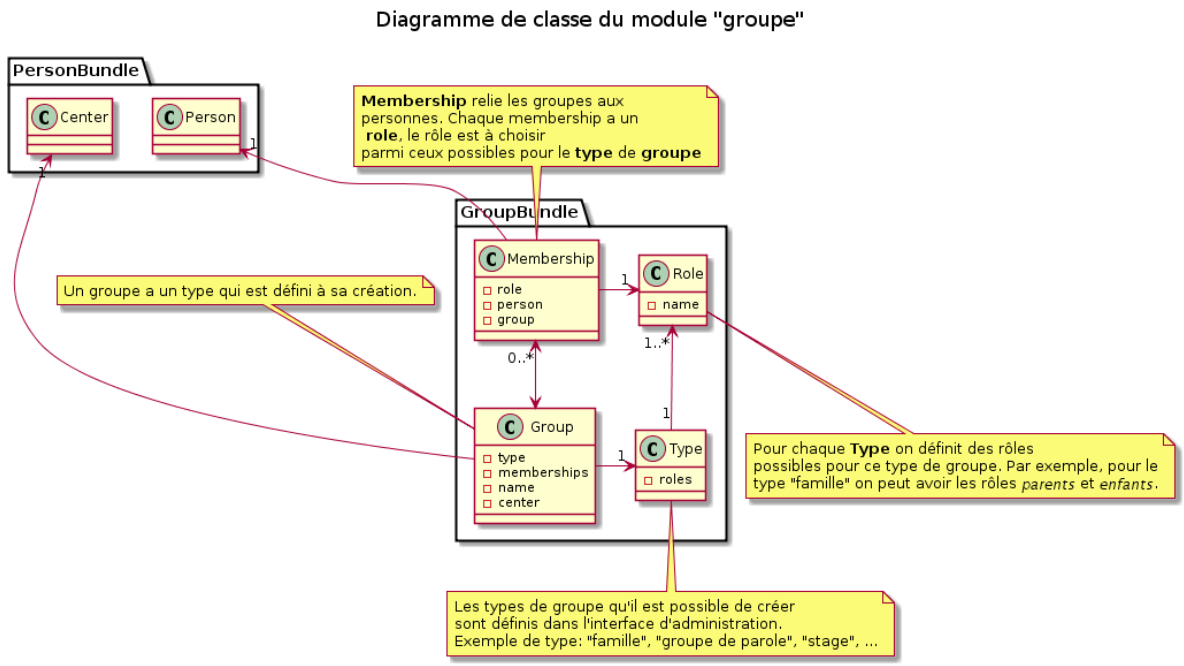
```
{% import 'ChillActivityBundle:ActivityReason:macro.html.twig' as m %}

{{ m.reason(r) }}
```

3.6 Group bundle

Allow to group people in a group. This group may be a family, an activity group, ...

Table of content

3.6.1 Entities



3.6.2 Macros

Group sticker

Macro file *ChillGroupBundle:Group:macro.html.twig*

Macro name `_render`

Macro envelope `group`, instance of `ChillGroupBundleEntityCGroup`

When to use this macro ? When you want to represent group.

Example usage :

```
{% import 'ChillGroupBundle:Group:macro.html.twig' as m %}

{{ m._render(g) }}
```

3.7 Event bundle

3.7.1 Template & Menu

The event bundle has a special template with a specific menu for actions on events. This menu is called *event*.

ChillEventBundle::layout.html.twig

This layout extends *ChillMainBundle::layoutWithVerticalMenu.html.twig* and add the menu *event*

It proposes a new block :

- event_content
  - where to display content relative to the event.

3.8 LDAP bundle

This bundle binds the database with an ldap directory.

The bundle synchronize the ldap directory with users in the database. It also provides a way to check user credentials against the ldap directory.

Table of content

3.8.1 Current limitations

- The length of the ldap dn must be < 255 characters
- if the username extracted from the ldap is updated, the changes are not reflected in the database and remains the same

3.8.2 Entities provided

This bundle provides only one entity : `UserLdapBinding`

3.8.3 How the synchronizer works ?

1. The synchronizer performs a query on `dn` and `query` defined in the *configuration*.
2. For each entry returned by the query, it looks if the `dn` exists in the database
   (a) If the entry does not exists :
      i. the synchronizer looks for user with same username as defined by `username_attr`, and bind it with the `dn` if it exists.
      ii. else, a user is created with username defined by `username_attr` (if the ldap contains more than one attribute, the first attribute returned is used)
   (b) if a user exists which is already binded with the `dn`, the entry is ignored.
3. The synchronizer looks for dn existing in database and which were not returned by the query performed in 1.
   (a) If they exists, those user are set to `enabled=false`: they are not allowed to login.

3.8.4 Installation

This bundle requires :

- PHP LDAP ext
- `symfony/ldap` with minimal version 3.1. Note that, currently, Chill uses Symfony 2.8: you should add the dependency on this single package manually

In your composer.json, for stable version :

```
"require": {
        // .. other dependencies
        "symfony/ldap" : "~3.1",
        "chill-project/ldap": "~1.0"
}
```

And for dev version :

```
"require": {
        // .. other dependencies
        "symfony/ldap" : "~3.1",
        "chill-project/ldap": "dev-master@dev"
}
```

3.8.5 Configuration

Configuration of the bundle

```
# Default configuration for extension with alias: "chill_ldap"
chill_ldap:
    server:               # Required

        # the host of the ldap directory
        host:                 ~ # Required, Example: localhost

        # the port to reach the ldap directory
        port:                 389

        # the version of the ldap directory
        version:              3
```

```
        # Is the use of ssl required to establish connection
        use_ssl:              false

        # Is the use of startssl required to establish connection
        use_starttls:         false

        # the user to bind to dn directory. Required for searching existing users.
        bind_dn:              ~ # Required, Example: cn=user,dn=chill,dn=social

        # the user's password to bind to dn directory.
        bind_password:        ~ # Required, Example: paSSw0rD
    user_query:           # Required

        # The DN where the query is executed
        dn:                   ~ # Example: ou=People,dc=champs-libres,dc=coop

        # The query which will allow to retrieve users
        query:                ~ # Example: (&(objectClass=inetOrgPerson)(userPassword=*))

        # The attribute which will provide username (=login)
        username_attr:        cn
```

Example :

```
chill_ldap:
    server:
        # host, bind_dn and bind_password are imported from parameters.yml
        host: "%ldap_host%"
        bind_dn: "%ldap_bind_dn%"
        bind_password: "%ldap_bind_password%"
    user_query:
        dn: dc=champs-libres,dc=coop
        query: "(&(objectClass=inetOrgPerson)(userPassword=*))"
```

Configuration of the security part of chill

Simply add the following config in the firewall of the security bundle : *chill_ldap_form_login: ~*. This config is located in *app/config/security.yml*

Example of a configuration :

```
# in app/config/security.yml

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        default:
            anonymous: ~
            # enable the login check by a form, agaisnt the database
            form_login:
                csrf_parameter: _csrf_token
                csrf_token_id: authenticate
                csrf_provider: form.csrf_provider
            # enable the login check by a form, against the ldap
            chill_ldap_form_login: ~ # this is the line you should add
```

Note that, if you enable the login check by form **and** by the ldap, the password will be checked against the database **and** against the ldap. If one of them match, the login will succeed.

If you want to completely disable login check against the database, simply remove the `form_login` entry and all his options.

Command and crontab

Synchronize the database :

```
php app/console chill:ldap:synchronize
```

For getting more debug message :

```
php app/console chill:ldap:synchronize -vvv
```

You should run this command regularly (using crontab or systemd timer) to synchronize ldap and database automatically.

3.9 Your bundle here ?

The contributors still do not have a policy about those bundle integration, but we would like to be very open on this subject. Please write to us or open an issue.

CHAPTER 4

Let's talk together !

Subscribe to the dev mailing-list to discuss your project and extend Chill with the feature you need!

- The dev mailing-list
- Read the archives of the mailing-list

CHAPTER 5

Contribute

- Issue tracker You may want to dispatch the issue in the multiple projects. If you do not know in which project is located your bug / feature request, use the project Chill-Main.
- The dev mailing-list

Source code is dispatched in multiple bundle, to improve re-usability. Each bundle has dependencies with other chill bundle, but the developer take care that bundles may not be installed if the user do not need it.

CHAPTER 6

User manual

An user manual exists in French and currently focuses on describing the main concept of the software.

Read (and contribute) to the manual

CHAPTER 7

Available bundles

- Chill-standard | https://git.framasoft.org/Chill-project/Chill-Standard This is the skeleton of the project. It does contains only few code, but information about configuration of your instance ;
- Chill-Main : https://git.framasoft.org/Chill-project/Chill-Main : the main, required bundle for all the subsequent chill bundles. It contains the framework to add features (like searching, timeline, ...). It also provides the user managements (authentification and authorization) ;
- Chill-Person : https://git.framasoft.org/Chill-project/Chill-Person This is the bundle which provides the possibility to create and add a person.
- Chill-CustomFields : https://git.framasoft.org/Chill-project/Chill-CustomFields This bundle allows you to create custom fields on other bundles. It provides the framework for this features, and modify the schema according to this. It is required by Chill-Person and Chill-Report.
- Chill-Report: https://git.framasoft.org/Chill-project/Chill-Report This bundle allow to add report about People recorded in your database ;
- Chill-Activity : https://git.framasoft.org/Chill-project/Chill-Activity This bundle allow to add activities about People recorded in your database ;
- Chill-ICPC2 : https://git.framasoft.org/Chill-project/Chill-ICPC2 This bundle provides a custom fields for ICPC code (international classification for primary care)
- Chill-Group: https://git.framasoft.org/Chill-project/Chill-Group This bundle provides a way to create link between accompanyed people
- Chill-Event: https://git.framasoft.org/Chill-project/Chill-Event This bundle provides a way to create event and associate people to event through a "participation"
- Chill-Ldap: https://git.framasoft.org/Chill-project/Chill-Ldap Allow to synchronize the database with a ldap directory.
- Chill-ONEStat : https://framagit.org/Chill-project/Chill-ONEStat Provide statistics for the Belgian one "Centre de vacances" and "Ecoles de devoir".

You will also found the following projects :

- The present documentation : https://git.framasoft.org/Chill-project/chill-documentation
- The website https://chill.social : https://git.framasoft.org/Chill-project/chill.social

And various project to build docker containers with Chill.

CHAPTER 8

TODO in documentation

Todo

Describe the entities provided.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/bundles/activity.rst, line 22.)

Todo

describe entities provided by person bundle

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/bundles/person.rst, line 22.)

Todo

The documentation about report is not writtend

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/bundles/report.rst, line 19.)

Todo

Waiting for a link between our api and this doc, we invite you to read the method signatures here

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/development/access line 125.)

Todo

Continue to explain the export framework

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/development/export line 136.)

Todo

We should write a section on “how to update your installation”.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/index.rs
line 61.)

Todo

We should write a section on "what are the concepts of chill" and explain what is a bundle, why we should install it, how to find them, ...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/index.rs
line 65.)

Todo

Add description of the bundle

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/install_
line 27.)

Todo

the section "Install production webserver" must be written. Help appreciated :-)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/install_
line 14.)

Todo

the section "Uninstall Chill" must be written. Help appreciated :-)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/uninstal
line 14.)

Todo

the section "Uninstall the docker database" must be written. Help appreciated :-)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/uninstal
line 23.)

Todo

the section "Uninstall the application" must be written. Help appreciated :-)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/chill/checkouts/latest/source/installation/uninstal
line 32.)

CHAPTER 9

Licence

The project is available under the GNU AFFERO GENERAL PUBLIC LICENSE v3.

This documentation is published under the GNU Free Documentation License (FDL) v1.3

C