# Intro

This building block will add the possibility to manage

1. Contact persons on company level
2. Manage employees
3. Manage users

All these functions are concentrated around the contact table.

# Contactpersons

A contactperson is a “contact” record, connected to a company. In real word: a contactperson can be the contactperson for billing or sales at the company.

In the tab contacts of the company view, contacts can be added to the company.

Adding a contact is done by filling the form with these parameters

|  |  |
| --- | --- |
| Firstname | Textfield |
| Initials | Textfield |
| Infixes | Textfield |
| Lastname | Textfield |
| Gender | Selectbox (Male, Female, Unknown), default Unknown (save as M, F, U) |
| Language code | Selectbox with languages (see below) |
| Phone number | Textfield |
| E-mail | Textfield, see below |
|  |  |
|  | [SAVE] [SAVE AND NEW] (see below) |
|  |  |

## Languages

Languages is an array which we may use throughout the application. A function like “getLanguageOptions()” would be very nice to have which can be reused in other places. This function call will return an array with the possible language. For now limit it to NL and EN. Default is NL. The language option for a contact defines, if we are going to communicate with a person using the system, in which language the communication should be done.

## Phone number and email

The phone number and email are a “multi” table connection to the contact table. In the wireframed views below is shown how visually the phonenumbers and emails are entered. Upon clicking “add” a new blank row is inserted. If left empty when saving the contact form, empty phonenumber/email lines are not stored.

Phone numbers and email are defined with a “type”. Use a reusable function to get the possible options for email type and phone type. Like: getPhoneTypeOptions() and getEmailTypeOptions(). Both function return an array with the possible options.

For phonenumbers we have the following options:

* office
* private
* cellphone

For email we have the following options:

* office
* private

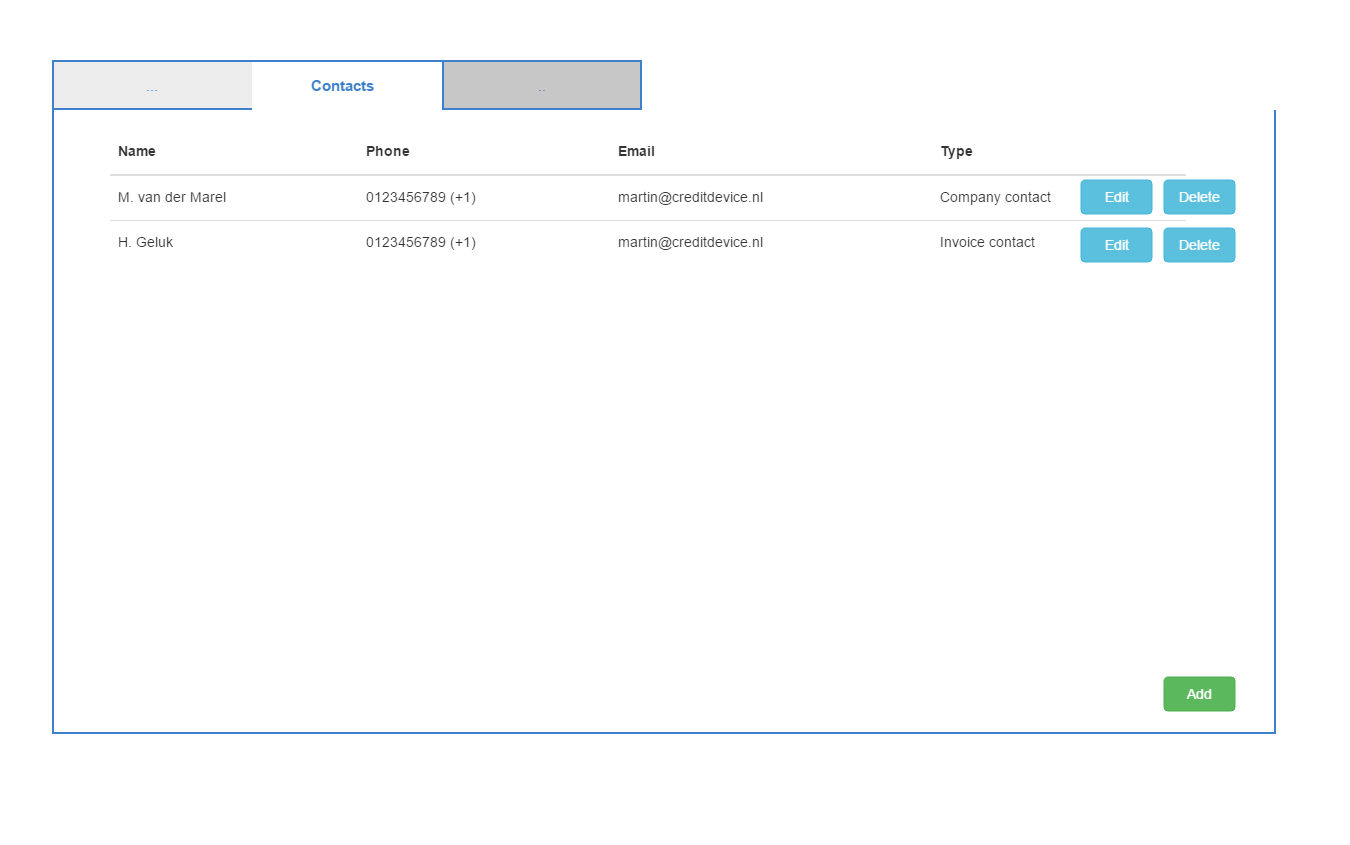
Like other labels, make sure these values are translatable in the front-end (like, the office value has a label “TYPES\_PHONE\_OFFICE” which then translates to “Office” in English and “Kantoor” in Dutch etc.

## Save buttons

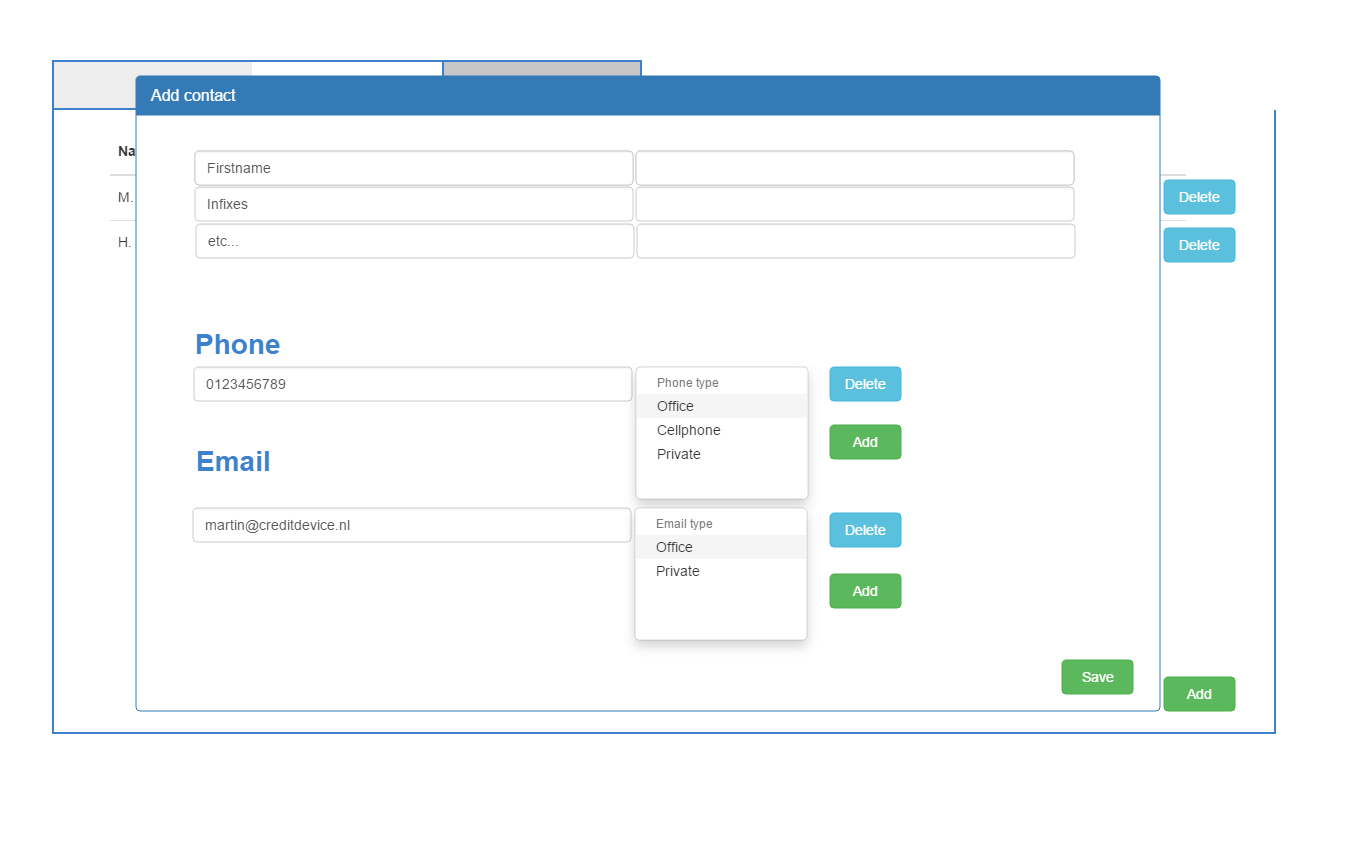
Save button will save the contactperson to the company and returns the user to the tab of contacts in the company view. Save and new will store the contactperson and forwards the user to another empty contactperson form, to allow the next contactperson to be entered.

## Wireframed

In wireframes, it looks like this



The grid above displays all connected contacts to the company (in the company edit page)



Adding a new contact opens a modal view. This modal view overlays over the main window. This is a new front-end principle we do not have at the moment. It is not required to have a draggable modal, bootstrap has some options for modal dialogs.

# Roles and permissions

With roles and permissions we like to achieve that certain users are able/unable to perform certain operations inside the application. Like: being able to add a user should only be possible for administrators.

We now define: 2 roles

* admin
* user

Default role for a new user is “user”

We want to disable some front-end elements and some api-features for the “user” role.

The value of the role for a user is stored as “user” or “admin” in the table, we do not need a role table for this.

I have prepared a work-in-progress class in the helper directory, called Permission (git pull to get this addition). Which holds a permission tree per role (now very basic, but you get the idea). With the hasPermission() function it should be possible to easily check if a user has permission to do something in the API or the front-end.

This can be done in various manners. Like so (this is pseudo code, not written for Laravel):

If($permissionInstance->hasPermission(‘COMPANY\_LIST’){

// show the link of company list in menu item

}

Or

$templateInstance->setVar(‘permissions’, $permissionIntance->getPermissions());

And then in the template test if the permission code is present to render a delete or add button

The permission tree I have created right now is self explaining. You can add the permission check around the application where the permission-code is applicable.

Important: If a user is not allowed to delete/add a company or something, this should also generate an error in the API. Here, it would be handy to have a generic this-action-is-not-allowed method to which you can redirect from another function. Like this:

If(!authcheck()){

return $someinstance->apiNotAllowed($params)

}

…. Do normal business here

The apiNotAllowed function would do something like this

Public function apiNotAllowed($params)

{

// return or echo

Return {“error”: etc}

}

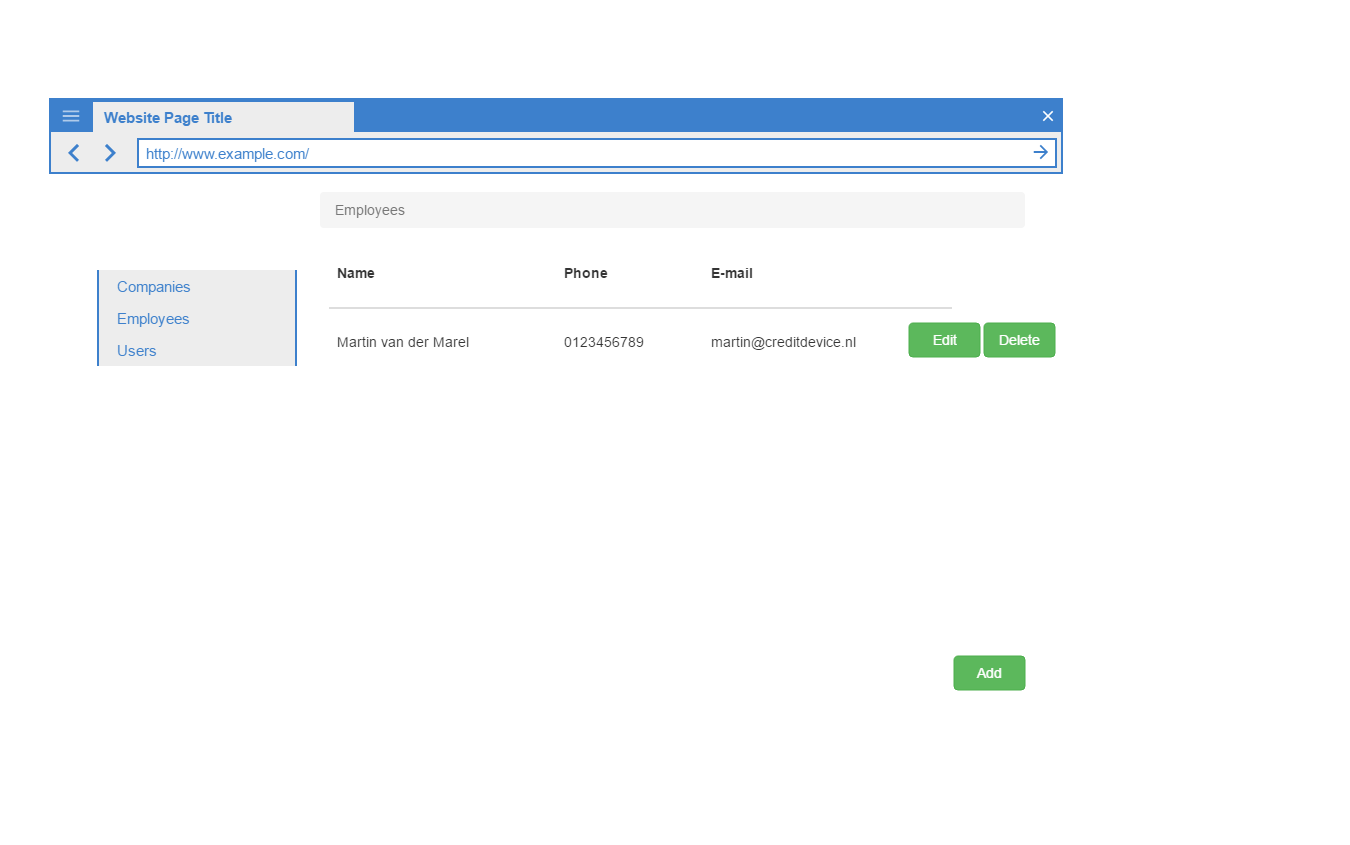
Purpose of this is that you have to write only one or a couple of error handlers and you can reuse the response they should give. When we want to change the error-repsone, we only have to do that in one place, instead of many places.

Same principle I’d like to have implemented in the controller (HTTP) requests, where a user is redirected to a not-allowed page if the user tries to access a page or function which the user cannot perform. I’m quite sure Laravel has built-in options for this that you can use (check documentation).

# Employees

Employees are defined outside the scope of a company. An employee is a record which can be used to create a user, or to connect to a company as a “company owner”.

Goal: create / edit remove employee, show list of all employees for distribution partner



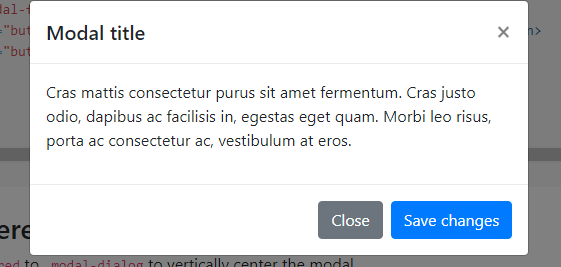
The grid view shows a list (paginated) of all employees currently active (not-soft deleted).

According to the user permissions (see previous chapter), a logged in user can see this menu option and list, can add or delete an employee

## Deleting

Upon clicking the delete button, a warning dialog is shown. Please think about a re-usable warning dialog system (from front-end javascript perspective). See bootstrap for warning/modal dialog standards.

Example:



Where the buttons in the warning dialog are “cancel” and “ok”

Cancel cancels the operation, “ok” proceeds the operation.

The title and the message inside the dialog should be variable.

We want to reuse this principle in other areas where we want to let the user confirm the action.

### Add employee

The form is similar to the “company contact” form. It contains the same fields. If possible, re-use the form of the company contact here so that we maintain one template for both situations (with exceptions that can be made inside the form for either an employee or a company contact).

Remember: an employee record does not have a relation with the company\_contact table, only the distribution\_partner\_id is stored at the contact table.

# Users

Goal: create / edit / remove user, show list of all users for distribution partner

The same logic of employees apply for the users list and create, remove, update functions.

The form allows to set a username (min length is 6 positions), a password (generated automatically) and displayed one, one cannot review the password once the create-form was completed.

The form contains a “role” selectbox, here one can select from the possible role options. Again, create a function in a reusable manner to get the role options so that it can be reused in other places in the application (getUserRoleOptions() or similar).

## Api key

Optional, for a user a API key can be generated. There should be a generate function for this, this generates a “GUID” string, which is the API key.

# Forgot password function

Now we can add users, it is useful to get the “forgot password” function operational. I think this is a standard function of Laravel, allowing one to send an email with a link to restore the password.

Be aware, that, if you followed the build instructions for your vagrant box, every mail from within the application (localhost) will be catched by the program “Mailhog”. Mailhog allows you to catch al outgoing e-mails from the application, without sending it to the real receiver. You can access this box via <http://localhost:8026/> (see vagrantfile for the actual portnumber which is specified there):

config.vm.network "forwarded\_port", guest: 8026, host: 8026 # mailhog