

# 11

## Creating an Application

*So far we've been quite website-centric: Creating pages, filling them with content and providing some add-ons. With the previous chapter's forms, we started to take it into a more advanced, application-centric direction. In our example, the data stored in the database wouldn't be directly displayed in the frontend again. Other, well-known examples would be Twitter as a one-page application, and CNN.com, which is mainly a website, but has reader logins, management interfaces and an API, and many more.*

In this chapter we'll keep going into this direction, but take some bigger steps:

- ◆ Retrieving values from our member card form definitely needs an improvement—at the moment we need to manually query the database for that
- ◆ Now that we've gathered the contact details of our customers, we should make some use of them—we'll start sending newsletters

Time to use some more features of SilverStripe to get our "application" going...



The distinction between website and application is definitely fuzzy; some wouldn't make it at all. In making the distinction, we are taking into account whether something is mainly based on pages or goes beyond that—requiring more application logic to drive it.

## Using the Model Admin class

Using DBPlumber (SilverStripe's own database frontend in the CMS, covered in *Chapter 4, Storing and Retrieving Information*) for finding the latest member cards is both inconvenient and dangerous. Imagine accidentally dropping the whole table with two bad clicks. That's also the reason why non-admin users can't use DBplumber by default.

To solve that problem, there's Model Admin—an automagically, on-the-fly generated admin interface that is tightly integrated into the CMS.

### Time for action – managing member cards the easy way

So how do we actually accomplish that?

1. In the Membercard class of the previous chapter (in `mysite/code/Membercard.php`, but you already know that) add the following statements:

```
public static $singular_name = 'Membercard';
public static $plural_name = 'Membercards';

public static $searchable_fields = array(
    'ID' => array(
        'field' => 'TextField',
        'filter' => 'PartialMatchFilter',
    ),
    'FirstName',
    'Surname',
    'City',
    'Zip',
    'MustCreateCard',
);

public static $summary_fields = array(
    'ID',
    'FirstName',
    'Surname',
    'Email',
    'MustCreateCard',
);

public function getCMSFields() {
    $fields = parent::getCMSFields();
    $fields->addFieldToTab(
        'Root.Main',
```

---

```

        new TextField('FirstName', 'First name', '', 32),
        'Surname'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('Surname', 'Surname', '', 32),
        'Birth'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('Address', 'Address', '', 64),
        'Zip'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('City', 'City', '', 32),
        'Phone'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new EmailField('Email', 'Email', '', 64),
        'MustCreateCard'
    );
    return $fields;
}

```

2. Rebuild the database for the settings to take effect.
3. Create a class `MembercardAdmin` with the following content:

```

class MembercardAdmin extends ModelAdmin {

    public static $managed_models = array(
        'Membercard',
    );

    public static $url_segment = 'membercard';

    public static $menu_title = 'Membercard';

}

```

4. Reload the CMS and the new menu item **Membercard** will appear in the upper navigation panel.
5. Open it up and click on the **Search** button, without filling in any information. By default, no records are shown, but if you leave the search fields empty, all database records will be found. Once you've added some data (we'll come to that in a minute), you should see something like the following screenshot:

The screenshot displays the 'Membercard' application interface. The top navigation bar includes 'Pages', 'Files & Images', 'Comments', 'Reports', 'Security', 'DB Plumber', 'Membercard', and 'Help'. The 'Membercard' section is active, showing a 'Search Results' table with two entries. The left sidebar contains sections for 'Add', 'Search', and 'Import'.

**Membercard**

**Add**

Create 'Membercard'

**Search**

ID:   
First Name:   
Surname:   
City:   
Zip:   
Create Card: (Any)

[Select result columns...](#)

[Clear Search](#)

**Import**

[Show Specification for Membercard](#)

Clear Database before import

**Search Results**

Displaying 1 to 2 of 2

ID	First Name	Surname	Email	Create Card	
1	<a href="#">Phillipp</a>	<a href="#">Krenn</a>	<a href="mailto:phillipp@test.com">phillipp@test.com</a>	1	✘
2	<a href="#">Other</a>	<a href="#">User</a>	<a href="mailto:someone@test.com">someone@test.com</a>	1	✘

6. On clicking on an entry you'll be taken to the following screen:

## What just happened?

All the changes that we've added in the Model are to configure the Model Admin CMS page.

## Singular and plural name

We start off by defining the singular (`$singular_name`) and plural name (`$plural_name`) for the view, so the labels are correctly named.

## Searchable fields

`$searchable_fields` defines the fields that should be available to the search function. When choosing which fields to make available, it's best to think about what you need to accomplish—for example:

- ◆ Find people by name
- ◆ See how many people are from a specific region—sharing the same ZIP code
- ◆ Get an overview of how many cards you've already created and how many are still missing

Except for the `ID` attribute, where we add some formatting options for the CMS, everything is nice and easy. If you added `ID` without the array defining further details, instead of a search box you'd get a drop-down list with all available values. This is fine when you only have a few dozen entries, but isn't efficient for more than that. By specifying that you want a `TextField` matching exact and partial values (`PartialMatchFilter`), you can freely enter any search term like in the other fields.

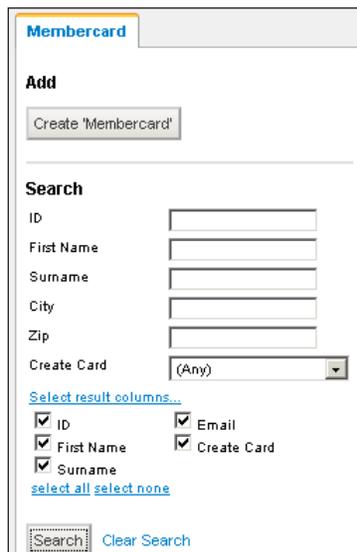
As our site is so good, we're expecting loads of people to sign up—so the drop down is no good for us. If you're a little less ambitious, you can easily switch to the other method.

After pressing the **Search** button the available data objects are searched based on your input. The more details you enter, the more limited the search results get. Leaving all fields empty fetches all data objects available.

## Summary fields

The `$summary_fields`, another array, are the fields shown in the overview after searching. You can also sort the table entries by the available columns. Often you'll display most of the searchable fields in the overview.

You can disable available values in the web interface but not add new ones:



The screenshot shows a web interface for 'Membercard'. It has a header 'Membercard' and a sub-header 'Add'. Below 'Add' is a button 'Create 'Membercard''. Underneath is a 'Search' section with input fields for 'ID', 'First Name', 'Surname', 'City', and 'Zip'. There is also a 'Create Card' dropdown menu set to '(Any)'. Below the search fields is a link 'Select result columns...' followed by a list of checkboxes: 'ID', 'First Name', 'Surname', 'Email', and 'Create Card'. At the bottom of the search section are links 'select all' and 'select none'. At the very bottom are 'Search' and 'Clear Search' buttons.

## Overwriting the CMS Field definition

Just like in the frontend we're limiting the number of characters according to the database definition with the method `getCMSFields()`. This is optional and can be left out, if you're satisfied with the default behavior.

We're not expecting too many edits in the Model Admin, but we can still avoid some potential problems. Define the limited length of text fields like you would for adding them on a regular Page. The major difference is the third argument of `addFieldToTab()`, that defines which field should follow after this element.

So whenever these fields are editable in the CMS, this definition will be used instead of a default one, not including a limit. Unfortunately text field size limits are currently not automatically picked up by SilverStripe.

## Defining the Model admin

What we need to do in the `MemberCardAdmin` class is very easy now:

- ◆ You need to define which model or models should be managed by the admin view.
- ◆ Additionally you need to select an URL for your view, being appended to `/admin`. If the frontend page is available under `/membercard`, `/admin/membercard` would be a good choice for the model admin.
- ◆ Finally (even though the order is up to you) you need to find a meaningful name for the top navigation—in this example an easy choice again.

## Final note

When we defined the underlying data object, we said that we'd take another look as to why we needed a default value for the attribute `sex`. So, click on the **Create 'Membercard'** button in the Model Admin view. For enumerations the first array element is always the default selection. Having added the "-" we won't accidentally set the wrong sex or add one if it's unknown. Just one more little trick to make our frontend more user-friendly.



If you're familiar with other frameworks, you may have seen some automatically generated code based on the Model. SilverStripe doesn't take this approach. In the frontend such code is generally not production-ready and rewriting it is often more work than starting from scratch. But for admin purpose it's perfect, and here auto-generation really shines—we don't need to change anything when the underlying Model changes. The drawback is that customizing the admin interface is a bit trickier, and can be even more than changing generated code.

## Permissions

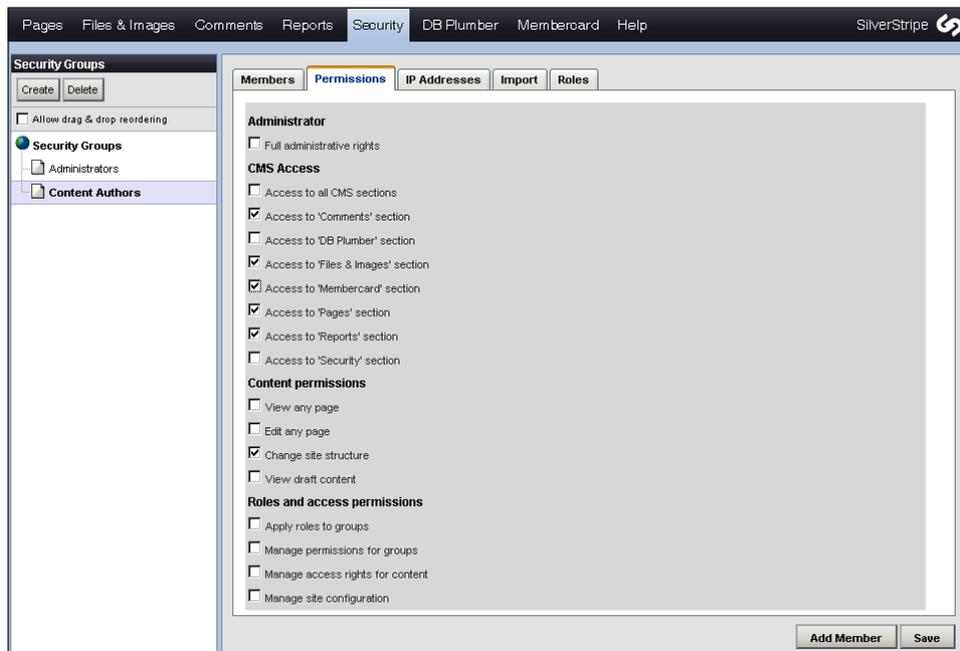
Something we've mostly ignored so far is the security of our backend. Let's integrate that aspect into our code.

## Time for action – limiting the permissions on member cards

1. First off, add the following code to the `Membercard` class. Note that it's not final yet, we'll extend it throughout this section.

```
public function canView($member = null){
    return true;
}
public function canEdit($member = null){
    return true;
}
public function canDelete($member = null){
    return true;
}
public function canCreate($member = null){
    return true;
}
public function canPublish($member = null){
    return true;
}
```

2. If you haven't already done so, create a new member of the **Content Authors** group under `/admin/security`. Allow members of this group to access the new member card section on the **Permissions** tab: **Access to 'Membercard' section**. This option is automatically added when creating a Model Admin subclass, but not activated by default:



3. If you don't activate this permission, **Content Authors** simply won't have the navigation element in the top navigation bar and also can't access the page directly through its URL.
4. Now logout through the bottom right link, then log in as a content author. Alternatively open up a second browser; not an instance of the same browser you're currently using, however, as the session would be shared between them. In the second browser, log in as the new user. Comment out the five methods starting with "can" (`canView()`, `canEdit()`, and so on) for temporary demonstration purposes. Go to the member card page in the backend—it should look like this:

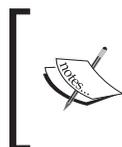
Note that you can't create or import new objects, nor can you edit or delete existing ones—the buttons are simply not there. Also the **Security** and **DBPlumber** tabs are not visible, as only admin users may use them (in the default configuration).

5. As soon as you activate the five methods again, members of the **Content Authors** group have full control over member card objects—just like admin users. You can only activate or deactivate the view in the CMS; the specific permissions must be set on the code level.

## What just happened?

The methods starting with "can" follow a special convention. As you've already seen, they manage permissions in the CMS. Depending on the current user, the return value of these methods is either `true` (allowing the action) or `false` (disallowing it).

Always returning `true` simply allows anyone able to view the Model Admin interface to execute the method name's action. The specific actions should be pretty obvious. `canCreate()` defines whether a user may create new objects, but not edit existing ones—that is handled by `canEdit()` and so on.



`canPublish()` is only useful for elements of the SiteTree. Many content editors can create new pages, but only a few selected ones could publish them online. For the sake of completeness we've included this option here, even though it's not really doing anything.

In case you need more fine-grained security settings, you need to know how SilverStripe organizes users and their permissions.

## Security roles

You can't define settings for specific users, only groups of them—the so-called roles. Once you think about it, this makes perfect sense. Sooner or later you'll need to have a new user with the same permission set as an existing one. Configuring each one individually is both time-consuming and error-prone. By using roles, you'll never be stuck with such a limitation.

Roles can be nested. As sub-elements inherit the rights of their parent, you can add permissions thorough inheritance but not remove them. Having more powerful users inherit from less powerful ones may seem counter-intuitive, but it works very well in practice. Let's take a look at an example (though not from our installation):

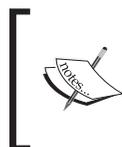
- ◆ Administrators
- ◆ Content Authors
  - Member Editors
- ◆ Members

On the first level, we've the two default roles and we've also added a new one for `Members`—not having any privileges. For `Content Authors` we've created a new sub-role `Member Editors`. Only this role (besides the almighty `Administrators`) is allowed to edit member cards. Regular `Content Authors` can only view them.

## Permission codes

Actual permissions are set by assigning permission codes to specific roles specific—these codes are pre- or self-defined strings. For example `ADMIN` is a pre-defined code, by default given to admin users. By convention the code should be uppercased.

In order to define a permission code yourself, you need to implement the interface `PermissionProvider`. It requires you to write a method `providePermissions()` which returns an array with your own codes. These are then available on the **Permissions** tab of the user groups. Don't worry, we'll try that out in a moment.



An interface (in object-oriented programming) is very similar to an abstract class. The major difference is that all interface methods are implicitly abstract. You define which functions must be implemented. There is no default implementation which can either be used or overwritten.

## Permissions in the controller

We've already used `$allowed_actions` in the controller. So far we've only used them in the most basic way, but there is more. Take a look at the following example:

```
public static $allowed_actions = array(
    'allowed',
    'alsoallowed' => true,
    'adminonly' => 'ADMIN',
    'advanced' => '->canAdvanced',
);
```

- ◆ The methods `allowed()` and `alsoallowed()` could be accessed by anyone.
- ◆ `adminonly()` could only be accessed by users with the `ADMIN` permission code.
- ◆ The final function `advanced()` could only be accessed by users for whom the method `$this->canAdvanced()` returns `true`. You'll need to create this check method yourself—starting its name with `can` is not required, but it's a sensible convention.

## Permissions in the model

We've already seen the different `can` methods. You can also check for specific permission codes: `Permission::checkMember($member, 'ADMIN')`, for example, checks for admin rights. Or you might implement any other check useful for your requirements.

## Using our powers

Now that we know so much about permissions, let's make use of them in our example.

### Have a go hero – set up roles and users

We'll extend our very basic example to use the roles that we've already discussed. We'll also need to define our own permission code for the system to be useful.

### Creating users

Start off by creating the roles we've discussed above in `SilverStripe` and add one user to `Member Editors`. Due to inheritance, users of a sub-role are also part of the super-role, so `Content Authors` should now have two participants. You did create the one in `Content Authors` itself before, right?

### Creating permission codes

Implement the interface by changing the class definition:

```
class Membercard extends DataObjectDecorator implements
PermissionProvider {
```

And add its required method in that class:

```
public function providePermissions(){
    return array(
        'MEMBER_EDITOR' => 'Create, edit and delete members',
    );
}
```

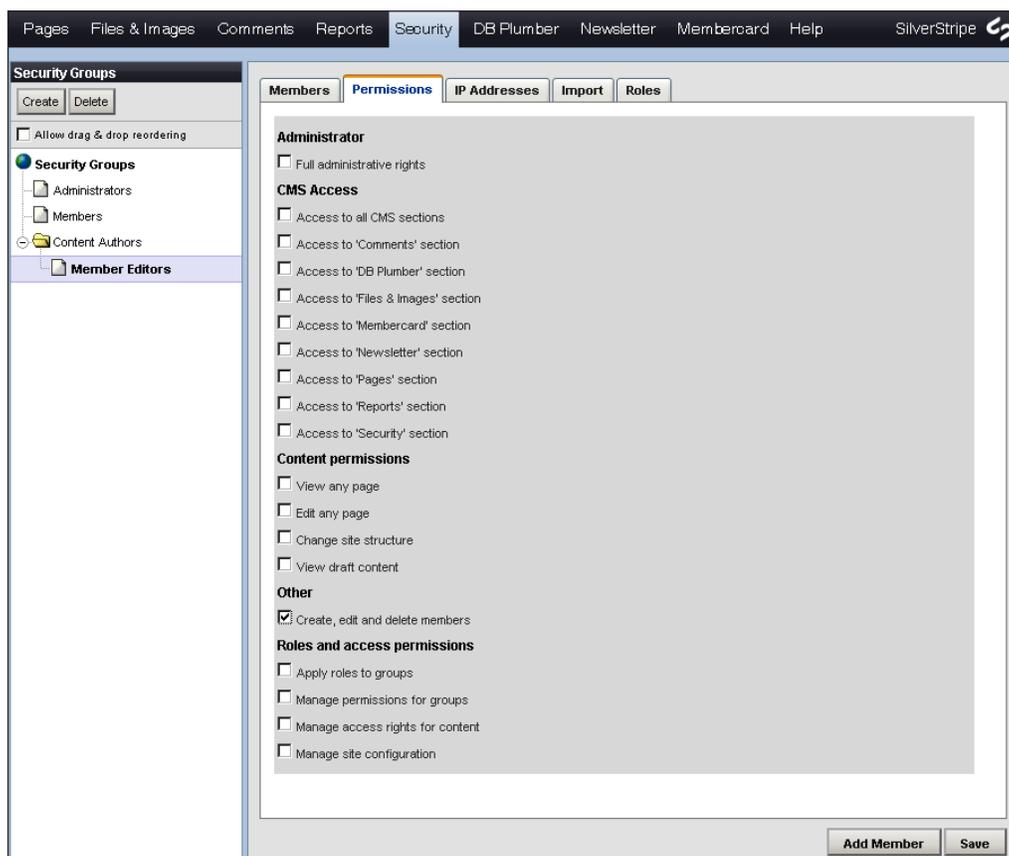
Within the array, you're first defining the code and then providing a labeling text for the CMS.

In general you would define the permission code within the class where it's used. However you're not bound to it; you could use `Page_Controller` for example. This would change your internal organization, keeping all codes in a single place rather than distributing them over various classes.

 Both approaches have their advantages and disadvantages, there isn't a clear winner. Go with the one you find more "natural" and you should be fine.

## Assigning permission codes

Now you can reload `/admin/security/` and the Permissions tab should include a new entry—the one we've just defined. Activate it for `Member Editors`.



 All other boxes are unchecked. These permissions are inherited from `Content Author`; you will need to check them to actually grant them to `Member Editors`.

Additionally leave the **Access to 'Membercard' section** activated for `Content Editors`. Also check that `Members` don't have any permissions at all—they shouldn't by default, but check just in case.

## Checking permission codes

Of the five methods, we can remove `canPublish()` as there isn't anything to publish here—we're not on a regular page.

The remaining four methods look like this:

```
public function canEdit($member = null){
    if(!$member){
        return false;
    }
    return Permission::checkMember($member, 'MEMBER_EDITOR');
}
public function canDelete($member = null){
    return $this->canEdit($member);
}
public function canCreate($member = null){
    return $this->canEdit($member);
}
public function canView($member = null){
    return $this->canEdit($member);
}
```

The `$member` parameter is optional. But as we're logged into the CMS when viewing this page, there should be a valid user object, otherwise we're refusing the action.

Depending on the permission settings of the current user, we're returning `true` or `false`. Admin users have all permissions by default, so there's no point in doing another check for them. For content authors we've specifically set the `MEMBER_EDITOR` permission.

As we're using the same settings for all four actions, we're simply reusing `canEdit()` each time. And that's it—log in with the different user accounts and see the difference. While these settings could get a lot more complicated, we now have a good starting point for any enhancements you may need to add to your own projects in the future.

## Internal organization

Before finishing off this section, let's take a look at how SilverStripe implements users internally, specifically the `Member` class. We'll need this information soon.

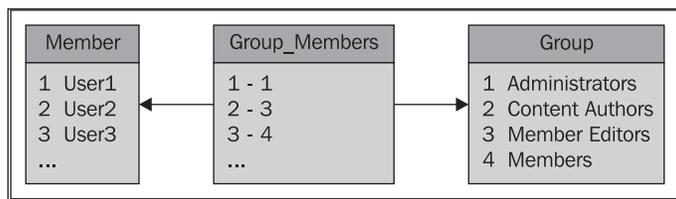
As you may have guessed, there is a table for the `Member` class, holding all the information of the individual members. Then there's a `Group` table—containing the title shown in the CMS, the hierarchy to children groups, a unique code and some more fields. It should look like the following image in **DBPlumber** :

The screenshot shows the SilverStripe CMS interface. On the left is a navigation menu with 'Group' selected. The main area displays a table titled 'Table Group' with the following data:

ID	ClassName	Created	LastEdited	Title	Description	Code	Locked	Sort	IPRestrict	HTMLEdit	ParentID
1	Group	2010-11-07 07:23:32	2010-12-20 01:56:37	Content Authors		content-authors	0	1			0
2	Group	2010-11-07 07:23:32	2010-11-07 07:23:32	Administrators		administrators	0	0			0
4	Group	2010-12-23 23:39:17	2010-12-20 02:40:27	Edit Members		edit-memk	0	0			1
5	Group	2010-12-23 23:44:11	2010-12-20 23:44:18	Members		members	0	0			0

We'll use the code later to find the right user group and add members to a specific code. This is done with the `Group_Members` table, join table (see *Chapter 4, Storing and Retrieving Information* for more on these).

Here is a schematic representation of this: Member with the ID 1 is connected to the Group 1 (an Administrator), while Member 2 is a Member Editor, and so on.



And there's also the table `MemberPassword`, storing the password history and how they are encrypted—each row belonging to exactly one user object. The current user passwords are stored in `Member.Password`.

## Preparing to send newsletters

Everything seems to be perfect, but now that our customers are providing their contact details, it would be pretty handy to send them newsletters.

Luckily there is a module just for that purpose, <http://silverstripe.org/newsletter-module/>. But there's one little problem: It only works on the default member class. We already started thinking about different implementation methods in the previous chapter—now it's time to switch from our own data object to a data object decorator.



Please note that this module is less powerful and scalable than dedicated mailing applications such as <http://www.mailchimp.com> and requires a properly configured server. While it's not officially maintained by SilverStripe, it has been put together by some of the core developers and integrates very well into the rest of the system. Specifically you can send newsletters to existing members and you can use SilverStripe templates to create the general layout.

## Decorating the member class

This is not too hard, but there are still some points to watch out for, so let's take a look at the code.

### Time for action – switching to a data object decorator

1. First we need to change our member card quite a bit—changes are highlighted:

```
<?php
```

```
class Membercard extends DataObjectDecorator implements  
PermissionProvider {
```

```
    public function extraStatics(){  
        return array(  
            'db' => array(  
                'Sex' => "Enum('-', male, female', '-')",  
                'Birth' => 'Date',  
                'Address' => 'Varchar(64)',  
                'Zip' => 'Int',  
                'City' => 'Varchar(32)',  
                'Phone' => 'Int',  
                'MustCreateCard' => 'Boolean',  
            ),  
            'searchable_fields' => array(  
                'ID' => array(  
                    'field' => 'TextField',  
                    'filter' => 'PartialMatchFilter',  
                ),  
            ),  
        );  
    }
```

```
        'FirstName',
        'Surname',
        'City',
        'Zip',
        'MustCreateCard',
    ),
    'summary_fields' => array(
        'ID',
        'FirstName',
        'Surname',
        'Email',
        'MustCreateCard',
    ),
);
}

public function updateCMSFields(&$fields){
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('FirstName', 'First name', '', 32),
        'Surname'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('Surname', 'Surname', '', 32),
        'Birth'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('Address', 'Address', '', 64),
        'Zip'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new TextField('City', 'City', '', 32),
        'Phone'
    );
    $fields->addFieldToTab(
        'Root.Main',
        new EmailField('Email', 'Email', '', 64),
        'MustCreateCard'
    );
}
```

```
public function providePermissions(){
    return array(
        'MEMBER_EDITOR' => 'Create, edit and delete members',
    );
}

public function canEdit($member = null){
    if(!$member){
        return false;
    }
    return Permission::checkMember($member, 'MEMBER_EDITOR');
}

public function canDelete($member = null){
    return $this->canEdit($member);
}

public function canCreate($member = null){
    return $this->canEdit($member);
}

}
```

2. Next we need to register our new decorator in the configuration file:

```
DataObject::add_extension('Member', 'Membercard');
```

3. The MembercardAdmin class needs to change its underlying model (\$managed\_models) from Membercard to Member.
4. Additionally add the following statement to the class so that we can exclude users which are not part of the Members group. We don't want to manage those in the model admin:

```
public static $collection_controller_class = 'MembercardAdmin_
CollectionController';
```

5. In the same file add the following class, which we registered in the previous point:

```
class MembercardAdmin_CollectionController extends ModelAdmin_
CollectionController {

    public function getSearchQuery($searchCriteria){
        $group = DataObject::get_one('Group', "\"Code\" = 'members'");
        if($group){
            $groupId = (int)$group->ID;
        } else {
            $groupId = 0;
        }
    }
}
```

```

    }
    $query = parent::getSearchQuery($searchCriteria);
    $query->from(' ', "Group_Members");
    $query->where(
        ' "Member"."ID" = "Group_Members"."MemberID" AND' .
        ' "Group_Members"."GroupID" = ' . $groupId
    );
    $query->orderBy('"MustCreateCard" DESC, "Created" ASC');
    return $query;
}
}

```

- 6.** In `MembercardPage::duplicateEmail($email)` be sure to change the following line from `Membercard` to `Member`:

```

$membercard = DataObject::get_one(
    'Member',
    "\"Email\" = '$SQL_email'"
);

```

- 7.** Finally rewrite the following method:

```

public function sendemail($data, $form){
    if($this->duplicateEmail($data['Email'])){
        $form->addErrorMessage(
            'Email',
            'The given email address <b>' . $data['Email'] .
            '</b> already exists',
            'error'
        );
    }
    Session::set('FormInfo.Form_Form.data', $data);
    Director::redirectBack();
    return;
} else {
    $member = new Member();
    $fields = array(
        'Sex',
        'FirstName',
        'Surname',
        'Birth',
        'Address',
        'Zip',
        'City',
        'Email',
    );
}

```

```
        'Phone',
    );
    $form->saveInto($member, $fields);
    $member->MustCreateCard = true;
    $member->Birth = $data['Birth'];
    $member->write();

    if(!$group = DataObject::get_one(
        'Group',
        "\"Code\" = 'members'"
    )){
        $group = new Group();
        $group->Code = 'members';
        $group->Title = 'Members';
        $group->write();
    }
    $group->Members()->add($member);

    Mailer_Controller::sendemail($data, $form);
}
}
```

8. That's it—rebuild your database and flush the cache.
9. Now all the information of your form should be saved in the default `Member` table. Try it out!



If you already have live data in your `Membercard` table, you will need to migrate it manually. With the help of DB Plumber you can export the existing table, make changes in your favorite editor and import the required data again. Unfortunately there isn't a module that does that for you at the moment, and the whole process is a little messy. Ideally you won't ever need to migrate such major changes on a live project, having selected the right approach from the start. In our example project we want to show both approaches and their differences. As you shouldn't have any production-grade data in our test site, this shouldn't be an issue after all.

From now on, we'll only use the `Member` class!

## ***What just happened?***

You already know how to add a decorator in the configuration from our custom site configuration, so we can jump right to our decorator.

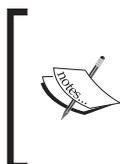
## The data object decorator

In order to understand the following steps, open up `sapphire/security/Member.php` as that's the basis we're building upon:

- ◆ We're using the method `extraStatics()` to add elements to the base class. As a rule of thumb, `public static` variables in the data object are added through this array to a decorated object. Simply put all elements into an array and make the variable names the key of an associative sub-array.
- ◆ Note that we don't need to add attributes such as `Email` as they already exist in the base class. This is also true for the index on `Email` as well as the variables `$singular_name` and `$plural_name`.
- ◆ For the definition of specific input fields, we need to change the method to a reference-based version—just like in the custom site configuration. The input definitions themselves are the same as before.
- ◆ The security-specific stuff can be left untouched. It already exists in the base class and is simply overwritten here.

## The Model Admin

You must change the managed model from `Membercard` to `Member`. However this leads to one undesired side effect: If we search for our members in the backend, we'll now find all users—including the admin and content editor users. To switch back to the previous behavior, we need to add our own collection controller.



Please note that this section gets pretty technical. On the one hand, if you don't require this feature, you can simply skip this part. On the other hand, it really shows how flexible SilverStripe is. Even though you might not need it right now, this kind of power could come in very handy in the future.

First we need to register the collection's class name and then we need to implement it. Following SilverStripe's convention, we're adding this class to the same file—just like with a page's Model and Controller.

In our own collection, we're overwriting `getSearchQuery($searchCriteria)`. This is a little more complex, but don't worry—it's not too hard:

- ◆ First we're fetching the group whose members we want to find. We're using the unique code for doing that. As we've set up a group with the name `Members`, its code is `members`.

- ◆ We're then fetching the group's ID—casting it to make sure it's a number. If there's no group with the given code, we're using 0 instead, which will find no object at all.
- ◆ Then we're taking the original search query, including the user-defined parameters of the searchable fields, and adding our own settings to it.
- ◆ We've already said that groups and members are linked together through the `Group_Members` table. The original query is only based on the classes given in `$managed_models`. So we need to add the `Group_Members` table, simply attach it to the already existing element with `->from()`. Double quoting it is not strictly necessary, but the core files generally follow this convention so it's a good idea to stick to it.
- ◆ Next we're adding the `->where` clause. This limits the results to members of the given group, but we won't go into the finer details of SQL here. If your knowledge of this area is a bit rusty, please take a look at a database manual.
- ◆ Then we're ordering the results, first showing the entries where the card hasn't been created yet. Inside of the two resulting groups (card already created or not), we're showing the oldest records first—using SilverStripe's automagical `Created` attribute, which exists for every object.  

This is helpful for finding out who has been waiting the longest for their card, but you may want to use a different approach.
- ◆ Finally we're returning the original query enriched with our custom settings.

## The page

So far we've totally ignored one crucial aspect of our redesigned member card: How do we actually add new users to a specific group?

We start off like before—creating a new object, adding all the attributes, and saving it. But then we're checking if the desired group exists, using its unique code. If it doesn't already exist, we're creating it. This isn't necessary in our case, as we've already created the group in the CMS manually, but you may need it in the future.

Finally we need to link members and groups together. All you need is `$group->Members()` `->add($member)`; you don't need to set and write values, SilverStripe automagically does that for you as it knows that `Group` and `Member` have a many-to-many relationship. Simply adding one object to the other accomplishes what we want.

## Optimizations

Before continuing with the newsletter itself, let's take a look at two minor optimizations. The code is already working fine, but there is one minor cleanup and one performance enhancement we can easily add.

### Hooks

Just like many other systems, SilverStripe can employ hooks. These are interfaces that allow you to add code before or after certain events.

Data objects currently support `onBeforeWrite()`, `onBeforeDelete()`, `onAfterWrite()`, and `onAfterDelete()`. You'll have guessed it, you can insert custom code before or after saving or deleting an object.

This is especially handy if you're using an object in multiple places and want to inject the same code whenever it's being used. You could call a dedicated method for that purpose, but hooks are automatically injected so you can't forget to call them.

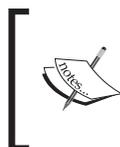
While this is not the case in our example, there's still a sensible use case. Right now we're "polluting" the method for storing new members with the code for creating their group (if required). That could be nicely done with a hook—ensuring our `Members` group is available before adding members.

But before hacking away, let's also take a look at the second optimization.

### Query caching

`DataObject::get_one('Group', "Code = 'members'")` is a pretty "cheap" query in performance terms. However it's still redundant, except for the first time it's being called. We've already covered partial caching in the template, but that is only applicable in the View. Wouldn't it be clever if we had a similar construct for queries? Actually we do. Let's look at it now.

If it's unlikely that data changes between queries (like in our example), you can cache the query result or some other expensive operation. This is done with the Sapphire class `SS_Cache`.



The cache life time you've defined for the partial caching in the View is also used for `SS_Cache`. That means you must change from development to test or live mode in the configuration file—otherwise caching won't work!

First you set up a unique cache key. The first time you access this key, you generate the result and store it in the cache. On any subsequent request you simply retrieve the cached value. We'll get to the code in a minute, which will make this much clearer.



#### Underscore methods

A similar feature is available via underscored methods. If you create a method which starts with an underscore (`public function _getValues()` for example), it will only hit the database once as the result is automatically stored. But as this approach will soon be deprecated, we won't use it, focussing instead on the future-proof version.

## Putting it into action

Our objective is clear, but it's a little tricky to set up, so let's take a look at the code.

### Time for action – optimizing

1. In the configuration file, add a new constant for our custom members' code:  
`define('MEMBERSCODE', 'members');`
2. In the class `MembercardPage_Controller` remove the creation of the group from the method `sendemail()` as we want to do that in our hook. We'll fetch the group from its own method, which we'll call both from the Controller and the Model (therefore it must be public):

```
$member->write();  
$group = $this->getMembersGroup();  
$group->Members()->add($member);
```

3. Add the method that we've just referenced in the same class:

```
public static function getMembersGroup() {  
    $cachekey = 'members_group';  
    $cache = SS_Cache::factory($cachekey);  
    if(!($result = unserialize($cache->load($cachekey))) {  
        $result = DataObject::get_one(  
            'Group',  
            "\"Code\" = '" . Convert::raw2sql(MEMBERSCODE) . "'"  
        );  
        $cache->save(serialize($result));  
    }  
    return $result;  
}
```

4. In the Membercard class add the following method:

```
public function onBeforeWrite() {
    $group = MembercardPage_Controller::getMembersGroup();
    if (!$group) {
        $group = new Group();
        $group->Code = MEMBERSCODE;
        $group->Title = 'Members';
        $group->write();
    }
    parent::onBeforeWrite();
    return;
}
```

5. That's it—when adding a new user you shouldn't notice any difference.

### ***What just happened?***

In case you're wondering where the tricky part is, let's take a closer look at the code—it definitely doesn't look like much:

- ◆ The change in the `sendemail()` method: Remove the creation of a group and simply fetch the object from a method in the same class.
- ◆ Fetching the group is now done in the method `getMembersGroup()`. The underlying database query hasn't changed. We've simply switched to the constant `MEMBERSCODE` and are also escaping it, just to be on the safe side.
- ◆ Additionally we've added some caching elements to the method `getMembersGroup()`:
  - First we need to set up a unique hash key: `$cache = SS_Cache::factory('unique');` In our example there is just a single object we want to cache, so using a fixed string is fine. If your method depends on input parameters (process a specific ID for example), you'd normally include these parameters in the cache key to make it unique.
  - Next we're trying to load the cached element: `$cache->load('unique')`
  - If there is no such element, we execute our query and save its result into our cache: `$cache->save($result)`
  - Finally we return the result, freshly fetched or from the cache, to the calling method.
  - Note that you need to `serialize()` and `unserialize()` anything that isn't a plain vanilla string. This includes our `DataObject`. But You'll get an error message if you don't stick to this rule, so you can't accidentally overlook it.
- ◆ Using `onBeforeWrite()` we're making sure that our custom group exists. Before even saving the user is just the right place for this.

- ◆ As `getMembersGroup()` is defined in `MembercardPage_Controller` and it's a static method, we don't need to instantiate it, but can access it through `MembercardPage_Controller::getMembersGroup()`.
- ◆ After fetching the group object, we're reusing the code from before these changes to create a new one, if required.
- ◆ Finally we should call `onBeforeWrite()` of the parent object. We don't want to overwrite the hook, but only extend it.

Additionally there are some architectural issues to consider:

- ◆ Why didn't we define `getMembersGroup()` in the decorator?
  - Normally you'd put this method in the Model—that's its area of operation after all. However we're only using a decorator, which hasn't been made for this purpose. So the Controller is the better choice in this case.
- ◆ Why did we make `getMembersGroup()` static?
  - Because we only need a single instance for the whole class (the member group is the same for all members). Additionally we don't need to instantiate a new `MembercardPage_Controller` object in the `Membercard` class.
- ◆ Was it a good idea to introduce the `onBeforeWrite()` method?
  - That's definitely a valid point. Currently it makes things more complicated, but if you ever want to add members inside another Controller, it will pay off as we won't need to duplicate the logic to create the group. Decide for yourself how extensible your solution needs to be without making it too complex in simple situations.

In case you're not sure that the caching is really working, here's how to check. Normally you'd simply use a URL parameter, for example `?debug_profile=1`, which shows the profiler for the current request (including our cache call). But due to the redirects it's not working for our example, as there are multiple requests. As we just want to do a quick check, we'll simply take a look at the cache files.



We've covered the location of SilverStripe's cache folder in *Chapter 6, Adding Some Spice with Widgets and Short Codes*.

Inside the cache folder there should be a file `cache/zend_cache---members_groupmembers_group`. If you open it up in a text editor, you should see something like the following screenshot. While serialized objects are hard to read, it's still obvious that our query caching is working as it should.

```

1 0:5:"Group":15:{s:9:"destroyed";b:0;s:9:"\NUL*\NULrecord";a:10:{s:9:"ClassNam
e";s:5:"Group";s:7:"Created";s:19:"2010-12-25
23:44:11";s:10:"LastEdited";s:19:"2010-12-25
23:44:18";s:5:"Title";s:7:"Members";s:4:"Code";s:7:"members";s:6:"Locked";s:
1:"0";s:4:"Sort";s:1:"0";s:8:"ParentID";s:1:"0";s:2:"ID";i:5;s:15:"RecordCla
ssName";s:5:"Group";}s:19:"\NUL*DataObject\NULchanged";a:0:{}s:11:"\NUL*\NULorig
inal";a:10:{s:9:"ClassName";s:5:"Group";s:7:"Created";s:19:"2010-12-25
23:44:11";s:10:"LastEdited";s:19:"2010-12-25
23:44:18";s:5:"Title";s:7:"Members";s:4:"Code";s:7:"members";s:6:"Locked";s:
1:"0";s:4:"Sort";s:1:"0";s:8:"ParentID";s:1:"0";s:2:"ID";i:5;s:15:"RecordCla
ssName";s:5:"Group";}s:13:"\NUL*\NULcomponents";N;s:17:"\NUL*\NULbrokenOnDelete
";b:0;s:16:"\NUL*\NULbrokenOnWrite";b:0;s:17:"\NUL*\NULcomponentCache";N;s:14:"
\NUL*\NULiteratorPos";N;s:21:"\NUL*\NULiteratorTotalItems";N;s:11:"\NUL*\NULfail
over";N;s:19:"\NUL*\NULcustomisedObject";N;s:22:"\NUL*\NULViewableData\NULobjCache";
a:0:{}s:5:"class";s:5:"Group";s:22:"\NUL*\NULextension_instances";a:1:{s:9:"H
ierarchy";O:9:"Hierarchy":7:{s:14:"\NUL*\NULmarkedNodes";N;s:16:"\NUL*\NULmarki
ngFilter";N;s:21:"\NUL*\NUL_cache_numChildren";N;s:8:"\NUL*\NULowner";N;s:17:"
\NUL*\NULownerBaseClass";s:5:"Group";s:20:"\NUL*\NULExtension\NULownerRefs";i:0;s:5:
"class";s:9:"Hierarchy";}}

```

## Sending newsletters

Now that we've done the hard work, let's get down to the fun part. Get the latest version of the newsletter module at <http://silverstripe.org/newsletter-module/> and install it, by rebuilding the database and flushing the cache. This will create many new tables, resulting in quite a colorful output.

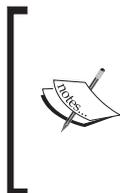
## Configuration and convention

There is only a single settings in the configuration file `mysite/_config.php`, you need to have and we've already added it in *Chapter 5, Customizing Your Installation*: `Email::setAdminEmail(EMAIL)`.

You should also have set up the constant `EMAIL` at the time. It defines the default sender address of all e-mails (or the newsletter in our specific case), but can be overridden when actually sending out the newsletter.

Secondly there's a convention to adding newsletter template files: It's either `mysite/templates/Email/` or `themes/bar/templates/Email/`—note that neither `mysite/` nor `themes/bar/` are hard-coded but dynamically set to the correct values. The folder name `Email/` could also be lowercased, but as both `Includes/` and `Layouts/` start with an uppercase letter, we'll stick to this convention.

As the newsletter template is definitely theme-specific, you should add it to the `themes/` folder. You might want to create the `themes/bar/templates/Email/` folder right away.



If your templates are not found, there are two possible reasons; First, you've forgotten to flush the template cache—after creating a template file always call `?flush=all` in your browser. If that doesn't work, you may need to move your newsletter templates from the `themes/` folder to `mysite/`. Due to a bug in some versions of the newsletter module, only the latter path is sometimes picked up.

## Templates

Now that we've defined our template folder, it's time to create the template itself. As SilverStripe's newsletter module is well integrated into the rest of the system, we can make use of the built-in template engine we already know. So let's get going!

## Placeholders

The following placeholders are available:

- ◆ `$Body`: The content of the e-mail, defined in the CMS.
- ◆ `$Member`: Our extended member object, including all the attributes such as `$Member.FirstName`, `$Member.Birth`, and so on.
- ◆ `$Subject`: Subject of the e-mail.
- ◆ `$From`: Sender of the e-mail.
- ◆ `$To`: Recipient of the e-mail.
- ◆ `$UnsubscribeLink`: A link to unsubscribe from the newsletter. We don't want to spam our customers, right?

## Creating a template

Before starting your design, a word of caution. The HTML support in e-mail applications varies considerably and is often pretty bad. In particular there are two points to consider:

- ◆ Web-based e-mail services such as GMail generally strip out the `<html>`, `<head>`, and `<body>` tags. After all, they're embedding your mail into their site, so they can't use that information. And they don't want to risk your styles colliding with their own settings. Therefore you can't embed your CSS, but feel free to add it inline (to the individual HTML tags). It will make you feel dirty, but at the moment there is no way around it.

 We didn't do that in our previous `Email.ss` template. But as that was just for internal notifications, we don't need to make it as rock-solid as the public newsletter.

- ◆ Desktop applications such as Microsoft Outlook only support a subset of current HTML and CSS standards. That's the reason why many newsletters are still created with a table-based design.

There are also people who prefer plain text e-mails. However that's a little too limited for us. We'll compromise by creating a simple newsletter, including our background image and logo, but without any advanced features. Instead let's focus on the content.

For the sake of modern web design we're avoiding tables. And we definitely won't use any JavaScript—which is considered dangerous and is generally disabled by default.

## Time for action – creating a newsletter template

Create the file `themes/bar/templates/Email/Default.ss`, add the following code to it and don't forget to flush after finishing:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html style="margin: 0;padding: 0;">
  <head>

  </head>
  <body style="margin: 0;padding: 0;">
    <div id="background" style="background-image:
      url(http://localhost/themes/bar/images/background.jpg);
      background-color: #eee;margin: 0; padding: 10px;">
    <div id="container" style="margin: 40px;
      padding: 10px 20px; background: #fff;">
      <img id="logo"
        src=http://localhost/themes/bar/images/logo.png
        alt="Logo" style="float: left;padding: 10px;">
      <div id="body" style="float: left; padding-left: 20px;">
        <h1 style="font-size: 2.5em; color: #f7d400;
          margin: 0 0 10px 0;">
          Bar Newsletter
        </h1>
        <p style="font-size: 1em; color: #333;">
          Dear $Member.FirstName,
        </p>
```

```
    $Body
  </div>
  <div id="foot" style="clear: both; text-align: center;
    margin: 0 auto; padding-top: 30px;">
    <p style="font-size: 1em;color: #333;">
      <a href="$UnsubscribeLink"
        style="color: #333;font-weight: bold;">
        Unsubscribe
      </a>
    </p>
  </div>
</div>
</div>
</body>
</html>
```

If your page isn't running on `http://localhost` or you want to send out newsletters to customers, you'll need to replace the two references with your custom domain.



#### Inline CSS

Creating inline CSS is a real pain. However there are tools available that convert embedded styles to inline ones, for example: <http://www.mailchimp.com/labs/inlinecss.php>. That should ease the pain a bit at least.

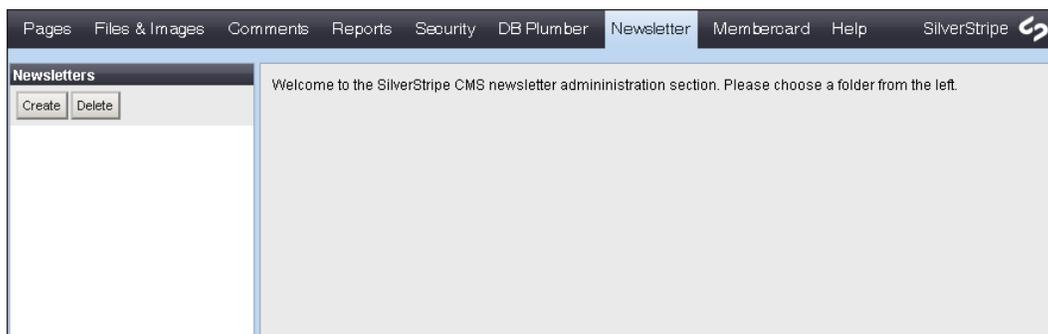
## What just happened?

This is pretty similar to our previous e-mail template. We could now move that to our newly created `Email` folder. But as the file's just for internal notifications and the folder for newsletters, we'll just leave it where it is.

The only uncommon thing about the code is that we're providing an absolute link to the logo. You'd avoid that on websites, but in e-mails you don't have much choice. Also, in a real newsletter you'd need a public domain name and not the one used for testing. But during development, however, `localhost` should do.

## In the CMS

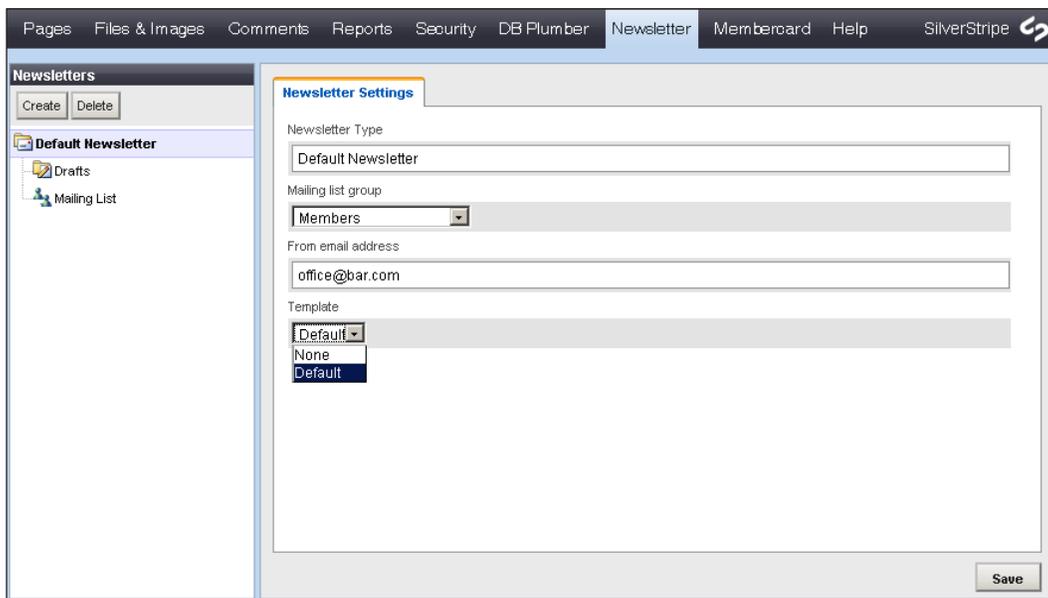
Now that we've finished the coding, it's time in the CMS. The first time you go to the newly created **Newsletter** tab, it's more or less empty:



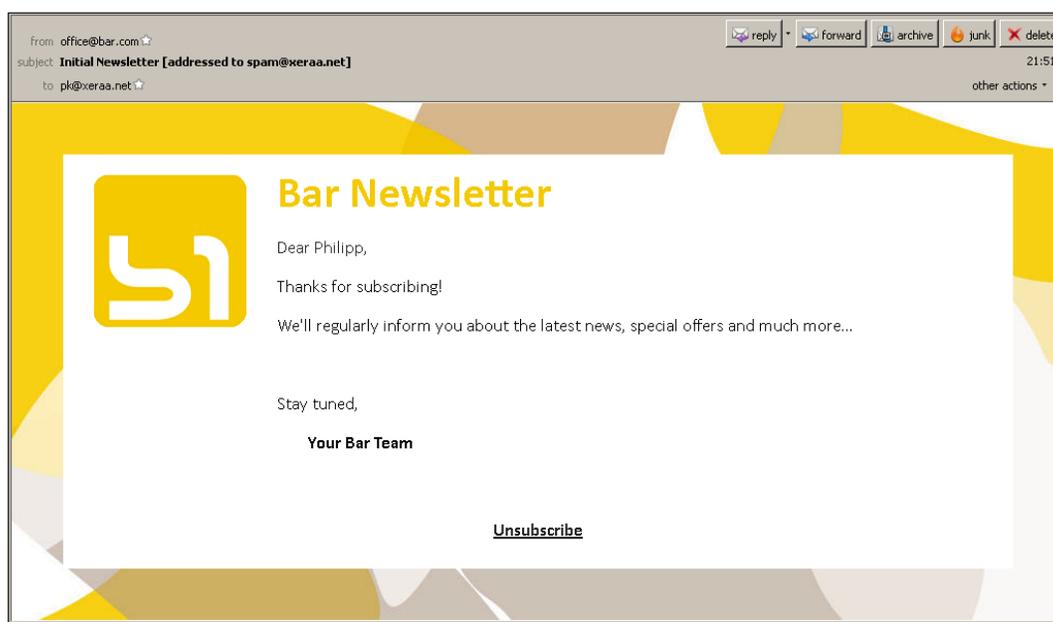
## Time for action – sending the newsletter

So let's fill it up:

1. Click on the **Create** button and **Add a new type**. This is more or less like a page type: You can have multiple ones for different purposes.
2. Then you can give your newly created newsletter a good name, select the group of recipients (**Members** in our case), optionally overwrite the from address and select our template. Don't forget to **Save!** The page should then look something like this:



3. Next you can create your first draft, just like for a regular page.
4. Using the **Preview this newsletter** link you can take a look at how the final result will look.
5. Using the **Sent Status Report** tab you can see the recipients and whether you've already sent them this draft or not.
6. Once you're satisfied with your settings, you can actually **Send...** the newsletter. You're given the option to send to a test address (you should definitely make use of that before sending hundreds of mails), to everyone on the list or just the people you've not yet sent this draft to.
7. A newsletter would look something like this, based on our template:



And that's it—you're ready to start your own e-mail campaign!

## Summary

By now you should have a good understanding of how to create features that don't focus on pages.

In particular we've started working with Model Admin—our new tool for easily managing data objects. With only a few lines of code we get full read and write access to a Model and its data. Additionally, we took an in-depth look at SilverStripe's permission and role system, which we've then integrated into our example. Finally we've made some changes to our code to use SilverStripe's newsletter module. Everyone requesting a member card is automatically added to our user base and the module allows us to easily send emails to all these users—tightly integrated into the CMS and making use of SilverStripe's template engine.

By now our page is more or less finished. It's so good that we're ready to expand our operations. This requires us to localize and internationalize our page—in short we want to globalize it. And that's exactly what we'll do in the next and final chapter.

