FREE eBook

# LEARNING silverstripe

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#silverstripe

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: silverstripe

It is an unofficial and free silverstripe ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official silverstripe.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with silverstripe

## Remarks

Silverstripe is an open source PHP content management system. A developer might want to use it because

- BSD License - meaning it can be rebranded as your own application
- Clean Object Oriented code very easy to understand and use - along with extend and customise
- Simple and powerful template engine making themes very easy to create

It can use most databases, primarily MySQL

## Versions

| Version | Release Date |
| --- | --- |
| 3.4.0 | 2016-06-03 |

## Examples

### Installation

SilverStripe can be installed via composer or through the extraction of downloaded zip file.

To install through composer we run the following command

```
composer create-project silverstripe/installer /path/to/project 3.4.0
```

A download zip file can be found on the download page of the SilverStripe website. Once downloaded, this file needs to be extracted into the root directory of the desired project.

Upon visiting the website for the first time an installation wizard will be presented to configure and set up the SilverStripe install.

### Customising the CMS / White Labeling

The SilverStripe CMS can be customised to change the CMS logo, link and application name.

This can be achieved with the following `config.yml` settings

```
LeftAndMain:
  application_name: 'My Application'
  application_link: 'http://www.example.com/'
```

```
  extra_requirements_css:
    - mysite/css/cms.css
```

### mysite/css/cms.css

```
.ss-loading-screen {
    background: #fff;
}
.ss-loading-screen .loading-logo {
    background: transparent url('../images/my-logo-loading.png') no-repeat 50% 50%;
}
.cms-logo a {
    background: transparent url('../images/my-logo-small.png') no-repeat left center;
}
```

Read Getting started with silverstripe online: https://riptutorial.com/silverstripe/topic/1771/getting-started-with-silverstripe

# Chapter 2: Add Ons and Modules

## Remarks

Addons and modules are encouraged to be registered with Packagist which then means they are found and registered with the SilverStripe add-on repository

Installation of modules is recommended through use of Composer

## Examples

### SilverStripe Grid Field Extensions Module

The SilverStripe Grid Field Extensions Module has some very nice features to enhance the basic `GridField`...

- `GridFieldAddExistingSearchButton` - a more advanced search form for adding items
- `GridFieldAddNewInlineButton` - builds on `GridFieldEditableColumns` to allow inline creation of records.
- `GridFieldAddNewMultiClass` - lets the user select from a list of classes to create a new record from
- `GridFieldEditableColumns` - allows inline editing of records
- `GridFieldOrderableRows` - drag and drop re-ordering of rows
- `GridFieldRequestHandler` - a basic utility class which can be used to build custom grid field detail views including tabs, breadcrumbs and other CMS features
- `GridFieldTitleHeader` - a simple header which displays column titles

More documentation is found within the module here.

### Better Buttons for GridField

The module Better Buttons for GridField adds new form actions and buttons to the GridField detail form.

- Save and add another: Create a record, and go right to adding another one, without having to click the back button, and then add again
- Save and close: Save the record and go back to list view
- User-friendly delete: Extracted from the tray of constructive actions and moved away so is less likely to be clicked accidentally. Includes inline confirmation of action instead of browser alert box
- Cancel: Same as the back button, but in a more convenient location
- Previous/Next record: Navigate to the previous or next record in the list without returning to list view
- and many more...

More documentation (and images) on the documentation for the module

## UserForms

The module UserForms enables CMS users to create dynamic forms via a drag and drop interface and without getting involved in any PHP code.

Main Features

- Construct a form using all major form fields (text, email, dropdown, radio, checkbox..)
- Ability to extend userforms from other modules to provide extra fields.
- Ability to email multiple people the form submission
- View submitted submissions and export them to CSV
- Define custom error messages and validation settings
- Optionally display and hide fields using javascript based on users input
- Displays a confirmation message when navigating away from a partially completed form

More documentation links can be found here in the github repository

## Display Logic

The Display Logic module allows you to add conditions for displaying or hiding certain form fields based on client-side behavior. This module is incredibly useful to make forms much more professional by showing only the appropriate fields and without adding a lot of custom JavaScript.

Example usage...

```
$products->displayIf("HasProducts")->isChecked();

$sizes->hideUnless("ProductType")->isEqualTo("t-shirt")
      ->andIf("Price")->isGreaterThan(10);

$payment->hideIf("Price")->isEqualTo(0);

$shipping->displayIf("ProductType")->isEqualTo("furniture")
           ->andIf()
              ->group()
                ->orIf("RushShipping")->isChecked()
                ->orIf("ShippingAddress")->isNotEmpty()
              ->end();
```

There are many more examples on the module readme.md

## Grouped CMS Menu

The Grouped CMS Menu Module allows you to group CMS menu items into nested lists which expand when hovered over. This is useful when there are so many CMS menu items that screen space becomes an issue.

## Dashboard

The Dashboard module provides a splash page for the CMS in SilverStripe 3 with configurable widgets that display relevant information. Panels can be created and extended easily. The goal of the Dashboard module is to provide users with a launchpad for common CMS actions such as creating specific page types or browsing new content.

There are Images and videos about this module can be found in this blog post.

There are some included Panels by default...

- Recently edited pages
- Recently uploaded files
- RSS Feed
- Quick links
- Section editor
- Google Analytics
- Weather

When you have this module installed it creates a dashboard per member, so if you have a large amount of members which will never use the admin and performance becomes an issue I recommend creating the members with these extra settings before writing it...

```
Member::create(array(
   'HasConfiguredDashboard' => 1
));
```

There is much more documentation in the modules readme.md

Read Add Ons and Modules online: https://riptutorial.com/silverstripe/topic/4339/add-ons-and-modules

# Chapter 3: DataExtensions

## Examples

### Adding fields to a DataObject

You can use the `DataExtension` mechanism to add extra database fields to an existing DataObject:

```
class MyMemberExtension extends DataExtension
{
    private static $db = [
        'HairColour' => 'Varchar'
    ];
}
```

And apply the extension:

```
# File: mysite/_config/app.yml
Member:
  extensions:
    - MyMemberExtension
```

This will add `HairColour` as a field to `Member` objects.

### Adding methods to a DataObject

You can add public methods to a DataObject using the extension mechanism, for example:

```
class MyMemberExtension extends DataExtension
{
    public function getHashId()
    {
        return sha1($this->owner->ID);
    }
}
```

When applied to the `Member` class, the example above would return the `sha1` hash of the `Member` ID by accessing the `Member` via the protected property `$this->owner`. Eg:

```
$member = Member::get()->byId(123);
var_dump($member->getHashId()); // string(40) "40bd001563085fc35165329ea1ff5c5ecbdbbeef"
```

### Applying a DataExtension to a Class

The most common way is to apply the extension via Config. Example:

```
# File: mysite/_config/config.yml
Member:
```

```
  extensions:
    - MyMemberExtension
```

The `extensions` config variable is of type "array", so you can add multiple extensions like this:

```
# File: mysite/_config/config.yml
Member:
  extensions:
    - MyMemberExtension
    - MyOtherMemberExtension
```

If you wrote the class that is to be extended, you can define the extension(s) as static variable:

```
<?php
class MyClass extends DataObject
{
    private static $extensions = ['MyCustomExtension'];
}
```

Read DataExtensions online: https://riptutorial.com/silverstripe/topic/3519/dataextensions

# Chapter 4: Forms

## Syntax

- Form::create($this, __FUNCTION__, $fields, $actions, $validator) // standard form creation
- Form::create(...)->addExtraClass('my-css-class another-class') // add CSS classes to your Form
- Form::create(...)->loadDataFrom(Member::get()->byID(1)); // populate a form with the details of an object

## Examples

### Creating a Form

Here is a basic example form with one required text field and one submit button, which submits to a custom function:

```
class Page_Controller extends ContentController {

    private static $allowed_actions = array(
        'ExampleForm'
    );

    public function ExampleForm() {
        $fields = FieldList::create(
            TextField::create('Name', 'Your Name')
        );

        $actions = FieldList::create(
            FormAction::create('doExampleFormAction', 'Go')
        );

        $requiredFields = RequiredFields::create('Name');

        $form = Form::create(
            $this,
            'ExampleForm',
            $fields,
            $actions,
            $requiredFields
        );

        return $form;
    }

    public function doExampleFormAction($data, Form $form) {
        $form->sessionMessage('Hello '. $data['Name'], 'success');

        return $this->redirectBack();
    }
}
```

To display this form we add `$ExampleForm` to our page template:

```
$ExampleForm
```

### Creating a simple AJAX Form

SilverStripe has reasonably good support for submitting form data using AJAX requests. Below is example code of how to set up a basic Form that accepts submissions by both AJAX and traditional default browser behaviour (as is good practice).

## Adding the form to our controller

First we need to define our form; your `Page_Controller` should look something like this:

```
class Page_Controller extends ContentController {

    /**
     * A list of "actions" (functions) that are allowed to be called from a URL
     *
     * @var array
     * @config
     */
    private static $allowed_actions = array(
        'Form',
        'complete',
    );

    /**
     * A method to return a Form object to display in a template and to accept form
submissions
     *
     * @param $request SS_HTTPRequest
     * @return Form
     */
    public function Form($request) {
        // include our javascript in the page to enable our AJAX behaviour
        Requirements::javascript('framework/thirdparty/jquery/jquery.js');
        Requirements::javascript('mysite/javascript/ajaxforms.js');
        //create the fields we want
        $fields = FieldList::create(
            TextField::create('Name'),
            EmailField::create('Email'),
            TextareaField::create('Message')
        );
        //create the button(s) we want
        $buttons = FieldList::create(
            FormAction::create('doForm', 'Send')
        );
        //add a validator to make sure the fields are submitted with values
        $validator = RequiredFields::create(array(
            'Name',
            'Email',
            'Message',
        ));
        //construct the Form
        $form = Form::create(
```

```
            $this,
            __FUNCTION__,
            $fields,
            $buttons,
            $validator
        );

        return $form;
    }

    /**
     * The form handler, this runs after a form submission has been successfully validated
     *
     * @param $data array RAW form submission data - don't use
     * @param $form Form The form object, populated with data
     * @param $request SS_HTTPRequest The current request object
     */
    public function doForm($data, $form, $request) {
        // discard the default $data because it is raw submitted data
        $data = $form->getData();

        // Do something with the data (eg: email it, save it to the DB, etc

        // send the user back to the "complete" action
        return $this->redirect($this->Link('complete'));
    }

    /**
     * The "complete" action to send users to upon successful submission of the Form.
     *
     * @param $request SS_HTTPRequest The current request object
     * @return string The rendered response
     */
    public function complete($request) {
        //if the request is an ajax request, then only render the include
        if ($request->isAjax()) {
            return $this->renderWith('Form_complete');
        }
        //otherwise, render the full HTML response
        return $this->renderWith(array(
            'Page_complete',
            'Page',
        ));
    }

}
```

Adding these functions to `Page_Controller` will make them available on **all** page types - this may not be desired and you should consider if it would be more appropriate to create a new page type (such as ContactPage) to have this form on

Here we've defined methods to:

- Create the `Form`
- A form handler (to save or send the submissions somewhere, this runs after the `Form` has successfully validated it's data)
- A `complete` action, which the user will be sent to after successfully completing the form submission.

## Customising out templates for easy content replacement

Next we need to set up our templates - modify your Layout/Page.ss file:

```
<% include SideBar %>
<div class="content-container unit size3of4 lastUnit">
    <article>
        <h1>$Title</h1>
        <div class="content">$Content</div>
    </article>
    <div class="form-holder">
        $Form
    </div>
        $CommentsForm
</div>
```

This is taken from the default simple theme, with a minor addition that the form is now wrapped in a `<div class="form-holder">` so that we can easily replace the form with a success message.

We also need to create a `Layout/Page_complete.ss` template - this will be the same as above except the `form-holder div` will be:

```
<div class="form-holder">
    <% include Form_complete %>
</div>
```

Next create the `Includes/Form_complete` include - it's important to use an include so that we can render **just** this section of the page for our responses to AJAX requests:

```
<h2>Thanks, we've received your form submission!</h2>
<p>We'll be in touch as soon as we can.</p>
```

# Creating the javascript form listener

Finally, we need to write our javascript to send the form by AJAX instead of the default browser behaviour (place this in mysite/javascript/ajaxform.js):

```
(function($) {
    $(window).on('submit', '.js-ajax-form', function(e) {
        var $form = $(this);
        var formData = $form.serialize();
        var formAction = $form.prop('action');
        var formMethod = $form.prop('method');
        var encType = $form.prop('enctype');

        $.ajax({
            beforeSend: function(jqXHR,settings) {
                if ($form.prop('isSending')) {
                    return false;
                }
                $form.prop('isSending',true);
```

```
            },
            complete: function(jqXHR,textStatus) {
                $form.prop('isSending',false);
            },
            contentType: encType,
            data: formData,
            error: function(jqXHR, textStatus, errorThrown) {
                window.location = window.location;
            },
            success: function(data, textStatus, jqXHR) {
                var $holder = $form.parent();
                $holder.fadeOut('normal',function() {
                    $holder.html(data).fadeIn();
                });
            },
            type: formMethod,
            url: formAction
        });
        e.preventDefault();
    });
})(jQuery);
```

This javascript will submit the form using AJAX and on completion it will fade the form out and replace it with the response and fade it back in.

# For advanced users:

With this example all forms on your site will be "ajaxified", this may be acceptable, but sometimes you need some control over this (for example, search forms wouldn't work well like this). Instead, you can modify the code slightly to only look for forms with a certain class.

Amend the `Form` method on `Page_Controller` like so:

```
public function Form() {
    ...
    $form->addExtraClass('js-ajax-form');
    return $form;
}
```

Amend the javascript like so:

```
$(window).on('submit', '.js-ajax-form', function(e) {
    ...
})(jQuery);
```

Only forms with the class `js-ajax-form` will now act in this way.

Read Forms online: https://riptutorial.com/silverstripe/topic/4126/forms

# Chapter 5: LeftAndMain

## Introduction

`LeftAndMain` is more of a lower-level API and not often required due to the existence of `ModelAdmin`. However if you wanted to create a custom user interface that did not necessarily require the functionality of `ModelAdmin` in the administration panel for your module, than `LeftAndMain` is where you would want to start.

## Examples

### 1. Getting Started

This guide is intended to get you started on creating your own User Interface by subclassing the `LeftAndMain` class.

By the end of this guide, you will have created your first `Hello World` interface in the Administration Panel.

# Requirements

This guide requires you to have at least version `3.*` of the framework AND CMS but less than version `4.*`.

If you wish to use this guide, then you will need to swap out any class references with the Fully Quality Class Name (FQCN) as defined in the SS4 upgrade guide.

# Preparation

> **tl;dr** Ignore the following steps and simply create the structure below them

1. Create a folder with a name of anything you choose in the root directory for your SilverStripe project, for this example we'll be using `/helloworld/` and create an empty file within that folder named `_config.php`. A `_config.php` at the very minimum is required in every module directory for SilverStripe to detect its existence.
2. Within your new folder, create a sub folder named exactly `/code/` and within that folder, for organisation purposes; create another folder called `/admin/`
3. Create `/helloworld/code/admin/HelloWorldLeftAndMain.php` and place the following code into it for now.

```
class HelloWorldLeftAndMain extends LeftAndMain {
```

```
}
```

4. Create the template file that will be used with this class called
   `/helloworld/templates/Includes/HelloWorldLeftAndMain.ss`

# Structure

```
/framework/
/cms/
/helloworld/
    + _config.php
    + /code/
        + /admin/
            + /HelloWorldLeftAndMain.php
    + /templates/
        + /Includes/
            + /HelloWorldLeftAndMain_Content.ss
```

### 2. Configuring HelloWorldLeftAndMain.php

If you haven't already lets simply start this file of with:

```
class HelloWorldLeftAndMain extends LeftAndMain {

}
```

# Configure

The first thing you should do, is define the `$url_segment` that will be used to access the interface, and the title (`$menu_title`) that will appear in the side navigation menu of the administration panel:

```
private static $url_segment = 'helloworld';
private static $menu_title  = 'Hello World';
```

*The following configuration variable(s) are optional and not used in this guide:*

```
private static $menu_icon   = 'helloworld/path/to/my/icon.png';
private static $url_rule    = '/$Action/$ID/$OtherID';
```

# Adding Stylesheets and Javascript

`LeftAndMain` allows you to override the `init` method in it's parent, we can use this to require specific files for our interface. Undoubtedly you should always need to require a CSS stylesheet that will style the elements for your user interface.

As a tip, it's recommended to never rely on the CSS classes provided by the CMS as these are subject to change without notice and will subsequently destroy the Look & Feel of your UI (for example, `3.*` to `4.*` has seen a complete makeover of the interface therefore any CSS classes you relied on in `3.*` need to be restyled for conversion to `4.*`)

So lets add our `helloworld/css/styles.css` file:

```
public function init() {
    parent::init();

    Requirements::css('helloworld/css/styles.css');
    //Requirements::javascript('helloworld/javascript/script.min.js');
}
```

We don't need any Javascript functionality for this example but in the above I have included how one would achieve adding Javascript a file using the Requirements class.

After which you can adopt what you have been used to when dealing `Page_Controller` such as `$allowed_actions` etc with one notable difference, however

> You **CANNOT** override `index()`.

Instead `index()` is assumed as `HelloWorldLeftAndMain_Content.ss` and from there, it's required to deal with the indexes display via template functions (see example below)

# Complete Code

```
class HelloWorldLeftAndMain extends LeftAndMain {
    private static $url_segment = 'helloworld';
    private static $menu_title  = 'Hello World';
    private static $allowed_actions = array(
        'some_action'
    );

    public function init() {
        parent::init();

        Requirements::css('helloworld/css/styles.css');
        //Requirements::javascript('helloworld/javascript/script.min.js');
    }

    public function Hello($who=null) {
        if (!$who) {
            $who = 'World';
        }

        return "Hello " . htmlentities($who);
    }
}
```

### 3. The Template (HelloWorldLeftAndMain_Content.ss)

The expected structure of this template can be a bit convoluted but it all boils down to this:

1. **There are 3 sections worth noting for this guide:**

    - `.north`
    - `.center`
    - `.south`

2. It must be wrapped entirely within an element that has the `data-pjax-fragment="Content"` attribute. This is so the AJAX calls generated from the sidemenu, know where the "Content" is so that it may display it appropriately:

```
<div class="cms-content center $BaseCSSClasses" data-layout-type="border" data-pjax-
fragment="Content">

</div>
```

I won't go into detail about template functionality, I have included comments where relevant but you shouldn't be reading this guide if you don't understand template syntax for SilverStripe

# Complete Code

The only thing from below; that you should expect to come out already styled is the `<% include CMSBreadcrumbs %>` everything else you must cater for yourself in the CSS file that was included earlier

```
<div class="cms-content center $BaseCSSClasses" data-layout-type="border" data-pjax-
fragment="Content">
    <%-- This will add the breadcrumb that you see on every other menu item --%>
    <div class="cms-content-header north">
        <div class="cms-content-header-info">
            <% include CMSBreadcrumbs %>
        </div>
    </div>

    <div class="center">
        <%-- Our function in HelloWorldLeftAndMain.php --%>
        $Hello('USER');
        <%-- ^ outputs "Hello USER" --%>
    </div>

    <div class='south'>
        Some footer-worthy content
    </div>
</div>
```

Now all thats left to do is for you to `/dev/build` and `?flush=1` then you can check out our useless little module in the Administration Panel!

Read LeftAndMain online: https://riptutorial.com/silverstripe/topic/8300/leftandmain

# Chapter 6: ModelAdmin

## Examples

### Simple Example

Given a simple `DataObject` like this:

```
class MyDataObject extends DataObject {
    private static $db = array(
        'Name' => 'Varchar(255)'
    );
}
```

To provide full Create-Read-Update-Delete for the objects then this is the `ModelAdmin` code required:

```
class MyModelAdmin extends ModelAdmin {
    private static $mangaged_models = array(
        'MyDataObject'
    );
    private static $url_segment = 'my-model-admin';
    private static $menu_title = 'My Model Admin';
    private static $menu_icon = 'mysite/images/treeicons/my-model-admin.png';
    private static $menu_priority = 9;
}
```

### Control the DataObject name displayed in the UI

```
class MyDataObject extends DataObject {

    private static $singular_name = 'My Object';
    private static $plural_name = 'My Objects';

    ...
}
```

### DataObjects can be sorted by default

```
class SortDataObject extends DataObject {

    private static $db = array(
        'Name' => 'Varchar',
        'SortOrder' => 'Int'
    );

    private static $default_sort = 'SortOrder DESC';
}
```

### Control columns displayed for the DataObject

```
class MyDataObject extends DataObject {

    private static $db = array(
        'Name' => 'Varchar'
    );

    private static $has_one = array(
        'OtherDataObject' => 'OtherDataObject'
    );

    private static $summary_fields = array(
        'Name',
        'OtherDataObject.Name'
    );

    private static $field_labels = array(
        'OtherDataObject.Name' => 'Other Data Object'
    );
}
```

ModelAdmin uses the `summary_fields` to generate the columns that it displays. To specify the name of the column, use `field_labels` as shown.

## Using searchable_fields to control the filters for that Object in ModelAdmin

```
class MyDataObject extends DataObject {

    private static $db = array(
        'Name' => 'Varchar'
    );

    private static $has_one = array(
        'OtherDataObject' => 'OtherDataObject'
    );

    private static $summary_fields = array(
        'Name',
        'OtherDataObject.Name'
    );

    private static $searchable_fields = array(
        'Name',
        'OtherDataObjectID' => array(
            'title' => 'Other Data Object'
        )
    );

}
```

Note the `OtherDataObjectID` which converts a text field into a drop down of the relating object to filter with.

## Remove scaffolded GridField for relationships

```
class MyDataObject extends DataObject {
```

```
    ...

    private static $has_many = array(
        'OtherDataObjects' => 'OtherDataObject'
    );

    function getCMSFields() {
        $fields = parent::getCMSFields();

        if ($gridField = $fields->dataFieldByName('OtherDataObjects')) {
            $gridField->getConfig()
                ->removeComponentsByType('GridFieldExportButton');
        }

        return $fields;
    }
}
```

## To remove the export button from ModelAdmin

```
class MyAdmin extends ModelAdmin {

    ...

    function getEditForm($id = null, $fields = null) {
        $form = parent::getEditForm($id, $fields);

        if ($this->modelClass == 'MyDataObjectName') {
            $form->Fields()
                ->fieldByName($this->sanitiseClassName($this->modelClass))
                ->getConfig()
                ->removeComponentsByType('GridFieldExportButton');
        }
        return $form;
    }
}
```

Read ModelAdmin online: https://riptutorial.com/silverstripe/topic/3836/modeladmin

# Chapter 7: The autoloader

## Remarks

When you make any changes to the classes then you need to run a dev/build?flush=1 to rebuild the manifest.

## Examples

### MyClass.php

```
<?php

class MyClass {
    ...
}

class OtherClass {
    ...
}

?>
```

Any class that has the same name as it's file name will be auto loaded by Silverstripe.

OtherClass will be loaded too because it is in a file which is being read.

MyPage.php

```
<?php

class MyPage_Controller extends BookingPage_Controller {
    ...
}

?>
```

For controller functions you can omit the "_Controller" part int he file name.

If a directory is to be ignored then include a file named "_manifest_exclude"

Read The autoloader online: https://riptutorial.com/silverstripe/topic/3817/the-autoloader

# Chapter 8: The Config System

## Remarks

# What is the config system

SilverStripe uses a global config system to store settings for classes and the application. These config variables can be used to define the structure of Models, security settings on Controllers or API keys for third party services.

# How it works

`Config` values are populated by the `SS_ConfigStaticManifest` during a `dev/build` and cache flush (appending `?flush` to any URL`) or on first ever run of the application code.

The `SS_ConfigStaticManifest` will scan all PHP classes and YAML config files for any config values and build a cache of these values.

> When making change to `Config` settings via YAML or `private static` variables, you'll need to flush the cache for these changes to take effect.

## Examples

### Setting config values

`Config` values can be set in three ways:

1. Via `private static` variables on any class within a SilverStripe project
2. Via yaml config files (stored in module-folder/_config/[file].yml)
3. Via PHP at run time (`Config::inst()->update('Director', 'environment_type', 'dev')`

> Generally it's best to set config values via the first 2 methods as these are statically cached when flushing the cache.

# Setting with private statics

```
class MyDataObject extends DataObject {

    private static $db = array(
        'Title' => 'Varchar',
    );

}
```

> All `private static` class variables in a SilverStripe project's code (including modules, but not packages in the `vendor/` directory) will be loaded into the `Config`.

## Setting with YAML

You can add this to `mysite/_config/config.yml` (or any other YAML file in that path).

```
Director:
  environment_type: dev
```

> Using YAML files is a great way to override default `Config` values for core classes or modules

## Setting at runtime

This would typically be done in `mysite/_config.php`

```
Config::inst()->update('Director', 'environment_type', 'dev');
```

> Updating the `Config` in PHP should be avoided where possible as it's slower than using the cached values

Read The Config System online: https://riptutorial.com/silverstripe/topic/4699/the-config-system

# Chapter 9: Using the ORM

## Examples

### Reading and writing DataObjects

DataObjects in SilverStripe represent a database table row. The fields in the model have magic methods that handle getting and setting data via their property names.

Given we have a simple DataObject as an example:

```
class Fruit extends DataObject
{
    private static $db = ['Name' => 'Varchar'];
}
```

You can create, set data and write a `Fruit` as follows:

```
$apple = Fruit::create();
$apple->Name = 'Apple';
$apple->write();
```

You can similarly retrieve the `Fruit` object as follows:

```
$apple = Fruit::get()->filter('Name', 'Apple')->first();
var_dump($apple->Name); // string(5) "Apple"
```

Read Using the ORM online: https://riptutorial.com/silverstripe/topic/3463/using-the-orm

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with silverstripe | 3dgoo, Barry, Community, zanderwar |
| 2 | Add Ons and Modules | Barry |
| 3 | DataExtensions | bummzack, Dan Hensby, Robbie Averill |
| 4 | Forms | 3dgoo, Dan Hensby |
| 5 | LeftAndMain | zanderwar |
| 6 | ModelAdmin | 3dgoo, Barry, Turnerj |
| 7 | The autoloader | Barry |
| 8 | The Config System | Dan Hensby |
| 9 | Using the ORM | 3dgoo, bummzack, Robbie Averill |