LEARNING DRUPAL 9 AS A FRAMEWORK

PRACTICAL GUIDE WITH FULL CODE INCLUDED

Requirements:
This is a coding book for programmers. At least one year of experience as a developer with drupal or a related framework is required. You must be able to install drupal on a local server.

Description
This course will teach you advanced concepts of drupal 9, Object-oriented PHP and symphony components. After the course, you'll be able to build robust and scalable software solutions of many kinds.

In this hands-on course a drupal expert with 10 years experience with the software will give you a deep-dive in the power that drupal core has to offer.

Advanced topics like custom entities, entity forms, access control, events, caching, workflows and many more are discussed while building an actual software.

With +2400 lines of custom code, the author offers you powerful and ready-to-use snippets for your next drupal projects.

Fun fact: you'll not even be using nodes at all but only custom entities.

Let's take a deep dive!

First edition: 05/05/2021
Modified edition: 24/06/2021
See changelog at https://stefvanlooveren.me/courses/drupal-9-framework#changelog

Foreword

The history of drupal as a content management system has been interesting in many ways. This open-source project hit an incredible 1 million powered websites worldwide in 2014 because of its truly great CMS features for developers and webmasters.

Since then, it has chosen an interesting path. While in 2014 the main competitors were Wordpress and Joomla, the community took the radical decision of a complete rewrite of the software to meet the challenges of the future. The result was the release of drupal 8: an object-oriented framework with Symfony components. It resulted in software that was ready to meet the high standards of the industry on security, scalability, cost-effectiveness, and reliability.

In 2021, with Drupal 9, it has become an enterprise software for ambitious and custom-tailored solutions that is comparable with Ruby on Rails and Django. Especially for startups, it offers a quick-to-market solution.

I wrote this course because from my experience the speed of building custom-tailored solutions with drupal 9 is stunning. A truly huge advantage is that the security updates come for free. From the community, for the community. I really like this slogan drupal has had for years: << Come for the code, stay for the community >>.

I've been around in the drupal ecosystem for ten years and have supported in many ways: code contributing, blogging, sharing solutions on DrupalAnswers, and attending conferences. With this book, I share almost all of the knowledge I've built up so far since using Drupal 8 and 9.

Allow me to say a few topics that do not get covered in this book. The ecosystem is so big that I had to let out chapters about these:

- Unit testing (would need a separate book)
- Server set-up (but take a look at Lando!)
- JSON-API and webservices (would need a separate book)
- Drupal theming guide
- How to install drupal

In this hands-on course, we'll be building a bidding software platform. While the platform will not be 100% finished, the backbone and layers are ready to use for your production websites. I hope you enjoy it as much as I did.

Part 1: Drupal developer essentials

👉 You may understand that building larger platforms also requires a more thorough development setup. In this section you will learn the more **advanced developer tools** real-life experts use to build their solutions.

First, we'll take a look at **compose**r and how we’ll manage installation, versioning and upgrades of the core and the modules.

After this we'll discuss **Git**. Interesting to know is that thanks to composer we do not have to add our modules to version control.

Then I want you to get to know **Xdebug**, which we will need to intercept the software at certain breakpoints to take a look at all the available data at that moment. I use this all the time during PHP development.

Webprofiler is a tool for using while developing. It indicates the current route, the load time of the controller and a lot of other stuff.

You might have heard of **drush** before, the cli for drupal developers. In this course, we’ll get to know this drupal command-line language a bit better and use it more and more while building custom modules.

Drupal is not meant for a simple blog anymore. The framework and CMS can be pretty complex and needs to get updated regularly and without downtime.
We will develop the platform in a way that when other developers of our team run a clean drupal install and import configuration, the platform is ready on-the-fly. But we do not want an empty box every time, so we’ll provide some seed data like users, offers, bids etc. that can be imported on installation.

We will develop the platform in a way that when you install, the platform is ready on-the-fly, with real-world content filled in. No more empty boxes or production data!

In the drupal 7 days, going from staging to production was hard. Drupal 8 has made a huge progression on this and even made it a strength. That’s why so many enterprises are now interested in drupal: it got easy to roll in new features in it while in production. **Configuration management** is a topic we’ll discuss more in depth at the end of this chapter.

Drupal 8 has made a strength out of its previous weakness: configuration management.

Composer

In this section we'll talk about composer and how it manages versioning.

Composer is an application-level package manager for PHP. It keeps track of the versions and dependencies of your core drupal, modules, themes and libraries.

In short, this will make sure you don't install something that is not compatible. In addition to the core drupal installation, every module or theme can add requirements and dependencies in it's .info file via a composer.json file.

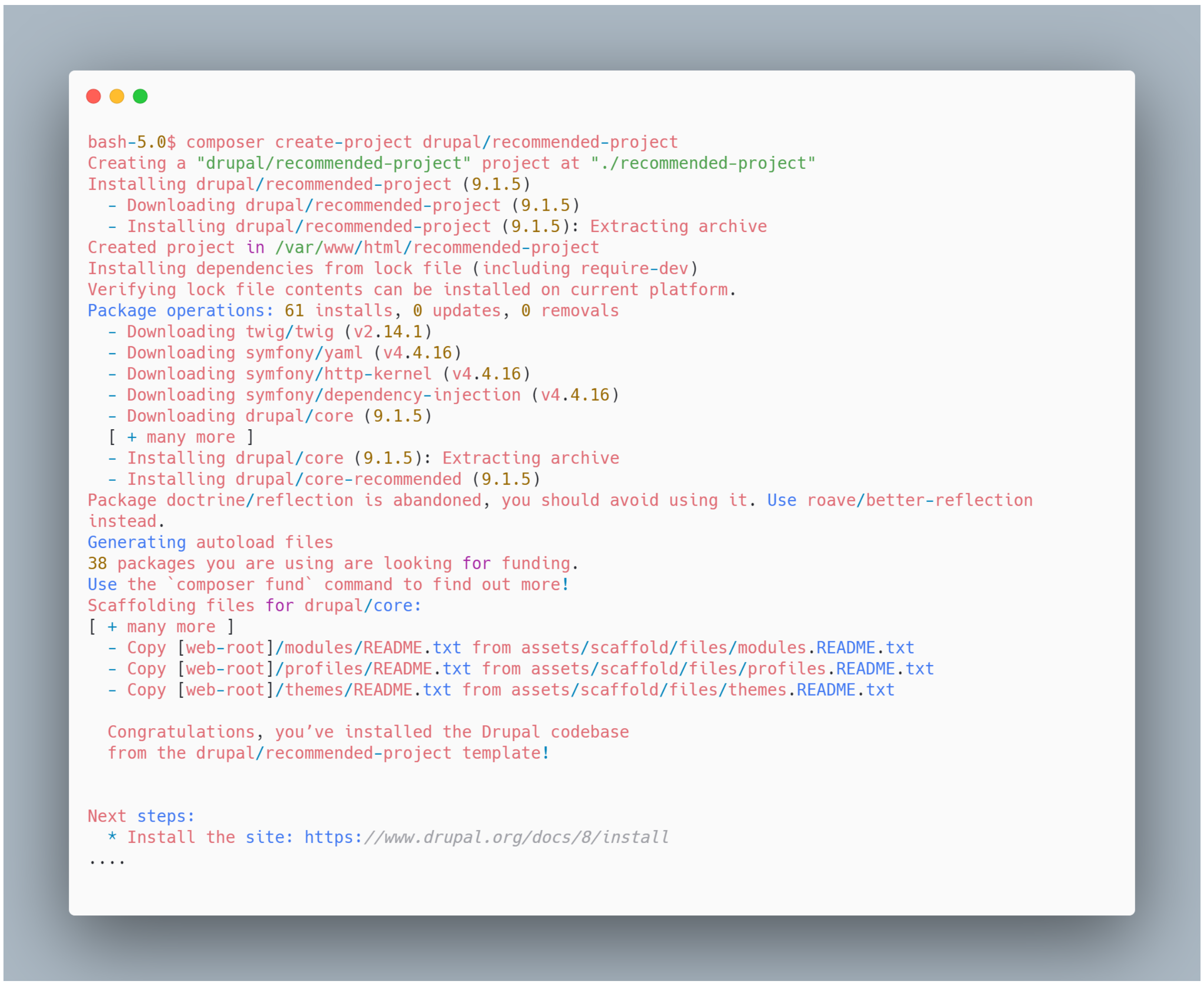
Composer was added in drupal 8 and is mandatory. Make sure you have it installed globally on your machine.

Take a look at a basic installation of drupal. Clone the files of the drupal/recommended-project to the folder you would like to use as your development environment.

Make sure you have composer installed, go to your terminal and type:

```
composer create-project drupal/recommended-project
```

Your terminal will go ahead and download all the required packages:



```
bash-5.0$ composer create-project drupal/recommended-project
Creating a "drupal/recommended-project" project at "./recommended-project"
Installing drupal/recommended-project (9.1.5)
  - Downloading drupal/recommended-project (9.1.5)
  - Installing drupal/recommended-project (9.1.5): Extracting archive
Created project in /var/www/html/recommended-project
Installing dependencies from lock file (including require-dev)
Verifying lock file contents can be installed on current platform.
Package operations: 61 installs, 0 updates, 0 removals
  - Downloading twig/twig (v2.14.1)
  - Downloading symfony/yaml (v4.4.16)
  - Downloading symfony/http-kernel (v4.4.16)
  - Downloading symfony/dependency-injection (v4.4.16)
  - Downloading drupal/core (9.1.5)
  [ + many more ]
  - Installing drupal/core (9.1.5): Extracting archive
  - Installing drupal/core-recommended (9.1.5)
Package doctrine/reflection is abandoned, you should avoid using it. Use roave/better-reflection
instead.
Generating autoload files
38 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
Scaffolding files for drupal/core:
[ + many more ]
  - Copy [web-root]/modules/README.txt from assets/scaffold/files/modules.README.txt
  - Copy [web-root]/profiles/README.txt from assets/scaffold/files/profiles.README.txt
  - Copy [web-root]/themes/README.txt from assets/scaffold/files/themes.README.txt

  Congratulations, you've installed the Drupal codebase
  from the drupal/recommended-project template!


Next steps:
  * Install the site: https://www.drupal.org/docs/8/install
....
```

After this, run

```
composer update
```

to make sure you have the latest packages available.

Using this *recommended* repository as a starter point, we avoid dependency problems by using only dependencies that have already been tested with your version of Drupal.

The provided **composer.json** file contains the best practice way of structuring a drupal project. It will automatically make sure your modules, themes and libraries get downloaded in the right folder. The created **composer.lock** file keeps track of every exact version you have installed.

You see a lot of **symfony components** go to your *vendor* map, but drupal also requires quite a lot of other extra packages.

Now that installation is ready, take a look at the folder structure we now have:

- **web** is the webroot of the project where we can see a fresh new drupal 9 installation . This is the folder where your domain name will be pointed to
- **vendor** is the folder with all the dependencies

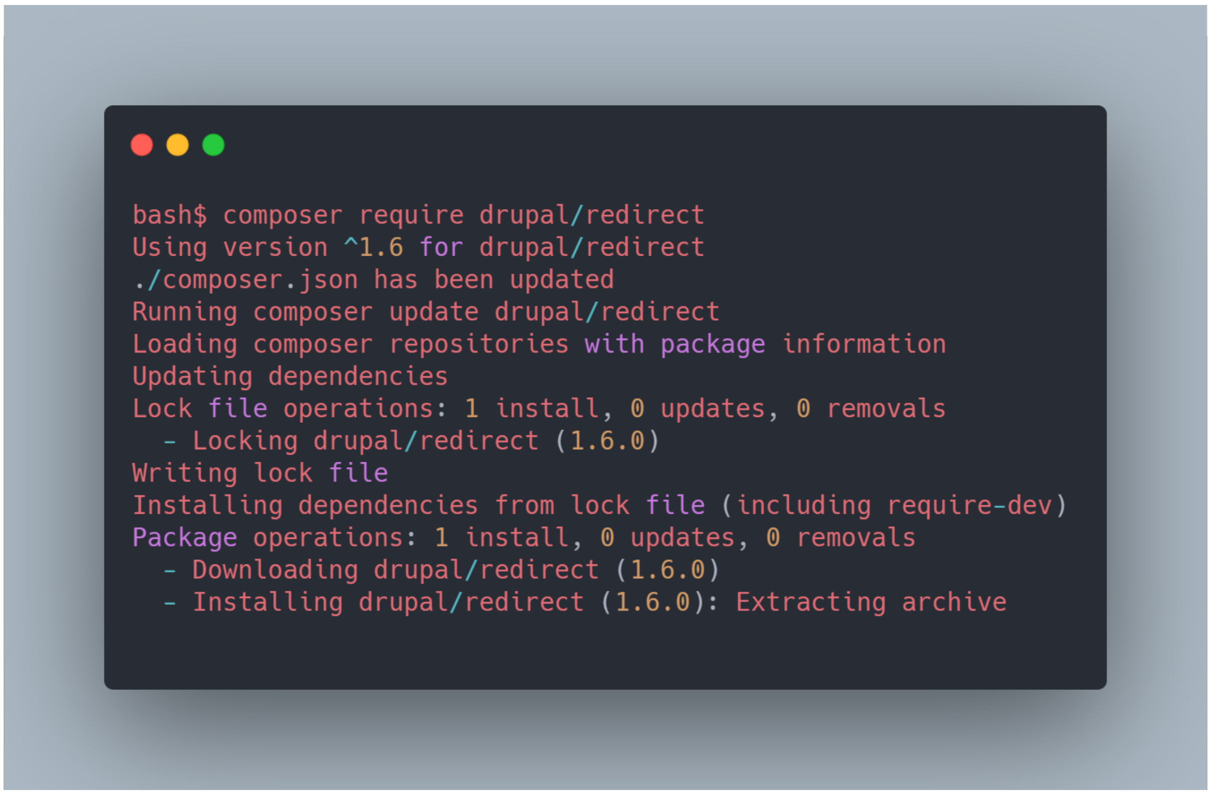
Notice that the **composer.json** file will always stay one level above our webroot, thus in the same level as *web* and *vendor*.

It is nice to have the vendor packages separated from the drupal core. This makes it cleaner in the long run. Also, dependencies are only accessible from the server because they are a level higher than the web root.

Basic composer commands

Let's move on with some basic composer commands:

```
composer require drupal/redirect
```

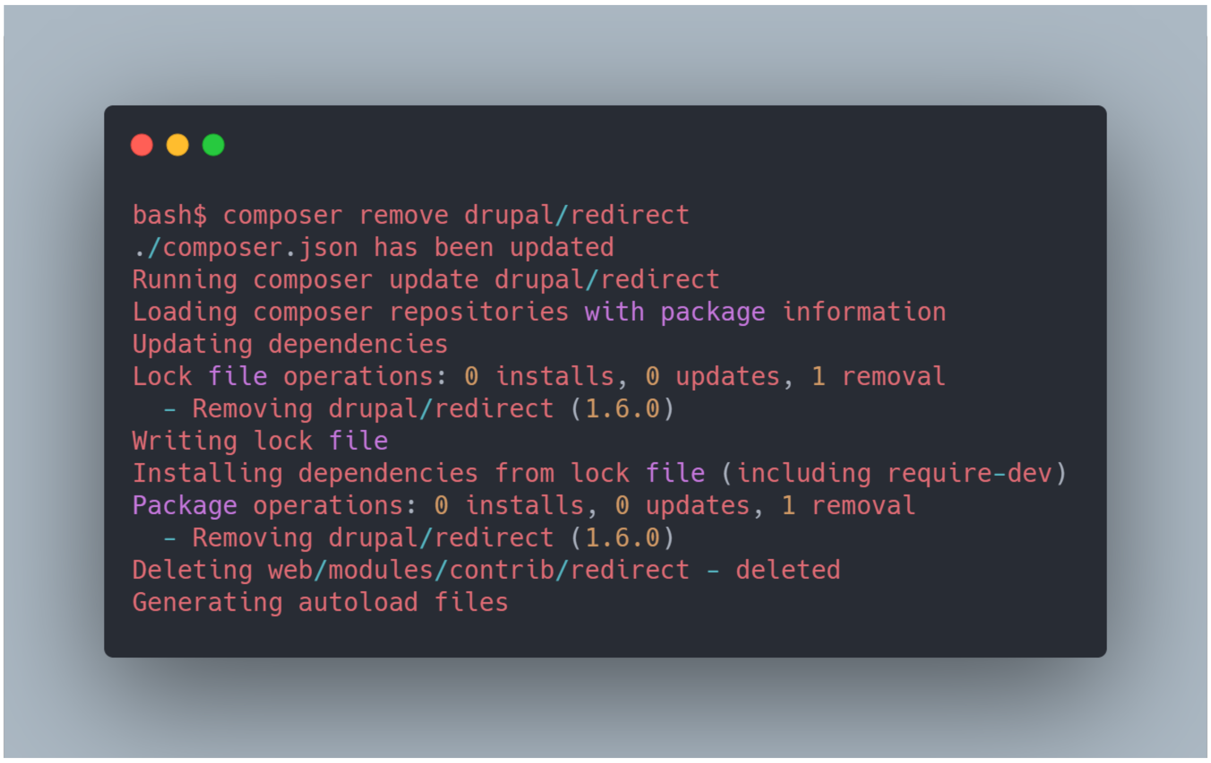


```
bash$ composer require drupal/redirect
Using version ^1.6 for drupal/redirect
./composer.json has been updated
Running composer update drupal/redirect
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking drupal/redirect (1.6.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Downloading drupal/redirect (1.6.0)
  - Installing drupal/redirect (1.6.0): Extracting archive
```

This will download the redirect module to **modules/contrib** and a reference to it gets added to our **composer.json** and **composer.lock** file. If the module has dependencies defined, they will get downloaded and put in the vendor map.

To remove it, we run:

```
composer remove drupal/redirect
```



```
bash$ composer remove drupal/redirect
./composer.json has been updated
Running composer update drupal/redirect
Loading composer repositories with package information
Updating dependencies
Lock file operations: 0 installs, 0 updates, 1 removal
  - Removing drupal/redirect (1.6.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 0 installs, 0 updates, 1 removal
  - Removing drupal/redirect (1.6.0)
Deleting web/modules/contrib/redirect - deleted
Generating autoload files
```

This removes the module. Always make sure you disabled the module first in drupal.

It is also possible to be more specific:

```
composer require drupal/redirect:1.5
```

This will get us a specific version of the module, instead of the latest one.

For a theme, it works the same:

```
composer require drupal/bootstrap
```

Thanks to the scaffolding in the composer.json file the system knows which is the theme directory, it gets placed in the right folder, which is **themes/custom/bootstrap**.

Comparing with drupal 7 and earlier

Why not just download the module from drupal.org, like we used to do?

Here’s the deal. If we want to be able to professionally use drupal as a software platform, you really really have to get comfortable with composer.
You’ll definitely get yourself into trouble if you are ignoring this way of working because **mixing the wrong dependencies will get you bugs** soon or late.
Second, by using composer you do not have to add all these modules to version control which **will dramatically decrease your git repository size**. Because a server can install everything that is in composer.json by himself. By this we keep everything clean and in the long-term it is just the only option to keep the software healthy when it gets bigger.

In addition: we can even self-host private modules in a repository of the company and add them as packages. It’s not what we’ll do in this course, but handy to know this is an option.

To update everything you run

```
composer update
```

To update one module only, go ahead and type

```
composer update drupal/redirect
```

Some extra helpful commands can be found at **https://getcomposer.org.** It is powerful, stable and well maintained.

Patching with composer

Sometimes we need to patch a module. This is the workflow: first we need an additional package for composer that makes patching possible. A side note here: since drupal 8 and the use of composer and symfony we got access to a broad ecosystem of php-contributors. This composer package, that wasn’t specifically built for drupal, is an example.

In our terminal, we type:

```
composer require cweagans/composer-patches
```

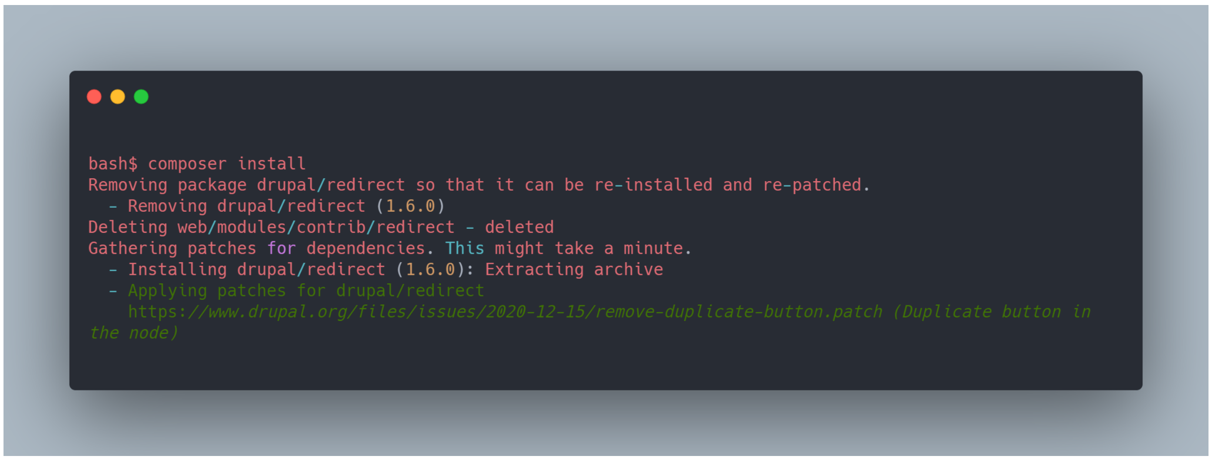
Then add this to the **composer.json** file in the “extra” section:

```
"patches": {
  "drupal/redirect": {
   "Duplicate button in the node":
"https://www.drupal.org/files/issues/2020-12-15/remove-duplicate-b
utton.patch"
  }
}
```

Now run

```
composer install
```

again. It will remove the module, reïnstall it and let you know if the patch was correctly applied.



```
bash$ composer install
Removing package drupal/redirect so that it can be re-installed and re-patched.
  - Removing drupal/redirect (1.6.0)
Deleting web/modules/contrib/redirect - deleted
Gathering patches for dependencies. This might take a minute.
  - Installing drupal/redirect (1.6.0): Extracting archive
  - Applying patches for drupal/redirect
    https://www.drupal.org/files/issues/2020-12-15/remove-duplicate-button.patch (Duplicate button in
the node)
```

If you have created patches yourself, add a patch to a folder you name **patches** in your root directory. Your patches section would look like this:

```
"patches": {
  "drupal/redirect": {
    "Fixing a PHP notice": "patches/redirect-php-notice-fix.patch"
  }
}
```

Git best practices

Version control systems are software tools that help a software team manage changes to source code over time. The current industry standard is Git.

This course will not be about best practices on git or how to organize your git workflow in a team. But I'll provide you some tips on how to keep our repository small and clean.

This section will not give you a git introduction, I assume you know the very basics.

I would always recommend using version control when developing software. It is just one of the things you'll heavily use for the rest of your career as a software developer.

In this section I'll discuss what the best way of working with git and drupal is.

In the source code you'll see there is a **.gitignore** file that was added to our code. This file says to git which files need to be ignored. Below you see the typical drupal folders we want to ignore.
The first section are all the directories that get generated by Composer. Like was said in the previous section, composer helps us to dramatically reduce our repository size:

```
# Ignore directories generated by Composer
/drush/contrib/
/vendor/
/web/core/
/web/modules/contrib/
/web/themes/contrib/
/web/profiles/contrib/
/web/libraries/
```

The second section contains all site specific data

```
# Site specific
/web/sites/*
```

The following are my IDE files, of course I do not want them in version control

```
# IDE specific
/.idea
```

Here are a few other typical things to ignore

```
# NPM packages & theming related
/node_modules/
*.css.map
```

I would like to emphasize on how composer and git are the two main ingredients for creating a whole new way of working in comparison with the older drupal way.

Composer is key for downloading packages and avoiding conflicts when upgrading, git is there for keeping track of history in changes of our custom modules, configuration and theme code.

It’s time to move on to the next section.

CLI tools

Drush is a command line shell and Unix scripting interface for Drupal. It is the most used cli, and although there is a second and similar one, called drupal console, drush is the most popular by far.

You can start by adding drush as a dependency in our **composer.json** by requiring the package:

```
composer require drush/drush
```

A note here: if you are running your site in the sites folder, and not the standard sites/default folder, or running a multi-site installation you have to go to the sites folder of your installation.

If your website is inside **web/sites/mysite**, you would do

```
cd web/sites/mysite
```

This way, drush knows which site you're talking about. To install a module

```
drush en views
```

While drush is the "standard", drupal console is interesting in some cases as well. Because it was introduced with the adoption of symfony, it is less opinionated about drupal than drush. Console is both a building tool as a debug tool. There is a lot of similar functionality, but I like to share this nice feature console has. You can get all information of a route with a simple command.

First:

```
Composer require drupal/console
```

Debug a route like this:

```
drupal debug:router entity.user_role.edit_form
```

This will return:

```
 Route             entity.user_role.edit_form
 Path             /admin/people/roles/manage/{user_role}
 Defaults
  _entity_form    user_role.default
  _title          Edit role
 Requirements
  _entity_access user_role.update
 Options
  compiler_class Drupal\Core\Routing\RouteCompiler
  utf8            1
  parameters      user_role:
                    type: 'entity:user_role'
                    converter:
drupal.proxy_original_service.paramconverter.configentity_admin

  _admin_route    1
  _access_checks access_check.entity
```

This is great. With one command, all of the routes info gets returned.

Another example is debugging of services containers. Let's lookup all the service containers we can find for sessions:

```
drupal debug:container|grep session
```

This will return:

```
 cache_context.session
Drupal\Core\Cache\Context\SessionCacheContext
 cache_context.session.exists
Drupal\Core\Cache\Context\SessionExistsCacheContext
 http_middleware.session
Drupal\Core\StackMiddleware\Session
 session
Symfony\Component\HttpFoundation\Session\Session
 session.flash_bag
Symfony\Component\HttpFoundation\Session\Flash\FlashBag
 session_configuration
Drupal\Core\Session\SessionConfiguration
 session_handler.storage
Drupal\Core\Session\SessionHandler
 session_handler.write_safe
```

```
Drupal\Core\Session\WriteSafeSessionHandler
 session_manager
Drupal\Core\Session\SessionManager
 session_manager.metadata_bag
Drupal\Core\Session\MetadataBag
```

Almost all of the things come from drupal core, but also some are symfony components. Interesting, right? Digging a little makes the layers more visible.

Apart from debugging, we can use console to generate custom modules, controllers, … etc.
I'll show this by generating a custom module called **offer,**, the name of our platform. It will ask us a few things.

Fyi: the drupal console will always ask for a valid uri, like

```
drupal generate:module --uri=mysite.test
```

Let's generate our first module:

```
bash$ drupal generate:module --uri=mysite.test

 // Welcome to the Drupal module generator

 Enter the new module name:
 > Offer

 Enter the module machine name [offer]:
 >

 Enter the module Path [sites/modules/custom]:
 >

 Enter module description [My Awesome Module]:
 > Main module for the offer platform

 Enter package name [Custom]:
 > Offer

 Enter Drupal Core version [9.x]:
```

```
 > 9.x

 Do you want to generate a .module file? (yes/no) [yes]:
 >

 Define module as feature (yes/no) [no]:
 > no

 Do you want to add a composer.json file to your module? (yes/no)
[yes]:
 > no

 Would you like to add module dependencies? (yes/no) [no]:
 > no

 Do you want to generate a unit test class? (yes/no) [yes]:
 > no

 Do you want to generate a themeable template? (yes/no) [yes]:
 > no

 Do you want to proceed with the operation? (yes/no) [yes]:
 > yes

Generated or updated files
 Generation path: /var/www/html/web
 1 - /sites/modules/custom/offer/offer.info.yml
 2 - /sites/modules/custom/offer/offer.module


 Generated lines: 29
```

Some other commands you might find interesting:

Creation of nodes:

```
drupal create:nodes article \
  --limit="5" \
  --title-words="5" \
  --time-range="1" \
  --revision \
  --language="und"
```

Exporting the configuration of a content type to a module

```
drupal config:export:entity node_type page \
  --module="demo"
drupal config:export:entity node_type page \
  --module="demo" \
  --optional-config \
  --remove-uuid \
  --remove-config-hash
```

(more on configuration in a later chapter)

Generate a custom block:

```
drupal generate:plugin:block  \
  --module="modulename"  \
  --class="DefaultBlock"  \
  --label="Default block"  \
  --plugin-id="default_block"  \
  --theme-region="header"  \
  --inputs='"name":"inputtext", "type":"text_format",
"label":"InputText", "options":"", "description":"Just an input
text", "maxlength":"", "size":"", "default_value":"",
"weight":"0", "fieldset":""'
```

In this course we will make a choice for drush. But we discussed some interesting drupal console commands as well. Make sure you check the official docs for more commands for console at https://drupalconsole.com/docs and for drush at https://www.drush.org/latest

Development & debug

In this section we'll talk about debugging techniques experts use while programming in drupal. These tools help you to get real deep into the layers of the software, and will accelerate quick development of custom modular development.

We'll discuss **Xcode** and **Webprofiler**, but first some typical drupal development setup tricks.

You enable "development mode" for a drupal site like this with console::

```
bash-5.0$ drupal site:mode dev --uri=mysite.test

 Configuration name: system.performance
 ------------------------- ---------------- ----------------
  Configuration key         Original Value   Override Value
 ------------------------- ---------------- ----------------
  cache.page.use_internal                    false
  css.preprocess            false            false
  css.gzip                  true             false
  js.preprocess             false            false
  js.gzip                   true             false
  response.gzip                              false
 ------------------------- ---------------- ----------------

 Configuration name: views.settings
 -------------------------------- ----------------
----------------
  Configuration key                Original Value   Override Value
 -------------------------------- ----------------
----------------
  ui.show.sql_query.enabled        false            true
  ui.show.performance_statistics   false            true
 -------------------------------- ----------------
----------------

 Configuration name: system.logging
 ------------------- ---------------- ----------------
  Configuration key   Original Value   Override Value
 ------------------- ---------------- ----------------
  error_level         hide             all
 ------------------ --------------- ----------------


 Services files "/var/www/html/web/sites/mysite/services.yml" was
overwritten


 New services settings
 ------------------------------------------ ------------- -------
  Service                                    Parameter     Value
 ------------------------------------------ ------------- -------
  http.response.debug_cacheability_headers                 true
  twig.config                                auto_reload   true
  twig.config                                cache         false
  twig.config                                debug         true
 ------------------------------------------ ------------- -------


 // cache:rebuild

 Rebuilding cache(s), wait a moment please.


 [OK] Done clearing cache(s).
```

Let's take a look at the services.yml under sites/mysite:

```
parameters:
   http.response.debug_cacheability_headers: true
   twig.config: { debug: true, auto_reload: true, cache: false }
services:
   cache.backend.null: { class:
Drupal\Core\Cache\NullBackendFactory }
```

This will disable all entity caching and twig caching so we do not have to bother about it while developing.

To re-enable production mode:

```
drupal site:mode prod --uir=mysite.test
```

If you don’t want to use drupal console for this you can

- copy **example.settings.local.php** in **sites/** folder to **settings.local.php**
- Put it in your **sites/mysite/** folder.

After doing this, uncomment the following in **settings.php**:

```
if (file_exists($app_root . '/' . $site_path .
'/settings.local.php')) {
  include $app_root . '/' . $site_path . '/settings.local.php';
}
```

And uncomment this:

```
$settings['container_yamls'][] = DRUPAL_ROOT .
'/sites/development.services.yml';
```

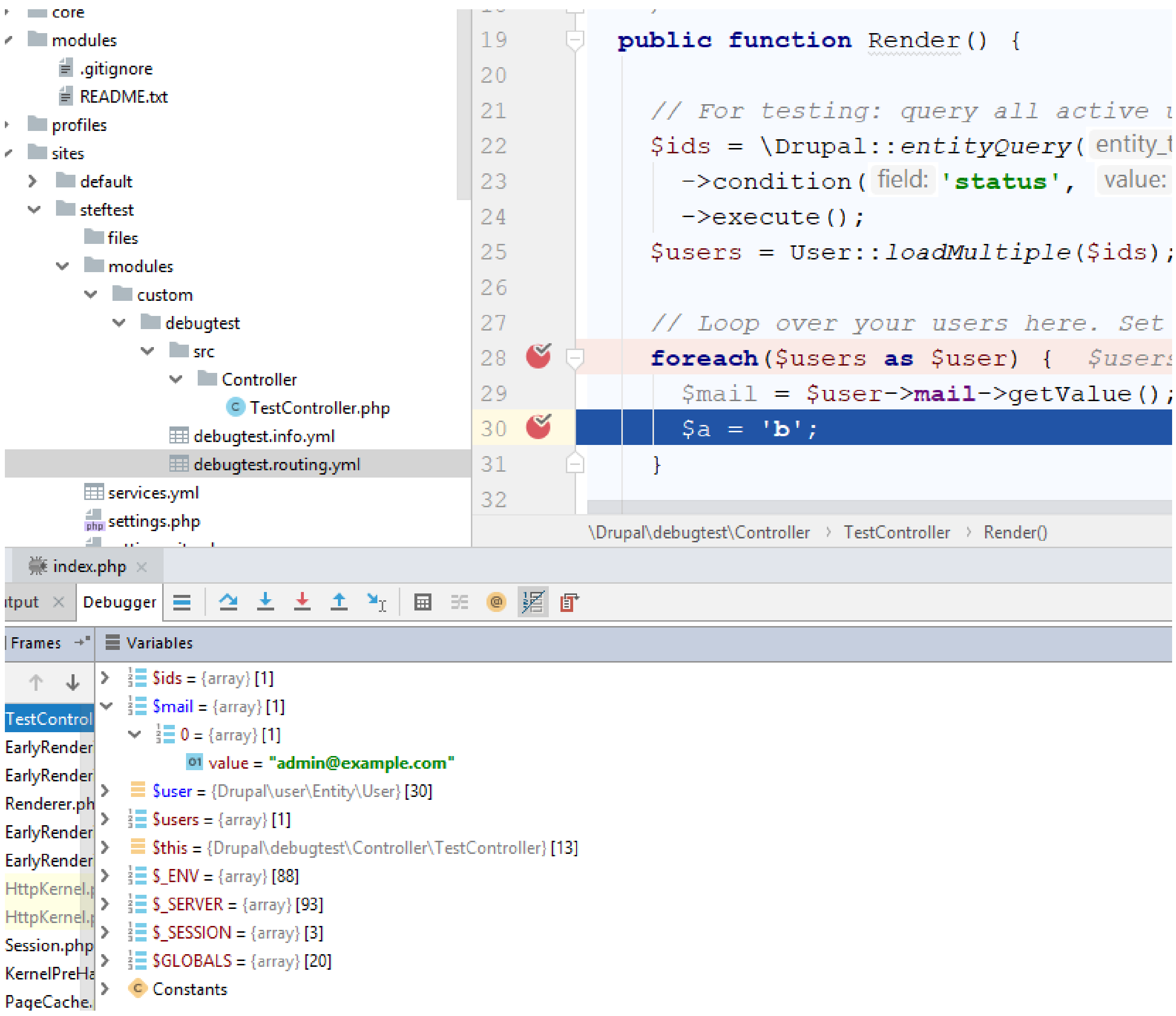
The end result is the same as running development mode in drupal console.

More info about development set-up on the official drupal.org website:
https://www.drupal.org/node/2598914

Xdebug

A real eye opener for me after a few years of programming with PHP was intercepting code with **Xdebug**. Most IDE’s have integration with Xdebug.

I use IntelliJ and have Xdebug connected with the IDE. This way, I can put my breakpoint anywhere in code and when I refresh the browser it intercepts the request and it keeps hanging right at my breakpoint. From there, I can take a look at my available variables, objects, etc.



```
core
modules
  .gitignore
  README.txt
profiles
sites
  default
  steftest
    files
    modules
      custom
        debugtest
          src
            Controller
              TestController.php
          debugtest.info.yml
          debugtest.routing.yml
    services.yml
    settings.php
```

```
public function Render() {

  // For testing: query all active u
  $ids = \Drupal::entityQuery( entity_t
    ->condition( field: 'status',  value:
    ->execute();
  $users = User::loadMultiple($ids);

  // Loop over your users here. Set
  foreach($users as $user) {   $users
    $mail = $user->mail->getValue();
    $a = 'b';
  }
```

\Drupal\debugtest\Controller › TestController › Render()

```
$ids = {array} [1]
$mail = {array} [1]
  0 = {array} [1]
    value = "admin@example.com"
$user = {Drupal\user\Entity\User} [30]
$users = {array} [1]
$this = {Drupal\debugtest\Controller\TestController} [13]
$_ENV = {array} [88]
$_SERVER = {array} [93]
$_SESSION = {array} [3]
$GLOBALS = {array} [20]
Constants
```

The interceptor stops exactly where I set my breakpoint. In my dummy line $a = 'b'; I see that my $mail variable is correct. So I can proceed coding further.

This kind of debugging is not only great for variables or objects. It can also give you a lot of knowledge about which stuff gets rendered in which order.

Some installation resources on debugging with Xdebug for php:

- Xdebug.org
- www.jetbrains.com/phpstorm/webhelp/configuring-xdebug.html
- https://7thzero.com/blog/configure-phpstorm-for-local-php-web-application-debugging
- https://www.jetbrains.com/help/phpstorm/debugging-with-phpstorm-ultimate-guide.html
- https://www.youtube.com/watch?v=rqDDJfG6ip4 (Advanced Debugging in PhpStorm - PhpStorm Video Tutorial)

Webprofiler

Drupal's well known Devel module contains a toolbar especially for developers. Let's install the module and enable Webprofiler.

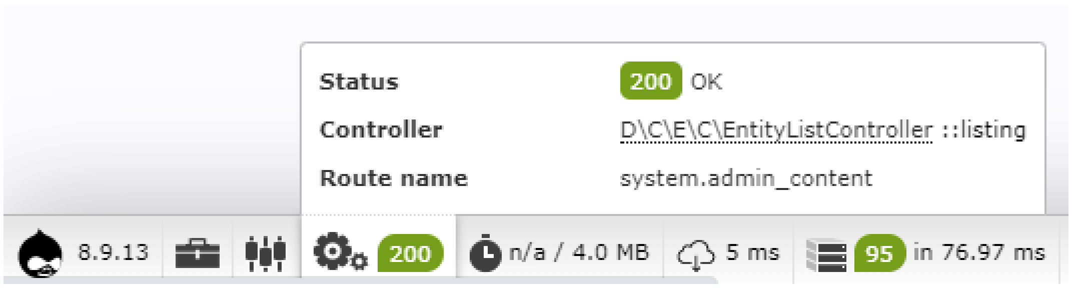
```
composer require drupal/devel
```

Once you have devel, install it via /admin/modules or with drush:

```
drush en devel webprofiler -y
```

After enabling the module, you'll get a toolbar at the bottom of your page on every request.

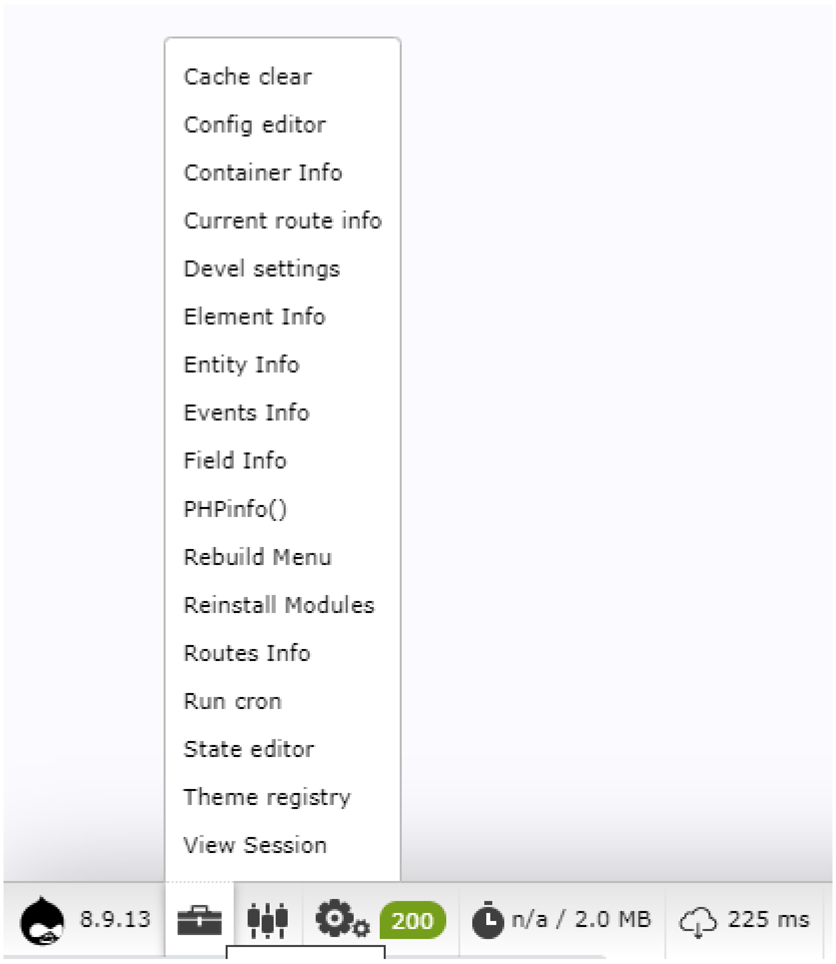
The toolbar gives me a ton of information about the current request. The one I use a lot is the Route name and Controller method used:



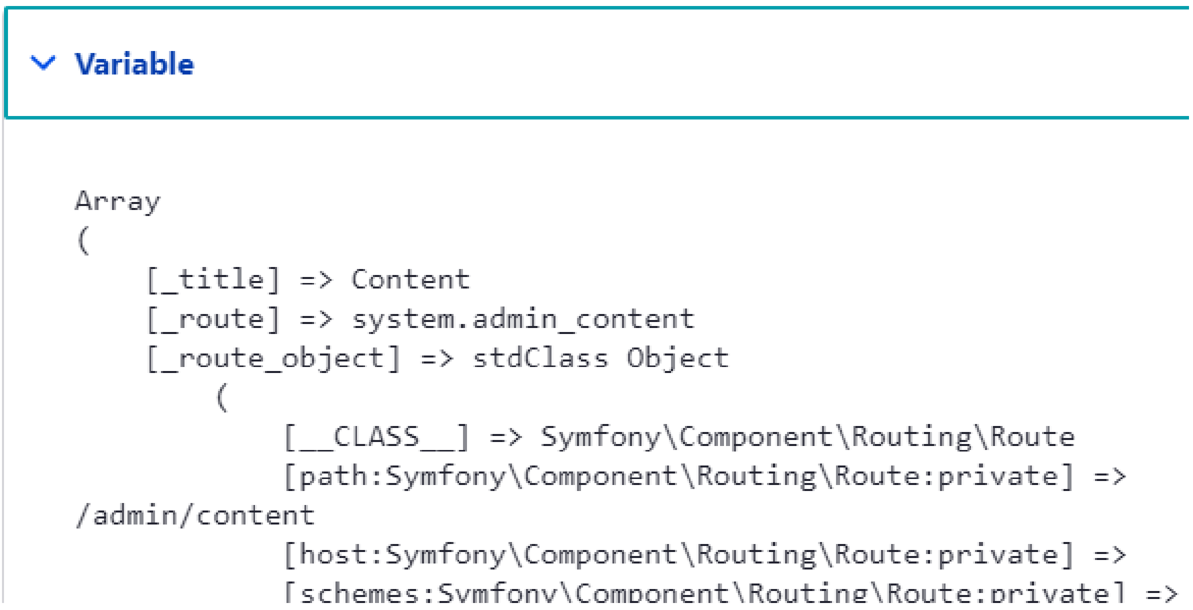
```
Status        200 OK
Controller    D\C\E\C\EntityListController ::listing
Route name    system.admin_content
```

Take a look at the current controller. We can see what the current controller name is, and which method is used.

The "Devel menu", the second button in the row, offers a lot of options:



The “Routes info” page, for example, let’s you lookup any registered route for debugging. This is the debug for the /admin/content page::



Variable

```
Array
(
    [_title] => Content
    [_route] => system.admin_content
    [_route_object] => stdClass Object
        (
            [__CLASS__] => Symfony\Component\Routing\Route
            [path:Symfony\Component\Routing\Route:private] =>
/admin/content
            [host:Symfony\Component\Routing\Route:private] =>
            [schemes:Symfony\Component\Routing\Route:private] =>
```

On the *Configure* page of this submodule (**/admin/config/development/devel/webprofiler**) there is even more information to display.

In the webprofiler toolbar you can check Events, Routing en Services. Click on an icon and go to the profiler reports page. Click around to see all routes, events, etc. that are registered.

Configuration management

Drupal 8 radically changed the way configuration is handled in comparison with the previous versions. Because this is a rather important topic to get, we start with some theoretical information on how drupal stores its configuration.

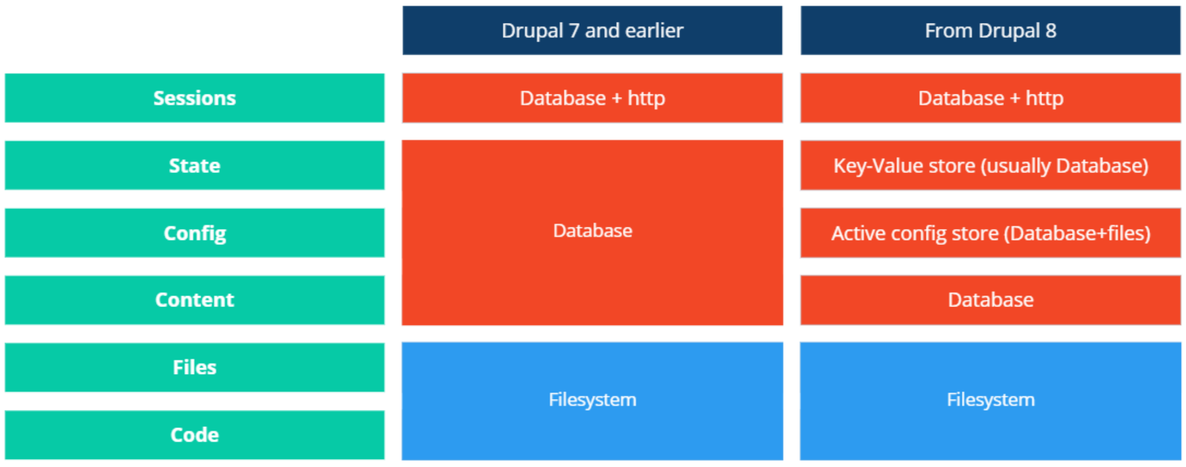
Basic configuration management

> Drupal uses a standard format for all site configuration—whether configuration of Drupal core components, or installed modules. This means that Drupal configuration can be exported and imported as YAML, which allows for staging of configuration changes, deploying configuration between sites, and easy version control.

Since drupal 8 configuration and content are separated. Configuration can be seen as literally ALL the settings of your platform. Some examples:

- Site name and slogan
- Which modules are enabled
- The settings of each module
- Placement of your blocks, block settings
- All your views settings
- Your entity types and fields
- Your google TagManager code
- … and so on

A visualisation of the difference in storing configuration in drupal 8 + 9 in comparison with earlier versions. Drupal moved from database-driven configuration in drupal 7 and earlier to configuration-driven in drupal 8.

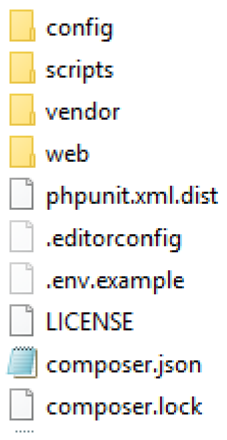


	Drupal 7 and earlier	From Drupal 8
Sessions	Database + http	Database + http
State	Database	Key-Value store (usually Database)
Config		Active config store (Database+files)
Content		Database
Files	Filesystem	Filesystem
Code		

The configuration storage exists of an Active store and a Sync store. The **Active store** is the current configuration that exists in your database. It can be different from the **Sync store**, which are yaml-files in a directory. We use this to import and export configuration between a development, staging and production version of websites.

Configuration management is shipped with drupal core by the **config** module. It provides import/export functionality for site configuration. Moreover it allows to **deploy configuration from one environment to another**, provided they are the same site.

We can export the configuration to yaml-files in a folder we choose. I prefer to keep configuration in the root, so our root folder becomes:



```
config
scripts
vendor
web
phpunit.xml.dist
.editorconfig
.env.example
LICENSE
composer.json
composer.lock
.....
```

We define the configuration folder in **settings.php**.. Look for the

```
$settings['config_sync_directory']
```

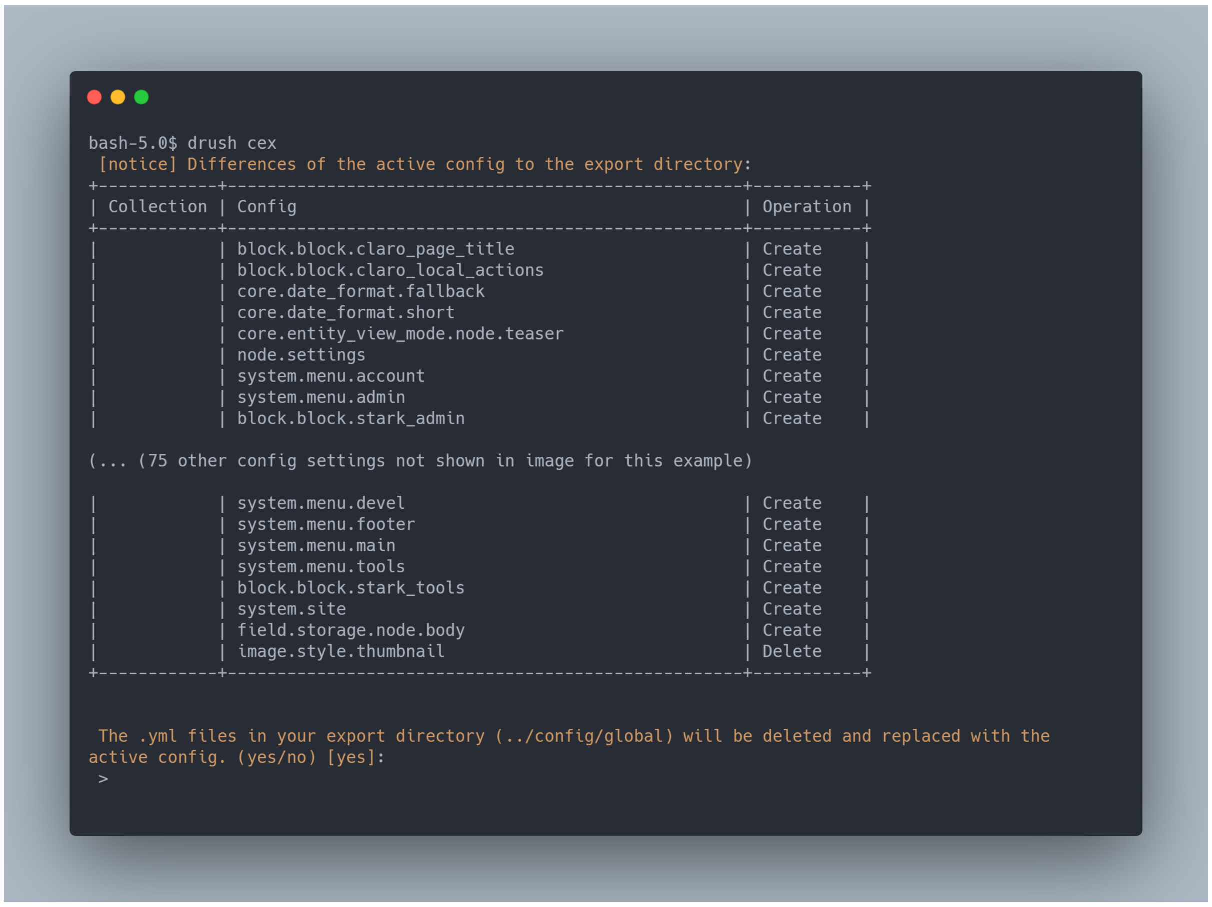
and set it to:

```
$settings['config_sync_directory'] = '../config/global';
```

Export all our current configuration with drush (drush cex is an alias for drush config-export)::

```
drush cex
```

Your terminal will return an overview of all the differences between your active storage (your database) and your sync storage (your config folder). Because your config folder is currently empty, it will list every setting your website has.



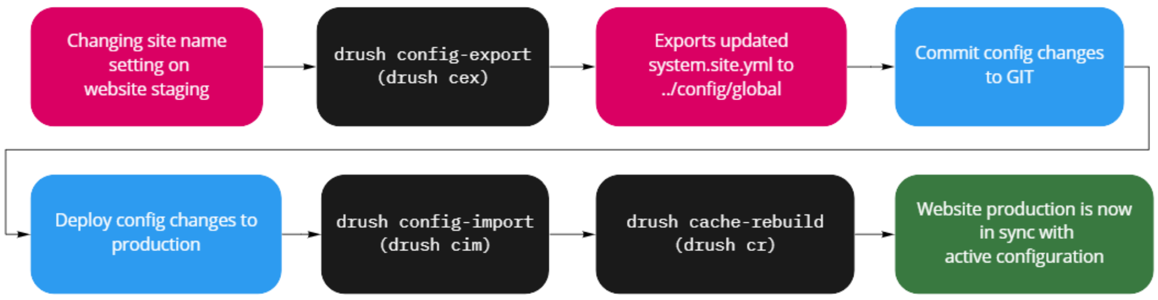
```
bash-5.0$ drush cex
 [notice] Differences of the active config to the export directory:
+------------+----------------------------------------------------+-----------+
| Collection | Config                                             | Operation |
+------------+----------------------------------------------------+-----------+
|            | block.block.claro_page_title                       | Create    |
|            | block.block.claro_local_actions                    | Create    |
|            | core.date_format.fallback                          | Create    |
|            | core.date_format.short                             | Create    |
|            | core.entity_view_mode.node.teaser                  | Create    |
|            | node.settings                                      | Create    |
|            | system.menu.account                                | Create    |
|            | system.menu.admin                                  | Create    |
|            | block.block.stark_admin                            | Create    |

(... (75 other config settings not shown in image for this example)

|            | system.menu.devel                                  | Create    |
|            | system.menu.footer                                 | Create    |
|            | system.menu.main                                   | Create    |
|            | system.menu.tools                                  | Create    |
|            | block.block.stark_tools                            | Create    |
|            | system.site                                        | Create    |
|            | field.storage.node.body                            | Create    |
|            | image.style.thumbnail                              | Delete    |
+------------+----------------------------------------------------+-----------+


 The .yml files in your export directory (../config/global) will be deleted and replaced with the
active config. (yes/no) [yes]:
 >
```

Answer with 'yes' or hit Enter key.

```
 [success] Configuration successfully exported to
../config/global.
```

Our config folder now contains all the configuration our website has. If we'd create a clean drupal installation with the same composer file and config folder, run 'composer install' and import our configuration we have a complete software platform with all functionalities ready. Except for the content, but we will use a custom drush command for that.

We use configuration management for bringing configuration from **website staging** to **website production**. Below is a schematic representation of the configuration management process:


Changing site name setting on website staging
drush config-export (drush cex)
Exports updated system.site.yml to ../config/global
Commit config changes to GIT
Deploy config changes to production
drush config-import (drush cim)
drush cache-rebuild (drush cr)
Website production is now in sync with active configuration

The structure of a configuration file, for example **system.site.yml**, looks like this:

```
uuid: e4a14ea3-a3fc-4e6f-b11a-e14fe1d8xxx
name: Offer platform
mail: noreply@offerplatform.mysite
slogan: ''
page:
  403: ''
  404: ''
  front: /user/login
```

Let’s change the slogan and see what happens. We can log in and fill it in at **/admin/config/system/site-information**, but we can also do it in the terminal with drush:

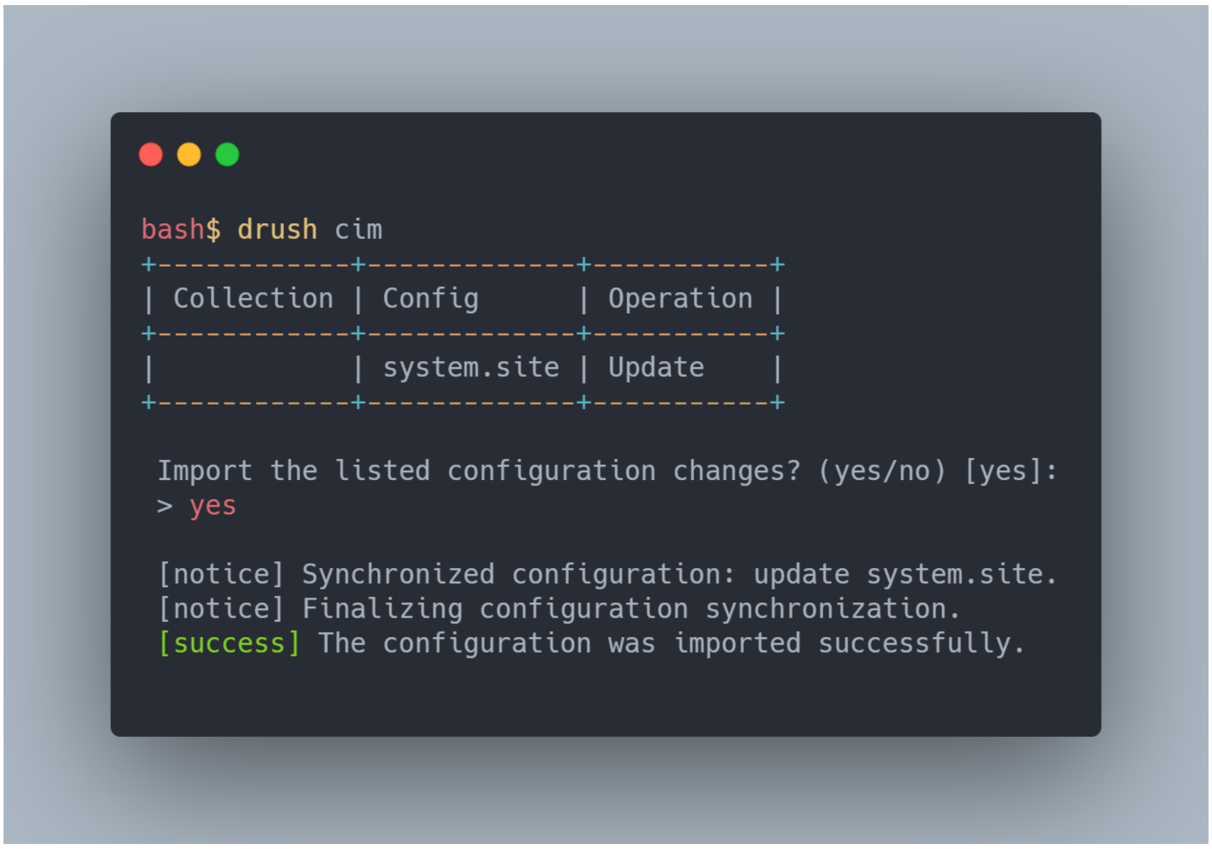
```
bash$ drush cset system.site slogan='Offer platform'
```

Note that this change only lives in the database (active store) for now. For every change we make, we have to make sure that if you like to keep the change or move the change to production you have to export the configuration to the config folder (sync store).

```
drupal cex
```

In production we can then run:

```
drupal cim
```



```
bash$ drush cim
+------------+-------------+-----------+
| Collection | Config      | Operation |
+------------+-------------+-----------+
|            | system.site | Update    |
+------------+-------------+-----------+

 Import the listed configuration changes? (yes/no) [yes]:
 > yes

 [notice] Synchronized configuration: update system.site.
 [notice] Finalizing configuration synchronization.
 [success] The configuration was imported successfully.
```

Putting configuration in yaml files is a smart thing to do. By running a config import command on production, all the new settings get imported and the site is updated. A huge improvement with earlier drupal versions, where we would have to do pretty advanced stuff for something simple as updating some settings.

This may look a very easy example, but it is an extremely scalable functionality. Whether it is only a simple change like a site name, or multiple entities, fields and module settings, configuration management will do the heavy lifting.

Creating custom configuration settings

Drupal core and (almost) every module come with configuration files. Most of these have settings that can be edited in the **/admin/config** section of the site. Let's add a custom settings form for a Google Tagmanager code, a typical thing we would like to have on a software platform. If you use drupal console you can do the following:

```
bash$ drupal generate:form:config --uri=mysite.local
```

First, add a module to **custom/modules** with the following structure:

- offer
    - offer.info.yml

In the **offer.info.yml** file, add:

```
name: offer
type: module
description: offer entity
core: 8.x
core_version_requirement: ^8 || ^9
```

Enable the module with drush:

```
bash$ drush en offer -y
 [success] Successfully enabled: offer
```

We add a file **offer/src/Form/CustomConfigForm.php** to build the configform. I started from a default extension of *ConfigFormBase()* and made sure the value gets saved to the right configuration file key in the submitForm() function.

To the *buildForm()* method I add a section for saving a textarea with a snippet.

The final code of our form:

```
 <?php

namespace Drupal\offer\Form;

use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;

/**
* Class CustomConfigForm.
*/
class CustomConfigForm extends ConfigFormBase {

 /**
  * {@inheritdoc}
  */
 protected function getEditableConfigNames() {
   return [
     'offer.customconfig',
   ];
 }

 /**
  * {@inheritdoc}
  */
 public function getFormId() {
   return 'custom_config_form';
 }

 /**
  * {@inheritdoc}
  */
 public function buildForm(array $form, FormStateInterface $form_state)
{
   $config = $this->config('offer.customconfig');

   $form['analytics'] = array(
     '#type' => 'details',
     '#title' => $this->t('Marketing & analytics'),
     '#open' => TRUE,
   );
   $form['analytics']['tagmanager'] = [
     '#type' => 'textarea',
     '#title' => $this->t('Tagmanager code'),
     '#default_value' => $config->get('tagmanager'),
     '#maxlength' => NULL,
   ];

   return parent::buildForm($form, $form_state);
 }

 /**
  * {@inheritdoc}
  */
 public function submitForm(array &$form, FormStateInterface
$form_state) {
   parent::submitForm($form, $form_state);

   $this->config('offer.customconfig')
     ->set('tagmanager', $form_state->getValue('tagmanager'))
     ->save();
 }

}
```

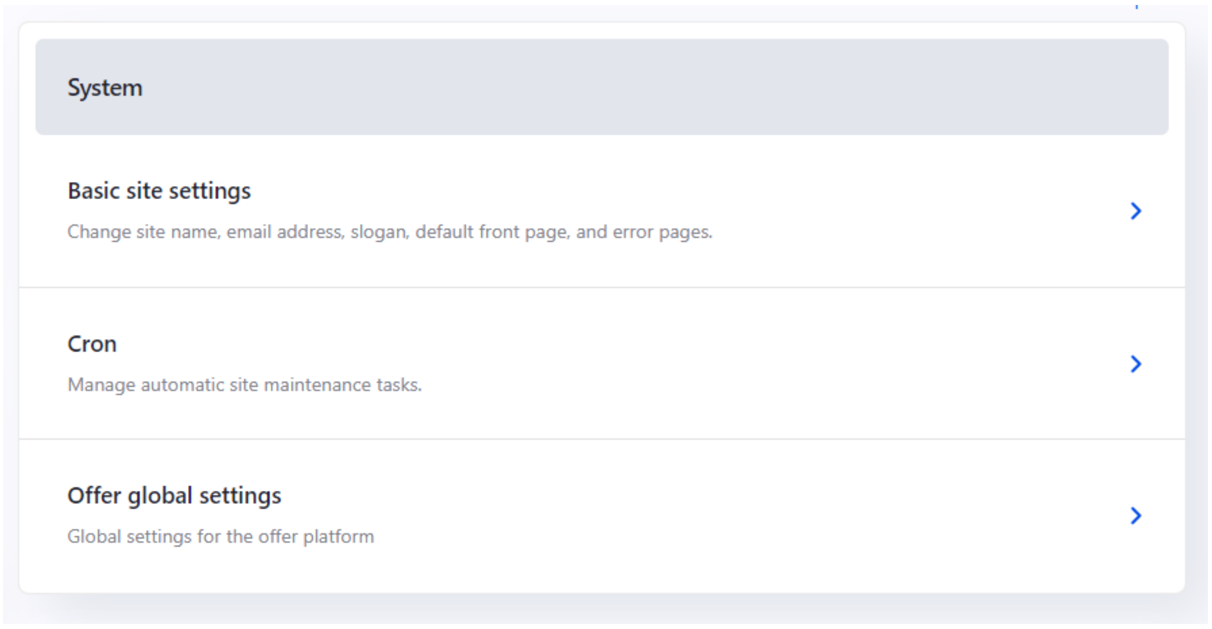
We now need to add **offer.links.menu.yml** to add the menu links in our back-end:

```
offer.config:
 title: 'Offer global settings'
 route_name: offer.config
 description: 'Global settings for the offer platform'
 parent: system.admin_config_system
 weight: 99
```

Then add a **offer.routing.yml** file to register the controller and route to the form:

```
offer.config:
 path: '/admin/config/offer/adminsettings'
 defaults:
   _form: '\Drupal\offer\Form\CustomConfigForm'
   _title: 'Offer platform global settings'
 requirements:
   _permission: 'administer site configuration'
```

Clear caches and head over to **/admin/config**



System

Basic site settings

Change site name, email address, slogan, default front page, and error pages.

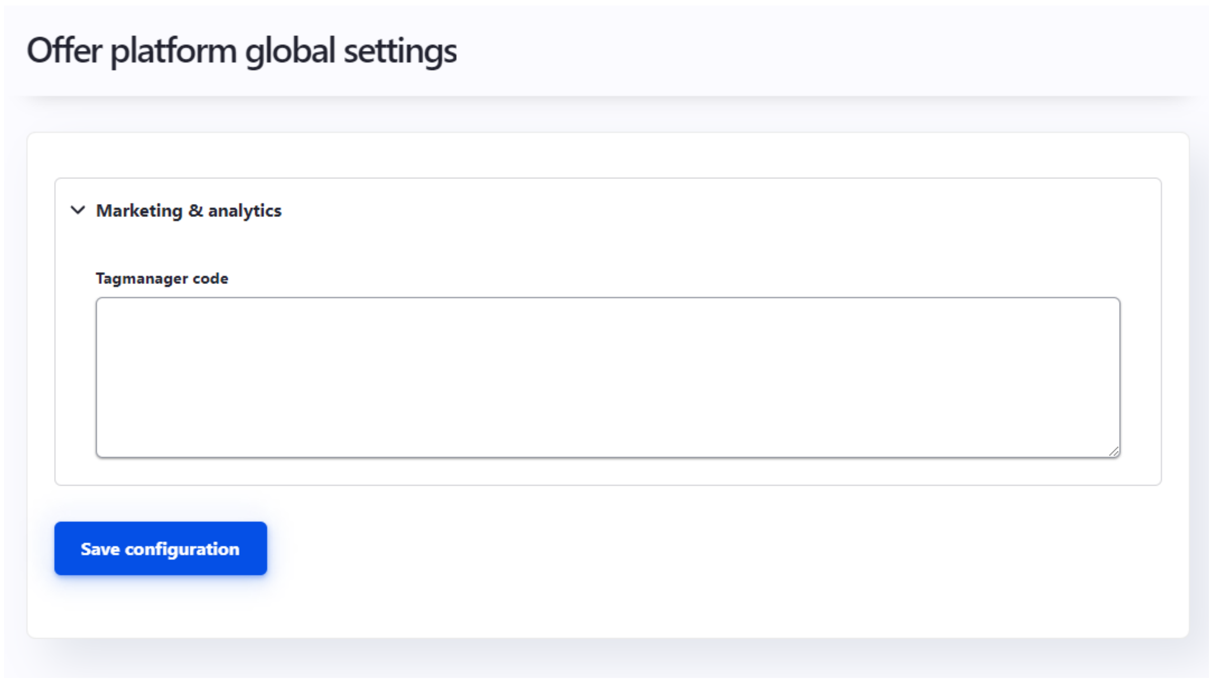
Cron

Manage automatic site maintenance tasks.

Offer global settings

Global settings for the offer platform

We see our custom menu link appearing under the “system” section. When we click we see our form:


Offer platform global settings

Marketing & analytics

Tagmanager code

When we save a value and export our configuration, we'll see a file called **offer.customconfig.yml** containing a key: 'tagmanager'.

Note that we could add a **config/install** folder to the offer module. This way, we can add a default setting for our tagmanager when enabling the module here. This is optional.

Creating custom configuration gives good insight in the behavior of configuration. It is possible to create modules with prepared configuration files shipped in its config folder.

Different configuration per environment

Because configuration management is the backbone of a good deployment process we need to dig a little deeper into the possibilities.

We talked about the Webprofiler module in an earlier section. This is typically something we only want in a development environment.

But there are more use cases where difference between configuration matters. Here are four examples. Note that in reality we would have a staging environment as well, but let's assume we only have development and production:

Functionality	Environment:	Environment:	Solution

		Development	Production	
A	Devel + Webprofiler	Enabled	Disabled	Settings.php (config exclude)
B	Caching	Disabled	Enabled	developer.services.yml file + settings.php (settings override)
C	Webform module and webforms fields	Enabled, bit not saved as configuration	Enabled, but not saved as configuration	Config filter module

Scenario A: Install a module on development, but not on production.

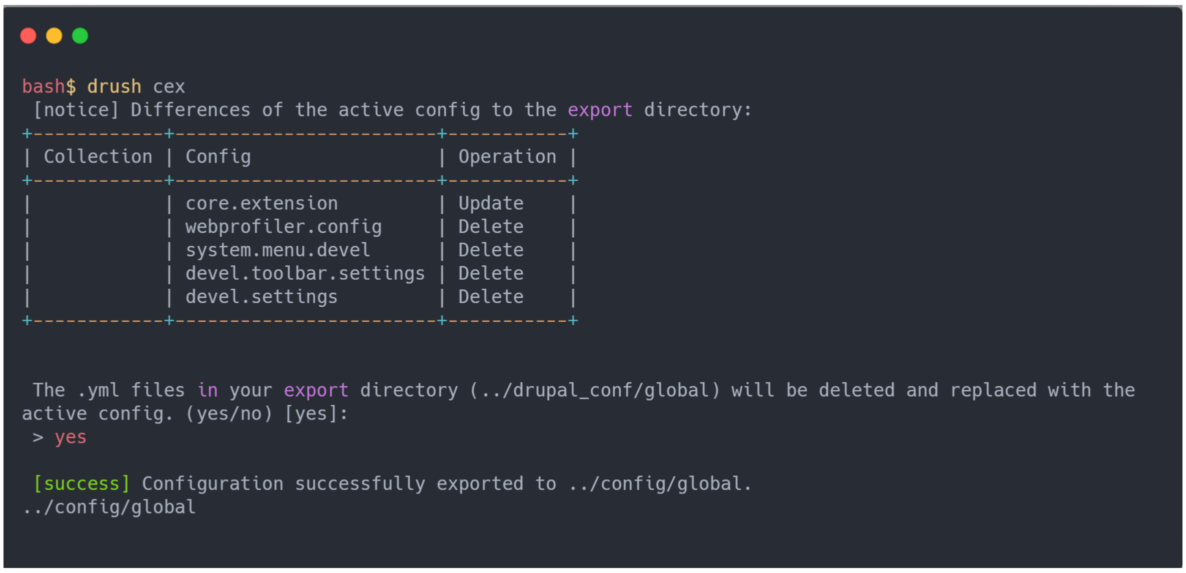
Since drupal 8.8 there is a core setting to exclude a module without any contrib modules.

The first step is to install *devel* and *webprofiler* module and make sure your sync store is up-to-date by doing *drush config-export -y*.

Then add this to your **settings.php file:**

```
$settings['config_exclude_modules'] = ['devel', 'webprofiler'];
```

After you've setup the modules you would like to exclude from your configuration, export your config again:



```
bash$ drush cex
 [notice] Differences of the active config to the export directory:
+------------+------------------------+-----------+
| Collection | Config                 | Operation |
+------------+------------------------+-----------+
|            | core.extension         | Update    |
|            | webprofiler.config     | Delete    |
|            | system.menu.devel      | Delete    |
|            | devel.toolbar.settings | Delete    |
|            | devel.settings         | Delete    |
+------------+------------------------+-----------+


 The .yml files in your export directory (../drupal_conf/global) will be deleted and replaced with the
active config. (yes/no) [yes]:
 > yes

 [success] Configuration successfully exported to ../config/global.
../config/global
```

You see that all of the configuration that was added by devel and webprofiler gets removed from the sync store. On a production site, there will be no devel and webprofiler.

We could go further and also make sure there is no devel module in our modules folder on production. For this, we would want to exclude it from our **composer.json** file by using the --*dev* flag. We would do the following. Note that it is best to do this while setting up your environments the first time:

```
$ composer require --dev drupal/devel
```

This results in those dependencies being added into the composer.json file under require-dev:

```
"require-dev": {
  "drupal/devel": "^4.0"
}
```

If you install the site without your dev modules you then would use:

```
$ composer install --no-dev
```

Scenario B: disable caching on development, but not on production.

You saw a lot of this in the development & debug chapter. Some can be solved by adding a development.services.yml file. But the following can be done by overwriting configuration on development. So in this case you enable all the caching mechanisms in your configuration (sync store) but you overwrite them locally in your **settings.local.php**:

```
$config['system.performance']['css']['preprocess'] = FALSE;
$config['system.performance']['js']['preprocess'] = FALSE;
```

Uncomment these lines to disable the render cache and disable dynamic page cache:

```
$settings['cache']['bins']['render'] = 'cache.backend.null';
```

```
$settings['cache']['bins']['dynamic_page_cache'] =
'cache.backend.null';
$settings['cache']['bins']['page'] = 'cache.backend.null';
```

Disabling caching has some drawbacks. The whole site responds slower, and rebuilding all caches takes a while too. Doing this a lot, and all those seconds waiting for a response add up. For an approach to developing with the cache enabled, see Drupal 8 development with caching on.

One bonus tip I can give you is to enable verbose error messaging on your local development. By doing this you get rich error messages instead of the default "something went wrong" message:

```
$config['system.logging']['error_level'] = 'verbose';
```

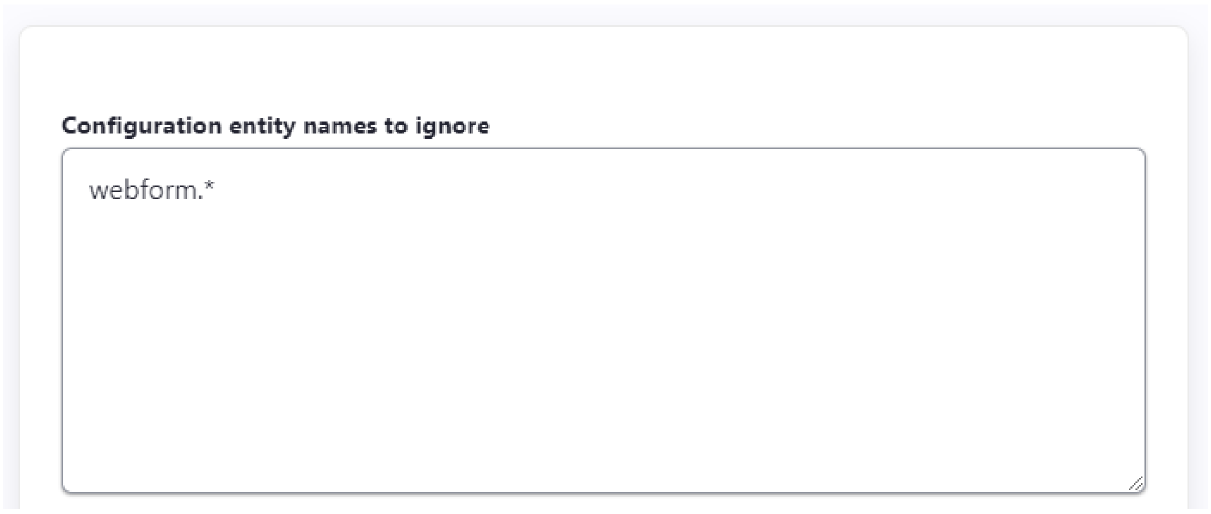
Scenario C: enable a module on both development and production, but ignore its configuration.

A real-life scenario you will be confronted with is when webmasters have access to modules that *create* configuration in your active store. An example is the webform module. Every form, form field or setting a webmaster adds will create additional configuration. The risk here is that when a developer deploys his development changes, and this configuration, he risks removing a newly created form.

The solution for this scenario is the config filter module. With this module you can tell both your development and production environment to ignore *parts* of your configuration.

First, install and enable the module on both development and production.

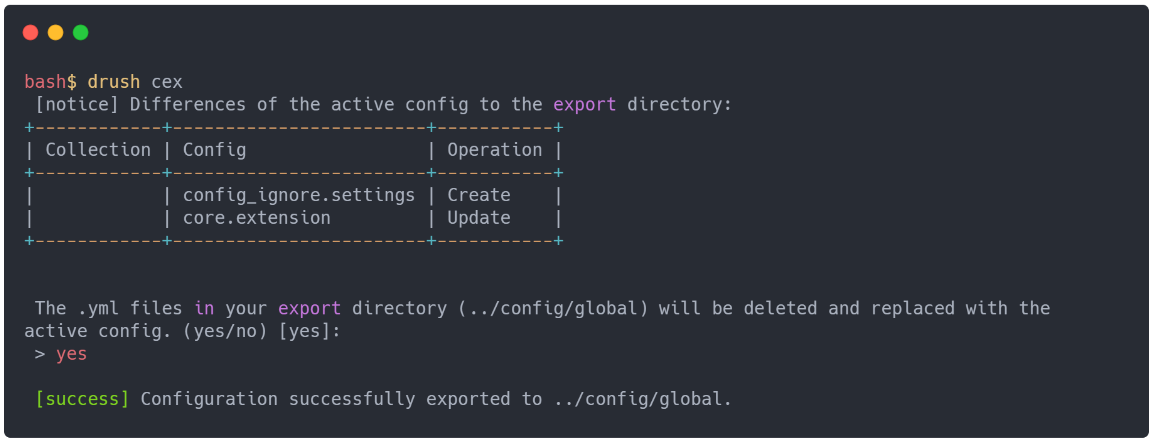
Second, go to **/admin/config/development/configuration/ignore.** There is a textarea where you can add the keys of the configuration you would like to ignore.


Configuration entity names to ignore

```
webform.*
```

Add *webform.** to the textarea and save.

Third, export your configuration and import it on production. This is important: both environments need to have the module installed before you can start ignoring certain config.



```
bash$ drush cex
 [notice] Differences of the active config to the export directory:
+------------+------------------------+-----------+
| Collection | Config                 | Operation |
+------------+------------------------+-----------+
|            | config_ignore.settings | Create    |
|            | core.extension         | Update    |
+------------+------------------------+-----------+


 The .yml files in your export directory (../config/global) will be deleted and replaced with the
active config. (yes/no) [yes]:
 > yes

 [success] Configuration successfully exported to ../config/global.
```

Fourth, enable the webform module and start adding forms. If you now export your content, it will still export your yaml files BUT your production will not read them when you import them.

The **config filter** module is great, but you must get comfortable using it. Start with easy configuration. I would also recommend either to ignore the entire module or not ignoring it. Ignoring parts of configuration is possible, but can be dangerous.

Reading configuration objects in your code

We know that our active configuration resides in the database. Often we want to read out configuration in our custom code. Below is a snippet on how to read out the google tagmanager snippet we've created in our previous chapter about custom configuration:

With this example snippet you make the variables available in your page.twig.html file. You can use *hook_preprocess_hook()* in your **mytheme.theme** file or like below In your **mymodule.module** file:

```
/**
 * Implements hook_preprocess_html().
 */
function MYMODULE_preprocess_html(&$variables) {
  $variables['tagmanager'] =
\Drupal::config('offer.customconfig')->get('tagmanager');
}
```

Now, in your html.html.twig file of your theme, just use the following in your *<head>* section:

```
{{ tagmanager }}
```

Clear caches and the snippet will appear.

Part 2: Project code and set-up

Seed data

Before we start setting up the project, I would like to talk about so-called *seed data*.

Drupal now has become an option for platforms that were typically built with tailor-made software solutions.

But one thing it lacks is **seed data**. Seed data is data that you populate the database with at the time it is created. You use seeding to provide initial values for lookup lists, for demo purposes, proof of concepts and of course for development.

We always want to achieve the following: a new developer comes into our team and installs everything that is in the composer file. After this, he imports the configuration (more on configuration later).

But then there is an empty software platform. Pretty annoying for debugging and programming.

What we did in the earlier drupal days, was use a database with production data. This is in many ways a bad practice:

- **Privacy**: when people put their data into a platform, they do not want their data to be visible for every developer. We should avoid this.
- **Security**: what about sending out emails to production customers by accident. Ask 10 developers and 5 of them will tell you this happened some time.

While there are some things we can do, console for example has a content generation function, we'll do this with a custom console command. I prefer realistic seed data instead of Lorem ipsum. Let's start by writing a class.

To **modules\custom\offer\src** we add a directory called **SeedData**. We add a file called **SeedDataGenerator.php to modules\custom\offer\src\SeedData.**

The class will for now just create a dummy user. In the project files it has become a large function to import all of our entity data with dummy users, offers and bids.

For now we'll just add a test user. in This is what our class looks like:

```php
<?php

namespace Drupal\offer\SeedData;

use Drupal\user\Entity\User;

/**
 * Class SeedGenerator
 * @package Drupal\offer
 */
Class SeedDataGenerator {

  /**
   * Function to create the Seed data
   * @param string $entity
   *   The type of entity that needs to be created.
   * @return integer $count
   *   The number of entities created.
   */
  public function Generate($entity) {

    $count = 0;

    switch ($entity) {
      case 'user':
        // USER SEEDS
        $user = User::create();
        $user->setUsername('test');
        $user->setPassword('test');
        $user->setEmail('test@mail.com');
        $user->activate();
        $user->enforceIsNew();
        if($user->save()) {
          $count += 1;
          return $count;
        }
        break;
    }

    return null;
  }
}
```

Proceed with adding a drush command that will trigger the class:

Add **custom/offer/src/Commands/SeedGeneratorCommand.php** and configure further. The final file looks like this:

```
<?php

namespace  Drupal\offer\Commands;

use Drush\Commands\DrushCommands;
use Drupal\offer\SeedData\SeedDataGenerator;
use Drush\Drush;

/**
 * Class SeedGeneratorCommand
 * @package Drupal\offer\Commands
 */
class SeedGeneratorCommand extends DrushCommands {

  /**
   * Runs the OfferCreateSeeds command. Will create all data for
the Offer platform.
   *
   * @command offer-create-seeds
   * @aliases offercs
   * @usage drush offer-create-seeds
   *  Display 'Seed data created'
   */
  public function OfferCreateSeeds() {
    $seed = new SeedDataGenerator();
    $count = $seed->Generate('user');
    Drush::output()->writeln($count . ' user(s) created');
  }
}
```

One more thing. Add a file **custom/offer/offer.services.yml** and add:

```
services:
 offer.commands:
   class: \Drupal\offer\Commands\SeedGeneratorCommand
   tags:
     - { name: drush.command }
```

That's it! Clear cache and see if our system has registered our command. Lookup all available commands for our offer module like this:

```
bash$ drush list | grep 'offer'
 offer-create-seeds (offercs)                Runs the
OfferCreateSeeds command. Will create all data for the Offer
platform.
```

This is nice. This way we can add commands for every module in our system. Developers can "grep" for every name of the module and see all the commands it has.
Let's run the command. If everything goes well we would create a new user with the command:

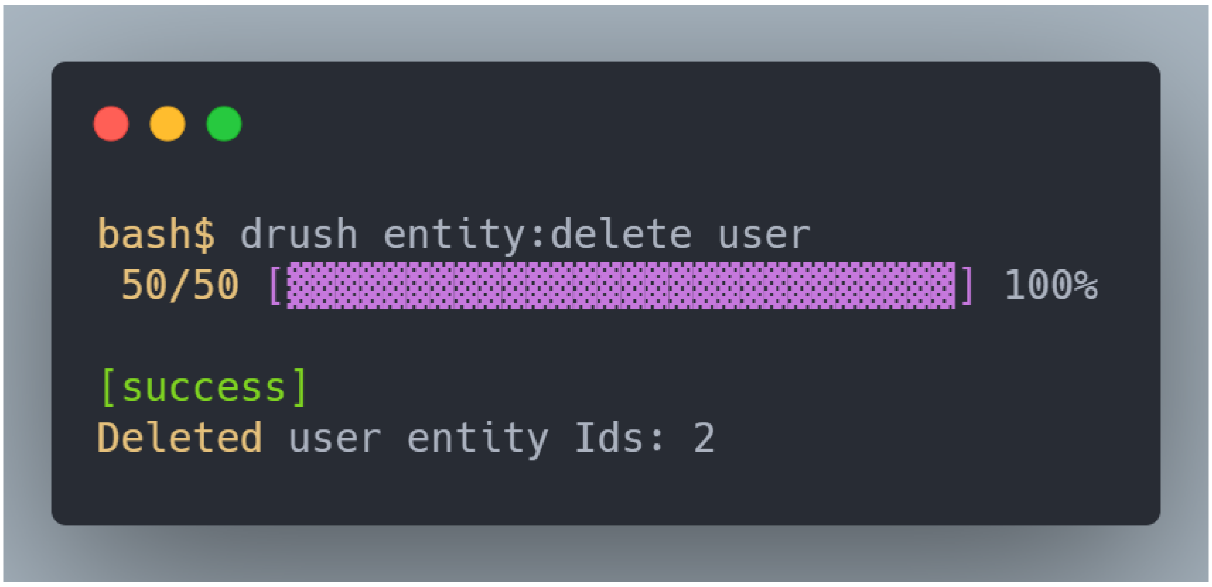
```
bash$ drush offer-create-seeds
1 user(s) created
```

We see we have imported the test user. We're forever done with using production databases. Extending the class after each entity we create won't take long and we'll just add 2 to 5 items per entity type.

This will become easier with every project you'll start. This way, we truly separated configuration from content, a necessary step to advanced software development!.

With drush it is easy to start playing to create and delete content. If you have created your user(s), you can easily delete them as well. Just type this in your terminal:

```
bash$ drush entity:delete <entityType>
```



```
bash$ drush entity:delete user
 50/50 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

[success]
Deleted user entity Ids: 2
```

Note that this will not delete the admin user, luckily!

Let's move on with setting up the project files

Project set-up

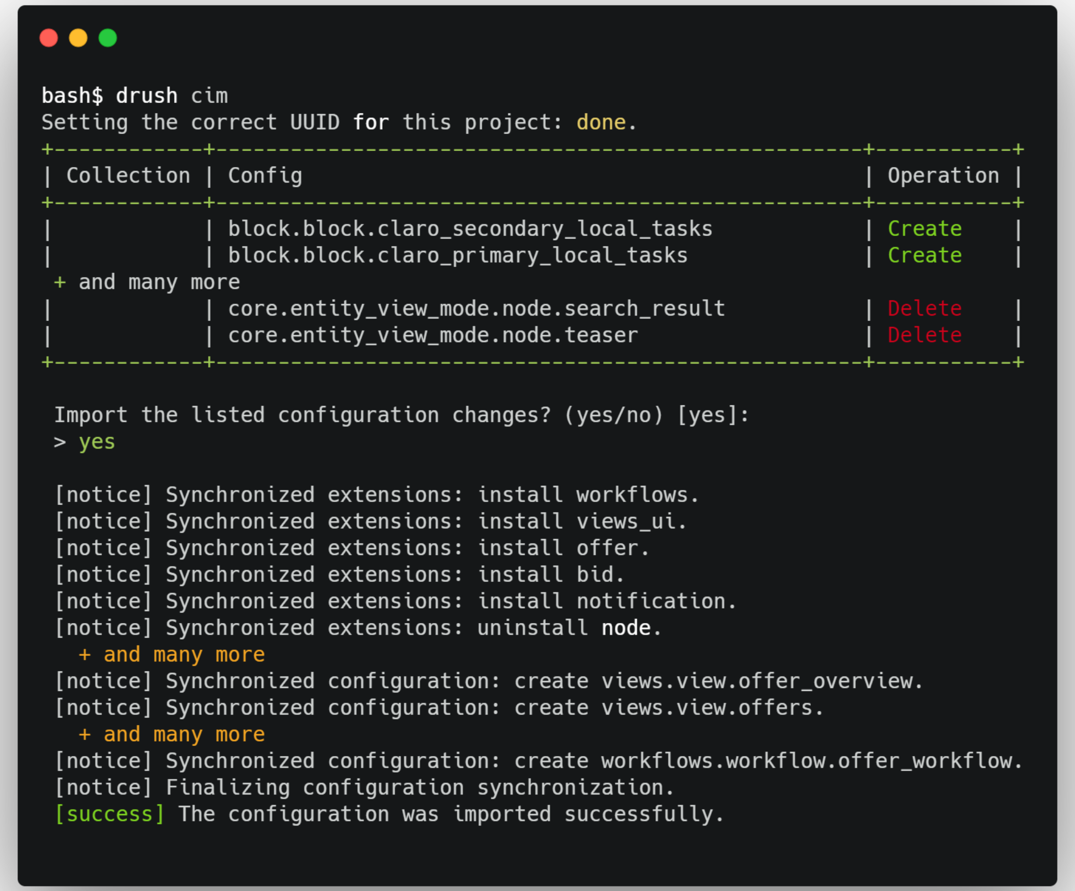
This chapter is your installation guide on how to set up the project code. Look at the last page of the book for the link to download the files.

The following steps are needed to set up the platform:

1. Past the project files into a directory. Make sure the *web* and *drush* folder are in the root of the project.
2. Make sure your local domain is pointed to the ***web*** folder
3. Run composer install in the root of your installation. This will download all the required packages for the platform, and add the map structure.
4. Fill in your database credentials via the UI and install your drupal site.
5. In your settings.php, as read in the configuration management chapter, at the bottom add

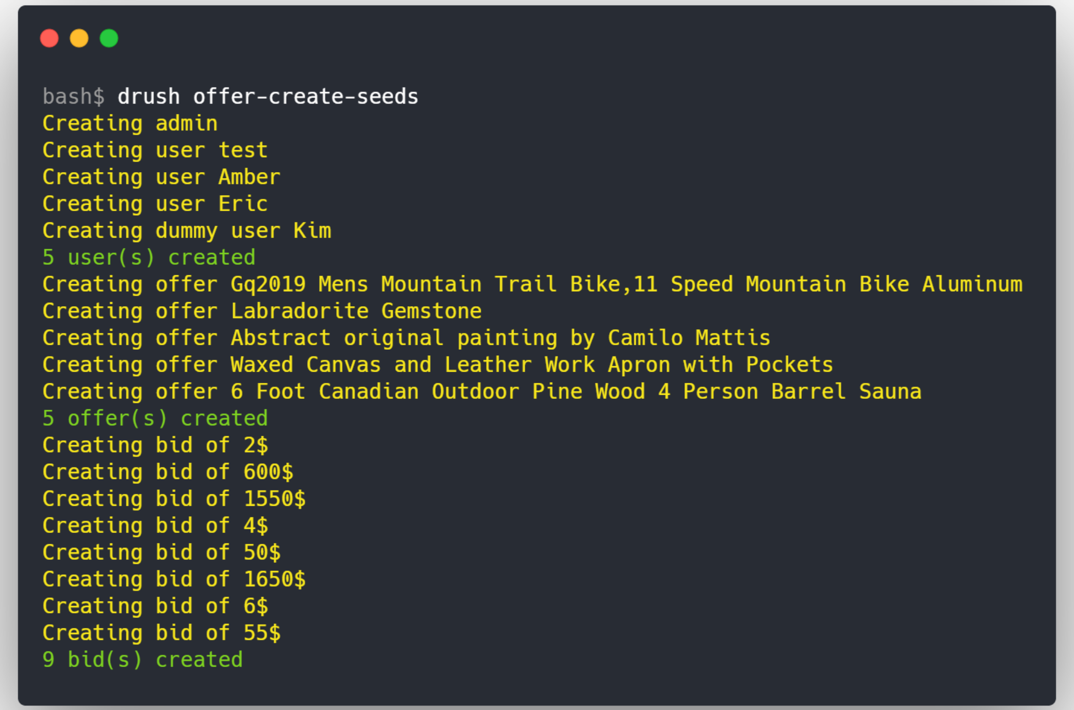
```
$settings['config_sync_directory'] = '../config/global';
```

6. Run drush config-import -y to enable the required modules and import all configuration that comes with them



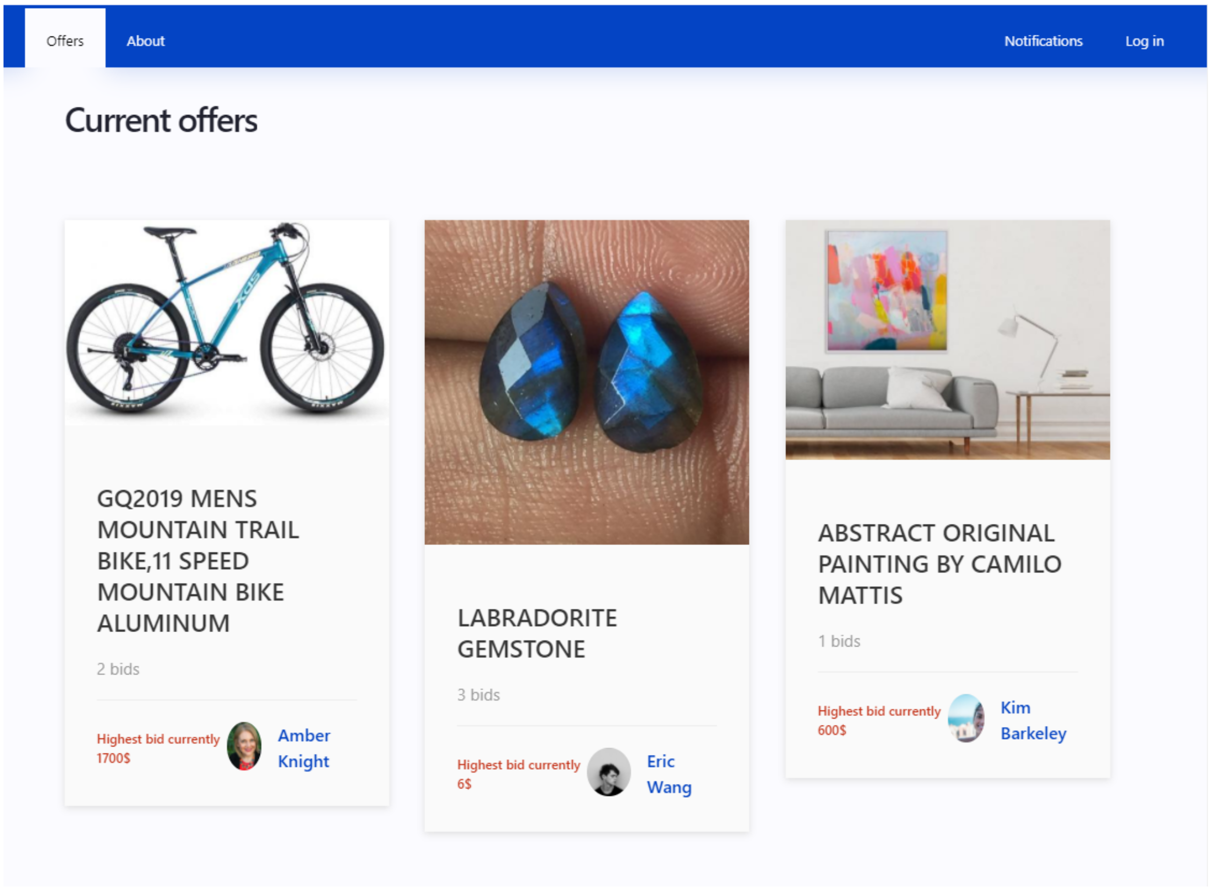
```
bash$ drush cim
Setting the correct UUID for this project: done.
+------------+-----------------------------------------------------+----------+
| Collection | Config                                              | Operation |
+------------+-----------------------------------------------------+----------+
|            | block.block.claro_secondary_local_tasks             | Create    |
|            | block.block.claro_primary_local_tasks               | Create    |
 + and many more
|            | core.entity_view_mode.node.search_result            | Delete    |
|            | core.entity_view_mode.node.teaser                   | Delete    |
+------------+-----------------------------------------------------+----------+

 Import the listed configuration changes? (yes/no) [yes]:
 > yes

 [notice] Synchronized extensions: install workflows.
 [notice] Synchronized extensions: install views_ui.
 [notice] Synchronized extensions: install offer.
 [notice] Synchronized extensions: install bid.
 [notice] Synchronized extensions: install notification.
 [notice] Synchronized extensions: uninstall node.
   + and many more
 [notice] Synchronized configuration: create views.view.offer_overview.
 [notice] Synchronized configuration: create views.view.offers.
   + and many more
 [notice] Synchronized configuration: create workflows.workflow.offer_workflow.
 [notice] Finalizing configuration synchronization.
 [success] The configuration was imported successfully.
```

7. Run drush offer-create-seeds to import all of your dummy content



```
bash$ drush offer-create-seeds
Creating admin
Creating user test
Creating user Amber
Creating user Eric
Creating dummy user Kim
5 user(s) created
Creating offer Gq2019 Mens Mountain Trail Bike,11 Speed Mountain Bike Aluminum
Creating offer Labradorite Gemstone
Creating offer Abstract original painting by Camilo Mattis
Creating offer Waxed Canvas and Leather Work Apron with Pockets
Creating offer 6 Foot Canadian Outdoor Pine Wood 4 Person Barrel Sauna
5 offer(s) created
Creating bid of 2$
Creating bid of 600$
Creating bid of 1550$
Creating bid of 4$
Creating bid of 50$
Creating bid of 1650$
Creating bid of 6$
Creating bid of 55$
9 bid(s) created
```

Clear the caches. You now have the full platforms ready-to-go that should look like this:



Current offers

GQ2019 MENS MOUNTAIN TRAIL BIKE,11 SPEED MOUNTAIN BIKE ALUMINUM

2 bids

Highest bid currently 1700$ Amber Knight

LABRADORITE GEMSTONE

3 bids

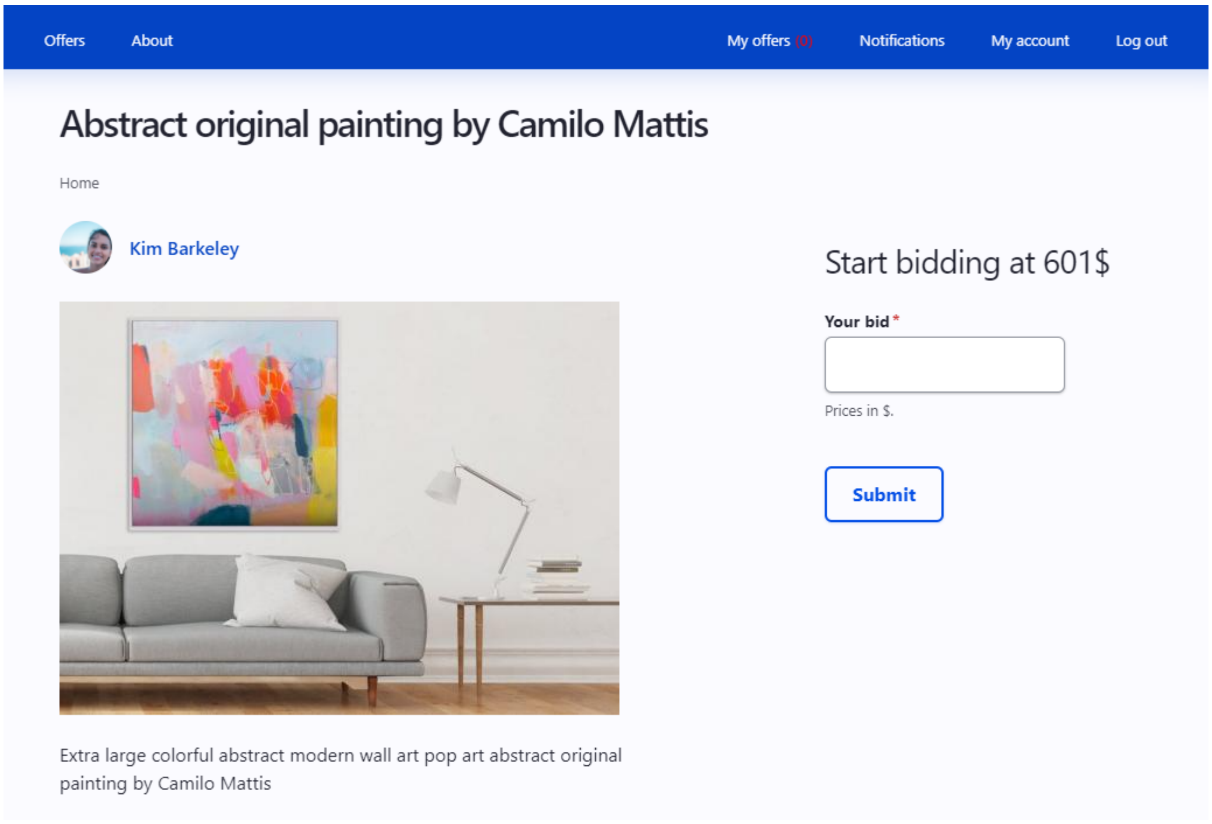
Highest bid currently 6$ Eric Wang

ABSTRACT ORIGINAL PAINTING BY CAMILO MATTIS

1 bids

Highest bid currently 600$ Kim Barkeley

Go to **/user/register** to register and directly log in with user "test" and password "test. Place a bid on an offer to see how it works.



Abstract original painting by Camilo Mattis

Home

Kim Barkeley

Start bidding at 601$

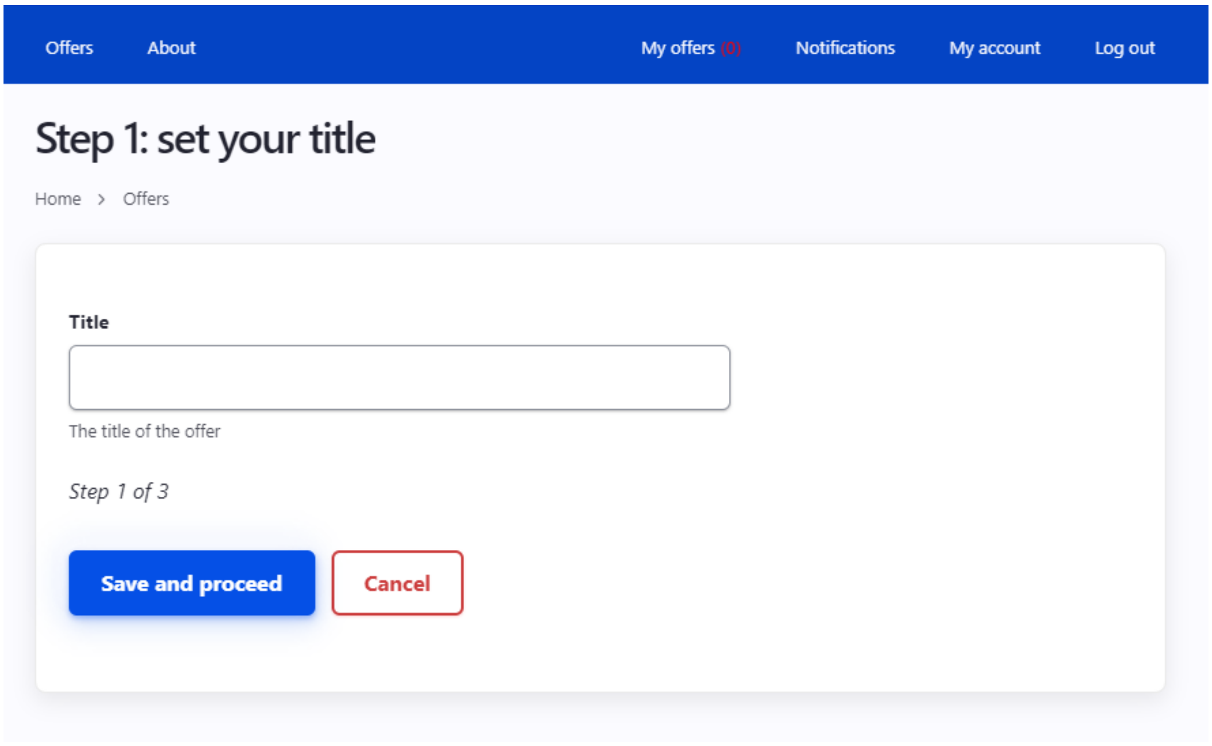
Your bid *

Prices in $.

Submit

Extra large colorful abstract modern wall art pop art abstract original painting by Camilo Mattis

Click on "**My offers**" to add your own offer via a multi-step interface.



Step 1: set your title

Home > Offers

Title

The title of the offer

Step 1 of 3

Save and proceed

Cancel

In the following chapters we will be building the platform (modules, settings, …) from scratch, step by step. You can use this course in two ways:

1. **Learn by reading** the chapters and looking into the final platform and code
2. **Learn by doing**: follow the course and create every file yourself. There are only a few additional things that were added to the final code, which is discussed at the end of this course. This second way is probably the most effective way to learn!

💻 If you follow this course **by doing** you will now and then see a grey box like this with additional info like when to enable a new module etc.

Having problems setting up the platform? Click the link below for some troubleshooting and how to contact me:
https://stefvanlooveren.me/troubleshooting

Part 3: Custom entities 101, CRUD operations, workflow states and access

👉 In this **3rd part** of the course you will learn everything concerning **custom entities**. At the end you will able to create your own entities, master how to add functionality using **annotations** to add **views support**, make the entities **fieldable**, add a **workflow** with transitions and importantly, detailed **access handling**.

On a software point of view this teaches you how to add data entry points using **form modes** and **view modes**. You will be able to create **multistep forms** for user friendly user input (CRUD) using drupal's typical field api and the core **media** module for images.

With our setup ready and the theory covered, it is time to get up speed with our platform. In this section we'll dive into the main core concepts we'll need to start adding data into our software. We start with diving into a very interesting core concept: entities.

A brief history: since drupal 7 the software contains generic data models that can be extended for specific purposes. Some examples in core are:

- Nodes
- Blocks
- Comments
- Taxonomy
- Users

Drupal became famous because of its **'fieldable'** functionality of these entities. Whether it is a user or a comment, they can easily be extended by extra fields in the same way:

- List items
- Textfield
- Entity references
- Media
- Link field
- …

Moreover these fields are translatable and can have multiple instances as well (multifield).

I believe there is no other software that has such a strong fieldable model as drupal 9. This makes me prefer drupal over a solely Symfony or Django platform in the majority of use cases. I hope I can show you why in this course.

Content entities

A content entity is a generic fieldable model that is defined in code. It inherits all functionality from the base Entity model defined in drupal core.

Take a look at **core/modules/user/src/Entity/User.php** and click further on the classes the current class you are in extends:

class User extends **ContentEntityBase** implements UserInterface
-> abstract class ContentEntityBase extends **EntityBase**
-> abstract class EntityBase implements **EntityInterface**

This Object-oriented way of data-modelling makes sense. All the classes that inherit from EntityBase get a ton of functionality for free:

- Makes the entity **fieldable** if desired (start adding fields via the ui on-the-go)
- Makes the entity **translatable,** if desired
- **Inherits various generic methods** for obtaining data
- Can make use of the core **workflow** (for moderation status of the entities: draft, published, expired, ...)
- Can make use of the powerful core **revisioning** system (keeps a history of the entity on every change)
- Create, read, update, and delete (**CRUD**) functionality with the desired security.
- **Views** integration by default. Views is the core module for creating lists with filtering, search etc.

The following generic methods for content entities are provided in core:

```
Entity::create()
Entity::load()
Entity::save()
Entity::id()
Entity::bundle()
Entity::isNew()
Entity::label()
```

An entity is purely defined in code, unless:

- We add bundles to the entity (f.e. Node entity has bundle ‘article’, ‘page’ in the standard profile). This is not what we will do in this course.
- We add fieldable functionality to the entity (this is an option when creating content entities) to add fields via the interface. We will do this in this course.

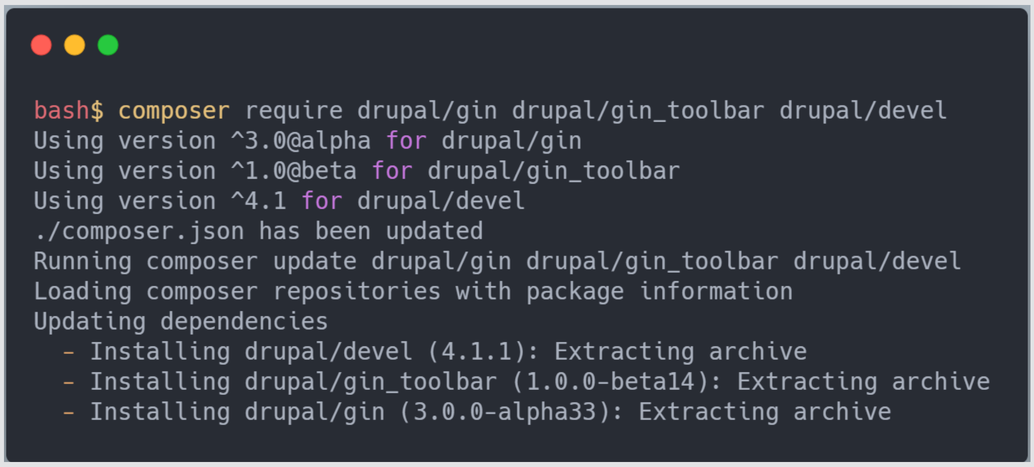
When one of the previous conditions is met, the extra configuration gets stored in the database and comes with yaml files when exporting configuration.

Building our first content entity

👉 This section teaches you how to define a custom entity and create it in the database. At the end you will be able to create your own custom entity with custom tailored **base fields** and revisions support.

💻 If you follow this course **by doing**:
After you installed the *recommended* drupal 9 set-up like described in the composer chapter, and added the custom configuration form in a custom module (offer), **install the following theme**:
- drupal/gin

And the following modules:
- drupal/gin_admin_toolbar
- drupal/devel



```
bash$ composer require drupal/gin drupal/gin_toolbar drupal/devel
Using version ^3.0@alpha for drupal/gin
Using version ^1.0@beta for drupal/gin_toolbar
Using version ^4.1 for drupal/devel
./composer.json has been updated
Running composer update drupal/gin drupal/gin_toolbar drupal/devel
Loading composer repositories with package information
Updating dependencies
  - Installing drupal/devel (4.1.1): Extracting archive
  - Installing drupal/gin_toolbar (1.0.0-beta14): Extracting archive
  - Installing drupal/gin (3.0.0-alpha33): Extracting archive
```

Make gin the default theme (also the administration theme) via admin/settings/appearance. Also in the theme settings, set the toolbar as “horizontal, modern toolbar” and disable the “Users can override Gin settings”.

The first question that gets raised is why would we use custom content entities. Isn’t the core node entity with it’s subtypes (bundles) enough?

If we'd have a simple website with just some blog posts and a portfolio, I'd always recommend to use the core Node content entity. It is the de facto out-of-the-box solution for this.

But our platform aims to have full control over all pages that create, edit and delete content, as well as the overviews. Custom entities give us more power to define our own access functions.

We build a platform that allows users to create offers as well as to make a bid on offers. It would not make sense to use Nodes with bundles like this:

Entity Node
  Bundle Offer
  Bundle Bid

I'd have to add numerous access checks to make sure users only get access to their own Offer entities and only their own Bids because they are from the same Entity. Drupal's node behaviour wasn't really meant to separate access between these kind of node types as well. No, instead we do:

Entity Offer
Entity Bid
...

Proper modelling of our data is crucial. The **Entity API** provides us all the tools for doing this.

We start with creating a content entity 'Offer'.

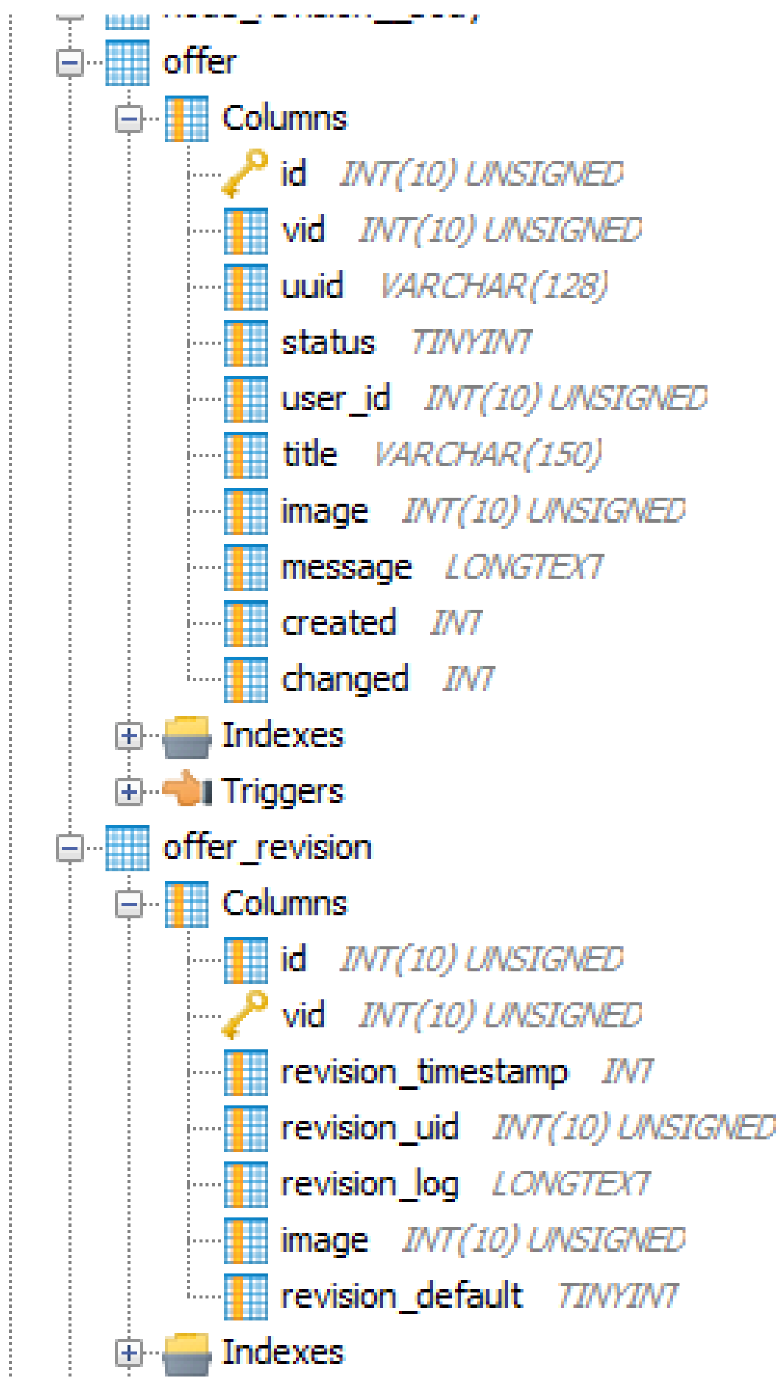
A file named **Offer.php** file inside **modules/custom/offer/src/Entity** will define our entity. Copy this code to define the entity:

```
<?php
/**
* @file
* Contains \Drupal\offer\Entity\Offer.
*/

namespace Drupal\offer\Entity;
```

```
use Drupal\Core\Entity\EditorialContentEntityBase;
use Drupal\Core\Field\BaseFieldDefinition;
use Drupal\Core\Entity\EntityTypeInterface;
use Drupal\Core\Entity\ContentEntityInterface;
use Drupal\Core\Entity\EntityStorageInterface;

/**
* Defines the offer entity.
*
* @ingroup offer
*
* @ContentEntityType(
*   id = "offer",
*   label = @Translation("Offer"),
*   base_table = "offer",
*   data_table = "offer_field_data",
*   revision_table = "offer_revision",
*   revision_data_table = "offer_field_revision",
*   entity_keys = {
*     "id" = "id",
*     "uuid" = "uuid",
*     "label" = "title",
*     "revision" = "vid",
*     "status" = "status",
*     "published" = "status",
*     "uid" = "uid",
*     "owner" = "uid",
*   },
*   revision_metadata_keys = {
*     "revision_user" = "revision_uid",
*     "revision_created" = "revision_timestamp",
*     "revision_log_message" = "revision_log"
*   },
* )
*/

class Offer extends EditorialContentEntityBase {

 public static function baseFieldDefinitions(EntityTypeInterface
$entity_type) {
   $fields = parent::baseFieldDefinitions($entity_type); // provides id
and uuid fields

   $fields['user_id'] = BaseFieldDefinition::create('entity_reference')
     ->setLabel(t('User'))
```

```
      ->setDescription(t('The user that created the offer.'))
      ->setSetting('target_type', 'user')
      ->setSetting('handler', 'default')
      ->setDisplayOptions('view', [
        'label' => 'hidden',
        'type' => 'author',
        'weight' => 0,
      ])
      ->setDisplayOptions('form', [
        'type' => 'entity_reference_autocomplete',
        'weight' => 5,
        'settings' => [
          'match_operator' => 'CONTAINS',
          'size' => '60',
          'autocomplete_type' => 'tags',
          'placeholder' => '',
        ],
      ])
      ->setDisplayConfigurable('form', TRUE)
      ->setDisplayConfigurable('view', TRUE);

    $fields['title'] = BaseFieldDefinition::create('string')
      ->setLabel(t('Title'))
      ->setDescription(t('The title of the offer'))
      ->setSettings([
        'max_length' => 150,
        'text_processing' => 0,
      ])
      ->setDefaultValue('')
      ->setDisplayOptions('view', [
        'label' => 'above',
        'type' => 'string',
        'weight' => -4,
      ])
      ->setDisplayOptions('form', [
        'type' => 'string_textfield',
        'weight' => -4,
      ])
      ->setDisplayConfigurable('form', TRUE)
      ->setDisplayConfigurable('view', TRUE);

    $fields['message'] = BaseFieldDefinition::create('string_long')
      ->setLabel(t('Message'))
      ->setRequired(TRUE)
      ->setDisplayOptions('form', [
        'type' => 'string_textarea',
        'weight' => 4,
        'settings' => [
          'rows' => 12,
        ],
      ])
      ->setDisplayConfigurable('form', TRUE)
      ->setDisplayOptions('view', [
        'type' => 'string',
        'weight' => 0,
        'label' => 'above',
      ])
      ->setDisplayConfigurable('view', TRUE);

    $fields['status'] = BaseFieldDefinition::create('boolean')
      ->setLabel(t('Publishing status'))
      ->setDescription(t('A boolean indicating whether the Offer entity
is published.'))
      ->setDefaultValue(TRUE);

    $fields['created'] = BaseFieldDefinition::create('created')
      ->setLabel(t('Created'))
      ->setDescription(t('The time that the entity was created.'));

    $fields['changed'] = BaseFieldDefinition::create('changed')
      ->setLabel(t('Changed'))
      ->setDescription(t('The time that the entity was last edited.'));

    return $fields;
  }
}
```

After clearing cache our entity is created and two extra database tables were added:


offer
Columns
id INT(10) UNSIGNED
vid INT(10) UNSIGNED
uuid VARCHAR(128)
status TINYINT
user_id INT(10) UNSIGNED
title VARCHAR(150)
image INT(10) UNSIGNED
message LONGTEXT
created INT
changed INT
Indexes
Triggers
offer_revision
Columns
id INT(10) UNSIGNED
vid INT(10) UNSIGNED
revision_timestamp INT
revision_uid INT(10) UNSIGNED
revision_log LONGTEXT
image INT(10) UNSIGNED
revision_default TINYINT
Indexes

Now let's proceed with the CRUD operations. For every offer, we'd like to have an add, edit and delete form. But first, we secure the access.

Securing access of our entities

👉 In this important section you will be taught how **entity access** works. At the end of the section you will be able to create custom permissions for users and translate it towards your entity. In the use case of this project it means that authors can only see/edit/delete their own created entities.

There is something worth noticing about our entities. While the author of the entity will be the owner (thanks to the *PreCreate()* function in our Offer entity) he has no exclusive access towards viewing, or even editing and deleting the entity.

While drupal will typically provide separate “view”, “create”, “edit” and “delete” options we will (for now) make this 1 single permission: ***administer own offers***.

But we did not specify which access this means towards the entity itself. Let us make sure that everyone with this access can create offers and more importantly that they can only edit and delete their own offers and not those of others.

First, add a file **modules/custom/offer/offer.permissions.yml** with the following:

```
administer own offers:
 title: 'Create/edit/delete own offers'
```

Second, add the following methods to the **custom/offer/src/Offer** class at the bottom, these are two methods that are used quite a lot. The first one is to make sure the user id gets stored as the author of the entity, The other ones are typical methods to quickly get info about the author of an entity:

```
/**
  * {@inheritdoc}
  *
  * Makes the current user the owner of the entity
  */
public static function preCreate(EntityStorageInterface
$storage_controller, array &$values) {
   parent::preCreate($storage_controller, $values);
   $values += array(
     'user_id' => \Drupal::currentUser()->id(),
   );
}

/**
* {@inheritdoc}
*/
```

```
public function getOwner() {
 return $this->get('user_id')->entity;
}

/**
* {@inheritdoc}
*/
public function getOwnerId() {
 return $this->get('user_id')->target_id;
}
```

In the end we want full CRUD access for our authenticated users. When accessing an entity in drupal, there are 4 operations that can be requested:

- view
- update
- edit
- delete

Add the following to the annotations of your entity inside the **modules/custom/offer/src/Entity/Offer.php** file:

```
*   handlers = {
*     "access" = "Drupal\offer\OfferAccessControlHandler",
*   }
```

This file will handle access towards our entity. Add a file called **OfferAccessControlHandler.php** inside **modules/custom/offer/src**:

```
<?php

namespace Drupal\offer;

use Drupal\Core\Access\AccessResult;
use Drupal\Core\Entity\EntityAccessControlHandler;
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Session\AccountInterface;

/**
* Access controller for the offer entity. Controls create/edit/delete
access for entity and fields.
*
* @see \Drupal\offer\Entity\Offer.
```

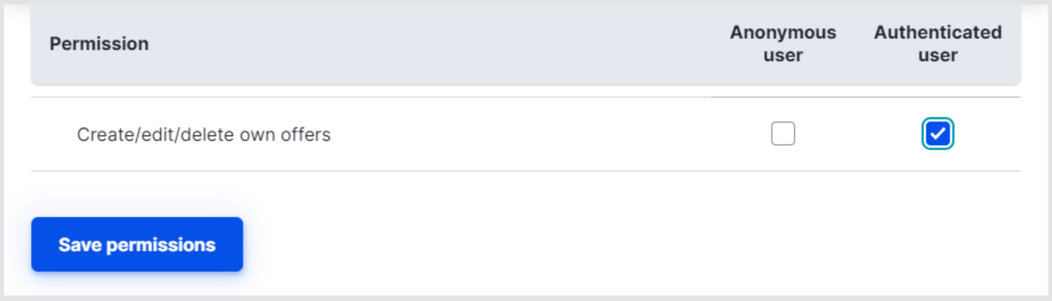
```
*/
class OfferAccessControlHandler extends EntityAccessControlHandler {

 /**
  * {@inheritdoc}
  *
  * Link the activities to the permissions. checkAccess is called with
the
  * $operation as defined in the routing.yml file.
  */
 protected function checkAccess(EntityInterface $entity, $operation,
AccountInterface $account) {

   $access = AccessResult::forbidden();

   switch ($operation) {
     case 'view':
       if ($account->hasPermission('administer own offers')) {
         $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
       }
       break;
     case 'update': // Shows the edit buttons in operations
       if ($account->hasPermission('administer own offers')) {
         $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
       }
       break;
     case 'edit': // Lets me in on the edit-page of the entity
       if ($account->hasPermission('administer own offers')) {
         $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
       }
       break;
     case 'delete': // Shows the delete buttons + access to delete this
entity
       if ($account->hasPermission('administer own offers')) {
         $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
       }
       break;
   }

   return $access;
 }
```

```
  /**
   * {@inheritdoc}
   *
   * Separate from the checkAccess because the entity does not yet exist,
 it
   * will be created during the 'add' process.
   */
  protected function checkCreateAccess(AccountInterface $account, array
 $context, $entity_bundle = NULL) {
    return AccessResult::allowedIfHasPermission($account, 'administer own
 offers');
  }

 }

 ?>
```

This access controller gives us a variety of power towards our entity. We now have full control in code on who can access different modes of our entity.

If you take a closer look, it is here that we integrate our permission (administer own offers) with our view/edit/update/delete access. As an extra we add a check to make sure there is only access to own entities.

💻 Add the newly created “administer own offers” permission to all authenticated users via admin/people/permissions.



Permission	Anonymous user	Authenticated user
Create/edit/delete own offers	☐	✓

In a later stage of the software, we can create different user roles for which entire access to the CRUD section can be granted with one click.

With our entity access completely nailed, we are about to use these access checks in our routing and crud forms. Let's move on to the next chapter!

Adding the create/edit/delete forms (CRUD)

👉 This section highlights the entity CRUD (create, read, edit, delete) process. At the end you will have learnt the ins and outs of how to show a user an **entity form** for **creation, editing and deleting** of a **custom entity**.

For our CRUD operations, the system needs to know where to put them. Let's update our **modules/custom/offer/src/Entity/offer.php** once again by adding the forms and links:

```
*   handlers = {
*     "access" = "Drupal\offer\OfferAccessControlHandler",
*     "form" = {
*       "add" = "Drupal\offer\Form\OfferForm",
*       "edit" = "Drupal\offer\Form\OfferForm",
*       "delete" = "Drupal\offer\Form\OfferDeleteForm",
*     },
*   },
*   links = {
*     "canonical" = "/offers/{offer}",
*     "delete-form" = "/offer/{offer}/delete",
*     "edit-form" = "/offer/{offer}/edit",
*     "create" = "/offer/create",
*   },
```

You may understand what we're adding here. The links are the routes we want to use for CRUD operations on our offer entity. Inside the handlers directory we placed the add/edit and delete form statements.

These routes to not exist yet, so we define them in **modules/custom/offer/offer.routing.yml**

```
offer.add:
  path: '/offers/create'
  defaults:
    _entity_form: offer.add
    _title: 'Add offer'
  requirements:
    _entity_create_access: 'offer'

entity.offer.edit_form:
  path: '/offers/{offer}/edit'
  defaults:
    _entity_form: offer.edit
    _title: 'Edit offer'
  requirements:
    _entity_access: 'offer.edit'

entity.offer.delete_form:
  path: '/offers/{offer}/delete'
  defaults:
    _entity_form: offer.delete
    _title: 'Delete offer'
  requirements:
    _entity_access: 'offer.delete'

entity.offer.canonical:
  path: '/offer/{offer}'
  defaults:
    _entity_view: 'offer'
    _title: 'Offer'
  requirements:
    _entity_access: 'offer.view'
```

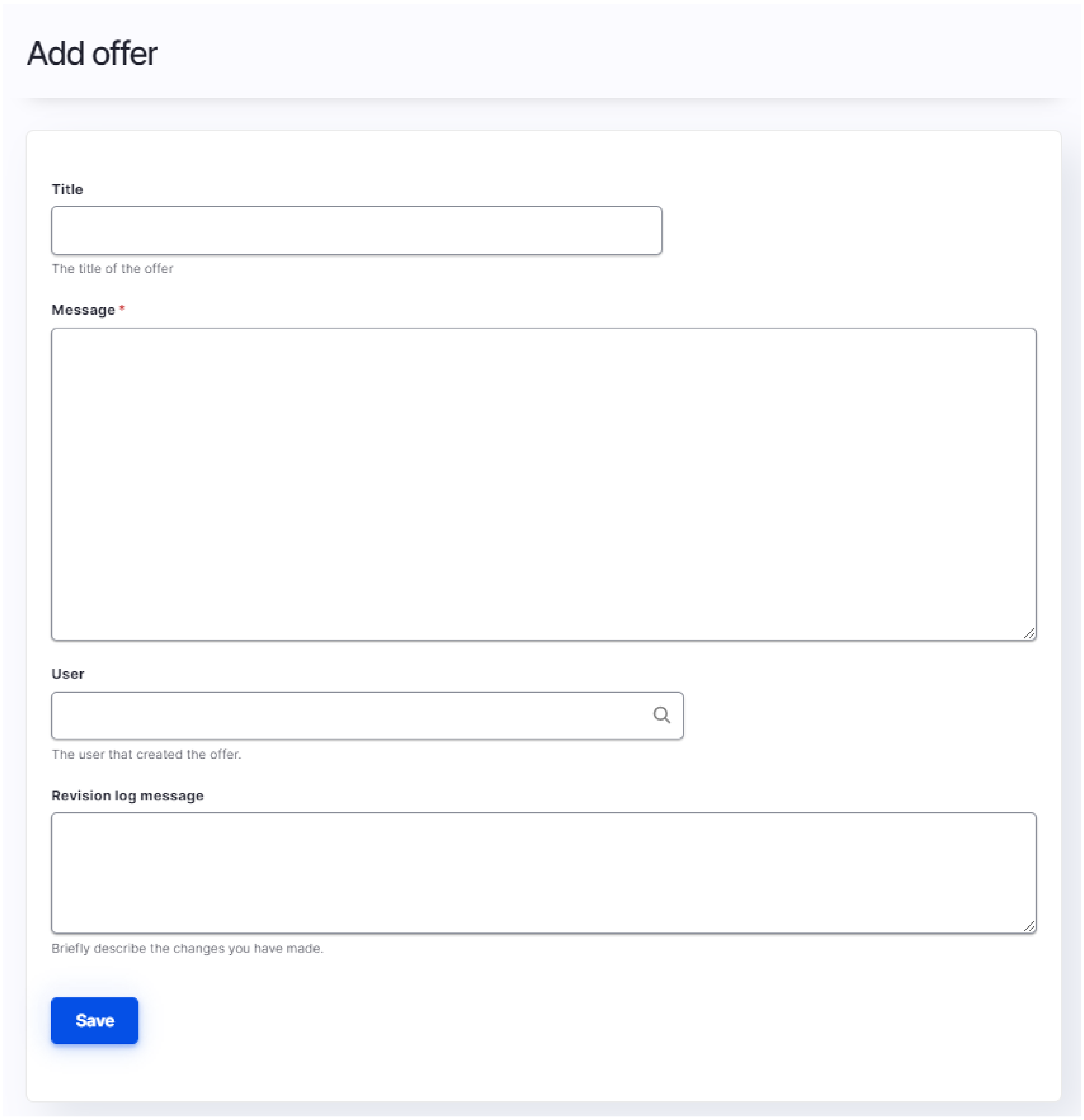
Check the *_entity_access* parameters. These will check the access conditions inside **OfferAccessControlHandler** (checkAccess and checkCreateAccess). By this, we control exactly who can access the /offers/create form for example.

Now we'll add a generic form for adding and editing. It inherits from *ContentEntityForm:* **custom/offer/src/Form/OfferForm.php**:

```
<?php
/**
* @file
* Contains Drupal\offer\Form\OfferForm.
*/

namespace Drupal\offer\Form;

use Drupal\Core\Entity\ContentEntityForm;
use Drupal\Core\Form\FormStateInterface;
```

```
/**
* Form controller for the offer entity edit forms.
*
* @ingroup content_entity_example
*/
class OfferForm extends ContentEntityForm {

 /**
  * {@inheritdoc}
  */
 public function buildForm(array $form, FormStateInterface $form_state)
{
   /* @var $entity \Drupal\offer\Entity\Offer */
   $form = parent::buildForm($form, $form_state);
   return $form;
 }

 /**
  * {@inheritdoc}
  */
 public function save(array $form, FormStateInterface $form_state) {
   // Redirect to offer list after save.
   $form_state->setRedirect('entity.offer.collection');
   $entity = $this->getEntity();
   $entity->save();
 }

}
```

Clear cache and head to /offers/create. There it is! We've gotten ourselves access to the create and edit mode of this entity.


Add offer

Title

The title of the offer

Message *

User

The user that created the offer.

Revision log message

Briefly describe the changes you have made.

💻 If you get an “Access denied”, make sure authenticated users have permission to ‘administer own offers’ via *admin/people/permissions*.

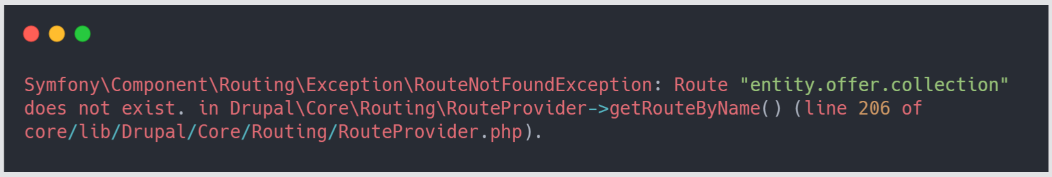
Saving an entity will not work at the moment. We get this error:

```
The website encountered an unexpected error. Please try again later.
```

This is not a readable error when developing. Add this to your **settings.php**:

```
$config['system.logging']['error_level'] = 'verbose';
```

Refresh the page to see the error:



```
Symfony\Component\Routing\Exception\RouteNotFoundException: Route "entity.offer.collection"
does not exist. in Drupal\Core\Routing\RouteProvider->getRouteByName() (line 206 of
core/lib/Drupal/Core/Routing/RouteProvider.php).
```

This tells us the page we redirect to after saving does not exist yet, We will fix this in the views integration chapter.

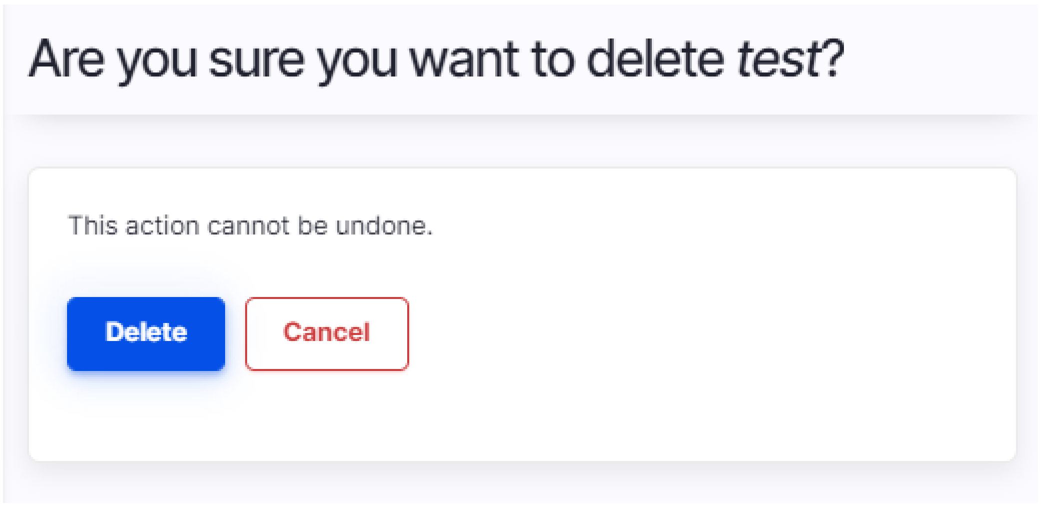
At /offers/1/edit you should see your created entity in an edit form with your submitted values filled in.

Apart from the overview page, a missing step towards full CRUD operations is the delete form. We've already added the definition in our entity and routing, so now add **custom/offer/src/Form/OfferDeleteForm.php**:

```
<?php

/**
* @file
* Contains \Drupal\offer\Form\OfferDeleteForm.
*/

namespace Drupal\offer\Form;

use Drupal\Core\Entity\ContentEntityConfirmFormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Url;

/**
* Provides a form for deleting a content_entity_example entity.
*
* @ingroup offer
*/
class OfferDeleteForm extends ContentEntityConfirmFormBase {

 /**
  * {@inheritdoc}
```

```
   */
  public function getQuestion() {
    return $this->t('Are you sure you want to delete %name?',
array('%name' => $this->entity->label()));
  }

   /**
    * {@inheritdoc}
    *
    * If the delete command is canceled, return to the offer.
    */
   public function getCancelUrl() {
     return Url::fromRoute('entity.offer.edit_form', ['offer' =>
$this->entity->id()]);
   }


  /**
   * {@inheritdoc}
   */
  public function getConfirmText() {
    return $this->t('Delete');
  }

  /**
   * {@inheritdoc}
   *
   * Delete the entity
   */
  public function submitForm(array &$form, FormStateInterface
$form_state) {
    $entity = $this->getEntity();
    $entity->delete();

    $this->logger('offer')->notice('deleted %title.',
      array(
        '%title' => $this->entity->label(),
      ));
    // Redirect to offer list after delete.
    $form_state->setRedirect('entity.offer.collection');
  }

}
```

We are now able to delete an offer with a confirmation form before deleting. If you tried creating an entity via the create form, you will now be able to visit offers/1/delete.


Are you sure you want to delete *test*?

This action cannot be undone.

We get an error after deletion because again, we get redirected towards the collection of this entity:

```
$form_state->setRedirect('entity.offer.collection');
```

The error happens because the entity has not defined a collection ("listing") class yet. Typically, drupal offers an entity listing functionality for entities that get defined in a list builder. Personally I think there is a better option because the ux and flexibility are not that great. Also, there is a much more powerful option for this. We will use the core Views module to offer a listing of our entities in the next chapter to fix the error and finalize our CRUD operations.

Views integration of our custom entity to add a listing

👉 In this section we go over the possibility of adding **views support** for **custom entities**. At the end you will be able to create powerful listings with views in the same way you can do with nodes.

```
bash-5.0$ drush en views views_ui
 [success] Successfully enabled: views, views_ui
```

💻 Enable the **views** and **views_ui** module.

```
bash$ drush en views views_ui
[success] Successfully enabled: views, views_ui
```

First, to ensure that our entities have integration with views we have to add an annotation with a link to the class.

In our **custom/offer/src/Entity/Offer.php** we add the views integration in the annotations like this:

```
*   handlers = {
*     "access" = "Drupal\offer\OfferAccessControlHandler",
*     "views_data" = "Drupal\offer\OfferViewsData",
*     "form" = {
*       "add" = "Drupal\offer\Form\OfferForm",
*       "edit" = "Drupal\offer\Form\OfferForm",
*       "delete" = "Drupal\offer\Form\OfferDeleteForm",
*     },
*   },
```

To the **modules/custom/offer/src/OfferViewsData.php**, add this code:

```
<?php

namespace Drupal\offer;

use Drupal\views\EntityViewsData;

/**
* Provides views data for Offer entities.
*
*/
class OfferViewsData extends EntityViewsData {

 /**
  * Returns the Views data for the entity.
  */
 public function getViewsData() {
   $data = parent::getViewsData();
```

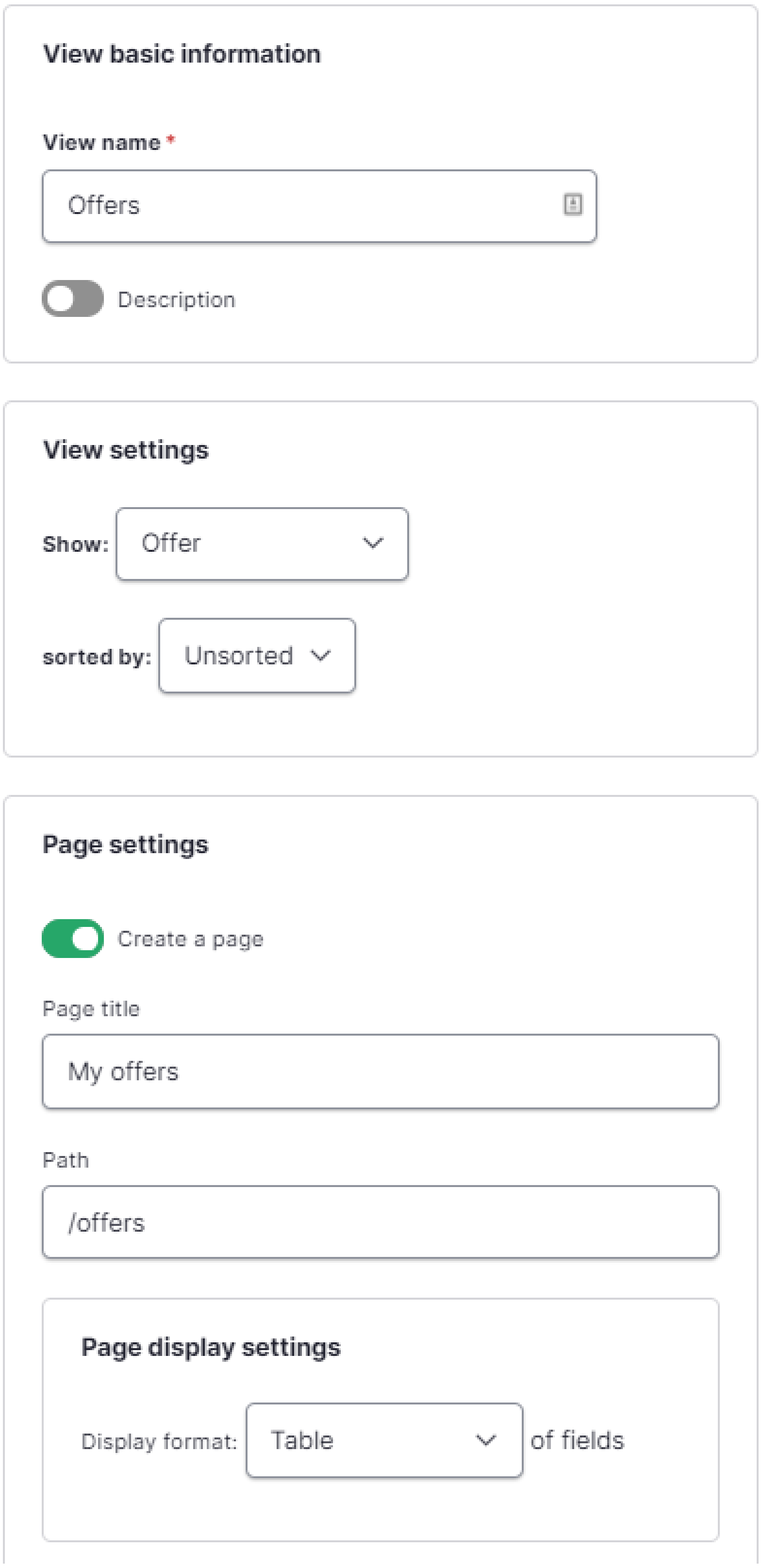
```
    return $data;
  }
}
```

Clear caches and head to *admin/structure/views/add*. When adding a new view, we can now select “offer” as an entity to make a listing. This is pretty powerful, we get full access to all of views finest functionality:

- Quickly make (advanced) listings
- Search
- Filtering
- Pagination
- Bulk operations
- …

But for now, we make an easy listing of our entities.

We add title, edit and a delete button as fields.


View basic information

View name *

Offers

Description

View settings

Show: Offer

sorted by: Unsorted

Page settings

Create a page

Page title

My offers

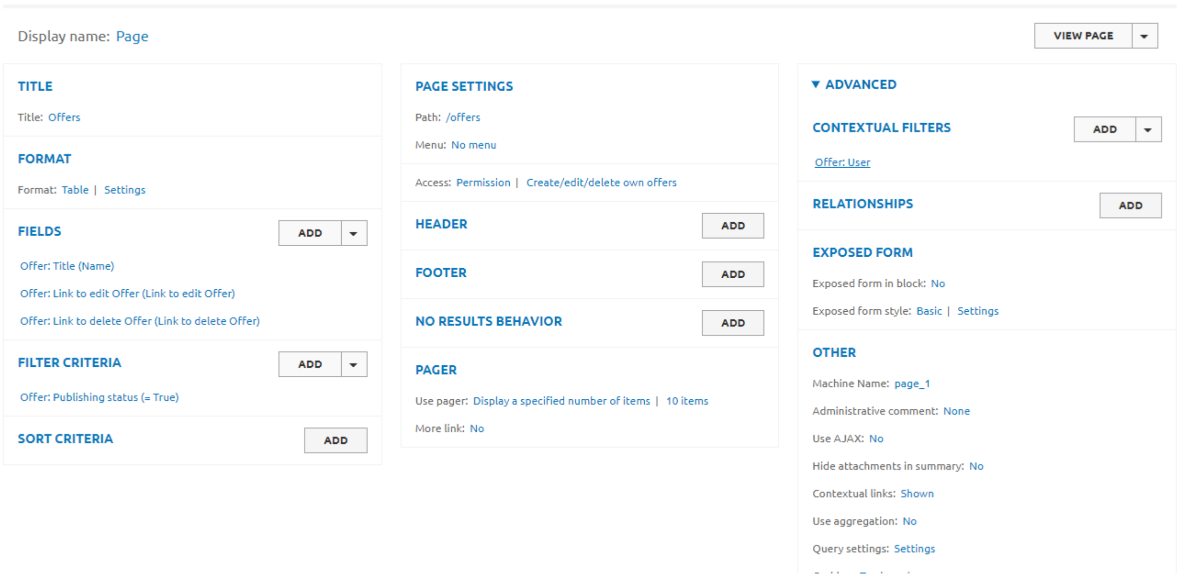
Path

/offers

Page display settings

Display format: Table of fields

We add title, edit and a delete button as fields.


Display name: Page

VIEW PAGE

TITLE
Title: Offers

FORMAT
Format: Table | Settings

FIELDS ADD
Offer: Title (Name)
Offer: Link to edit Offer (Link to edit Offer)
Offer: Link to delete Offer (Link to delete Offer)

FILTER CRITERIA ADD
Offer: Publishing status (= True)

SORT CRITERIA ADD

PAGE SETTINGS
Path: /offers
Menu: No menu
Access: Permission | Create/edit/delete own offers

HEADER ADD

FOOTER ADD

NO RESULTS BEHAVIOR ADD

PAGER
Use pager: Display a specified number of items | 10 items
More link: No

▼ ADVANCED

CONTEXTUAL FILTERS ADD
Offer: User

RELATIONSHIPS ADD

EXPOSED FORM
Exposed form in block: No
Exposed form style: Basic | Settings

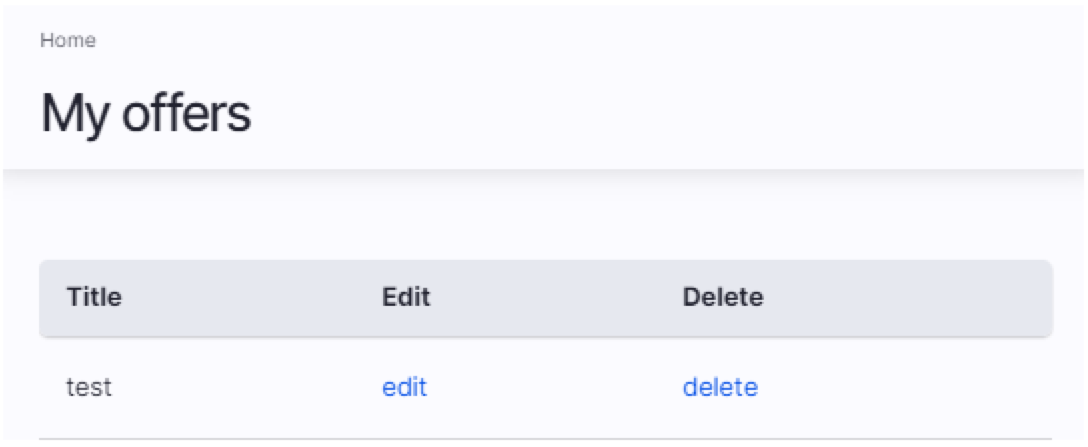
OTHER
Machine Name: page_1
Administrative comment: None
Use AJAX: No
Hide attachments in summary: No
Contextual links: Shown
Use aggregation: No
Query settings: Settings

Important

Views is an extremely powerful tool. But think about it as a querying interface and a tool for building easy and advanced entity listings. If your entities are secured like we did with the offers, this does not mean they can not appear in views listings for other users.

For that, always make sure you only show entities that have the logged in user as author (*Advanced > Contextual filter > User > Provide default value > User id from logged in user*). Of course, if you make the users author of their own entities like we are doing.

Also, check for permissions on the listing. In this case, we gave permission to users that have “administer own offers“ (Create/edit/delete own offers) permission.


Home

My offers

Title	Edit	Delete
test	edit	delete

We’ve got our listing! We will tweak it further in a later chapter, but this is the functionality we want at this moment.

💻 Time to get our CRUD operations complete by adding our entity listing to as the collection of this entity. To **offer/offer.routing.yml**, add:

```
entity.offer.collection:
  path: '/offers'
  requirements:
    _permission: 'administer own offers'
```

Test by adding, editing and deleting an offer. But rebuild caches first!

Export your configuration now

```
drush cex -y
```

and look for **views.view.offers.yml**. Copy the file to **modules/custom/offer/config/install**.
This will keep our view inside our offers module and on next installation, the view will be installed.

To our **offer.info.yml** file, we add:

```
dependencies:
 - drupal:views
```

We'll disable the module and reïnstall to see if it all works. Delete our entities first like this:

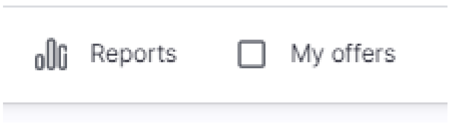
```
bash$ drush entity:delete offer
[success] Deleted offer entity Ids: 1, 2
bash$ drush pmu offer
[success] Successfully uninstalled: offer
bash$ drush en offer
[success] Successfully enabled: offer
```

We have now full CRUD operations up and working in a generic and reusable module.

To make our lives a bit more comfortable, we add a menu link to the toolbar for direct access. To **custom/modules/offer/offer.links.menu.yml** add:

```
offer.toolbar.my_offers:
 title: 'My offers'
 route_name: entity.offer.collection
 parent: system.admin
 weight: 99
```

This shows a link to the “My offers” overview page in the toolbar.

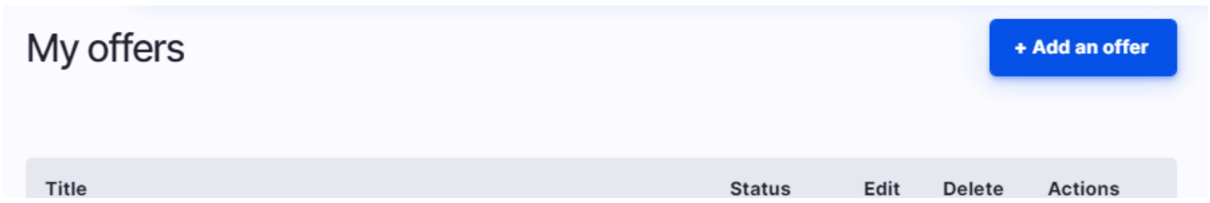


💻 Give authenticated users access to the toolbar first (via permissions), but note that this is only for the development phase of the platform.

Next, this listing needs a big “Add an offer” button. Drupal has a system for this. Add **offer.links.action.yml** and add:

```
offer.add_offer:
  route_name: offer.add
  title: 'Add an offer'
  appears_on:
    - entity.offer.collection
```

Clear caches and we now have our action button:



Getting up-to-speed: making the entity fieldable

👉 This section is about making your custom entity fieldable. At the end you will be able to connect you **custom entity** with the power of the **drupal user-interface for adding fields, form modes and view modes**. At the end you will be able to add fields to your entity via the UI and manage the display. The created fields will be exportable as configuration.

As told earlier, the entity data structure in drupal made the software famous. We can extend our custom entity with this dynamic and extremely feature-rich UI.

Advantages of working this way are:

- **Definition and naming can be done in the UI**. You get access to powerful fields with features as autocomplete entity reference, multifield, private files, ...
- Create **view modes** of the entity via the UI (a teaser and full entity view for example) and set the display of each field
- Manage **form modes** (configure the appearance of the add and edit form via the UI)

With that UI, we can add the desired fields to our entity by just clicking:

- Images
- A PDF upload
- Description
- Short text
- Tags
- … There are hundreds of modules available that define all sorts of fields!

Note that we already have a description defined in our **Offer.php** entity structure. We're about to delete it. We'll delete our entities first and uninstall again:

```
bash$ drush entity:delete offer
[success] Deleted offer entity Ids: 1, 2
bash$ drush pmu offer
[success] Successfully uninstalled: offer
```

Now **remove** the *$fields['message']* from the *BaseFieldDefinitions* in your **Offer.php** entity file.

💻 make sure the core **field_ui** module is enabled.

```
bash$ drush en field_ui
[success] Successfully enabled: field_ui
```

To the annotations in **modules/custom/offer/src/Entity/Offer.php** add a fiel_ui_base_route key:

```
*     "create" = "/offer/c...
*   },
*   field_ui_base_route = "entity.offer.settings",
*   revision_me....
```

The route **entity.offer.settings** will be the landing page of the ui settings of our entity. If we configure this route to /admin/structure/offer, the following routes will become available automatically:

- admin/structure/offer/fields (fields ui)
- admin/structure/offer/form-display (add form display)
- admin/structure/offer/display (view modes display management)

The settings landing page is usually a settings form. While we will not use additional settings in this form, let’s keep the good practice and add this route as a form.

To **modules/custom/offer/routing.yml** we add:

```
entity.offer.settings:
 path: 'admin/structure/offer'
 defaults:
   _form: '\Drupal\offer\Form\OfferSettingsForm'
   _title: 'Offer settings'
 requirements:
   _role: 'administer own offers'
```

To **modules/custom/offer/src/Form/OfferSettingsForm**:

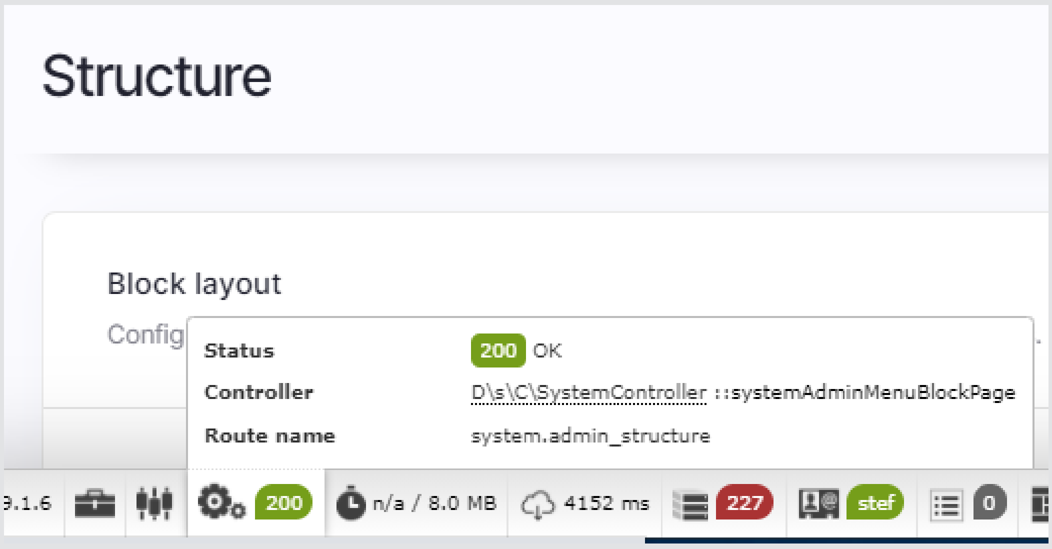
```
<?php
/**
* @file
* Contains \Drupal\offer\Form\OfferSettingsForm.
*/

namespace Drupal\offer\Form;

use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
```

```
/**
* Class OfferSettingsForm.
*
* @package Drupal\offer\Form
*
* @ingroup offer
*/
class OfferSettingsForm extends FormBase {
 /**
  * Returns a unique string identifying the form.
  *
  * @return string
  *   The unique string identifying the form.
  */
 public function getFormId() {
   return 'offer_settings';
 }

 /**
  * {@inheritdoc}
  */
 public function submitForm(array &$form, FormStateInterface
$form_state) {
   // Empty implementation of the abstract submit class.
 }

 /**
  * {@inheritdoc}
  */
 public function buildForm(array $form, FormStateInterface $form_state)
{
   $form['offer_settings']['#markup'] = 'Settings form for offer. We
don\'t need additional entity settings. Manage field settings with the
tabs above.';
   return $form;
 }

}
```

We want to do this clean, with adding a link on the admin/structure page to our offer settings form. With the webprofiler toolbar (see developer tools) we check the route name for admin/structure.

💻 Enable the **weprofiler** module, which is part of the **devel** module and re-nable the **offer** module.

```
bash$ drush en webprofiler offer
[success] Successfully enabled: webprofiler, offer
```



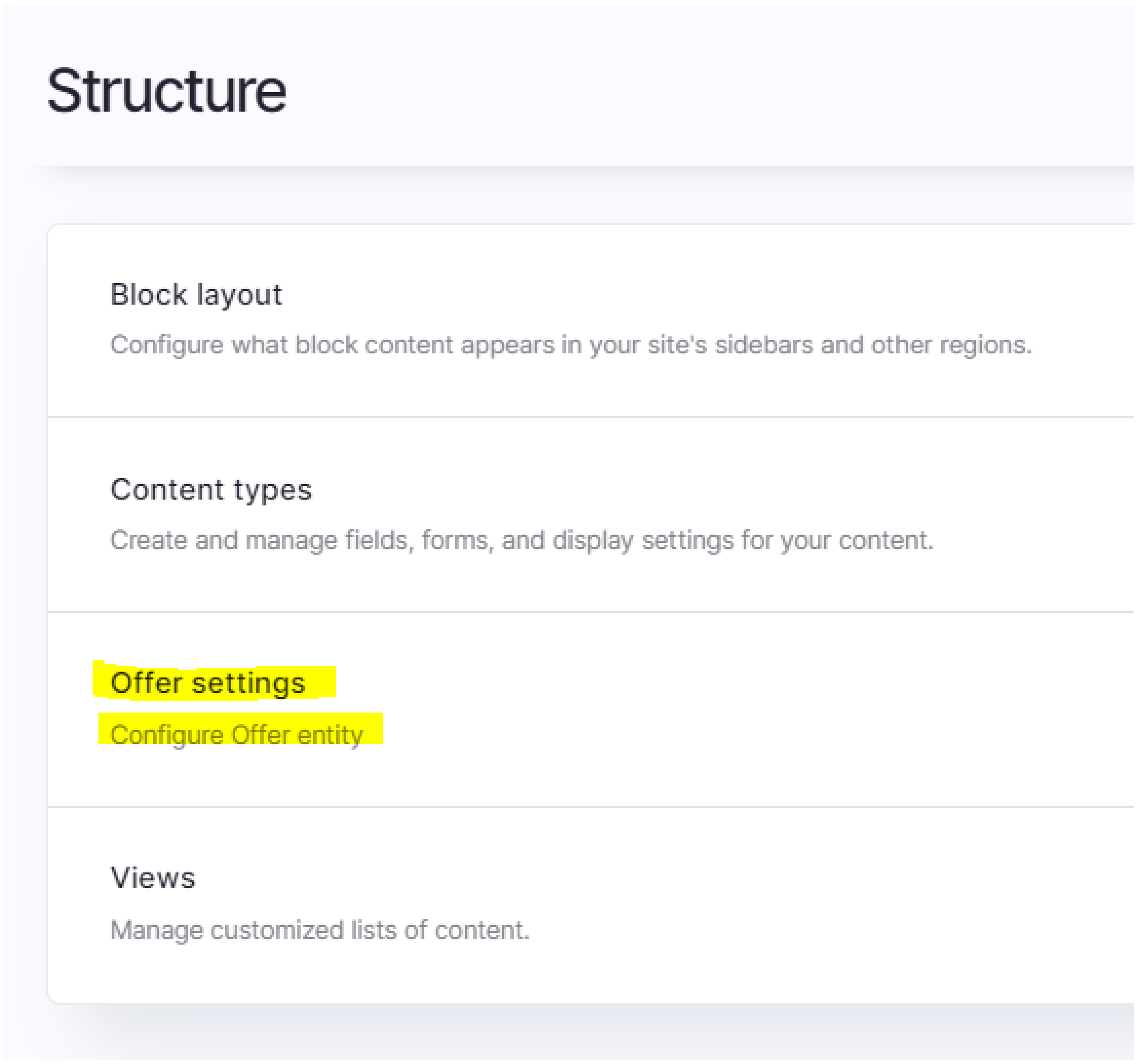
```
Status        200 OK
Controller    D\s\C\SystemController ::systemAdminMenuBlockPage
Route name    system.admin_structure
```

We discovered the route name is *system.admin.structure*. We'll use it next as a parent for our settings link.

Next, we add a file named **custom/offer/offer.links.menu.yml**:

```
offer.admin.structure.settings:
 title: Offer settings
 description: 'Configure Offer entity'
 route_name:  entity.offer.settings
 parent: system.admin_structure
```

💻 Always clear the caches after adding/editing new routes, menu links or other YAML config!


Structure

Block layout
Configure what block content appears in your site's sidebars and other regions.

Content types
Create and manage fields, forms, and display settings for your content.

Offer settings
Configure Offer entity

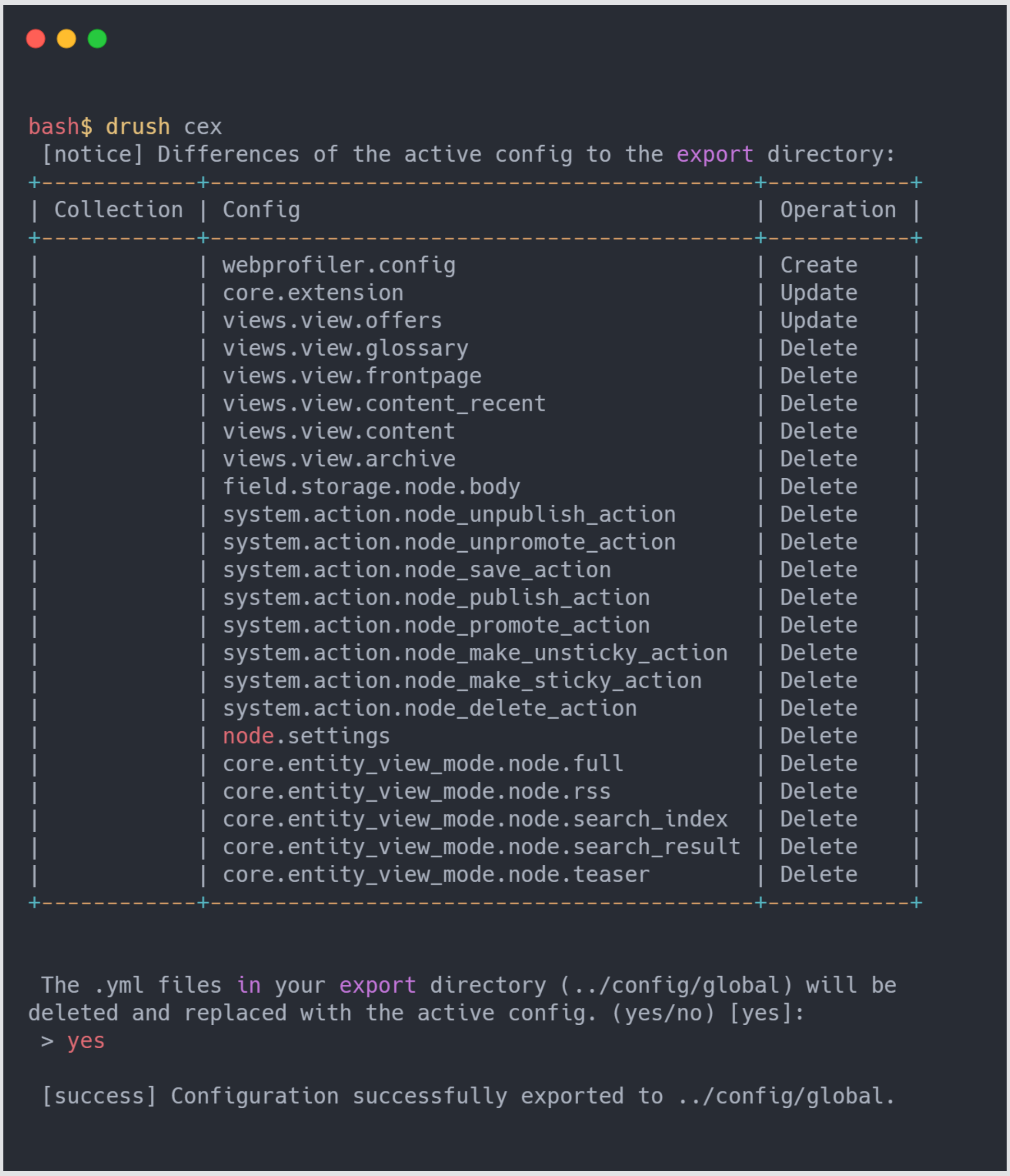
Views
Manage customized lists of content.

The link appeared in our structure tree, and the build-up is the same way as "content types", which we know from the Node entity.

💻 If you still see "Content types", the time is here to uninstall the Node entity type:

```
bash$ drush pmu node
[success] Successfully uninstalled: node
```

If you export config, this will delete some default settings from the node entity type as well.

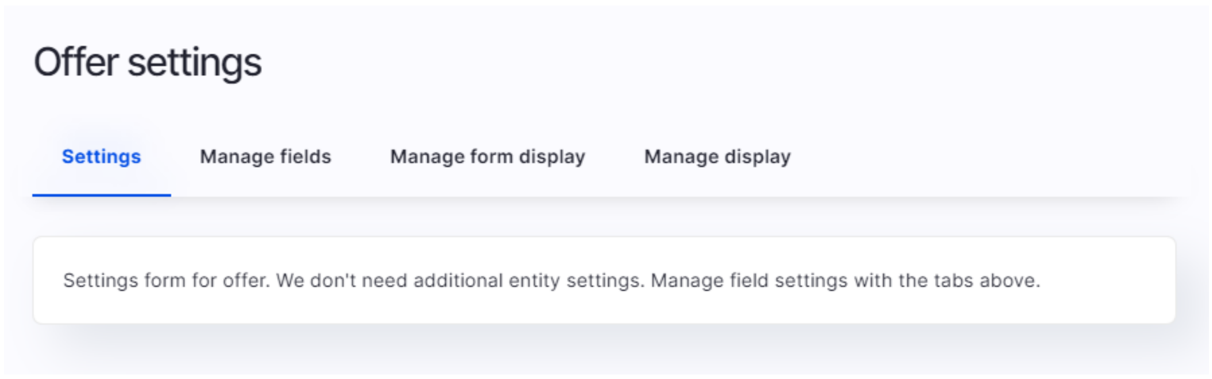


```
bash$ drush cex
 [notice] Differences of the active config to the export directory:
+------------+-------------------------------------------+-----------+
| Collection | Config                                    | Operation |
+------------+-------------------------------------------+-----------+
|            | webprofiler.config                        | Create    |
|            | core.extension                            | Update    |
|            | views.view.offers                         | Update    |
|            | views.view.glossary                       | Delete    |
|            | views.view.frontpage                      | Delete    |
|            | views.view.content_recent                 | Delete    |
|            | views.view.content                        | Delete    |
|            | views.view.archive                        | Delete    |
|            | field.storage.node.body                   | Delete    |
|            | system.action.node_unpublish_action       | Delete    |
|            | system.action.node_unpromote_action       | Delete    |
|            | system.action.node_save_action            | Delete    |
|            | system.action.node_publish_action         | Delete    |
|            | system.action.node_promote_action         | Delete    |
|            | system.action.node_make_unsticky_action   | Delete    |
|            | system.action.node_make_sticky_action     | Delete    |
|            | system.action.node_delete_action          | Delete    |
|            | node.settings                             | Delete    |
|            | core.entity_view_mode.node.full           | Delete    |
|            | core.entity_view_mode.node.rss            | Delete    |
|            | core.entity_view_mode.node.search_index   | Delete    |
|            | core.entity_view_mode.node.search_result  | Delete    |
|            | core.entity_view_mode.node.teaser         | Delete    |
+------------+-------------------------------------------+-----------+


 The .yml files in your export directory (../config/global) will be
deleted and replaced with the active config. (yes/no) [yes]:
 > yes

 [success] Configuration successfully exported to ../config/global.
```

Finally, we add a “local tasks”-link to the settings page. This will activate the tasks tabs above to get the structure that is known when administering nodes. By this I mean the “Edit”, “Manage fields”, “Manage form display” and “Manage display” tabs. To **modules/custom/offer/offer.links.task.yml**, add:

```
# Activates the tabs on the entity admin pages
(/admin/structure/offer)
offer.settings_tab:
 route_name: entity.offer.settings
 title: 'Settings'
 base_route: entity.offer.settings
```

We re-enable the module and see the desired entity ui extensions on admin/structure/offer.



Settings form for offer. We don't need additional entity settings. Manage field settings with the tabs above.

Cool! Time to get up to speed by adding some fields.

Media field with library

👉 This section shows how to add a media field to your entity, using the media library, and to only show media items you have uploaded.

💻 Enable the core **media** module as well as the core **media library** module:

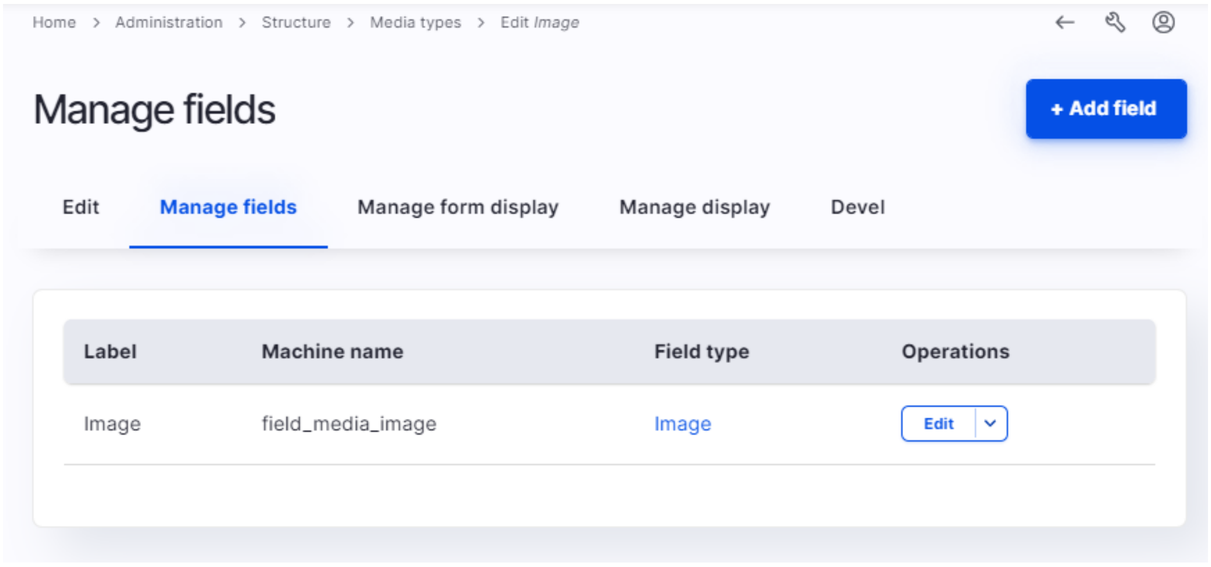
```
bash$ drush en media media_library
The following module(s) will be enabled: media, media_libraty,
image

 Do you want to continue? (yes/no) [yes]:
 > yes

 [success] Successfully enabled: media, media_library, image
```

A real advantage of drupal is that time-consuming development practices like file uploads and managing a file library are easily installed with some clicks.

💻 Define a media type at Structure > Media types > Add media Type and choose this:

Name: Image
Media source: Image

After saving the media type, you'll discover 'Image' is a bundle of entity media. It has fields which are preconfigured at admin/structure/media/manage/image/fields. In this case, the entity has an image field that is shipped with the media source: image.



Label	Machine name	Field type	Operations
Image	field_media_image	Image	Edit

We tweak the image field at /admin/structure/media/manage/image/fields/media.image.field_media_image with some settings, so resizing is automatically done when a file is too large.

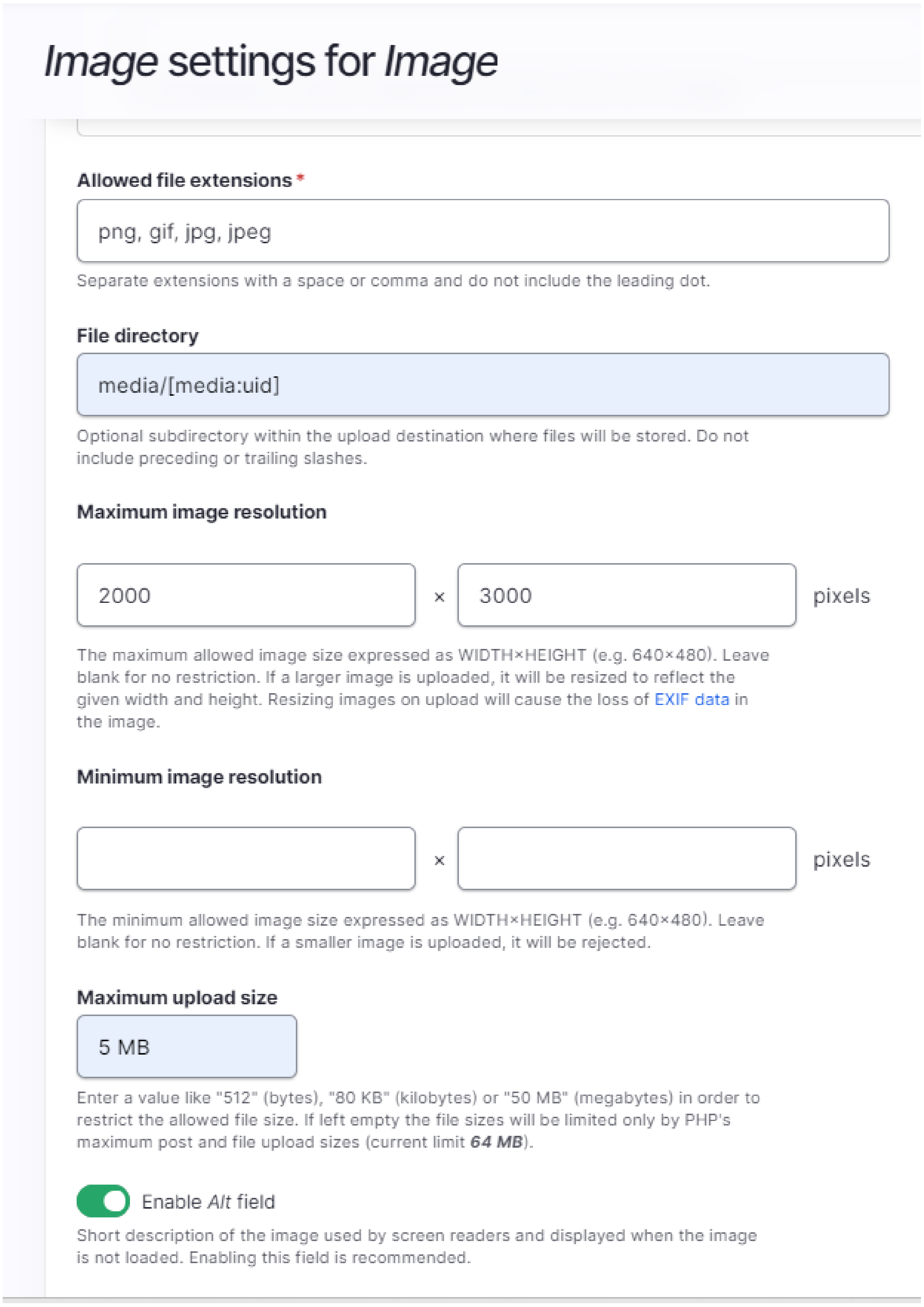


Image settings for *Image*

Allowed file extensions *

png, gif, jpg, jpeg

Separate extensions with a space or comma and do not include the leading dot.

File directory

media/[media:uid]

Optional subdirectory within the upload destination where files will be stored. Do not include preceding or trailing slashes.

Maximum image resolution

2000 × 3000 pixels

The maximum allowed image size expressed as WIDTH×HEIGHT (e.g. 640×480). Leave blank for no restriction. If a larger image is uploaded, it will be resized to reflect the given width and height. Resizing images on upload will cause the loss of EXIF data in the image.

Minimum image resolution

× pixels

The minimum allowed image size expressed as WIDTH×HEIGHT (e.g. 640×480). Leave blank for no restriction. If a smaller image is uploaded, it will be rejected.

Maximum upload size

5 MB

Enter a value like "512" (bytes), "80 KB" (kilobytes) or "50 MB" (megabytes) in order to restrict the allowed file size. If left empty the file sizes will be limited only by PHP's maximum post and file upload sizes (current limit ***64 MB***).

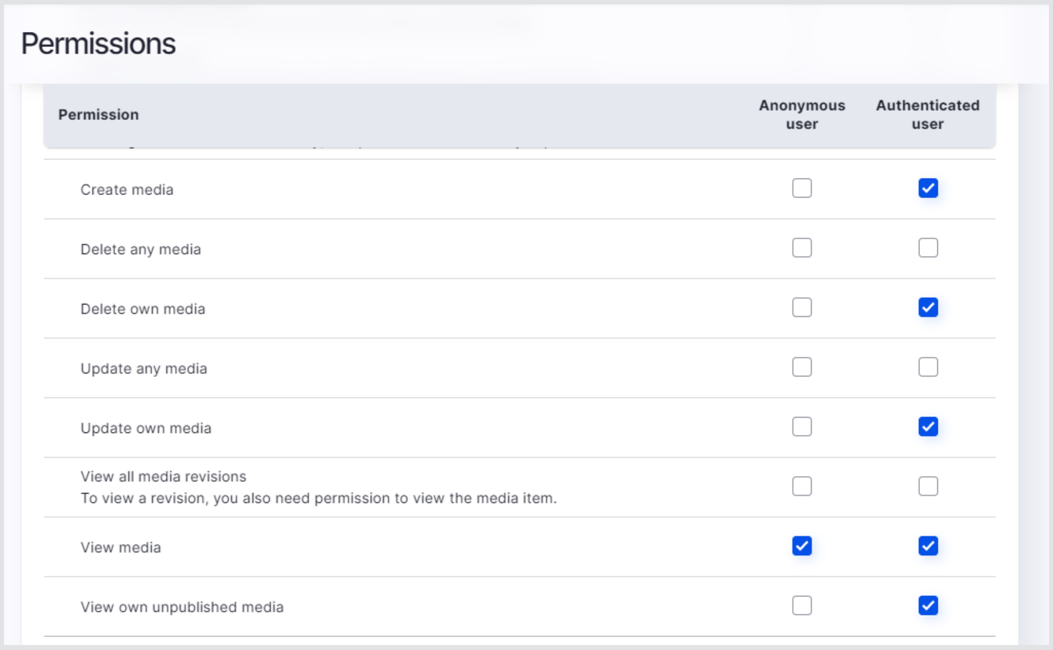
Enable *Alt* field [✓]

Short description of the image used by screen readers and displayed when the image is not loaded. Enabling this field is recommended.

💻 We'll add a media field for our offer at */admin/structure/offer/fields* > *add field*

Add a new field (Media)
Add a label (Image)
Help text (Market your offer with a tremendous image!)
Reference type (Default)
Media Type (Image)
(save and continue)
Allowed number of values (1)
Required field (yes)

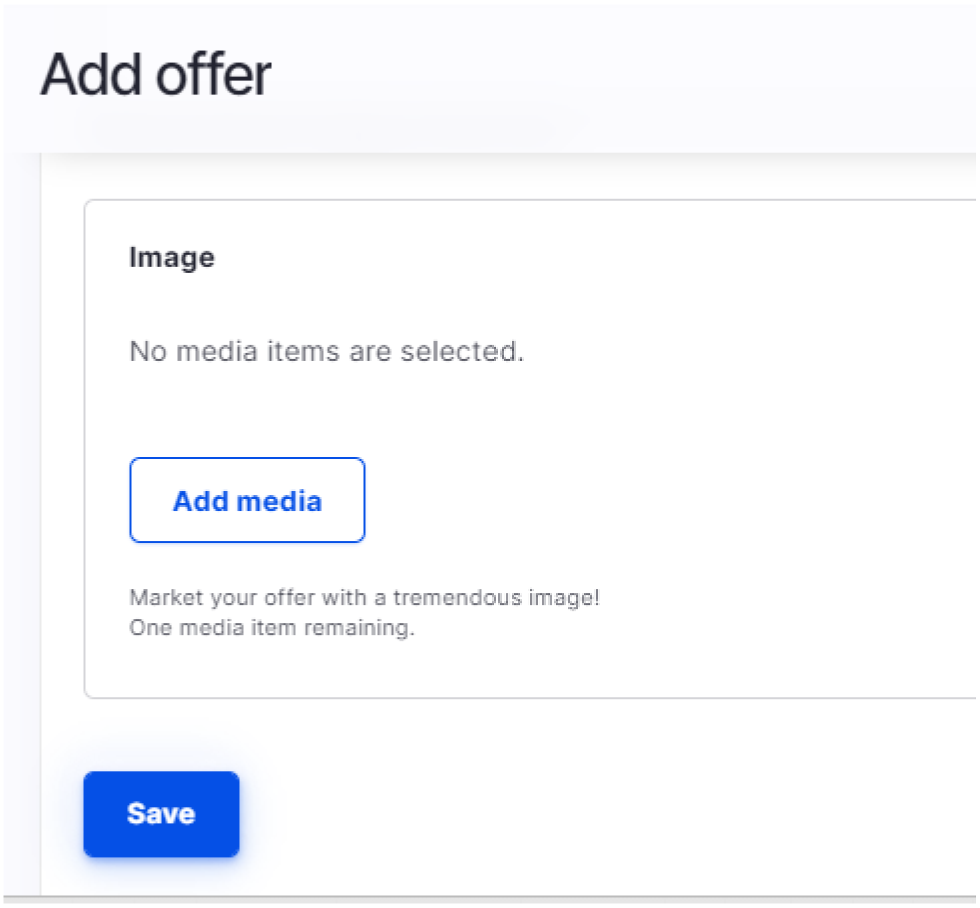
An additional step is giving our authenticated users the permissions to add these media items. Because the core media module works like our offer entity, it has its own permissions. Set them to make sure users can create and edit media entities:



Permissions

Permission	Anonymous user	Authenticated user
Create media	☐	✓
Delete any media	☐	☐
Delete own media	☐	✓
Update any media	☐	☐
Update own media	☐	✓
View all media revisions To view a revision, you also need permission to view the media item.	☐	☐
View media	✓	✓
View own unpublished media	☐	✓

Save and head back to our offer create form (*/offers/create*) we now have a widget with a media library that works for us out-of-the box!


Add offer
Image
No media items are selected.
Add media
Market your offer with a tremendous image!
One media item remaining.
Save

💻 You'll notice that when adding media items to your library, also media items of others are visible. This is because the media library is using a view to display the content. At /admin/structure/views/view/media_library **add the same contextual filter as we did with the offers view**: contextual filters > media:authored by > provide default value > User ID from logged in user.

Important! Do this for every display mode:
/admin/structure/views/view/media_library/edit/page
/admin/structure/views/view/media_library/edit/widget
/admin/structure/views/view/media_library/edit/widget_table

This will *only* make sure we don't see media items from other users inside our library. Media entities are not secured further in this course. If you would like more security, you could overwrite the Media module *AccesControlHandler()* by adding a *offer_entity_access()* hook to your **modules/custom/offer/offer.module**. You could also use private files for uploads.

But we'll leave this public, as this is not a feature that is needed.

Adding fields to the custom entity via the UI

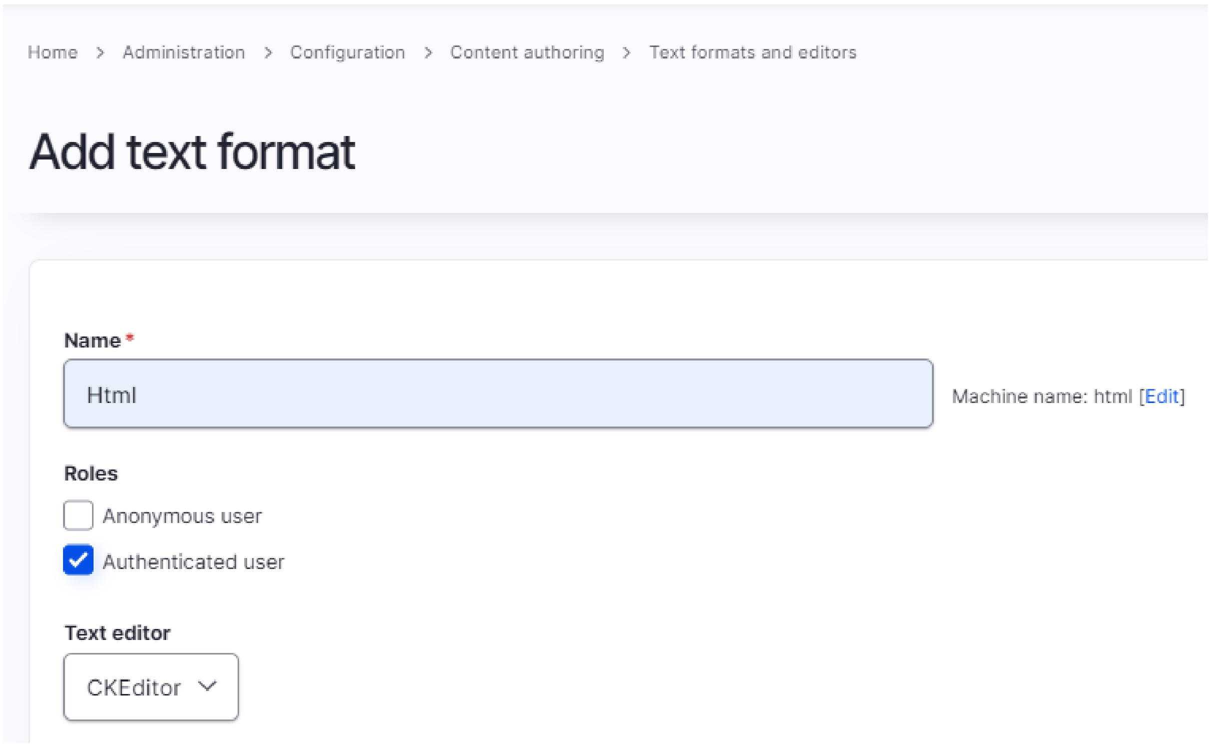
👉 In this section, the author shows how to add a textarea with a rich-text editor (CKEditor) so that users have a few options to optimize their content. Further, a price field and an options field.

Drupal ships with a core CKEditor module, which is the industry standard for rich text editing. We want to add a description field to our entity.

💻 Enable the **ckeditor** module:

```
bash$ drush en ckeditor -y
[success] Successfully enabled: ckeditor
```

For authenticated users to be able to use the editor, go to /admin/config/content/formats > add text format:
Name (Html)
Roles (Authenticated users)
Text editor (CKeditor)


Home > Administration > Configuration > Content authoring > Text formats and editors
Add text format

Name *

Html

Machine name: html [Edit]

Roles

☐ Anonymous user

☑ Authenticated user

Text editor

CKEditor

Now choose the rich text buttons you’d like to enable and save.

We’ll add a description field for our offer at /admin/structure/offer/fields > *add field*

Add a new field (Text, formatted, long)
Add a label (Description)
Description (Be as complete as possible about your offer)

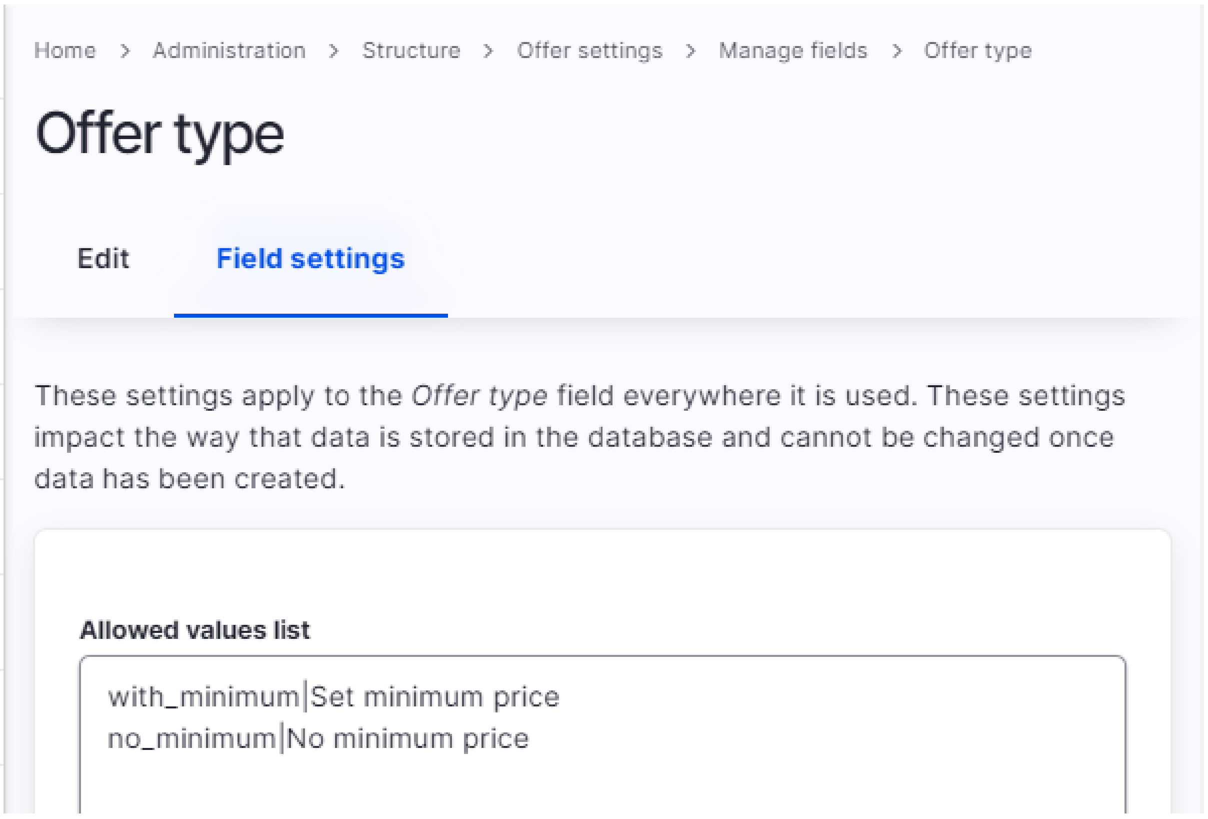
Next, we want the users to choose between a minimum price or without a minimum price.

💻 We enable the core options module for this:

```
bash$ drush en options
[success] Successfully enabled: options
```

Add a new field (Text, list)
Add a label (Offer type)
Allowed values list:

with_minimum|Set minimum price
no_minimum|No minimum price


Home > Administration > Structure > Offer settings > Manage fields > Offer type

Offer type

Edit | Field settings

These settings apply to the *Offer type* field everywhere it is used. These settings impact the way that data is stored in the database and cannot be changed once data has been created.

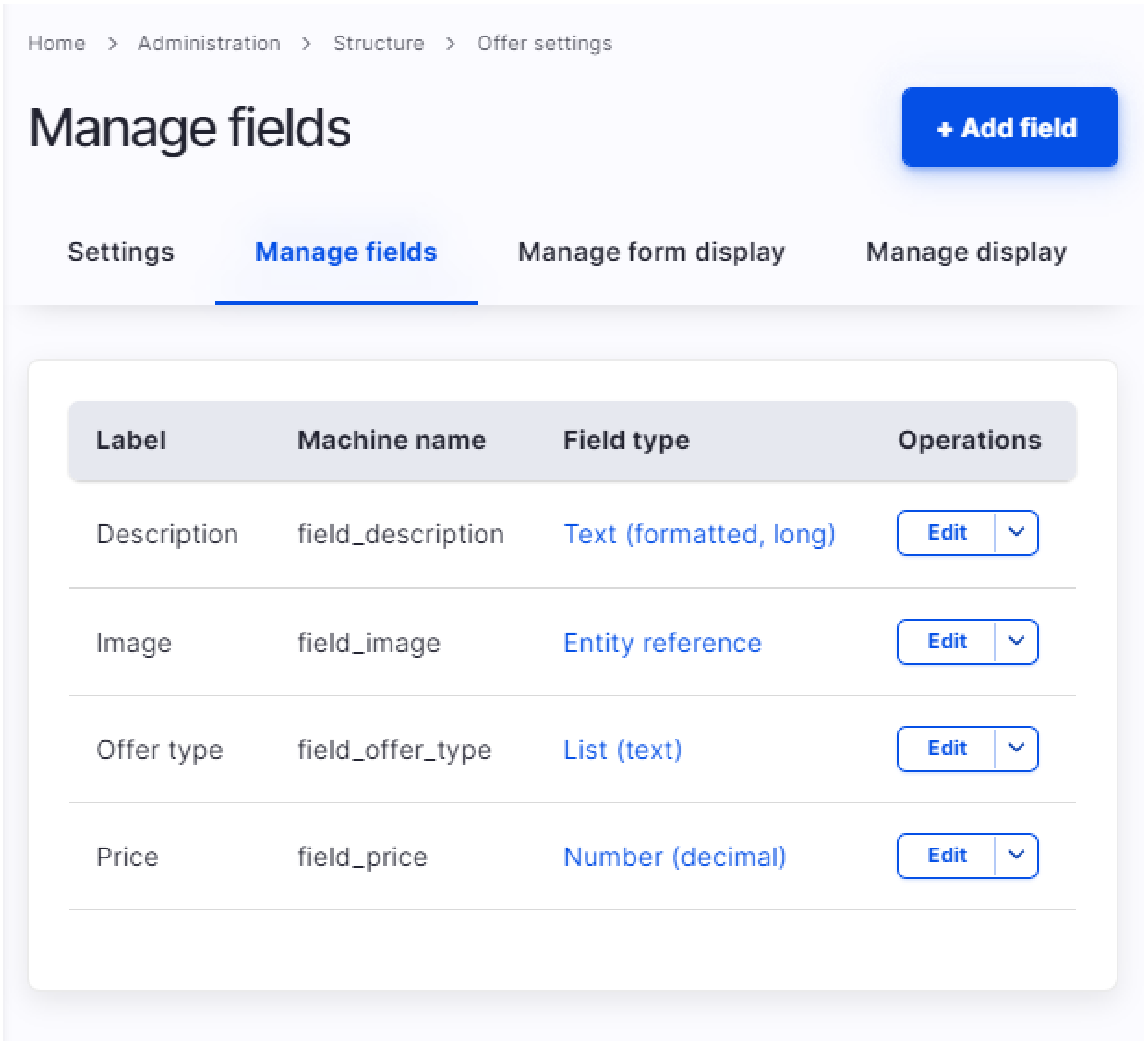
Allowed values list

```
with_minimum|Set minimum price
no_minimum|No minimum price
```

Description (Set the type of offer)

If a user wants to ask for a minimum price, he needs a field to enter the price. We'll add a last field for this:

Add a new field (Number, decimal)
Add a label (Price)
Add a description (Your minimum price (in $))
Minimum (0)
Suffix ($)


Home > Administration > Structure > Offer settings

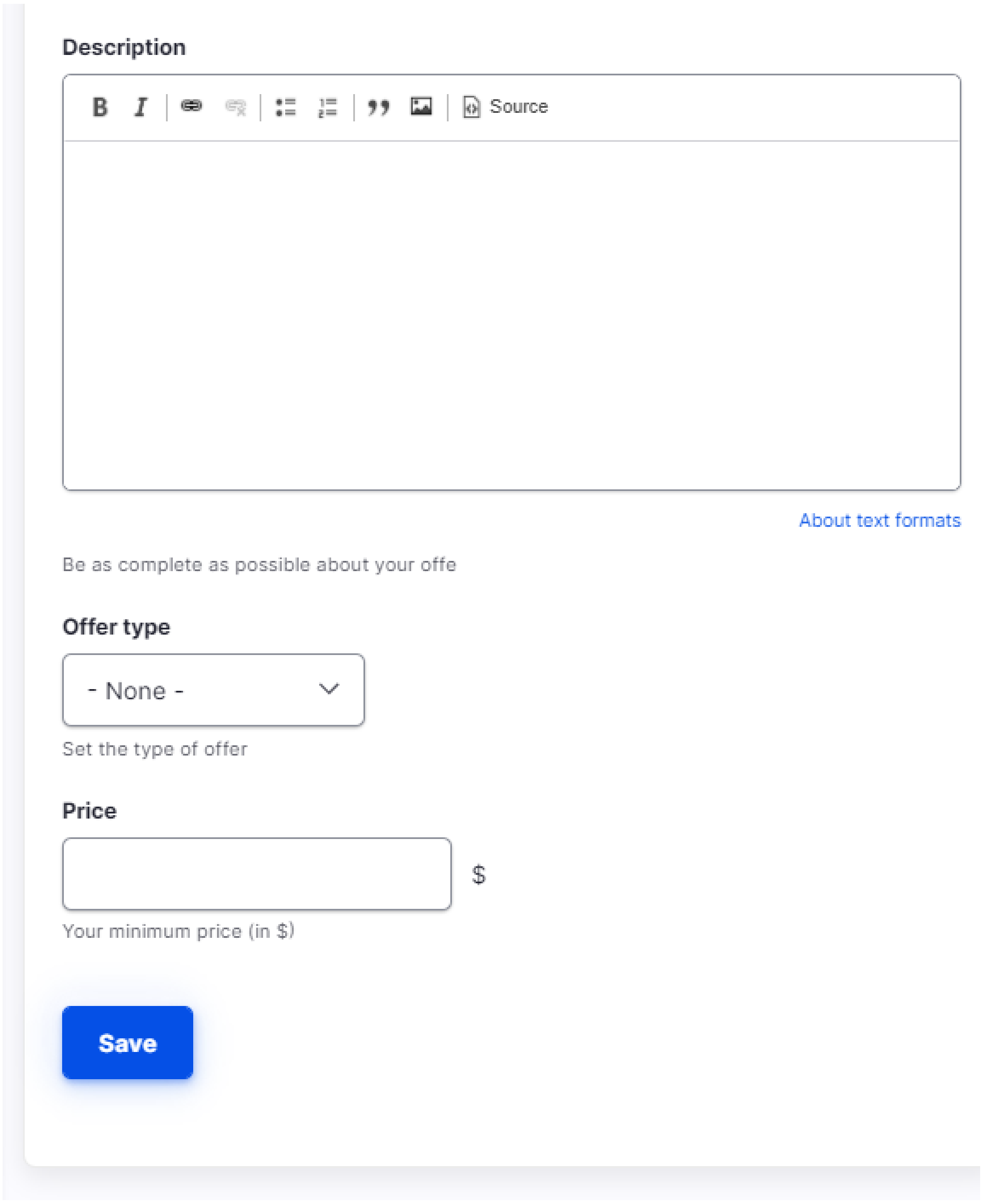
Manage fields

+ Add field

Settings | Manage fields | Manage form display | Manage display

Label	Machine name	Field type	Operations
Description	field_description	Text (formatted, long)	Edit
Image	field_image	Entity reference	Edit
Offer type	field_offer_type	List (text)	Edit
Price	field_price	Number (decimal)	Edit

The fields appear automatically in the entity creation form at /offers/create. Via /admin/structure/offer/form-display we can manipulate order and display settings.



Be as complete as possible about your offe

Offer type

- None -

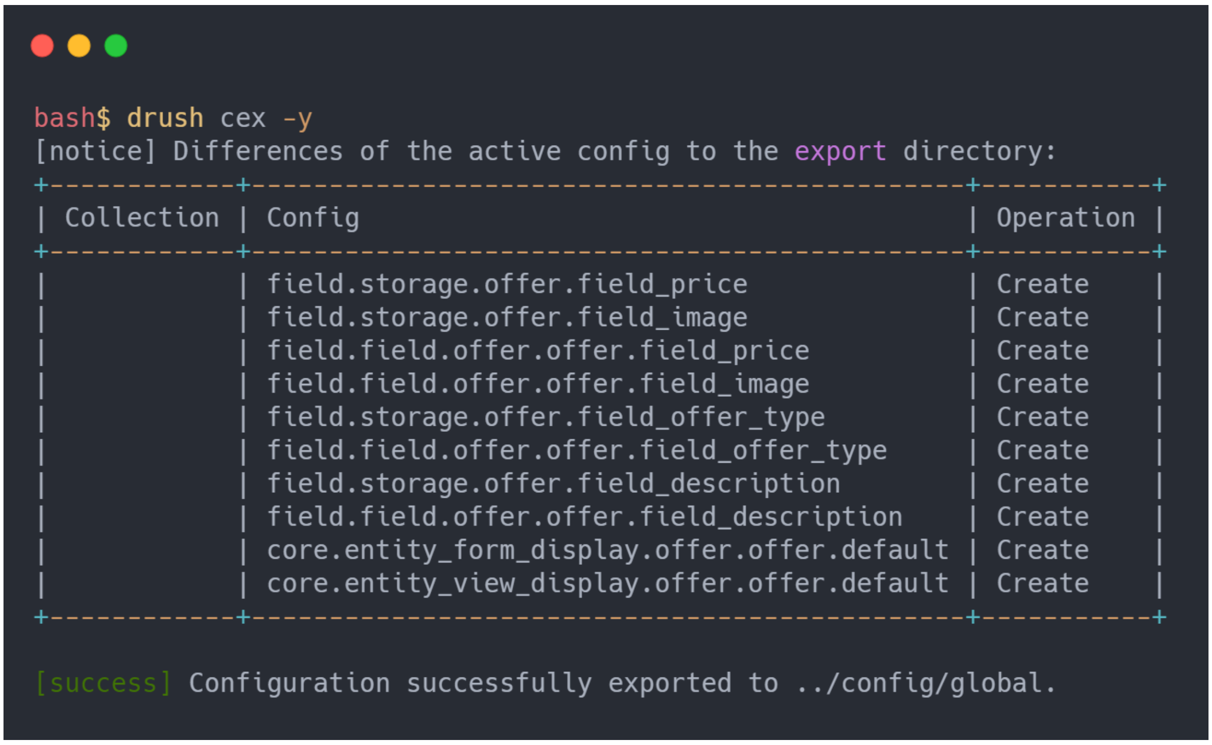
Set the type of offer

Price

$

Your minimum price (in $)

We have now added all fields to our offer entities. We learned that besides fields defined in code, there is also a way to add them via the UI. For the first time, this will create extra configuration via configuration management for our offer entity. Export it with drush to see which files are created.



```
bash$ drush cex -y
[notice] Differences of the active config to the export directory:
+------------+----------------------------------------------+-----------+
| Collection | Config                                       | Operation |
+------------+----------------------------------------------+-----------+
|            | field.storage.offer.field_price              | Create    |
|            | field.storage.offer.field_image              | Create    |
|            | field.field.offer.offer.field_price          | Create    |
|            | field.field.offer.offer.field_image          | Create    |
|            | field.storage.offer.field_offer_type         | Create    |
|            | field.field.offer.offer.field_offer_type     | Create    |
|            | field.storage.offer.field_description        | Create    |
|            | field.field.offer.offer.field_description    | Create    |
|            | core.entity_form_display.offer.offer.default | Create    |
|            | core.entity_view_display.offer.offer.default | Create    |
+------------+----------------------------------------------+-----------+

[success] Configuration successfully exported to ../config/global.
```

Adding workflows and moderation to custom entities

👉 This section teaches you the power of using workflows. We extend our **custom entity** with a state of **Draft, Published** and **Expired**. Based on these states, the published state (yes/no) is set..

After years of building, the **workflows** and **content moderation** modules were added to core. These enormous powerful modules give us the chance to add workflow states like “draft”, “review” and “published” to our entities. Additionally we can give permissions to the different states or add actions when states change.

💻 We enable the core options module for this:

```
bash$ drush en workflows, content_moderation
[success] Successfully enabled: workflows,
content_moderation
```

In our project, we want to add the following **moderation states**:

Moderation state	Is published?	Description
Draft	No	only authors can view their own drafts
Published	Yes	accessible for everyone, bidding enabled
Expired	Yes	visible for everyone, bidding disabled

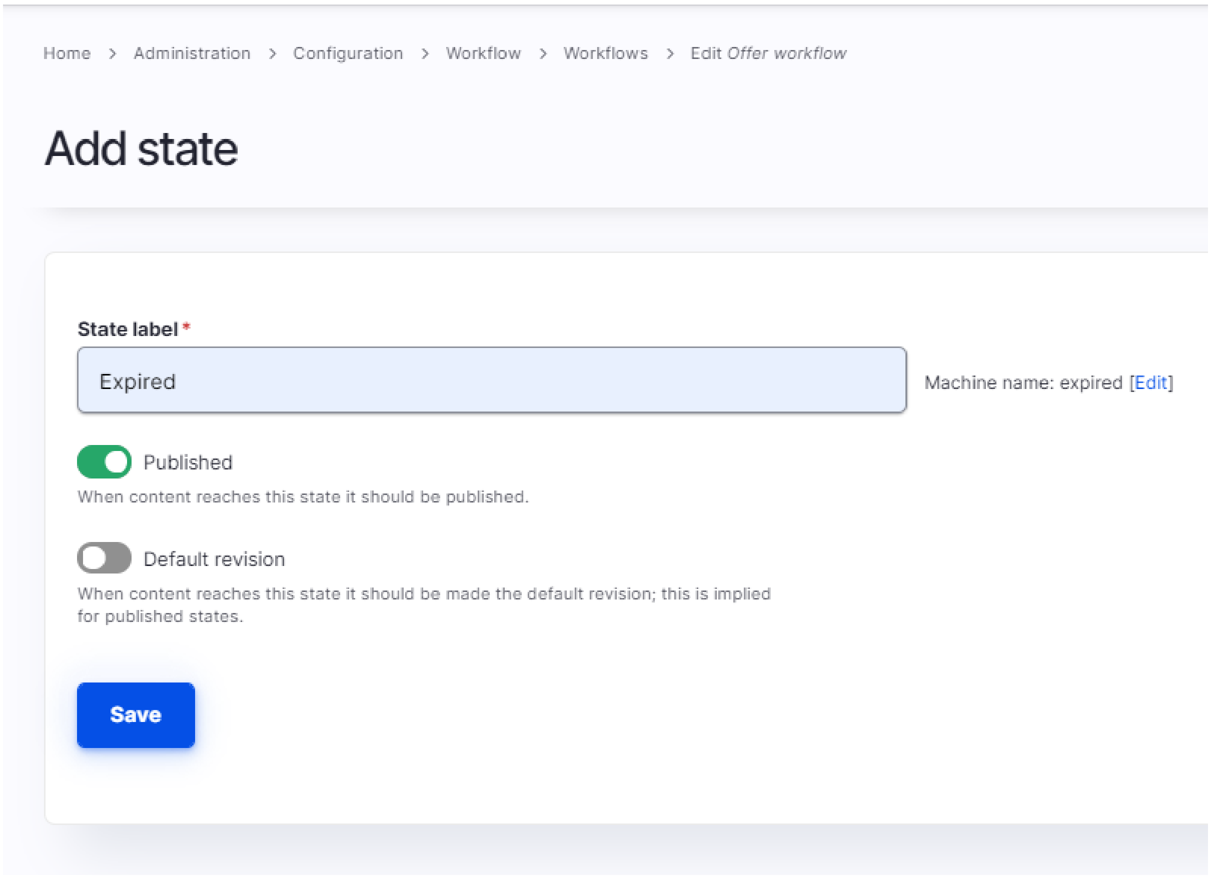
Further, we want to use Workflow for the following **transitions**:

- When an offer goes to an expired status, we want to send an email to all users who have made a bid on the offer.

Via admin/config/workflow/workflows we add a workflow called 'Offer workflow'.

Label (Offer workflow)
Content moderation (Worfklow type)

Under **the states tab**, draft and published are default shipped with the content moderation module. We need to add 'Expired'. Click 'Add a new state':


Home > Administration > Configuration > Workflow > Workflows > Edit *Offer workflow*
Add state
Save

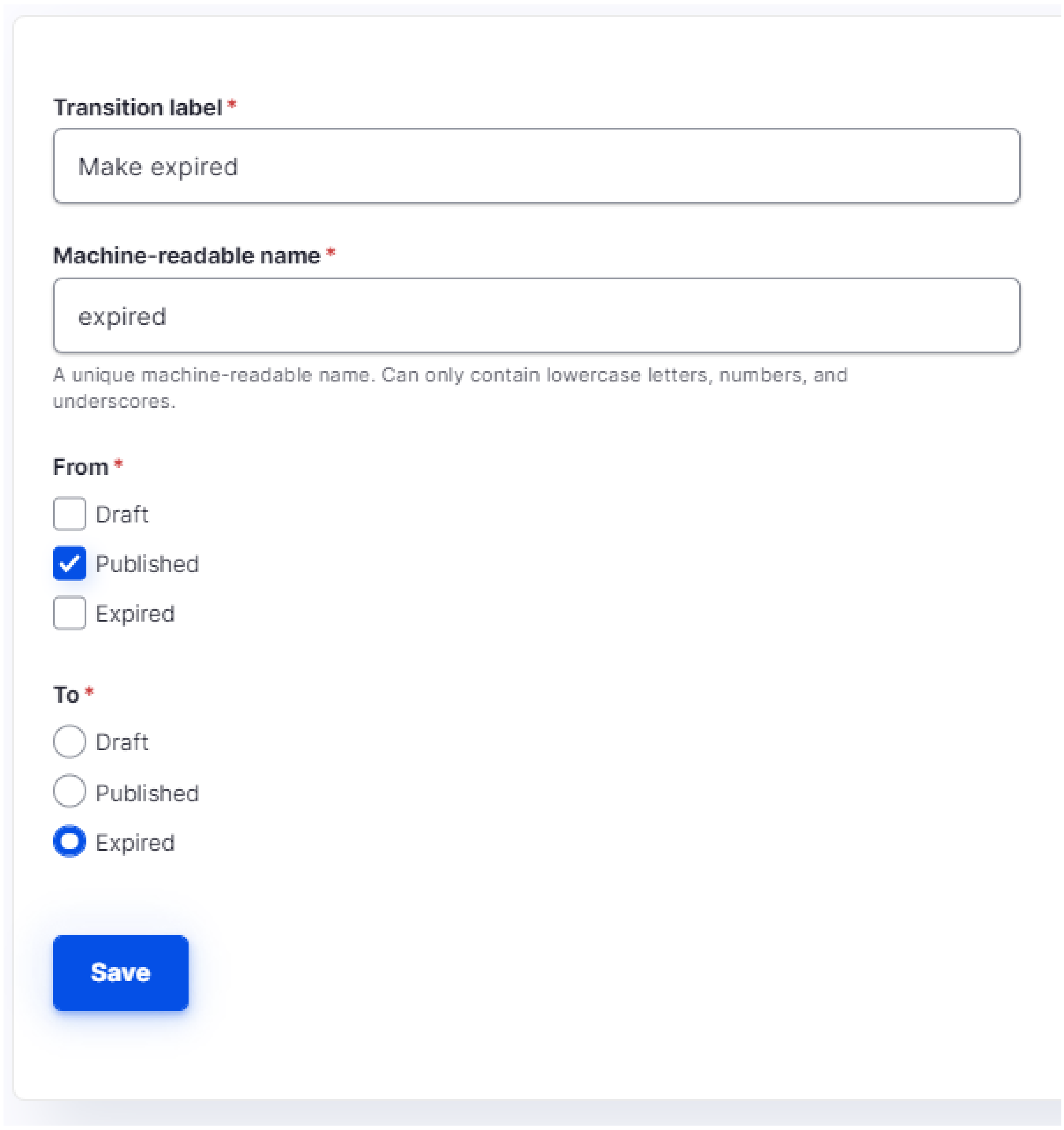
State label *: Expired
Machine name: expired [Edit]

Published: on
When content reaches this state it should be published.

Default revision: off
When content reaches this state it should be made the default revision; this is implied for published states.

Under **the transitions tab**, we see “Create new draft” and “Publish” are already enabled. We add “Make expired” by clicking the “Add a new transition” button.

💻 Use the machine name “expired” here. We will need this for our events later.



Transition label *

Make expired

Machine-readable name *

expired

A unique machine-readable name. Can only contain lowercase letters, numbers, and underscores.

From *

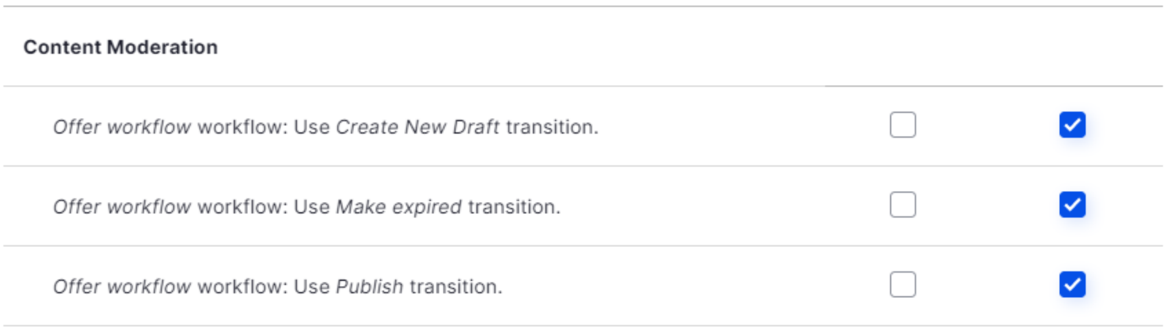
☐ Draft
☑ Published
☐ Expired

To *

○ Draft
○ Published
● Expired

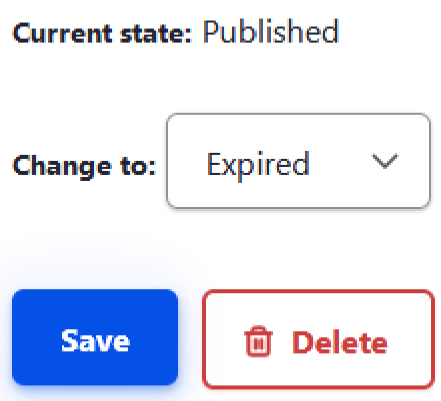
Finally, in the **this workflow applies to** - tab, select the offer entity. The **default moderation state** should be set to Draft.

Via /admin/people/permissions set the permission for transitions to authenticated users:



Content Moderation		
Offer workflow workflow: Use *Create New Draft* transition.	☐	☑
Offer workflow workflow: Use *Make expired* transition.	☐	☑
Offer workflow workflow: Use *Publish* transition.	☐	☑

It will activate a transition button on the edit forms:



Current state: Published

Change to: Expired

In this chapter we learned to add workflow and transition to our entity. Via the the UI we can set up these pretty advanced workflows. In the rest of this course we'll discover what we can do with them in terms of access, visibility, etc.

Change entity access based on workflow states

👉 This short section aims to change the existing access on our **custom entity** to an access based on **workflow state** of the entity.

At this moment, offer entities are only viewable for the owner (creator) of the offer. Of course we want our offers to be visible for everyone once it is in a published state.

We head back over to **custom/offer/src/Entity/OfferAccessControlHandler.php** and change the 'view' state from:

```
case 'view':
 if ($account->hasPermission('administer own offers')) {
   $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
 }
 break;
```

to:

```
case 'view':
 // owners of the offer can always view
 if ($account->id() == $entity->getOwnerId()) {
   $access =
AccessResult::allowed()->cachePerUser()->addCacheableDependency($entity)
;
 } else {
   // Other users only when published or expired
   $allowed = ['published', 'expired'];
   $access =
AccessResult::allowedIf(in_array($entity->get('moderation_state')->getSt
ring(), $allowed))->addCacheableDependency($entity);
 }
 break;
```

This will make my offer visible, even for anonymous visitors of our platform if the offer is in the published or expired state. With a second check, we ensure authors (owners) can see their offers anytime (in all states). This way, we secured the draft (unpublished) state for outside viewers.

While permissions in drupal are often based on a **permission** level via admin/permission, we extended it now with **ownership** and **workflow** of an entity. This allows us very specific and advanced control.

Adding views plugins: custom fields and operation links

👉 This section is meant to teach you how to add **custom fields** to your **views**. At the end you will be able to show custom output in views listings like tables. More specifically you will master how to add your own **operation links** in a **dropbutton** to add a direct publishing button, for example.

Our entity already has views access defined. We can now extend **custom/offer/src/OfferViewsData.php** with custom fields. First we want to add a nice badge to the table to show the moderation states:

```
<?php

namespace Drupal\offer;

use Drupal\views\EntityViewsData;

/**
* Provides views data for Offer entities.
*
*/
class OfferViewsData extends EntityViewsData {

  /**
   * Returns the Views data for the entity.
   */
  public function getViewsData() {
    $data = parent::getViewsData();

    $data['offer']['offer_entity_moderation_state_views_field'] = [
      'title' => t('Moderation status'),
      'field' => array(
        'title' => t('Moderation status'),
        'help' => t('Shows the state of the offer entity.'),
        'id' => 'offer_entity_moderation_state_views_field',
      ),
    ];

    return $data;
  }
}
```

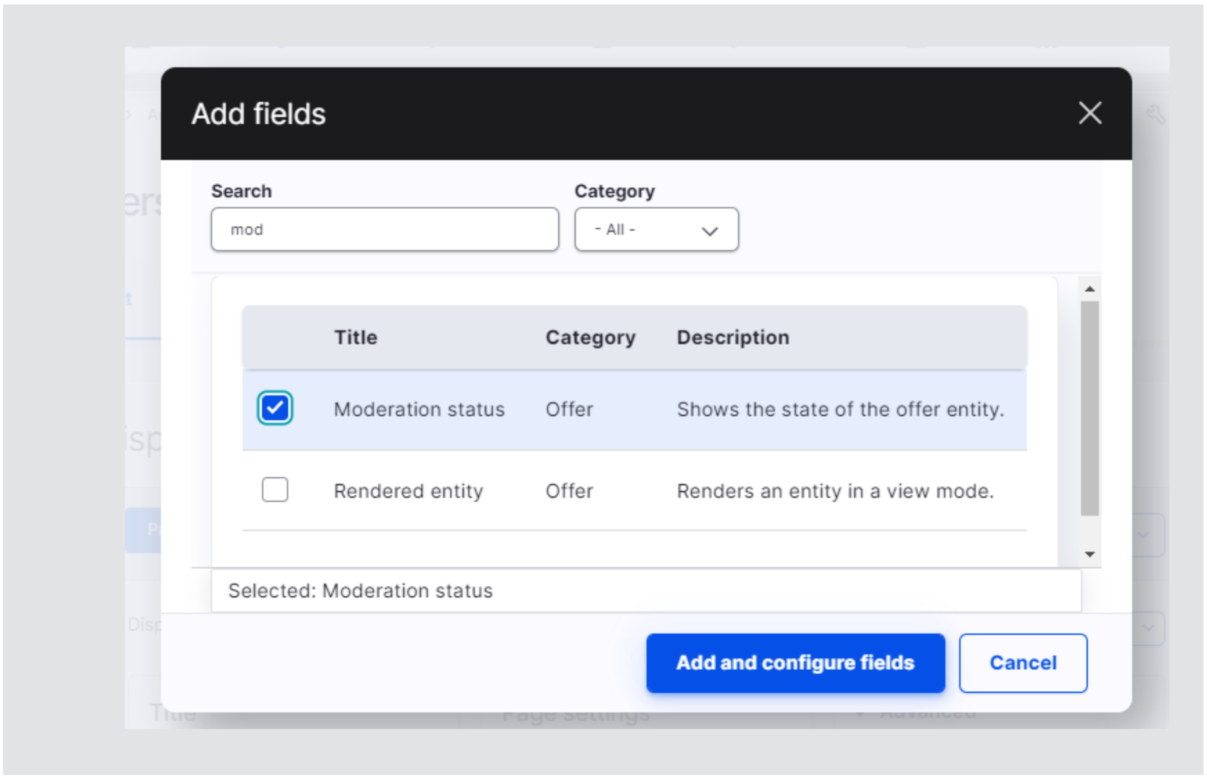
For this to work, we need to add a views plugin. Add a file to **custom/offer/src/Plugin/views/field/OfferEntityModerationState.php.** This provides us many functions, for which we won't go into detail. But take a look at what we did at the render() function. There we load the value of the moderation state and return it.

```
<?php
```

```
namespace Drupal\offer\Plugin\views\field;

use Drupal\views\Plugin\views\field\FieldPluginBase;
use Drupal\views\ResultRow;
use Drupal\views\Plugin\views\display\DisplayPluginBase;
use Drupal\views\ViewExecutable;

/**
* A handler to provide a field that is completely custom by the
administrator.
*
* @ingroup views_field_handlers
*
* @ViewsField("offer_entity_moderation_state_views_field")
*/
class OfferEntityModerationState extends FieldPluginBase {

 /**
  * The current display.
  *
  * @var string
  *   The current display of the view.
  */
 protected $currentDisplay;

 /**
  * {@inheritdoc}
  */
 public function init(ViewExecutable $view, DisplayPluginBase
$display, array &$options = NULL) {
   parent::init($view, $display, $options);
   $this->currentDisplay = $view->current_display;
 }

 /**
  * {@inheritdoc}
  */
 public function usesGroupBy() {
   return FALSE;
 }
```

```
  /**
   * {@inheritdoc}
   */
  public function query() {
    // Do nothing -- to override the parent query.
  }

  /**
   * {@inheritdoc}
   */
  protected function defineOptions() {
    $options = parent::defineOptions();
    $options['hide_alter_empty'] = ['default' => FALSE];
    return $options;
  }

  /**
   * {@inheritdoc}
   */
  public function render(ResultRow $values) {
    $entity = $values->_entity;
    $state = $entity->get('moderation_state')->getString();
    return $state;
  }

}
```

💻 Clear caches and add the moderation state to the offer views table. Also, make sure you remove the “Published status = TRUE”-filter from the views filters so draft offers also appear in the listing.



Search

mod

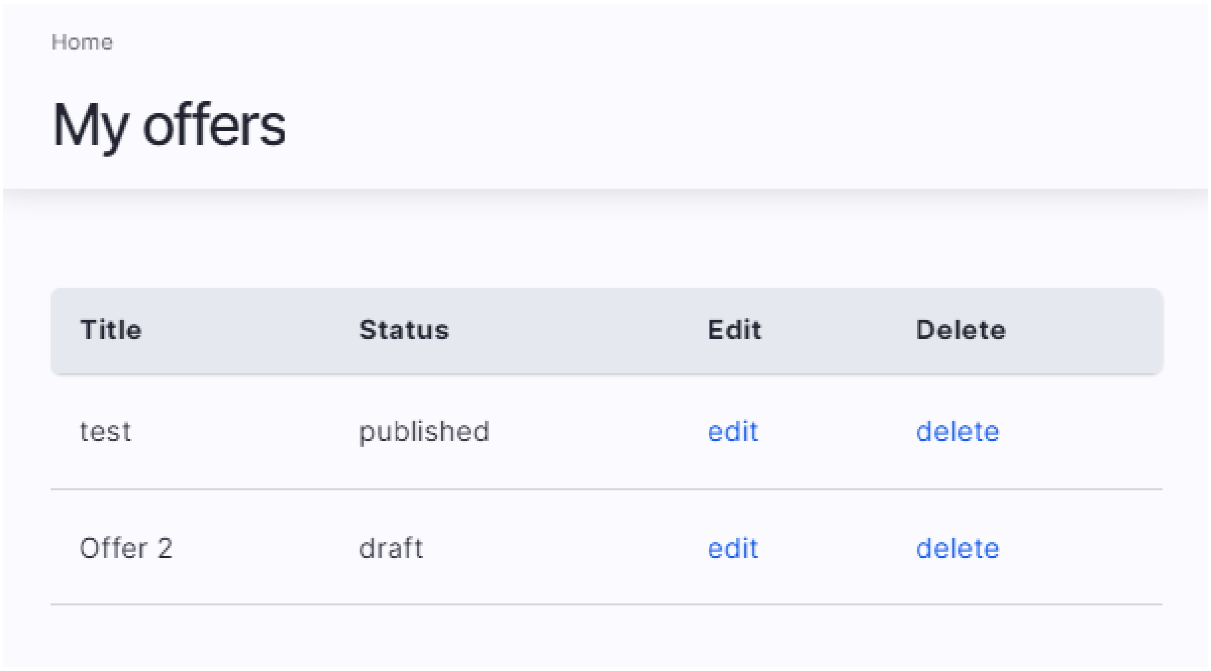
Category

- All -

	Title	Category	Description
✓	Moderation status	Offer	Shows the state of the offer entity.
☐	Rendered entity	Offer	Renders an entity in a view mode.

Selected: Moderation status

Now our listing shows the moderation status as well at /offers:



My offers

Title	Status	Edit	Delete
test	published	edit	delete
Offer 2	draft	edit	delete

Let’s move on with a second field. What I always advise people when doing custom applications with drupal, is to use the power of the core render arrays. There are so many core things you can render, like dropdowns, tables, links, …

A render array is an associative array which conforms to the standards and data structures used in Drupal's Render API. The Render API is also integrated with the Theme API.

The advantage of using them is they stay consistent when upgrading. Plus when you change your layout with a theme from the community, you are sure they will look nice as well.

We want to make another custom views plugin field that shows a **dropbutton** for editing and deleting. This makes our table a bit more compact. When an offer is still in draft mode, we also want a "Publish" option added to the dropbutton. We start by extending the **custom/offer/src/OfferViewsData.php** *getViewsData()* function with a new field:

```
$data['offer']['offer_dynamic_operation_links'] = [
 'title' => t('Dynamic operations'),
 'field' => array(
   'title' => t('Dynamic operations'),
   'help' => t('Shows a dropbutton with dynamic operations for 
offers.'),
   'id' => 'offer_dynamic_operation_links',
 ),
];
```

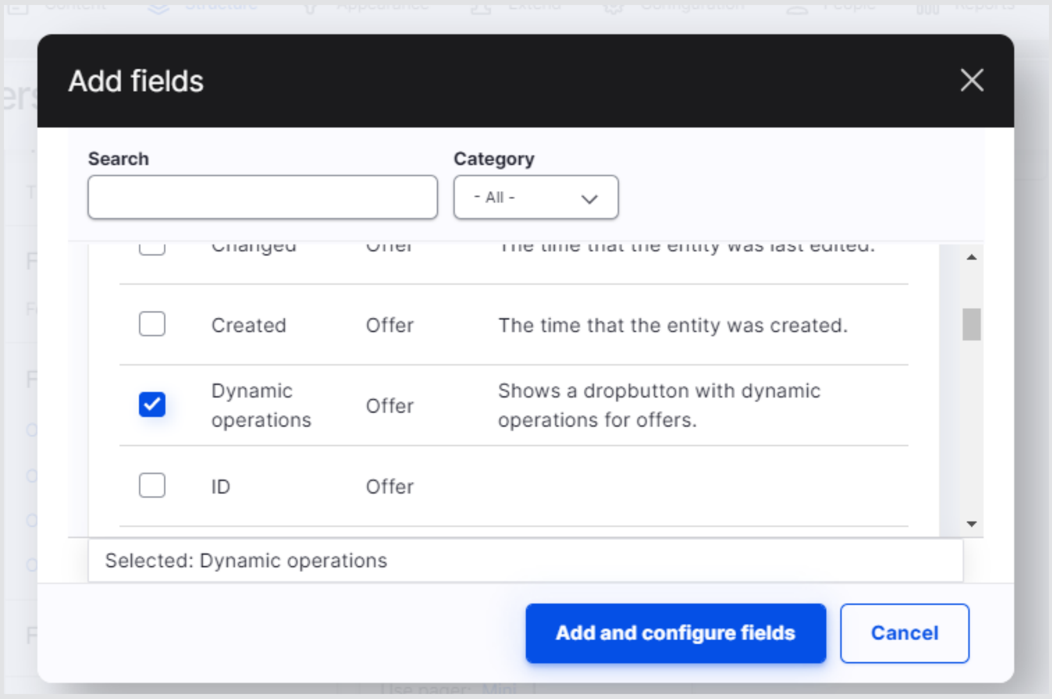
After this, we need to add a new views plugin:
custom/offer/src/Plugin/views/field/OfferDynamicOperationLinks.php.

```
<?php

namespace Drupal\offer\Plugin\views\field;

use Drupal\views\Plugin\views\field\FieldPluginBase;
use Drupal\views\ResultRow;
use Drupal\views\Plugin\views\display\DisplayPluginBase;
use Drupal\views\ViewExecutable;
use Drupal\core\Url;

/**
 * A handler to provide a field that is completely custom by the 
administrator.
```

```
 *
 * @ingroup views_field_handlers
 *
 * @ViewsField("offer_dynamic_operation_links")
 */
class OfferDynamicOperationLinks extends FieldPluginBase
{

  /**
   * The current display.
   *
   * @var string
   *   The current display of the view.
   */
  protected $currentDisplay;

  /**
   * {@inheritdoc}
   */
  public function init(ViewExecutable $view, DisplayPluginBase $display,
array &$options = NULL)
  {
    parent::init($view, $display, $options);
    $this->currentDisplay = $view->current_display;
  }

  /**
   * {@inheritdoc}
   */
  public function usesGroupBy()
  {
    return FALSE;
  }

  /**
   * {@inheritdoc}
   */
  public function query()
  {
    // Do nothing -- to override the parent query.
  }

  /**
   * {@inheritdoc}
   */
  protected function defineOptions()
  {
    $options = parent::defineOptions();
    $options['hide_alter_empty'] = ['default' => FALSE];
    return $options;
  }

  /**
   * {@inheritdoc}
   */
  public function render(ResultRow $values)
  {
    $entity = $values->_entity;
    $state = $entity->get('moderation_state')->getString();

    switch ($state) {
      case 'draft':
        $operations['publish'] = [
          'title' => $this->t('Publish'),
          'url' => Url::fromRoute('entity.offer.edit_form', ['offer' =>
$entity->id()])
        ];
        break;
    }

    $operations['edit'] = [
      'title' => $this->t('Edit'),
      'url' => Url::fromRoute('entity.offer.edit_form', ['offer' =>
$entity->id()])
    ];

    $operations['delete'] = [
      'title' => $this->t('Delete'),
      'url' => Url::fromRoute('entity.offer.delete_form', ['offer' =>
$entity->id()])
    ];

    $dropbutton = [
      '#type' => 'dropbutton',
      '#links' => $operations
    ];

    return render($dropbutton);

  }
```

```
}
```

💻 Now we add the field to the view. It is now available at admin/structure/views/view/offers:

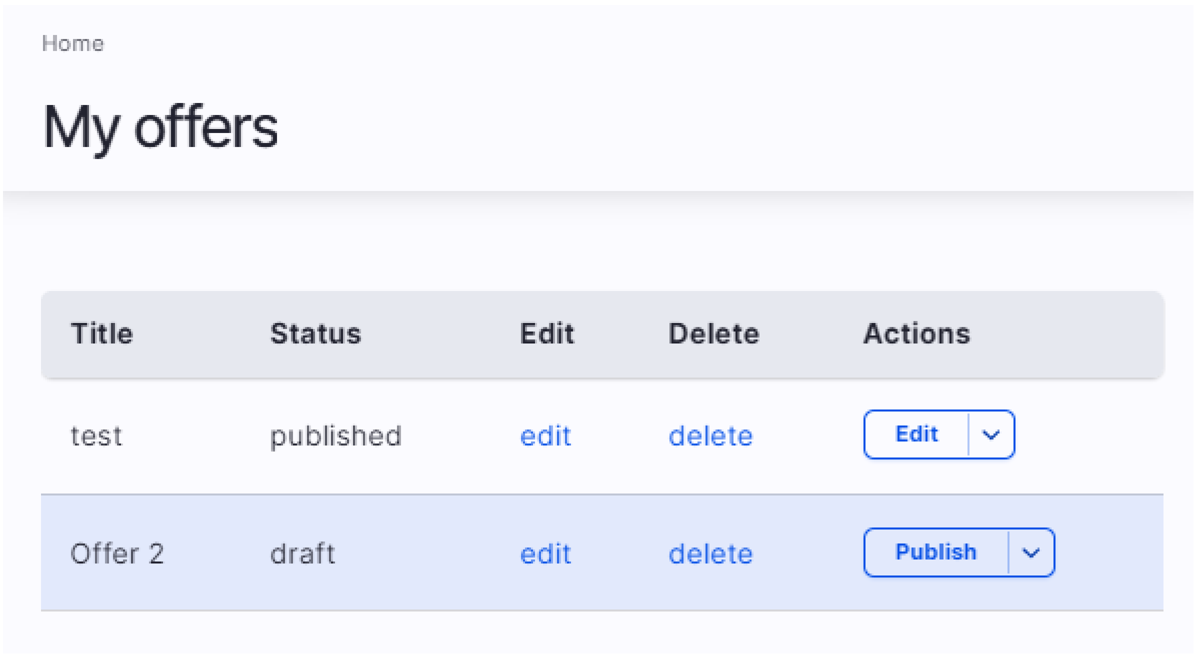


☐	Changed	Offer	The time that the entity was last edited.
☐	Created	Offer	The time that the entity was created.
✓	Dynamic operations	Offer	Shows a dropbutton with dynamic operations for offers.
☐	ID	Offer	

Selected: Dynamic operations

After we save, you see we've added dynamic dropdown buttons based on the entity state. When an entity has the 'draft' state, a publish button is visible. **At this moment, the 'Publish' button is just linked to the edit page**. In the next chapter we'll add a custom controller in order for this to work.

Adding a


Home

My offers

Title	Status	Edit	Delete	Actions
test	published	edit	delete	Edit
Offer 2	draft	edit	delete	Publish

💻 Now all our actions are in the Actions column, we can delete the Edit and Delete column in views as well.

custom controller for direct publishing

👉 This section focuses on adding a custom controller. At the end you will master the possibilities of **routing** in drupal, know how **parameter upcasting** works and how to **validate** the slug.

The 'publish' button is pointed to the edit page. We would like to provide a direct publish controller, with redirect to the published offer.

First, to **custom/offer/offer.routing.yml** we add:

```
offer.publish:
  path: '/offers/publish/{offer}'
  defaults:
    _controller:
'\Drupal\offer\Controller\OfferPublishController::Render'
```

```
    _title_callback: 'Publish offer'
  requirements:
    _custom_access:
'\Drupal\offer\Controller\OfferPublishController::Access'
  options:
    parameters:
      offer:
        type: entity:offer
```

Let’s go over the different keys of the snippet:

Key	Explanation
offer.publish	This is the uri that gets registered in drupal for this page.
path	The url of this page. We add a slug with {id} to identify the id of the offer we want to publish.
_controller	The class that gets called when visiting the page.
_title callback	The title of the page. We can add a string or a method inside the class.
id	We add a validator to the id slug. By saying ‘ \d+’, we mean that id has to be numeric.
_custom_access	That method can then check access and return an access result object. The advantage of this is that menu links that point to the controller can automatically check access and decide to hide the link when necessary.

parameters:
  offer:
    type: entity:offer
	A route can have route parameters. In this case our slug is {offer}. We tell the system these are of type entity:offer. This is called route parameter upcasting.This way our routing is automatically secured against wrong, non-numerical calls etc.
	A bonus is that we can use the entity directly in our Render() and Access() functions like this:
	Render(Offer $offer) {}

Add a controller to **custom/offer/src/Controller/OfferPublishController**

```
<?php
```

```php
namespace Drupal\offer\Controller;

use Drupal\Core\Access\AccessResult;
use Drupal\Core\Controller\ControllerBase;
use Drupal\Core\Entity\RevisionLogInterface;
use Drupal\Core\Url;
use Drupal\offer\Entity\Offer;
use Symfony\Component\HttpFoundation\RedirectResponse;

/**
* Class OfferPublishController
*/
class OfferPublishController extends ControllerBase {

 public function Render(Offer $offer) {

   // We set the moderation to published
   $new_state = 'published';
   $offer->set('moderation_state', $new_state);
   if ($offer instanceof RevisionLogInterface) {
     $offer->setRevisionLogMessage('Changed moderation state to
Published.');
     $offer->setRevisionUserId($this->currentUser()->id());
   }
   $offer->save();

   $publishedOffer = Url::fromRoute('entity.offer.canonical',
['offer' => $offer->id()]);

   \Drupal::messenger()->addMessage($offer->label() . ' is
published.');

   return new RedirectResponse($publishedOffer->toString());

 }

 public function Access(Offer $offer) {

   // Securing no one is trying to publish other people's offers.
   $access = AccessResult::allowedIf($offer->access('view'));
   // Make sure state is draft
```

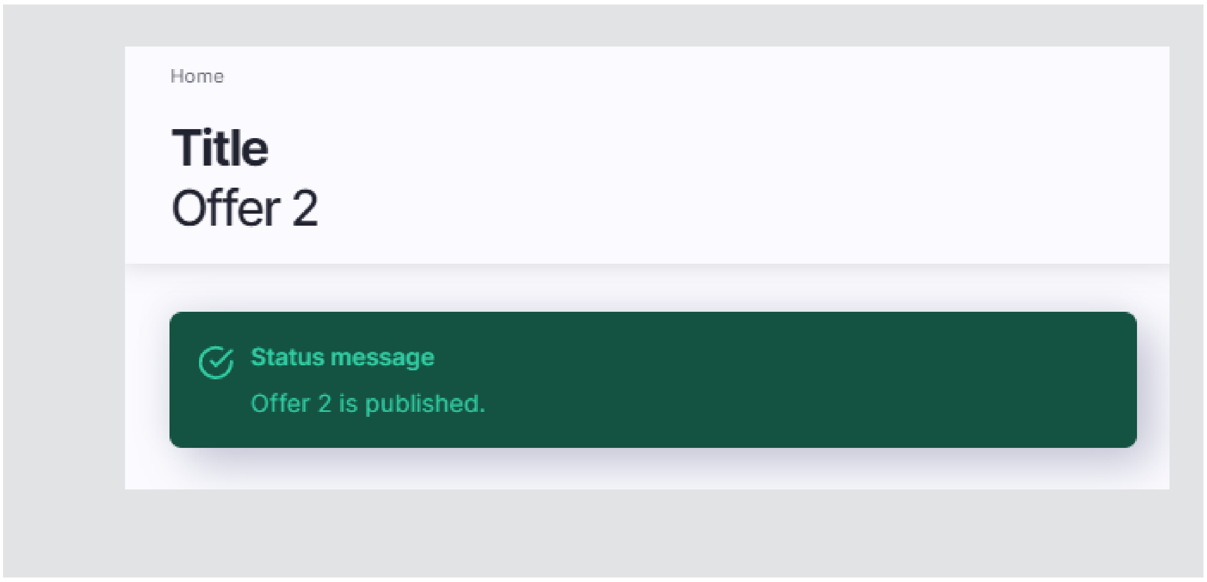
```
    if($offer->get('moderation_state')->getString() != 'draft') {
      $access = AccessResult::forbidden();
    }

    return $access;
  }
}
```

In the ***Render()*** method, we programmatically set the moderation state to published and redirect to the collection overview. Thanks to the **Access()** method, we secure the page for abuse: we do an **$offer->access()** check that verifies the visitor (it goes over the **OfferAccessControlHandler.php)** is the owner and we check if the moderation state is draft.

💻 In the previous chapter we added a publish button in the view that was linked to the edit page. Link it to the publish page at **custom/offer/src/Plugin/views/field/OfferDynamicOperationLinks.php** on line **71**:

```
switch ($state) {
  case 'draft':
    $operations['publish'] = [
      'title' => $this->t('Publish'),
      'url' => Url::fromRoute('offer.publish', ['offer'
=> $entity->id()])
    ];
    break;
}
```

Clear caches and publish a draft offer.


Home

Title
Offer 2

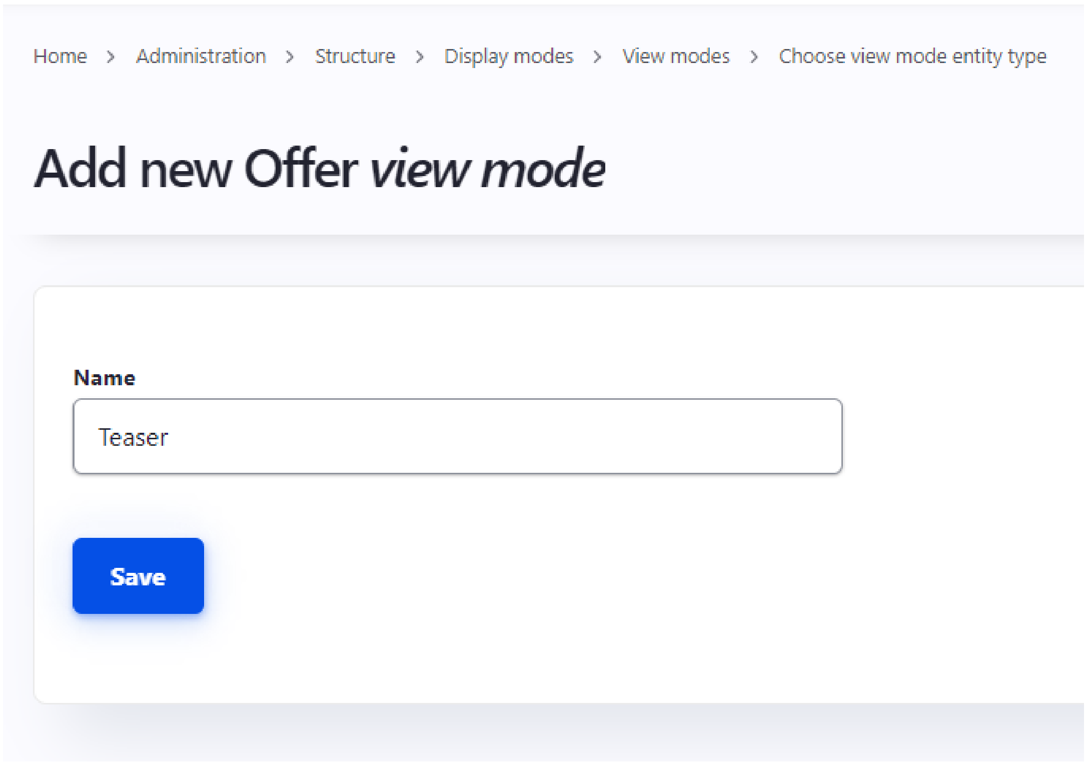
Status message
Offer 2 is published.

Building an overview page

👉 In this section you will be taught how to make **entity listings** with views using **view modes**.

We need an overview page where all the offers are viewable for users that want to bid. The core views module comes back into play.

But first, we want a *teaser* view mode for the offer entities. At /admin/structure/display-modes/view/add/offer. we add a new one named *teaser.*

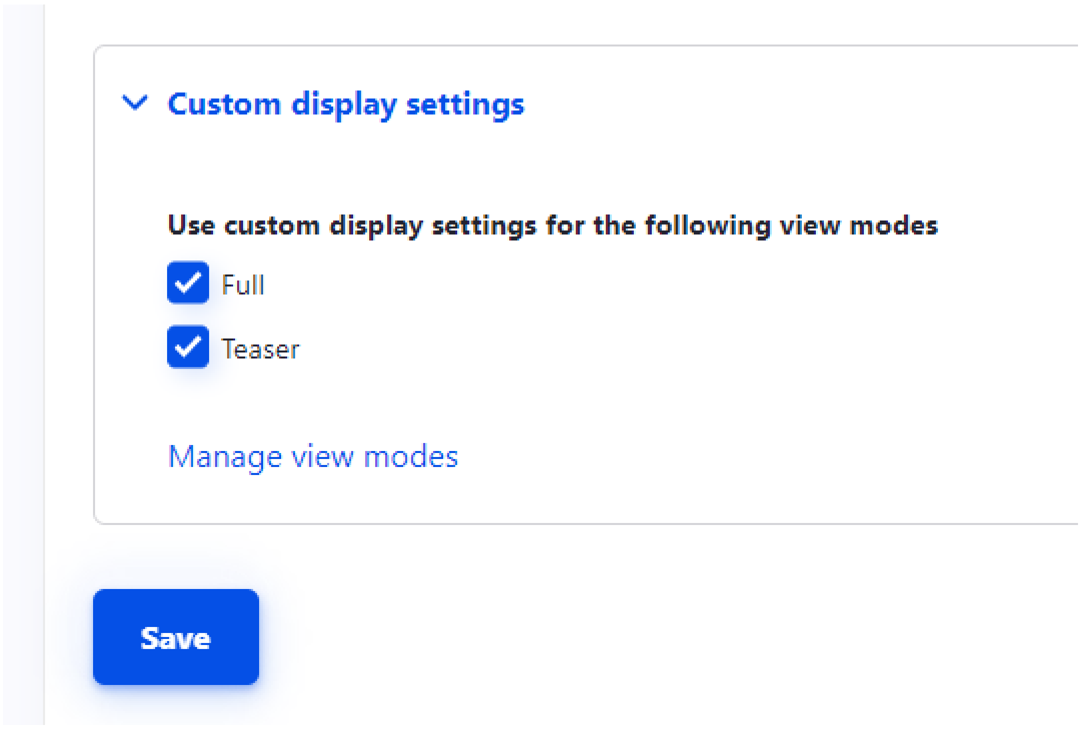

Home › Administration › Structure › Display modes › View modes › Choose view mode entity type

Add new Offer *view mode*

Name: Teaser

💻 Also, add a display mode called ‘full’, for later in the course.

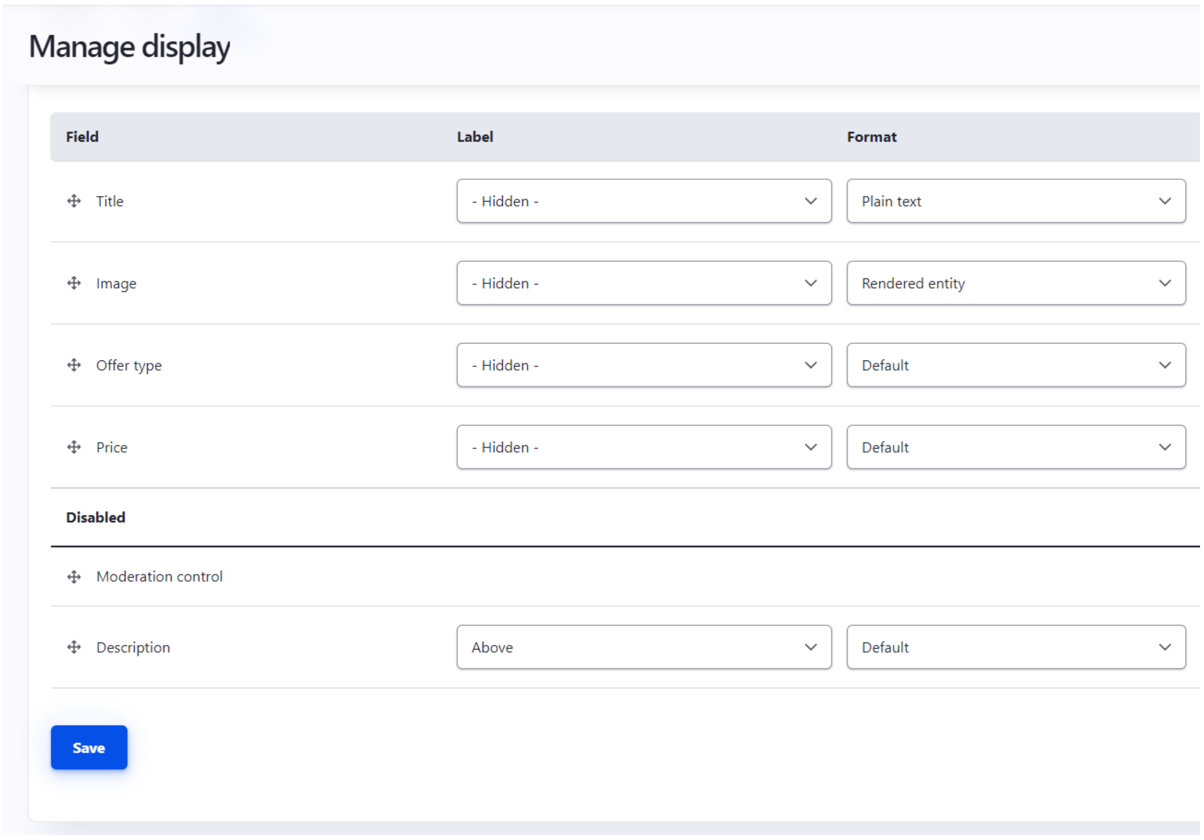
Make sure you activate the view modes via /admin/structure/offer/display:



Use custom display settings for the following view modes

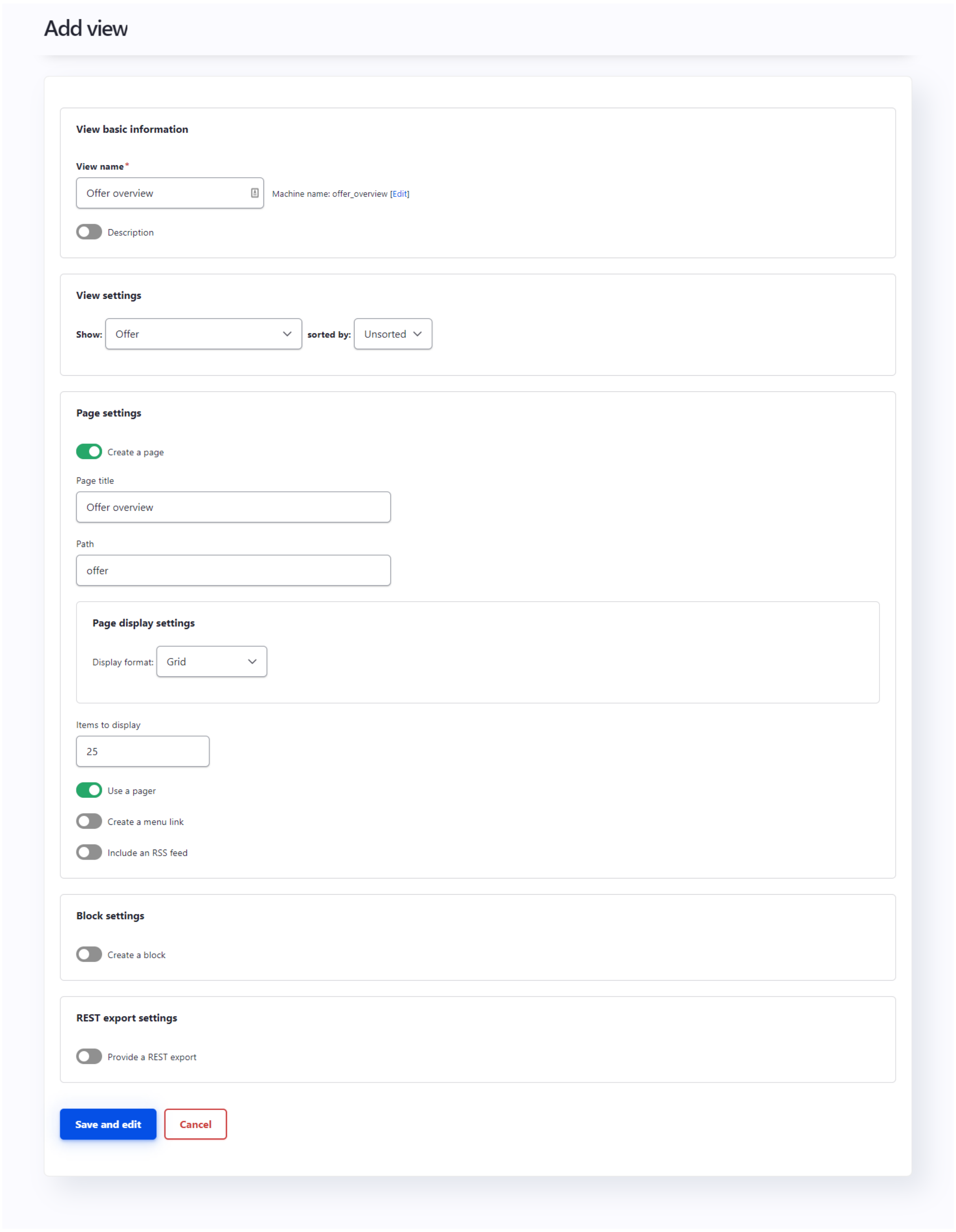
- ✓ Full
- ✓ Teaser

At /admin/structure/offer/display/teaser we add the fields we want to use in our twig files for the teasers of offer entities. For the teaser, we need title, offer type, price, and image.


Manage display

Field	Label	Format
Title	- Hidden -	Plain text
Image	- Hidden -	Rendered entity
Offer type	- Hidden -	Default
Price	- Hidden -	Default
Disabled		
Moderation control		
Description	Above	Default

Time to build our view. Go to admin/structure/views/*add* to add a view named “Offer overview”



Add view

View basic information

View name *

Offer overview

Machine name: offer_overview [Edit]

Description

View settings

Show: Offer sorted by: Unsorted

Page settings

Create a page

Page title

Offer overview

Path

offer

Page display settings

Display format: Grid

Items to display

25

Use a pager

Create a menu link

Include an RSS feed

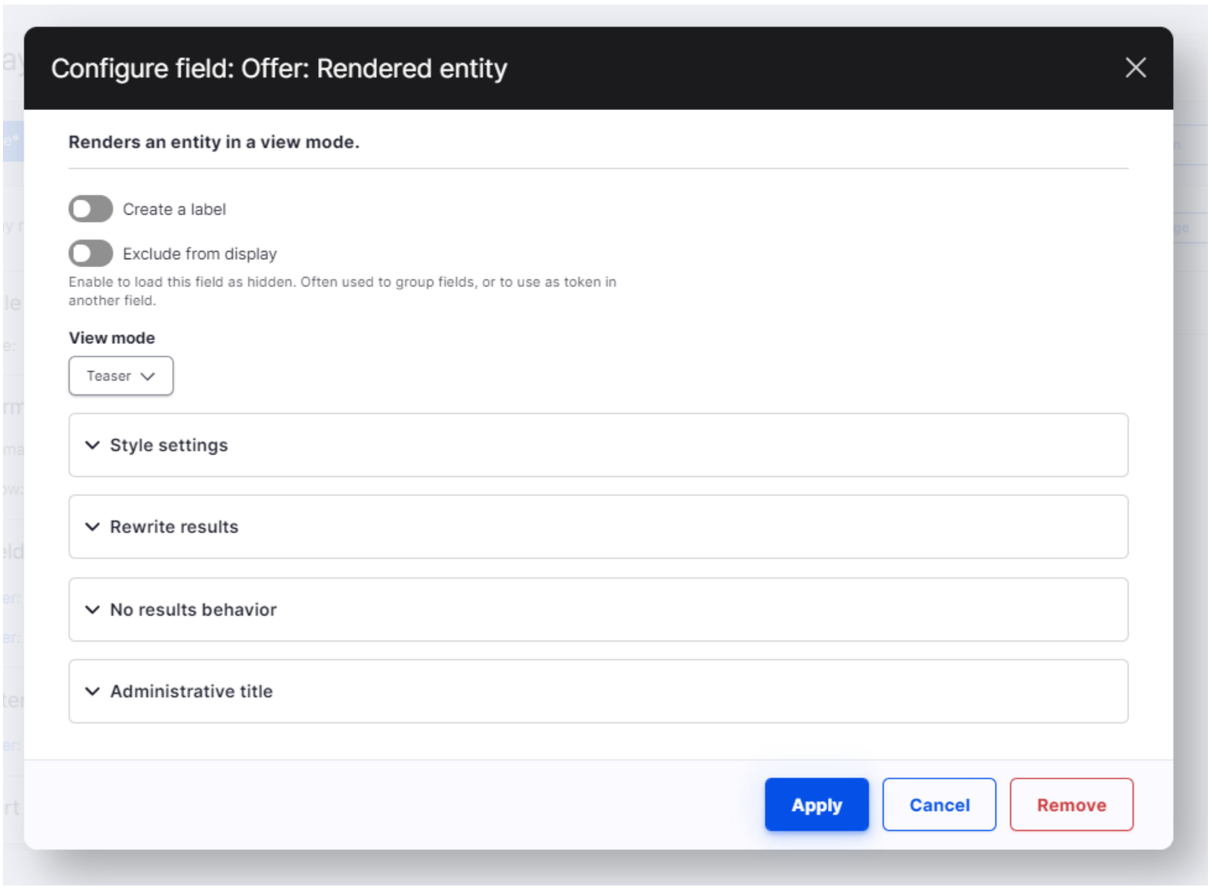
Block settings

Create a block

REST export settings

Provide a REST export

At the **fields** section, add “rendered entity”, choos display mode **teaser**.


Configure field: Offer: Rendered entity

Renders an entity in a view mode.

Create a label

Exclude from display

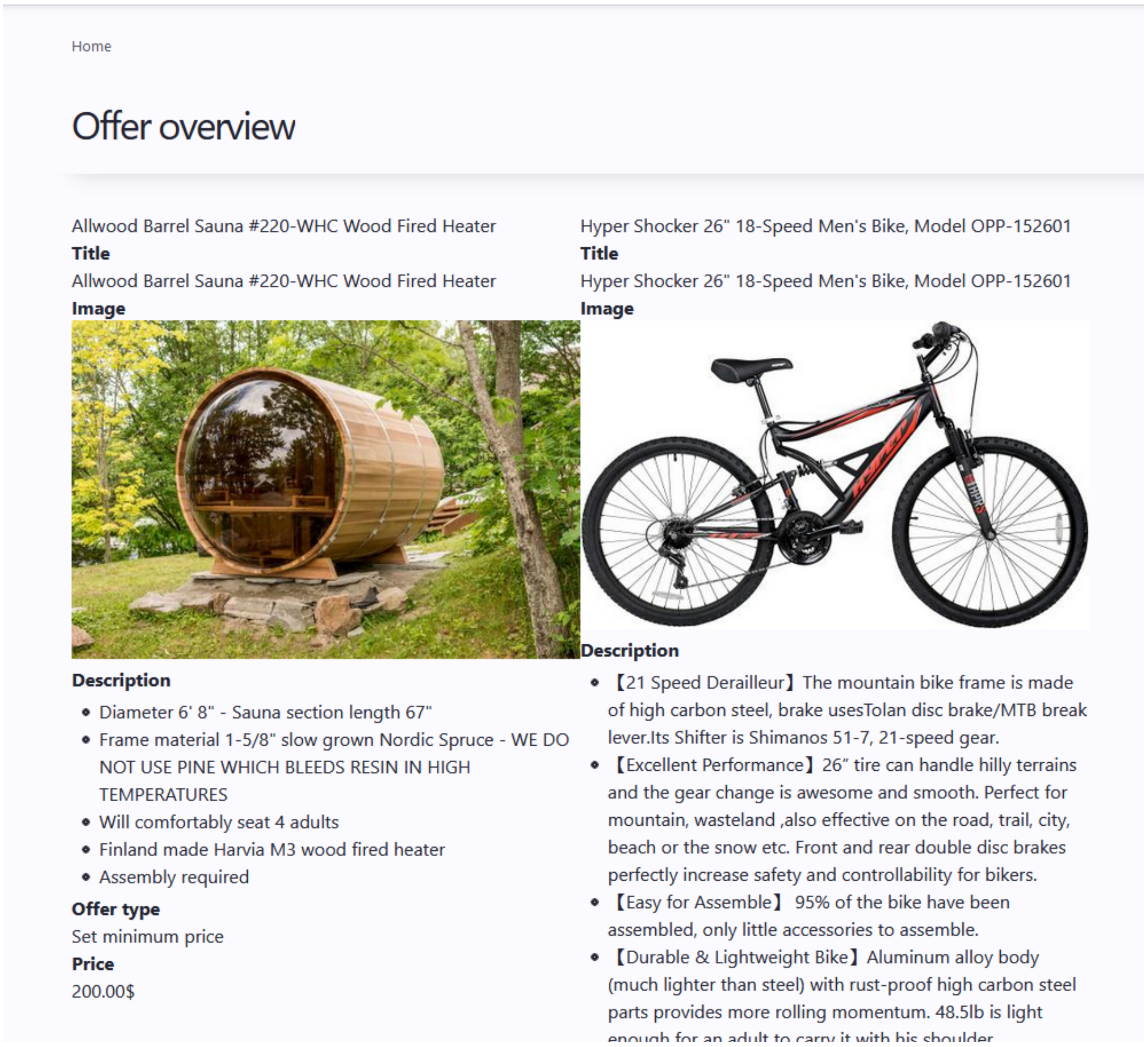
Enable to load this field as hidden. Often used to group fields, or to use as token in another field.

View mode

Teaser

We'll take a look at what we have now:

(



Home

Offer overview

Allwood Barrel Sauna #220-WHC Wood Fired Heater
Title
Allwood Barrel Sauna #220-WHC Wood Fired Heater
Image

Description

- Diameter 6' 8" - Sauna section length 67"
- Frame material 1-5/8" slow grown Nordic Spruce - WE DO NOT USE PINE WHICH BLEEDS RESIN IN HIGH TEMPERATURES
- Will comfortably seat 4 adults
- Finland made Harvia M3 wood fired heater
- Assembly required

Offer type
Set minimum price
Price
200.00$

Hyper Shocker 26" 18-Speed Men's Bike, Model OPP-152601
Title
Hyper Shocker 26" 18-Speed Men's Bike, Model OPP-152601
Image

Description

- 【21 Speed Derailleur】The mountain bike frame is made of high carbon steel, brake usesTolan disc brake/MTB break lever.Its Shifter is Shimanos 51-7, 21-speed gear.
- 【Excellent Performance】26" tire can handle hilly terrains and the gear change is awesome and smooth. Perfect for mountain, wasteland ,also effective on the road, trail, city, beach or the snow etc. Front and rear double disc brakes perfectly increase safety and controllability for bikers.
- 【Easy for Assemble】 95% of the bike have been assembled, only little accessories to assemble.
- 【Durable & Lightweight Bike】Aluminum alloy body (much lighter than steel) with rust-proof high carbon steel parts provides more rolling momentum. 48.5lb is light enough for an adult to carry it with his shoulder

There is a bit of a problem here. We only want the ones with moderation state ‘published’ in the overview. Not the ones with ‘expired’ status. For a solution for this, head on to the next chapter.

Adding a custom views filter based on moderation state

👉 This section spotlights **custom views filters**. At the end you will be able to use a self-defined filter in the user interface of views. More specifically an example with a filter on certain moderation state (filter only on offers with moderation state “Published”) is elaborated.

Moderation state is not an available filter in drupal core (*yet?*). But this is a good opportunity to learn how to add a custom filter in views, based on a state of the entity.

We extend our *viewsData()* function inside **custom/offer/src/OfferViewsData.php** with a new field, this time a filter:

```
$data['offer']['offer_moderation_state_filter'] = [
 'title' => t('Moderation state'),
 'filter' => [
   'title' => t('Moderation state'),
   'help' => 'Filters on moderation state',
   'field' => 'id',
   'id' => 'offer_moderation_state_filter',
 ]
];
```

We add a file to **custom/offer/src/Plugin/views/filter** named **ModerationStateFilter.php**:

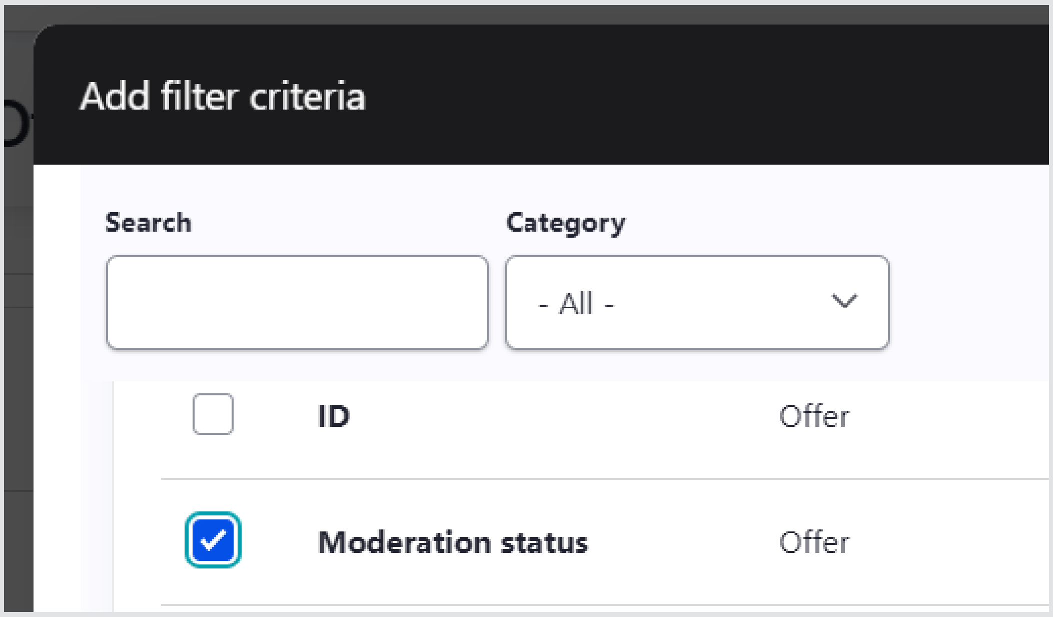
```
<?php

namespace Drupal\offer\Plugin\views\filter;

use Drupal\Core\Database\Connection;
use Drupal\views\Plugin\views\filter\InOperator;
use Drupal\views\ViewExecutable;
use Drupal\views\Plugin\views\display\DisplayPluginBase;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
* Filter class which filters by the available ModerationStates.
*
* @ViewsFilter("offer_moderation_state_filter")
*/
class ModerationStateFilter extends InOperator {


  /**
   * @var \Drupal\Core\Database\Connection
   */
  protected $database;
```

```
 /**
   * Constructs a Bundle object.
   *
   * @param array $configuration
   *   A configuration array containing information about the plugin
instance.
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   */
 public function __construct(array $configuration, $plugin_id,
$plugin_definition, Connection $database) {
   parent::__construct($configuration, $plugin_id, $plugin_definition);
   $this->database = $database;
 }

 /**
   * {@inheritdoc}
   */
 public static function create(ContainerInterface $container, array
$configuration, $plugin_id, $plugin_definition) {
   return new static(
     $configuration,
     $plugin_id,
     $plugin_definition,
     $container->get('database')
   );
 }

 /**
   * Override the query so that no filtering takes place if the user
doesn't
   * select any options.
   */
 public function query() {

   // Get the selected value first
   $selected = $this->value;

   // If 'all' is selected, do not filter. This would mean all offers
   if(!in_array('all', $selected)) {

     $configuration = [
       'table' => 'content_moderation_state_field_data',
        'field' => 'content_entity_id',
        'left_table' => 'offer',
        'left_field' => 'id',
        'operator' => '='
      ];
      $join =
\Drupal::service('plugin.manager.views.join')->createInstance('standard'
, $configuration);


$this->query->addRelationship('content_moderation_state_field_data',
$join, 'offer');

      $this->query->addWhere('AND',
'content_moderation_state_field_data.moderation_state', $selected,
'IN');

    }
  }

  /**
   * {@inheritdoc}
   */
  public function init(ViewExecutable $view, DisplayPluginBase $display,
array &$options = NULL) {
    parent::init($view, $display, $options);
    $this->valueTitle = t('Moderation state');
    $this->definition['options callback'] = [$this,
'getModerationStates'];
  }

  /**
   * Generates the list of ModerationStates that can be used in the
filter.
   */
  public function getModerationStates() {
    $result = [
      'draft' => 'Draft',
      'published' => 'Published',
      'expired' => 'Expired'
    ];
    return $result;
  }
}
```

Inside *getModerationStates()*, we define the search options. We could dynamically get the available moderation states on an offer entity, but this would take us too far. For the sake of example, we offer the options in an hard-coded array.

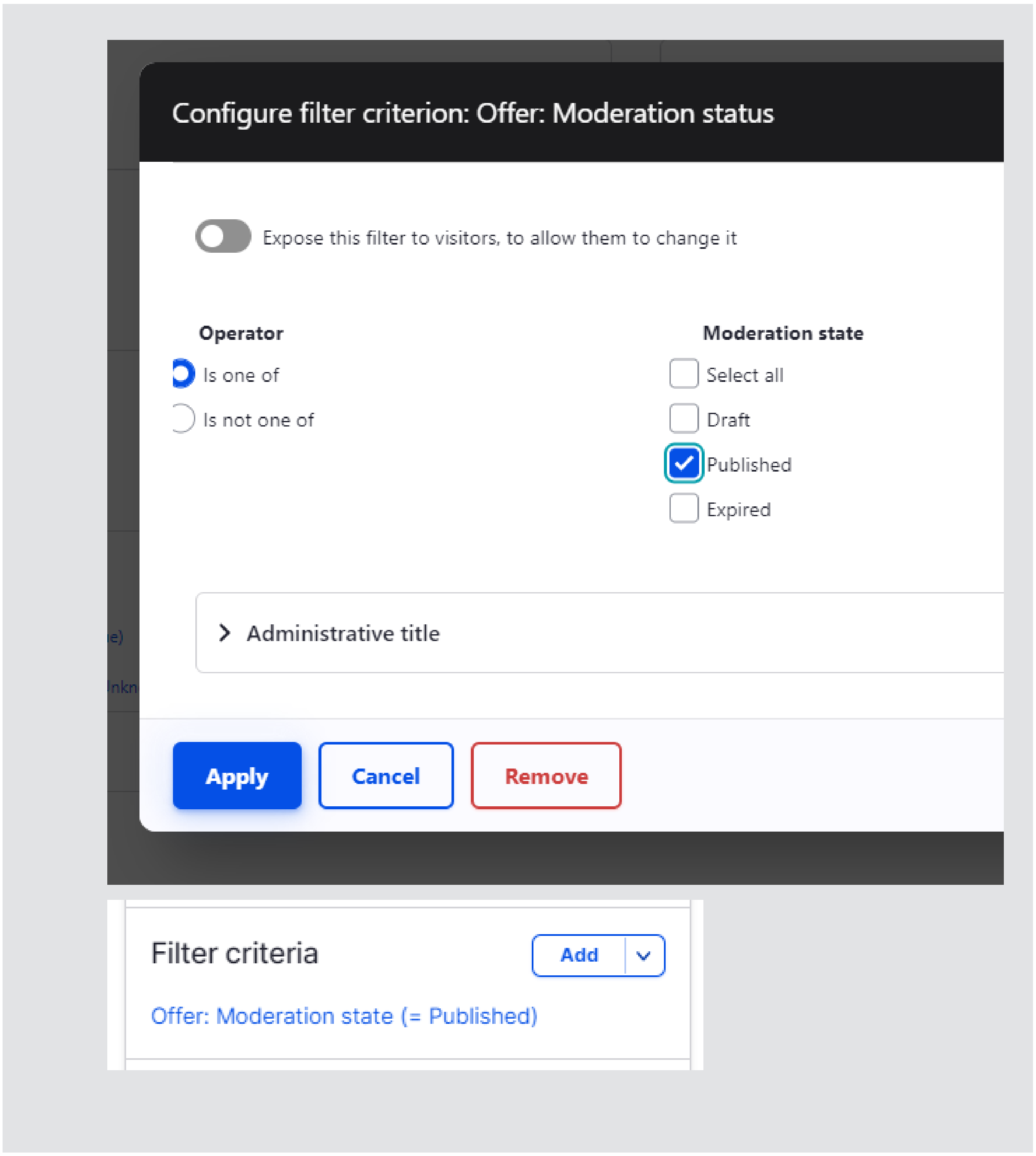
Inside *query()*, we add a relationship to the moderation table and left join it on the offer id.

💻 Add the filter at /admin/structure/views/view/offer_overview.



☐	ID	Offer
☑	Moderation status	Offer

Select ‘published’:


Configure filter criterion: Offer: Moderation status

Expose this filter to visitors, to allow them to change it

Operator
- [✓ Is one of]
- Is not one of

Moderation state
- [ ] Select all
- [ ] Draft
- [x] Published
- [ ] Expired

Filter criteria

Offer: Moderation state (= Published)

One last thing we need to to before saving the view is add a file to **custom/modules/offer/config/schema** named **offer.schema.yml**:

```
views.filter.offer_moderation_state_filter:
 type: views.filter.in_operator
 label: 'Filter for moderation state'
```

Views plugins all need to have their own configuration schema in order for them to work.

Clear caches, save the view and you'll see it works! If you check 'Show SQL query' at admin/structure/views/settings you can see the actual query views does:

```
SELECT offer.id AS id
FROM
{offer} offer
LEFT JOIN {content_moderation_state_field_data}
content_moderation_state_field_data ON offer.id =
content_moderation_state_field_data.content_entity_id
WHERE content_moderation_state_field_data.moderation_state IN
('published')
LIMIT 11 OFFSET 0
```

Before drupal 8, we always used the 'Published' status which is a built-in boolean for being published or not. Here we use states.

An overview of the moderation states vs published status of our offers:

Moderation state	Offer is published?
Draft	No
Published	Yes
Expired	Yes

So while the expired offers *are* published, we do not show them in our view. On a software level we want to keep our expired offers online, for SEO and archiving. Users will not be able to place a bid on expired offers anymore.

Optimize the entity teaser with custom variables for twig

👉 In this chapter we will learn how to optimize **rendered output** of an entity. You will see how we **prepare variables** for output with **twig**. For example, if an offer has no bids yet, show a message. Or show the amount of bids the offer has so far.

First, we prepare some extra variables we want to show in our teasers:

Variable name	Description
promo	Adds a teasing message ‘Be the first!’ if the offer has no bids yet.
price	Based on the type of offer, show the minimum price

We can make the variables available by using the *template_preprocess_offer()* hook in **modules/custom/offer/offer.module**:

```
use Drupal\Core\Render\Element; // on top of file

/**
 * Prepares variables for templates.
 * implements hook_preprocess_HOOK()
 */
function template_preprocess_offer(array &$variables) {
  foreach (Element::children($variables['elements']) as $key) {
    $variables['content'][$key] = $variables['elements'][$key];
  }

  $offer = $variables['elements']['#offer'];
  // The full offer object
  $variables['offer'] = $offer;

  // The current price: minimum or highest bid if available
  switch($offer->get('field_offer_type')->getString()) {
    case 'with_minimum':
      $variables['price'] = 'Start bidding at ' .
$offer->get('field_price')->getString() . '$';
      break;
    case 'no_minimum';
      $variables['price'] = 'Start bidding at 0$';
      break;
  }

  // a promo badge (we'll change this later)
  $variables['promo'] = 'Be the first!';

}
```

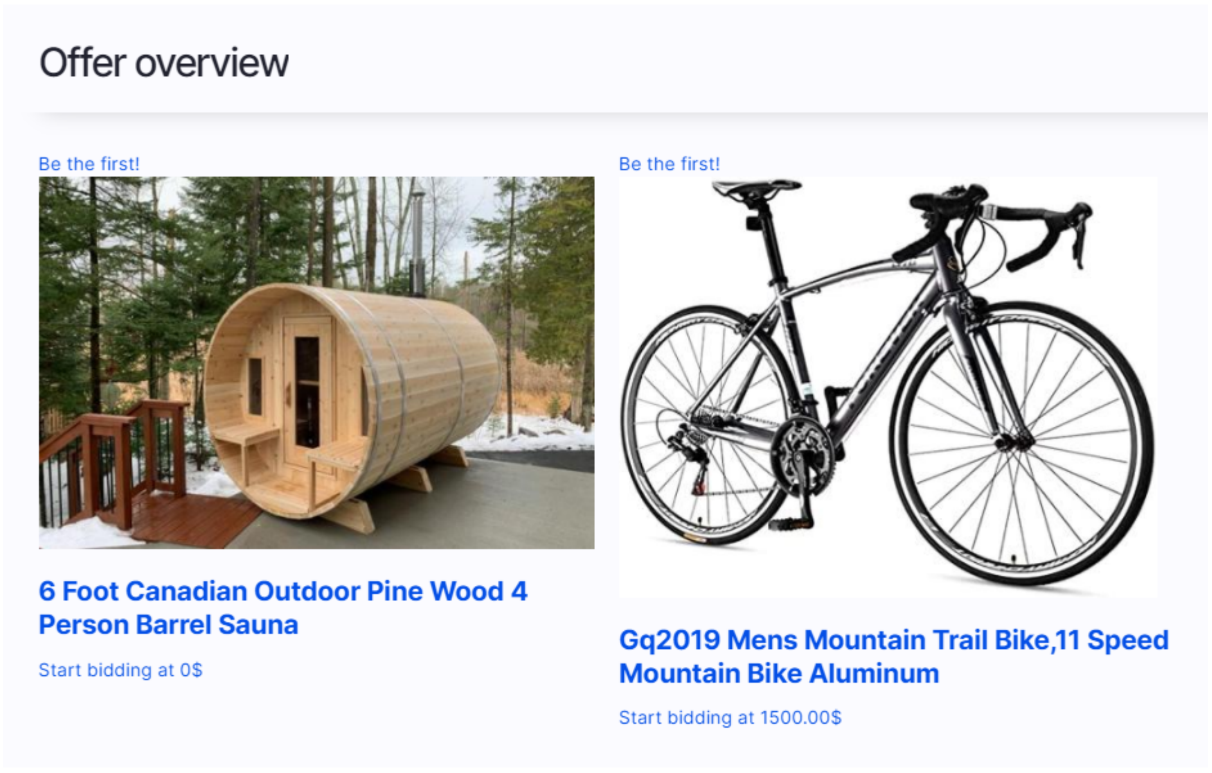
We add a teaser template file **custom/offer/templates/offer--teaser.html.twig:**

```
<a href="{{ path('entity.offer.canonical', {'offer': offer.id()})
}}" {{ attributes.addClass('offer-teaser') }}>
 {% if promo %}<div class="badge">{{promo }}</div>{% endif %}
 <div class="product-tumb">
   {{ content.field_image }}
 </div>
 <div class="product-details">
   <h4>{{ offer.label() }}</h4>
   <p>{{ bid_amount }} bids</p>
   <div class="product-bottom-details">
     <div class="product-price">{{ price }}</div>
   </div>
 </div>
</a>
```

For drupal to be able to recognize this template, extend the *offer_theme()* hook inside **custom/offer/offer.module**, we add the offer__teaser key. We also add the default offer key, and a offer__full for the offer page which we will use later:

```
/**
* Provides a theme definition for custom content entity offer
* {@inheritdoc}
*/
function offer_theme($existing, $type, $theme, $path) {
 return [
   'offer' => [
     'render element' => 'elements',
   ],
   'offer__full' => [
     'base hook' => 'offer',
   ],
   'offer__teaser' => [
     'base hook' => 'offer'
   ]
 ];
}
```

Clear caches and take a view. Drupal now recognizes the teaser file and shows the correct values in the teasers.



Offer overview

Be the first!

6 Foot Canadian Outdoor Pine Wood 4 Person Barrel Sauna

Start bidding at 0$

Be the first!

Gq2019 Mens Mountain Trail Bike,11 Speed Mountain Bike Aluminum

Start bidding at 1500.00$

Head to the next chapter to add some styling to them.

Adding css to views

👉 In this chapter we will learn how to add **a libraries.yml** file with css to be included when rendering a specific view. At the end you will be able to include styles where a certain view is loaded.

I would recommend including the theming in a custom theme, but sometimes it is needed to have custom javascript or css in a module to target only specific components.

Add a file **custom/offer/offer.libraries.yml** file to include a library for the overview page:

```
offer_overview_page:
 css:
   theme:
     css/overview.css: {}
```

To add the css only when the view is showed, we can use the following hook that we add to our **custom/offer/offer.module** file:

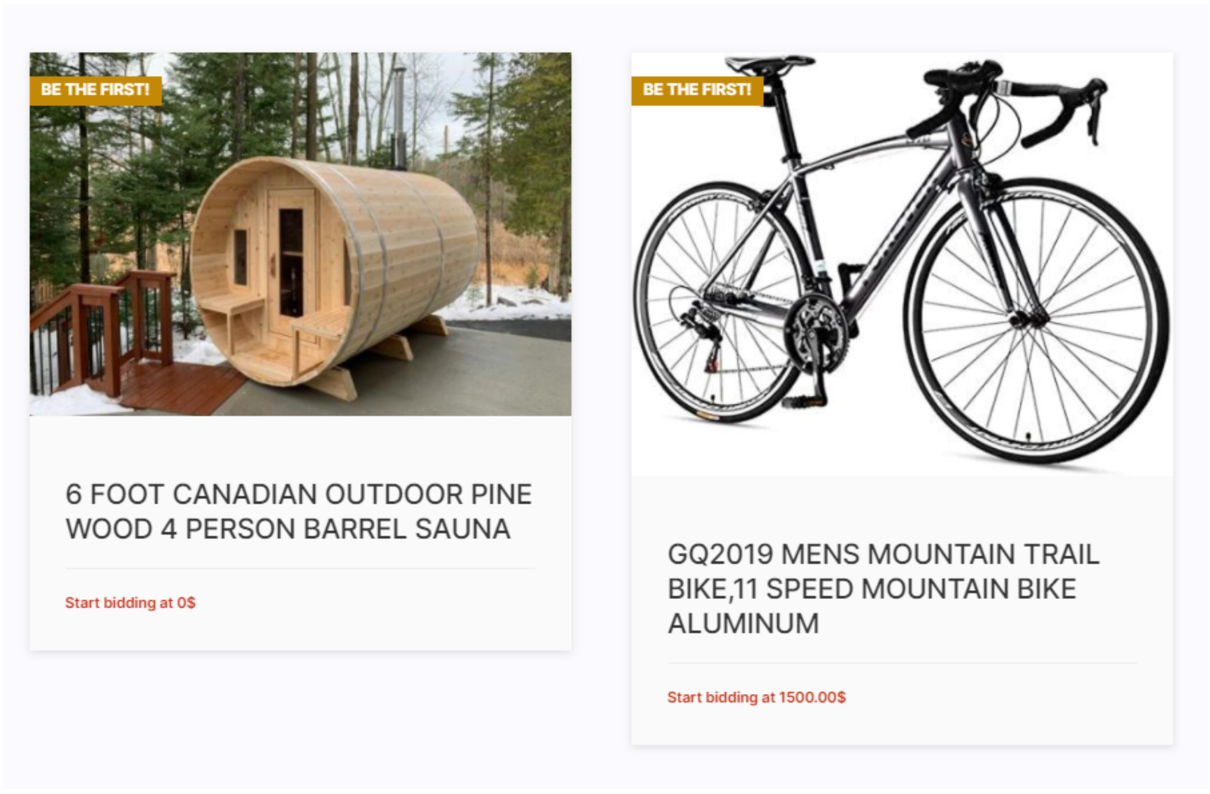
```
use Drupal\views\ViewExecutable; // on top of file

/**
* Implements hook_views_pre_render().
*/
function offer_views_pre_render(ViewExecutable $view) {
 if (isset($view) && ($view->storage->id() == 'offer_overview')) {
   $view->element['#attached']['library'][] =
'offer/offer_overview_page';
 }
}
```

We use *hook_views_pre_render()* to tell the system to always attach the extra library whenever this view is rendered.

We add some css to **offer/css/overview.css** and our teasers are ready!

```
.offer-teaser {
  width: 90%;
  position: relative;
  box-shadow: 0 2px 7px #dfdfdf;
  margin: 50px 0;
  background: #fafafa;
  display:block;
}

.offer-teaser * {
  text-underline:none;
}

.badge {
  position: absolute;
  left: 0;
  top: 20px;
  text-transform: uppercase;
  font-size: 13px;
  font-weight: 700;
  background: #C48904;
  color: #fff;
  padding: 3px 10px;
}

.product-tumb {
  display: flex;
  align-items: center;
  justify-content: center;
  height: auto;
  width: 100%;
}

.product-tumb img {
  max-width: 100%;
  max-height: 100%;
}

.product-details {
  padding: 30px;
}

.product-details h4 {
  font-weight: 500;
  display: block;
  margin-bottom: 18px;
  text-transform: uppercase;
  color: #363636;
  text-decoration: none;
  transition: 0.3s;
}

.product-details p {
  font-size: 15px;
  line-height: 22px;
  margin-bottom: 18px;
  color: #999;
}

.product-bottom-details {
  overflow: hidden;
  border-top: 1px solid #eee;
  padding-top: 20px;
}

.product-price {
  font-size: 12px;
  color: #C43218;
  font-weight: 600;
}
```

Clear all caches and let's have a look:



6 FOOT CANADIAN OUTDOOR PINE WOOD 4 PERSON BARREL SAUNA

Start bidding at 0$

GQ2019 MENS MOUNTAIN TRAIL BIKE,11 SPEED MOUNTAIN BIKE ALUMINUM

Start bidding at 1500.00$

This looks nice and appealing. Later in this course we will make the variables more dynamic. Let's move forward!

Sidenote here: if the use case is to show the teasers on other pages also, it would be better to attach the css to the teasers and not the overview. In general, you should style your view modes in a way that they look the same everywhere on the platform.

Adding a user-friendly multistep form for entity creation

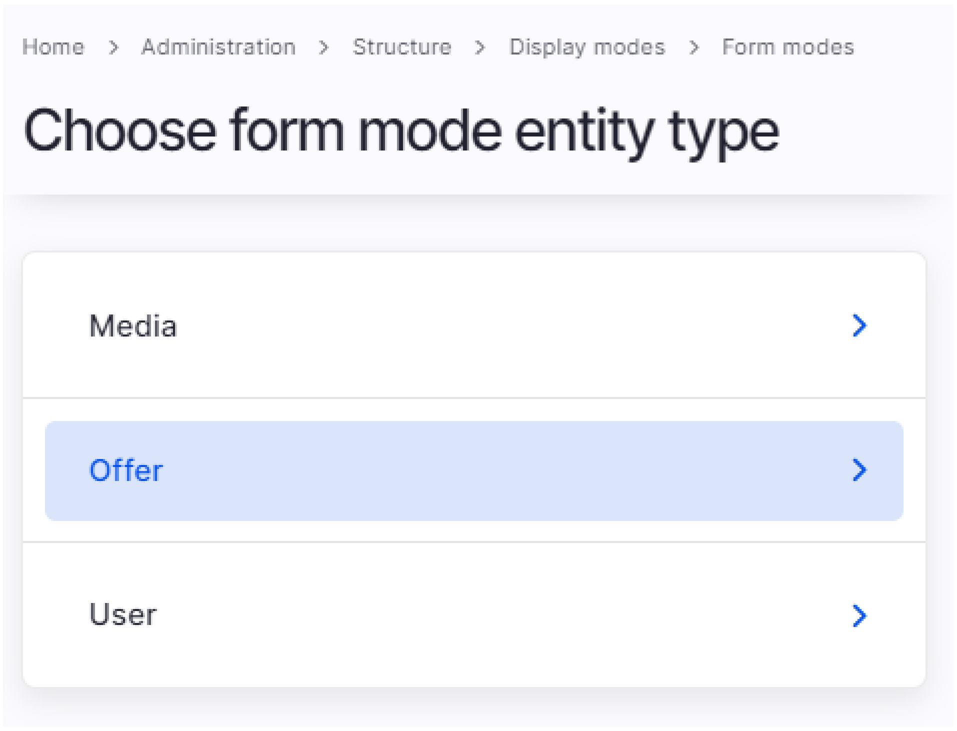
👉 In this chapter you will learn how to create **form modes** for your data entry forms. More specifically we will show how to add a **multistep form** for creating offers in a user-friendly way.

If we head over to *offers/create* we see the default form that can be configured under admin/structure/offer/form-display.

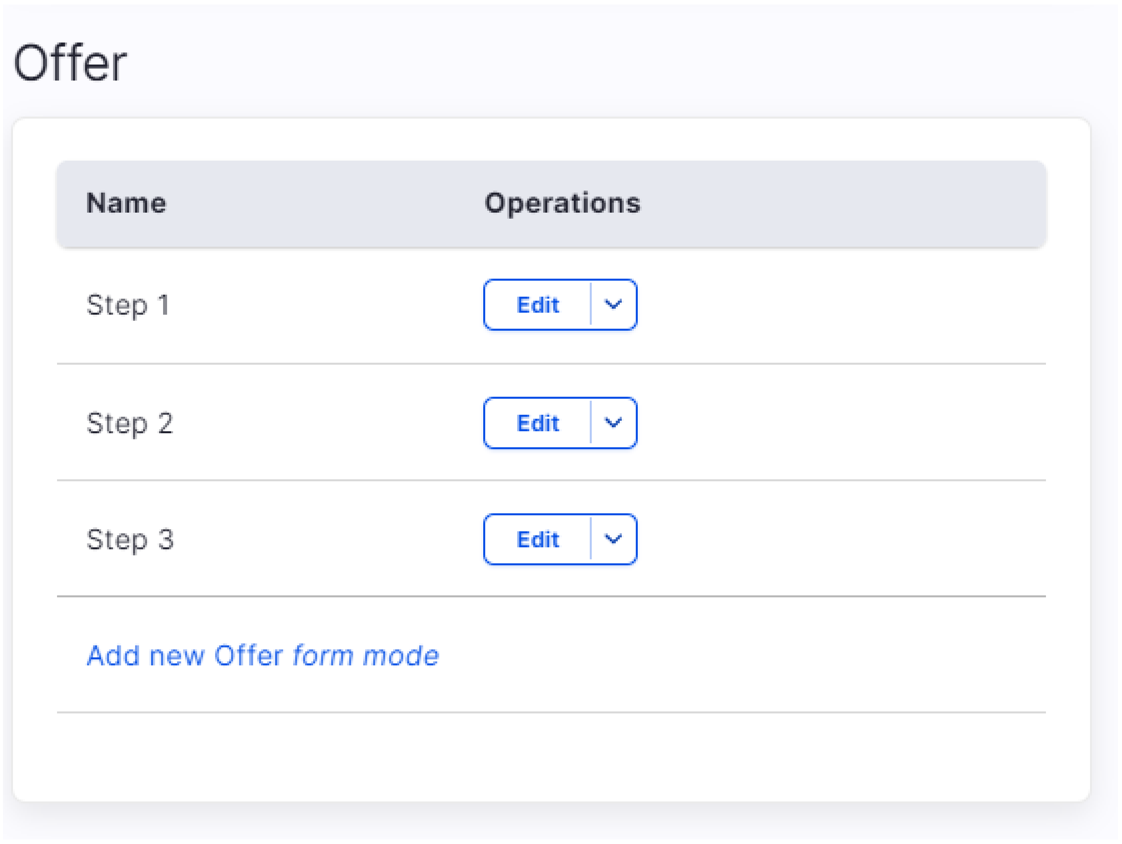
But we want a more user-friendly creation form. This is the flow we want:

- **Step 1:** Add the title of the offer
- **Step 2**: Add a description and an image
- **Step 3:** Add the type of offer

That's where Form Modes come into play. Go to /admin/structure/display-modes/form and add a new one for Offer:


Home > Administration > Structure > Display modes > Form modes
Choose form mode entity type
Media
Offer
User

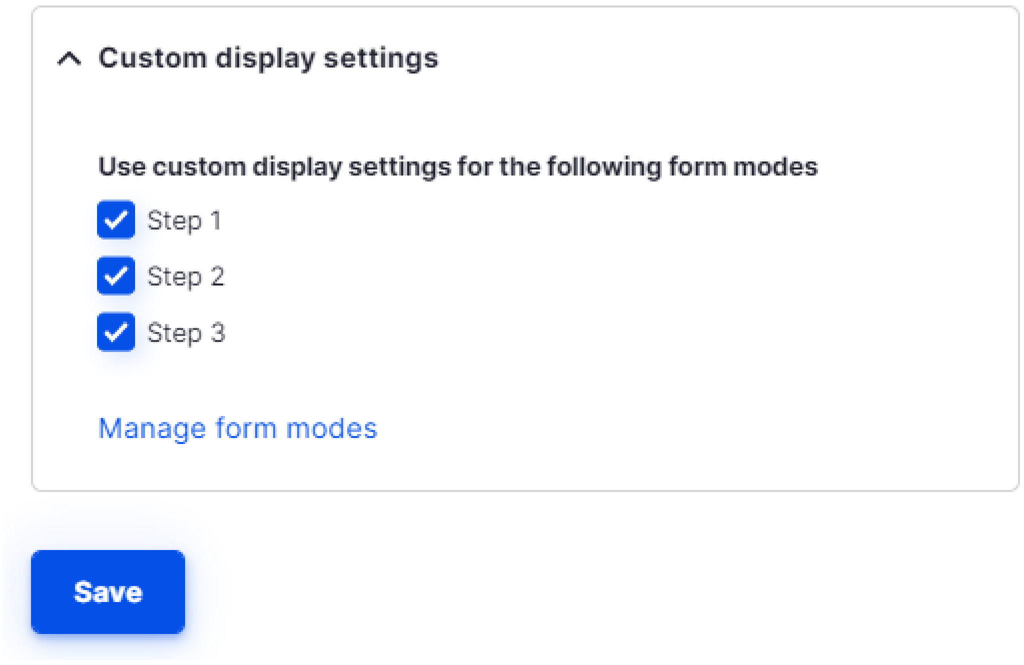
Add **step_1 (Step 1), step_2 (Step 2), step_3 (Step 3)**:


Offer

Name	Operations
Step 1	Edit
Step 2	Edit
Step 3	Edit

Add new Offer *form mode*

At /admin/structure/offer/form-display go to “Custom display settings” and enable all three:


Custom display settings

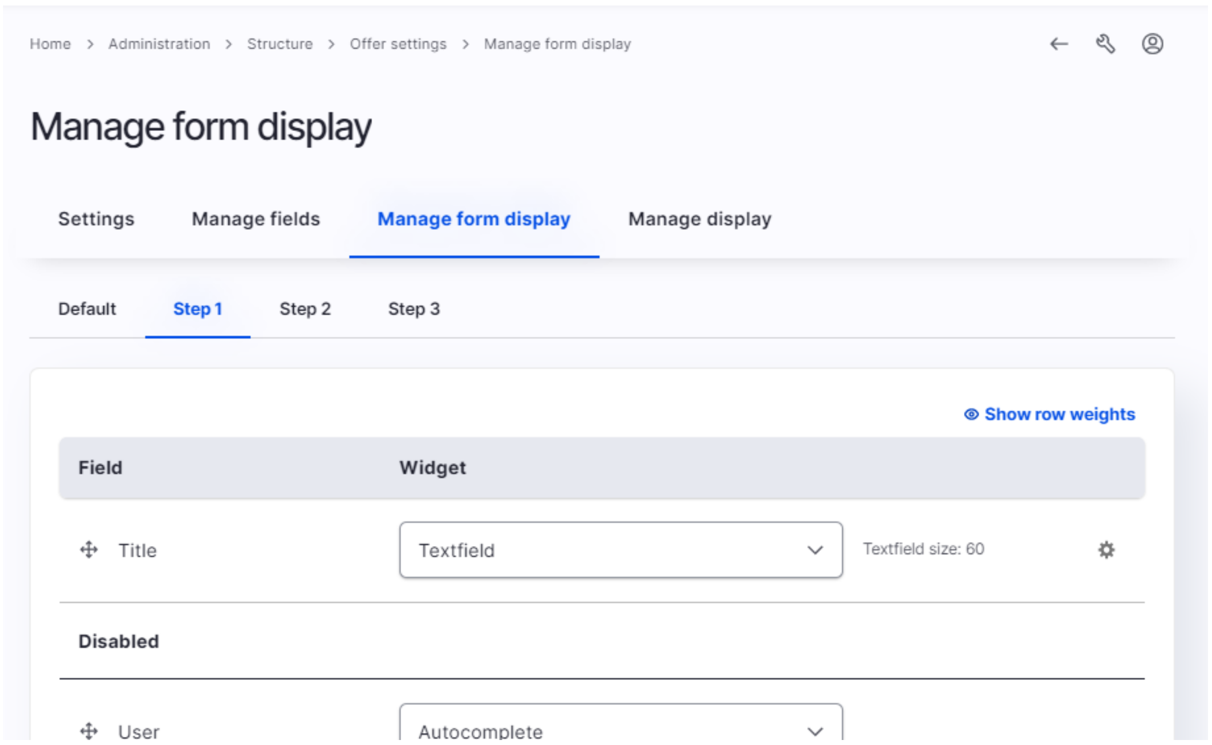
Use custom display settings for the following form modes

- [✓] Step 1
- [✓] Step 2
- [✓] Step 3

Manage form modes

Save

Next, we drag the desired fields to each of the steps. In the case of “Step 1”, this is only the title field.


Home > Administration > Structure > Offer settings > Manage form display

Manage form display

Field	Widget		
Title	Textfield	Textfield size: 60	
Disabled			
User	Autocomplete		

Do the same for step 2 and step 3.
We have our new form modes, but this will initially do nothing. First, we have to "inform" our custom entity that there are three new form modes available. Let's add them to the offer entity right after our default form:

```
 *   handlers = {
*     "access" = "...",
*     "views_data" = "...",
*     "form" = {
*       "default" = "Drupal\offer\Form\OfferForm",
*       "step_1" = "Drupal\offer\Form\OfferAddFormStep1",
*       "step_2" = "Drupal\offer\Form\OfferAddFormStep2",
*       "step_3" = "Drupal\offer\Form\OfferAddFormStep3",
*       "edit" = "Drupal\offer\Form\OfferForm",
```

Clear caches to see the result.

We've informed drupal that there are now new forms for our entity. **Make sure your keys in the annotations ("step_1", "step_2", …) are the same as the machine name of your custom form modes.**

Now **copy** the content entity form under **Form\Offerform.php to Form\OfferAddFormStep1, Form\OfferAddFormStep2.php, and Form\OfferAddFormStep3.php**. This way you copied the default entity form to the steps. This way we can customize them later.

💻 When copying the **Offerform.php** to **OfferAddFormStep1**, make sure you also change the Class name in the file itself!

Time to make new routes for our multi-step pages. To **custom/offer/offer.routing.yml**:

```
offer.add:
 path: '/offers/create'
 defaults:
   _entity_form: offer.step_1
   _title: 'Step 1: set your title'
 requirements:
   _entity_create_access: 'offer'

offer.step1:
 path: '/offers/create/{offer}'
 defaults:
   _entity_form: offer.step_1
   _title: 'Step 1: set your title'
 requirements:
   _entity_create_access: 'offer'

offer.step2:
 path: '/offers/create/step2/{offer}'
 defaults:
   _entity_form: offer.step_2
   _title: 'Step 2: add a description and image'
 requirements:
   _entity_access: 'offer.edit'

offer.step3:
 path: '/offers/create/step3/{offer}'
 defaults:
   _entity_form: offer.step_3
   _title: 'Step 3: choose the type of offer'
 requirements:
    _entity_access: 'offer.edit'
```

The entity forms remain basically the same (although they have different form modes), but the difference is in the *Save()* function. We change it in **src\Form\OfferAddFormStep1.php** like this:

```
public function save(array $form, FormStateInterface $form_state) {
    // Redirect to step 2.
    $entity = $this->getEntity();
    $entity->save();
    $id = $entity->id();
    $form_state->setRedirect('offer.step2', ['offer' => $id]);
  }
```

Step 1 is where the entity gets saved on submission. We use it to redirect to step 2 via entity parameter upcasting.

We do the same in step 2, to redirect to step 3. Change it in **src\Form\OfferAddFormStep2.php** to this:

```
public function save(array $form, FormStateInterface $form_state) {
 // Redirect step 3 after save.
 $entity = $this->getEntity();
 $entity->save();
 $id = $entity->id();
 $form_state->setRedirect('offer.step3', ['offer' => $id]);
}
```

In our final step, **src\Form\OfferAddFormStep3.php**, we add redirect to the overview and add a message:

```
/**
* {@inheritdoc}
*/
public function save(array $form, FormStateInterface $form_state) {
 // Redirect to offer overview after save.
 $form_state->setRedirect('entity.offer.collection');
 \Drupal::messenger()->addMessage('Your offer was created. Click the
publish button to start earning.');
 $entity = $this->getEntity();
 $entity->save();
}
```

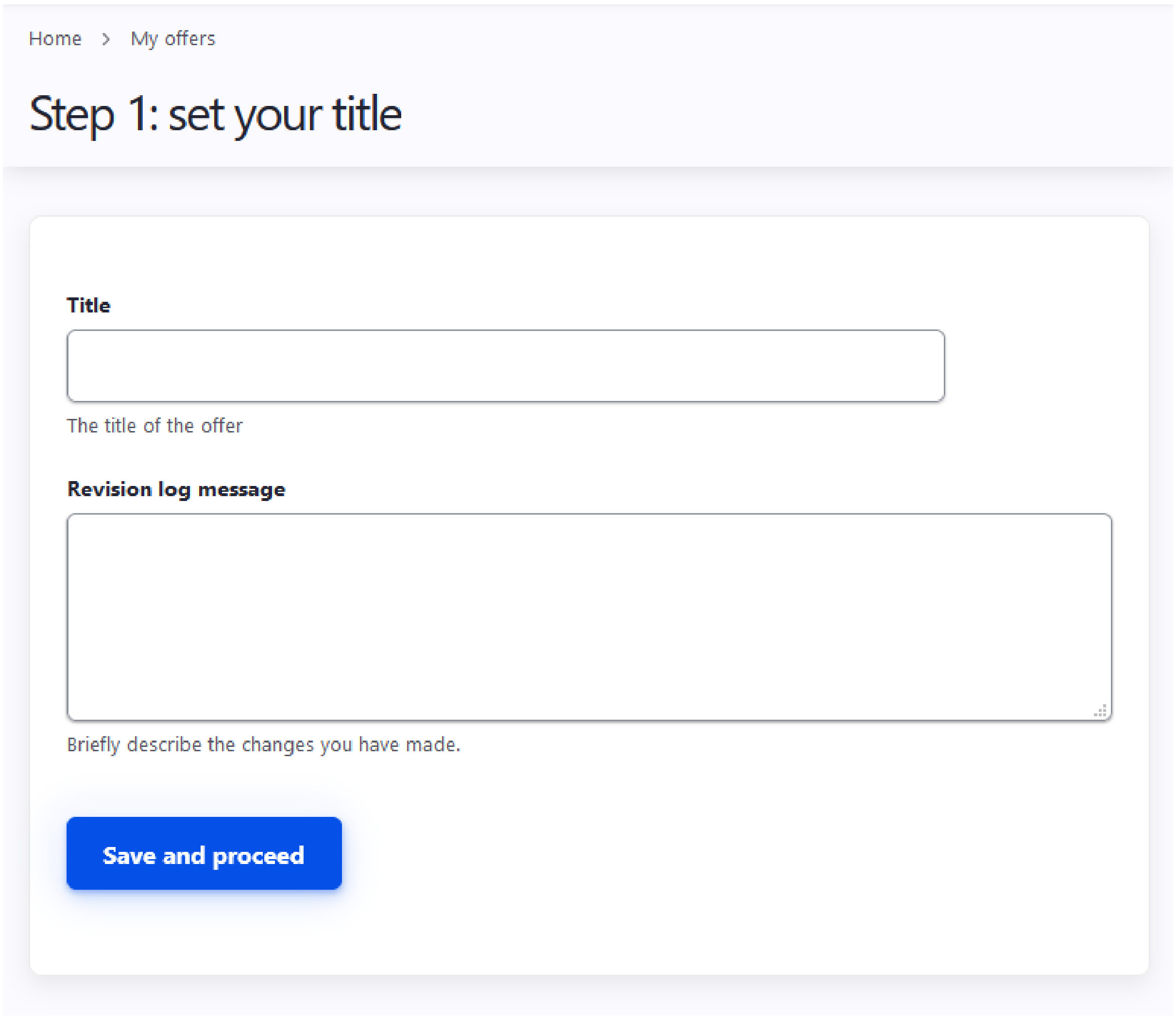
In the *buildForm()* function of each of the forms, you can tweak some more. I changed the submit button text like this:

```php
public function buildForm(array $form, FormStateInterface
$form_state) {
 /* @var $entity \Drupal\offer\Entity\Offer */
 $form = parent::buildForm($form, $form_state);
 $form['actions']['submit']['#value'] = t('Save and proceed');
 return $form;
}
```

We're done! We've created an advanced multi-step form, which saves our entity on each step. Let's enhance the process a bit for further user experience.

💻 Best way to test is to create a test user at admin/people who is just an authenticated user. This way you can check better if all permissions are set correctly.

Go to offers/create and have a look:



Home > My offers

Step 1: set your title

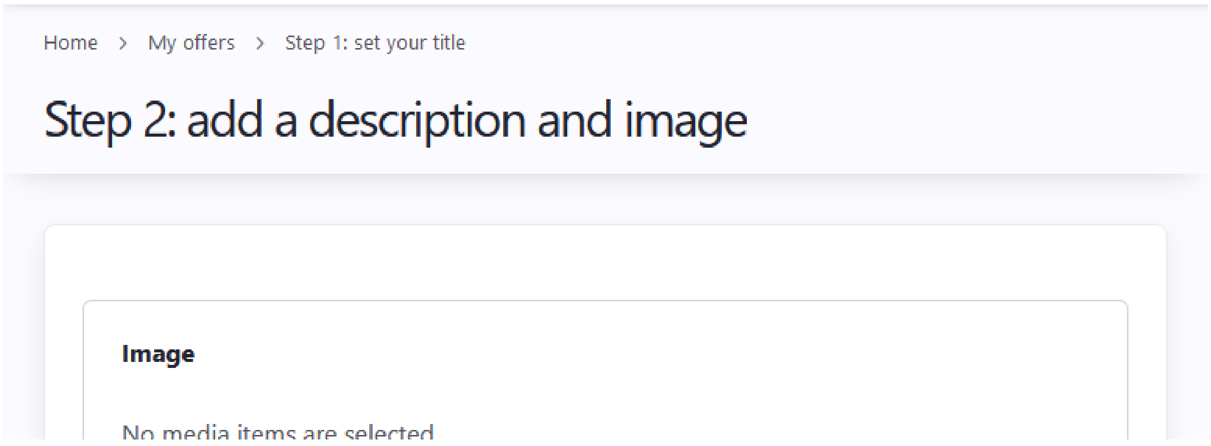
Title

The title of the offer

Revision log message

Briefly describe the changes you have made.

If we click *Save and proceed* we go to the next step:



Home > My offers > Step 1: set your title

Step 2: add a description and image

Image

No media items are selected

💻 There is still a field “Revision log” visible. This is because the entity is revisionable. But we do not want to use it here. We make it invisible with this snippet. Clear caches and you’ll see it is gone.

```
use Drupal\Core\Form\FormStateInterface; // on top of file

/**
 * Implements hook_form_alter().
 */
function offer_form_alter(array &$form, FormStateInterface
$form_state, $form_id) {
  $forms = ['offer_step_1_form', 'offer_step_2_form',
'offer_step_3_form', 'offer_edit_form'];
  if (in_array($form_id, $forms)) {
    // Prevent revision log box access
    $form['revision_log']['#access'] = FALSE;
  }
}
```

Add custom actions to the form

👉 In this chapter you will learn how to add additional form buttons (actions) like a cancel button or a “Go back to step 1” redirect on entity forms.

Actions are the typical buttons drupal groups below a form. Because we use multiple steps we want to add functionality to be able to navigate inside the form as well.

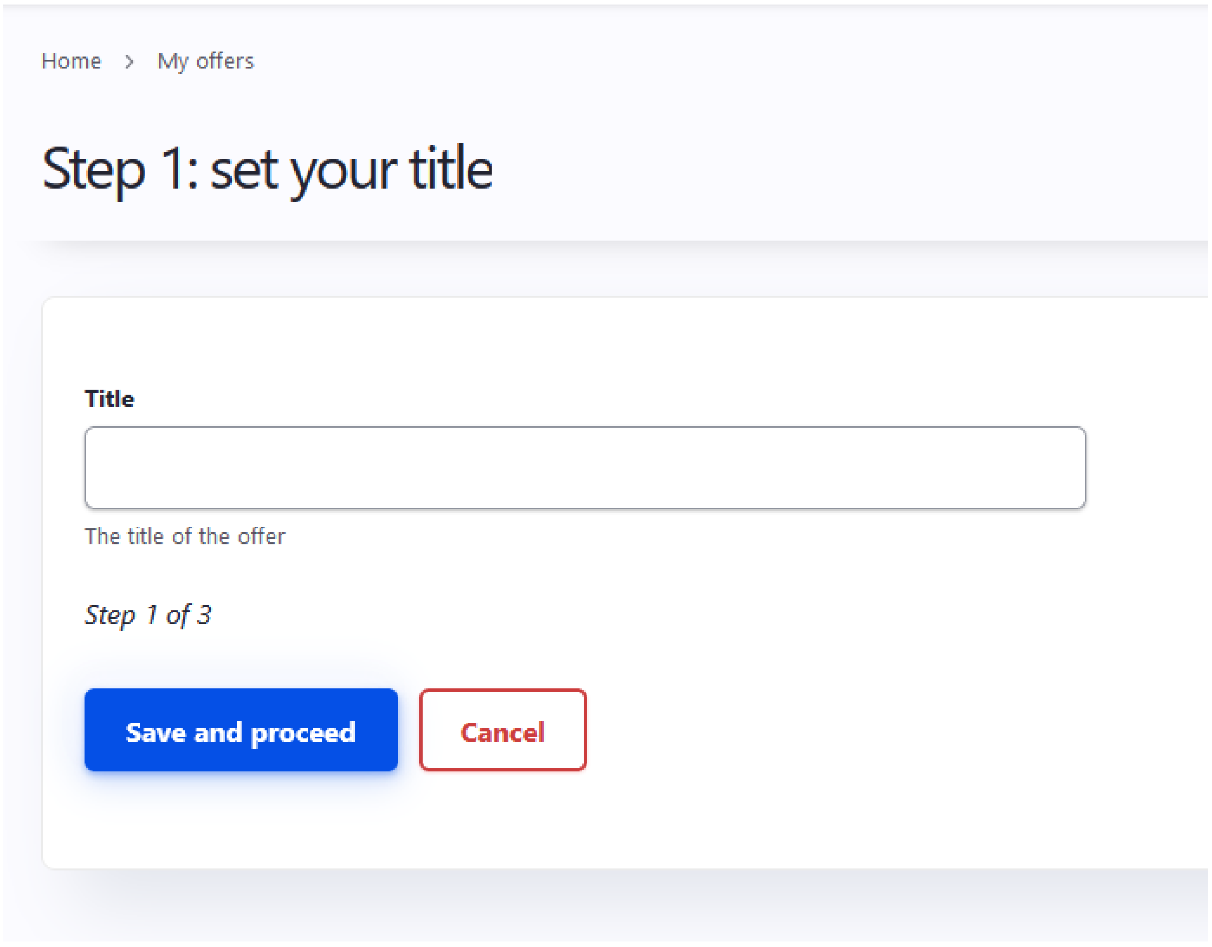
In our first step, we’d like to provide a ‘**Cancel**’ button next to the ‘**Save and proceed**’ button. Also, remove the ‘delete’ button.

To **src\Form\OfferAddFormStep1.php** we add:

```
protected function actions(array $form, FormStateInterface $form_state)
{
 $actions =  parent::actions($form, $form_state);
 $actions['cancel'] = [
   '#type' => 'submit',
   '#value' => $this->t('Cancel'),
   '#submit' => ['::cancelSubmit'],
   '#weight' => 90,
   '#limit_validation_errors' => []
 ];
 if (array_key_exists('delete', $actions)) {
   unset($actions['delete']);
 }
 $actions['#prefix'] = '<i>Step 1 of 3</i>';

 return $actions;
}

public function cancelSubmit(array $form, FormStateInterface
$form_state) {
 $form_state->setRedirect('entity.offer.collection');
}
```

This is Object Oriented code in practice. The actions method allows us to extend and add an extra action. In the callback we specify a redirect to cancel the operation.

This is what **step 1** now looks like:


Home > My offers

Step 1: set your title

Title

The title of the offer

Step 1 of 3

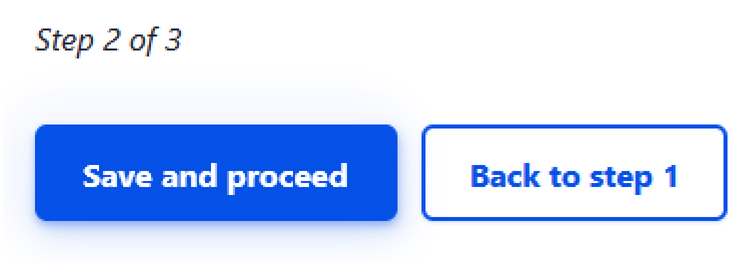
To **src\Form\OfferAddFormStep2.php** we add:

```
protected function actions(array $form, FormStateInterface
$form_state) {
 $actions =  parent::actions($form, $form_state);
 $actions['go_back'] = [
   '#type' => 'submit',
   '#value' => $this->t('Back to step 1'),
   '#submit' => ['::goBack'],
   '#weight' => 90,
   '#limit_validation_errors' => []
 ];
 if (array_key_exists('delete', $actions)) {
   unset($actions['delete']);
 }
 $actions['#prefix'] = '<i>Step 2 of 3</i>';
 return $actions;
}
```

```
public function goBack(array $form, FormStateInterface
$form_state) {
 $entity = $this->getEntity();
 $id = $entity->id();
 $form_state->setRedirect('offer.step1', ['offer' => $id]);
}
```

You'll understand what I did here. The *goBack()* function will get called when clicking the button '**Back to step 1**'. We'll get redirected to */offer/create/{offer}*.

This is what the actions in **step 2** now look like:



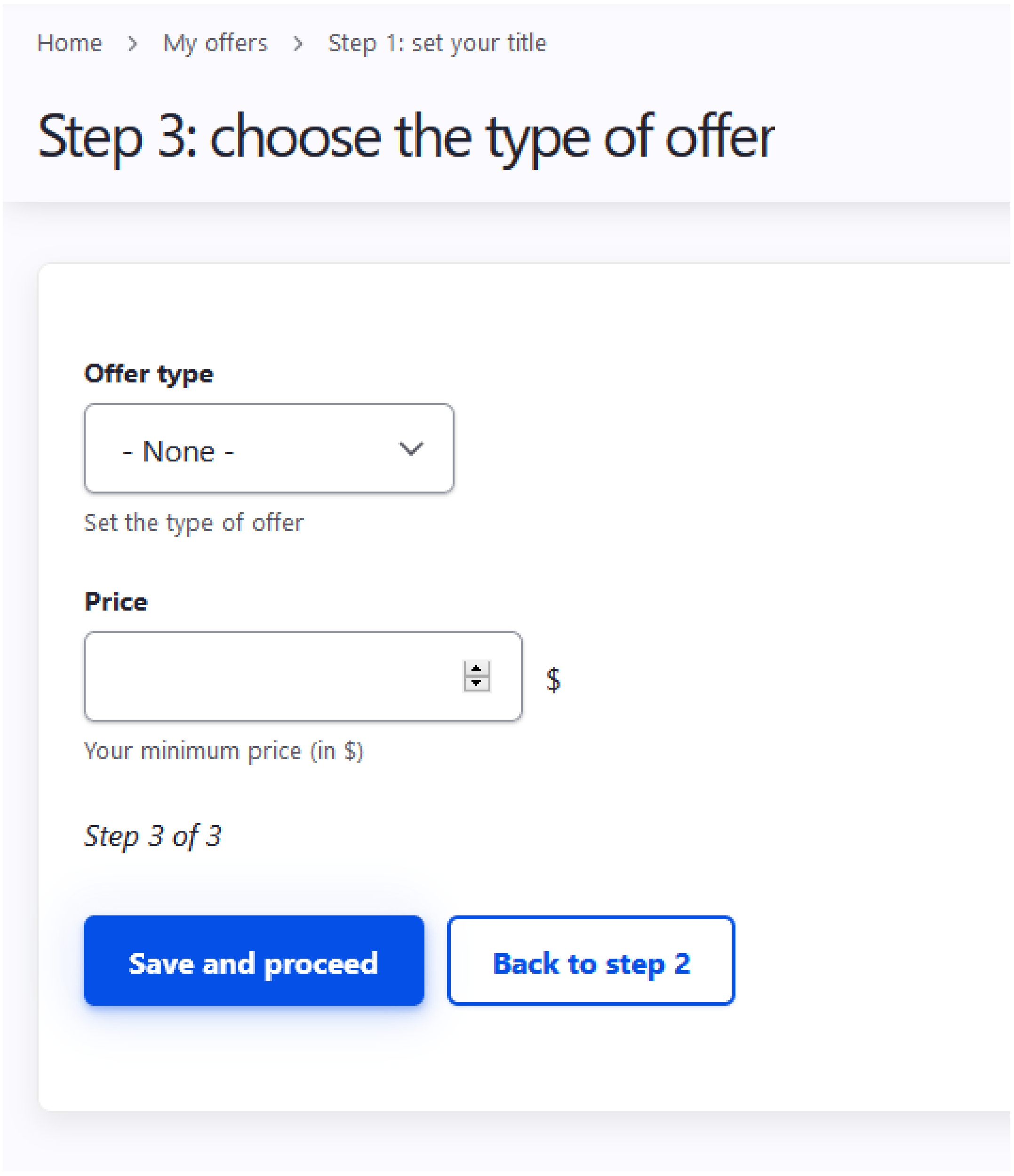
Finally, we optimize our step three in the same way. To **src\Form\OfferAddFormStep3.php** we add:

```
protected function actions(array $form, FormStateInterface $form_state)
{
 $actions =  parent::actions($form, $form_state);
 $actions['go_back'] = [
   '#type' => 'submit',
   '#value' => $this->t('Back to step 2'),
   '#submit' => ['::goBack'],
   '#weight' => 90,
   '#limit_validation_errors' => []
 ];
 if (array_key_exists('delete', $actions)) {
   unset($actions['delete']);
 }
 $actions['#prefix'] = '<i>Step 3 of 3</i>';
 return $actions;
}

public function goBack(array $form, FormStateInterface $form_state) {
 $entity = $this->getEntity();
 $id = $entity->id();
 $form_state->setRedirect('offer.step2', ['offer' => $id]);
}
```

Because this is the last step in the form, we redirect to our overview and add a message via the Messenger API Drupal provides us:

```
public function save(array $form, FormStateInterface $form_state)
{
 // Redirect to offer overview after save.
 $form_state->setRedirect('entity.offer.collection');
 \Drupal::messenger()->addMessage('Your offer was created. Click
the publish button to start earning.');
 $entity = $this->getEntity();
 $entity->save();
}
```

Step three looks like this now:


Home > My offers > Step 1: set your title

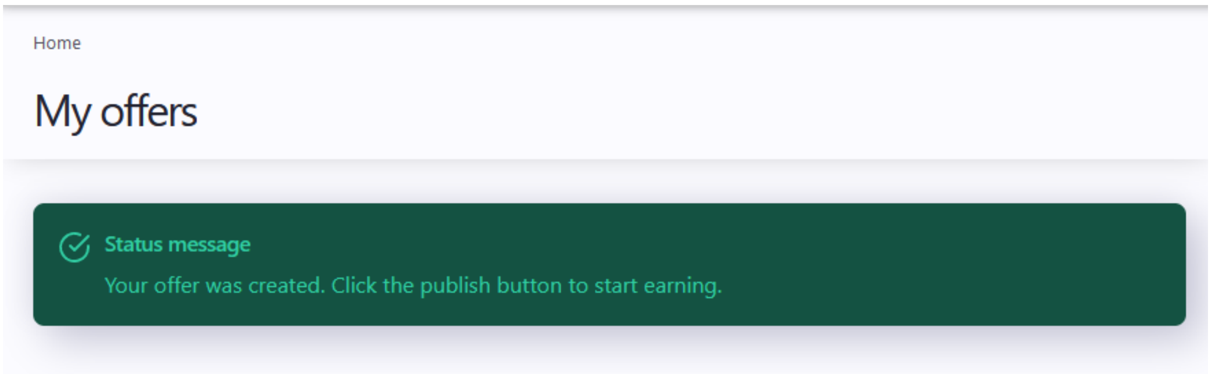
Step 3: choose the type of offer

Set the type of offer

Your minimum price (in $)

Step 3 of 3

After we finished our offer, we get redirected to the My offers page:


Home

My offers

Status message

Your offer was created. Click the publish button to start earning.

The last step (step 3) is not yet how we want it. We want to show the Price field only if the Offer type is set to ‘Set minimum price’ and hidden when the Offer type is set to ‘No minimum price’. We’ll do this in the next chapter.

Conditional fields in the Form API

👉 In this chapter we show how to use **conditional fields** with the core **form API**. By the end of this chapter you will know how to show/hide fields based on user input on another field.

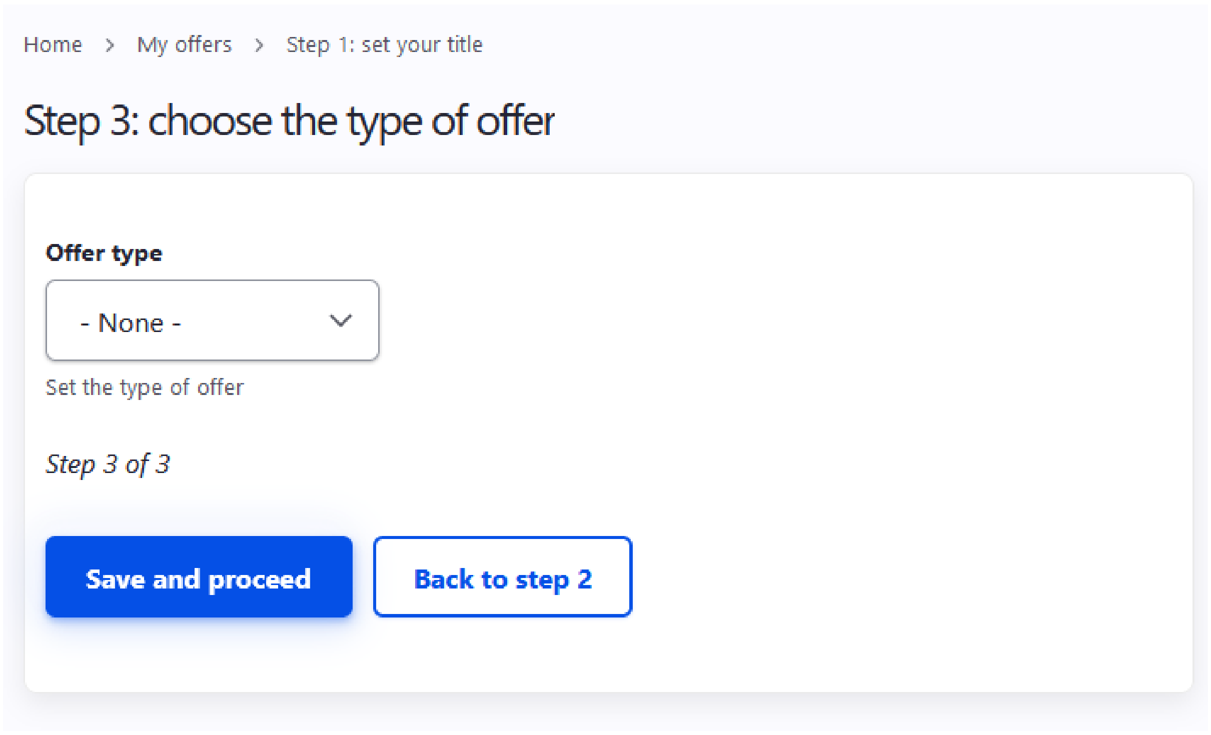
While there were quite a lot of new things introduced last year, one of the most stable things is the Form API. It grew out to be one of the best API’s drupal has. In our step 3, we want to have our price field only visible when users select ‘Set minimum price’.

In **src\Form\OfferAddFormStep3.php** tweak the buildForm function so that it becomes this:

```
public function buildForm(array $form, FormStateInterface $form_state) {
  /* @var $entity \Drupal\offer\Entity\Offer */
  $form = parent::buildForm($form, $form_state);
  $form['actions']['submit']['#value'] = t('Save and proceed');
  unset($form['actions']['delete']);

  $form['field_price']['#states'] = [
    'visible' => [
      ['select[name="field_offer_type"]' => ['value' => 'with_minimum']],
    ]
  ];

  return $form;
}
```

This is actually quite readable. **WHEN** the offer type has the selection ‘with_minimum’ (this is the key of ‘Set minimum price’) selected, **THEN** make the price field visible.

In action:


Home > My offers > Step 1: set your title

Step 3: choose the type of offer

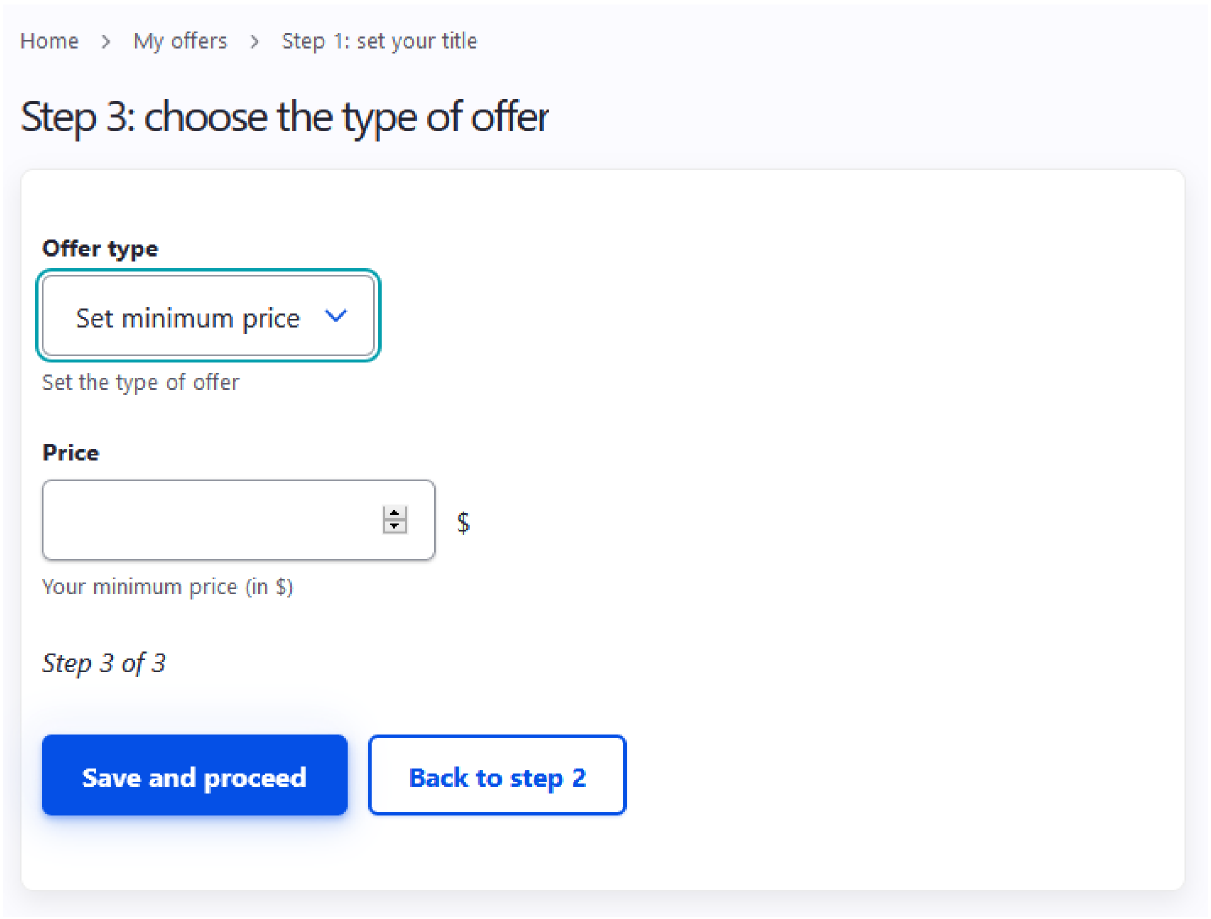
Offer type

- None -

Set the type of offer

Step 3 of 3

When selecting ‘Set a minimum price’:


Home > My offers > Step 1: set your title

Step 3: choose the type of offer

Offer type

Set minimum price

Set the type of offer

Price

$

Your minimum price (in $)

Step 3 of 3

This is it! Our multi step form is completely done.

Updating our data seeds

Remember from the first part of this course that we created some seed data. This was meant for using real dummy data on our platform. Now that our offer entity is ready, we can extend our import with offers.

To **offer/src/SeedData/SeedDataGenerator.php** we add a function with an array of offers. We'll add one below for example:

```
public function getOfferList() {
    $data = [
      [
        'title' => 'Gq2019 Mens Mountain Trail Bike,11 Speed
Mountain Bike Aluminum',
        'field_description' => '
            <ul>
              <li>Photochromic mountain bike, the use of aluminum
alloy frame, hydraulic disc brake system, wheel diameter size:
27.5 inches ...</li>
            </ul>
          ',
        'field_price' => 1500,
        'field_offer_type' => 'with_minimum',
        'field_image' =>
'https://images-na.ssl-images-amazon.com/images/I/61-HR1eqFuL._AC_
SL1001_.jpg',
        'moderation_state' => 'published',
        'user_id' => 1
      ],
    ];
    return $data;
  }
```

We'll use this to loop over our offer entities to create. We add this to the switch statement under 'user' in de *generate()* method:

```
case 'offer':
 $offers = $this->getOfferList();
 foreach($offers as $offerListItem) {
   $offer = Offer::create();
   $offer->set('title', $offerListItem['title']);
   $offer->set('field_description', ['value' =>
$offerListItem['field_description'], 'format' => 'html']);
   $offer->set('field_offer_type',
$offerListItem['field_offer_type']);
   $offer->set('field_price', $offerListItem['field_price']);
   $directory = 'public://';
   $url = $offerListItem['field_image'];
   $file = system_retrieve_file($url, $directory, true);
   $drupalMedia = Media::create([
     'bundle' => 'image',
     'uid' => '0',
     'field_media_image' => [
       'target_id' => $file->id(),
     ],
   ]);
   $drupalMedia->setPublished(TRUE)
     ->save();
   $offer->set('field_image', ['target_id' =>
$drupalMedia->id()]);
   $offer->set('user_id', 1);
   $offer->set('moderation_state',
$offerListItem['moderation_state']);
   if($offer->save()) {
     $count += 1;
   }
 }
 return $count;
 break;
```

Then, we add the following lines to our *OfferCreateSeeds()* method in **custom/offer/src/Commands/SeedGeneratorCommand.php:**

```
$count = $seed->Generate('offer');
Drush::output()->writeln($count . ' offer(s) created');
```

In the final project code, you’ll see I’ve ended up with multiple offers that get created this way. Now developers who install the platform locally can work with real-life offers instead of an empty installation.

Part 4: building the application. Storing, validating and rendering data

Add dynamic menu links with Menu plugins

👉 In this chapter we go over **custom menu links** defined in **code**. At the end you will be able to create dynamic menu links with counters such as “My offers (2)”.

If we log in with a user who is not the administrator we lack navigation. Drupal has a built-in menu system which content managers can use to add content to menus. But because this is more a framework course than a CMS course we want to add these in code. A nice advantage is that you can cache these items individually and cleverly. More about this in the chapter about caching. A second advantage is you can add *variable* data such as counters.

To **modules/custom/offer/offer.links.menu.yml**, we add:

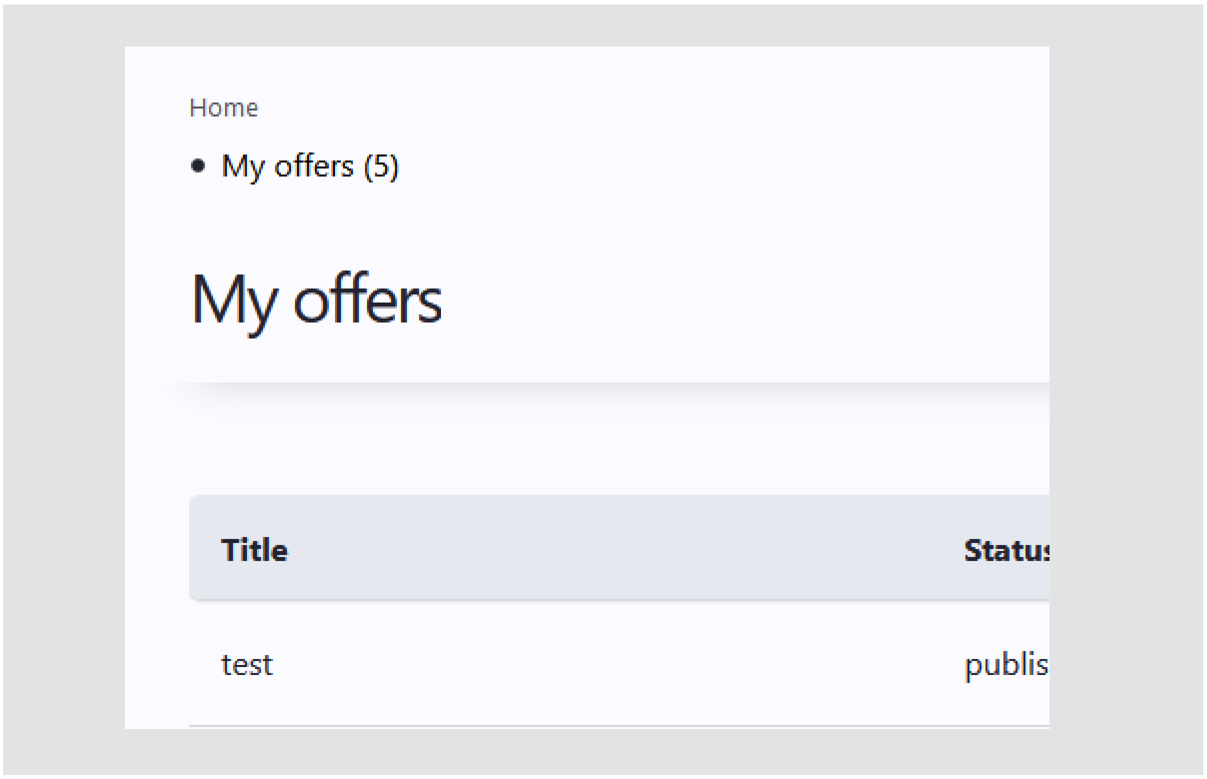
```
offer.account.my_offers:
  route_name: entity.offer.collection
  menu_name: main
  class: Drupal\offer\Plugin\Menu\MyOffers
  weight: -50
```

We need to add a Menu Plugin to our module so the Menu module can detect our menu link. Add **modules/custom/offer/Plugin/Menu/MyOffers.php:**

```
<?php
namespace Drupal\offer\Plugin\Menu;

use Drupal\Core\Menu\MenuLinkDefault;

/**
* displays number of offers.
*/
class MyOffers extends MenuLinkDefault {

  /**
   * {@inheritdoc}
   */
  public function getTitle() {
    $count = 0;
    if(\Drupal::currentUser()->isAuthenticated()) {
      $offers = \Drupal::entityTypeManager()
        ->getStorage('offer')
        ->loadByProperties(['user_id' => \Drupal::currentUser()->id()]);
      $count = count($offers);
      return $this->t('My offers (<span
class="count-badge">(@count)</span>)', ['@count' => $count]);
    } else {
      return null;
    }
  }

  /**
   * {@inheritdoc}
   */
  public function getCacheMaxAge() {
    return 0;
  }

}
```

💻 Add the **main navigation** block to the ‘breadcrumb’ region at */admin/structure/blocks*, we see our dynamic link to the offers page, with a number counter to show the amount of offers we’ve created so far.

If we clear the caches, we see the ‘My offers’ menu link appearing on top.



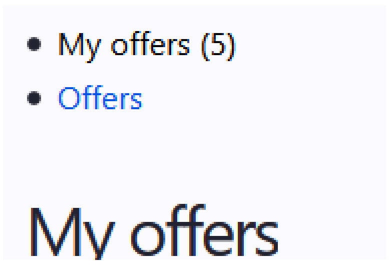
The link is not cached as it needs to show an updated value after adding a new offer. See the chapter about caching for details on how to provide custom caching mechanisms for this. We will style this main menu in a further chapter.

Next, we add a link to the overview page (to the main menu) and the notifications (to the account menu):

To **custom/offer/offer.links.menu.yml**

```yaml
offer.overview:
 title: 'Offers'
 route_name: view.offer_overview.page_1
 menu_name: main
 weight: 5
```

The second link is visible after a cache clear:



Building the offer page with twig: theming a custom content entity

Now it's time for us to start building the offer page. We'll use twig files, because this is the standard in drupal 9. I would recommend not to install contributed modules for styling as they may cause troubles in the long run (upgrading?).

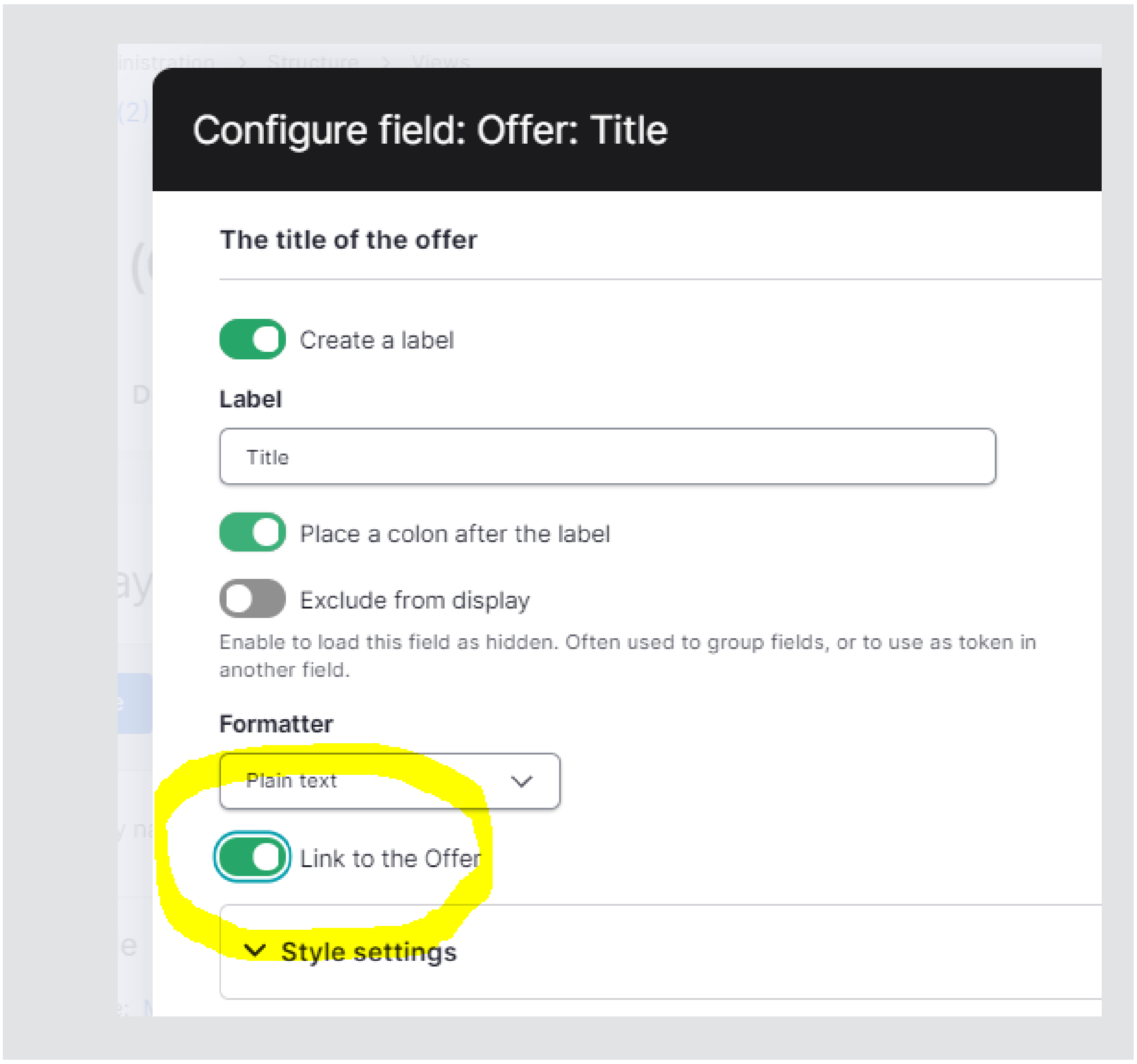
If you followed this course, thanks to the instruction in the rendering a teaser the full template should already be registered in our **custom/offer/offer.module:**

```
/**
* Provides a theme definition for custom content entity offer
* {@inheritdoc}
*/
function offer_theme($existing, $type, $theme, $path) {
 return [
   'offer' => [
     'render element' => 'elements',
   ],
   'offer__full' => [
     'base hook' => 'offer',
   ],
   'offer__teaser' => [
     'base hook' => 'offer'
   ]
 ];
}
```

First, we want to theme the full view, for that we clear caches and add a file to **custom/offer/templates** named **offer--full.html.twig**. To the file, add:

```
{{ content }}
```

💻 In the 'fields'-section of your offers view, at /admin/structure/views/view/offers, make sure you've clicked the "Link to Offer" checkbox. This way you will be able to visit your entity detail page from the /offers page.



Configure field: Offer: Title

The title of the offer

Create a label: on

Label: Title

Place a colon after the label: on

Exclude from display: off

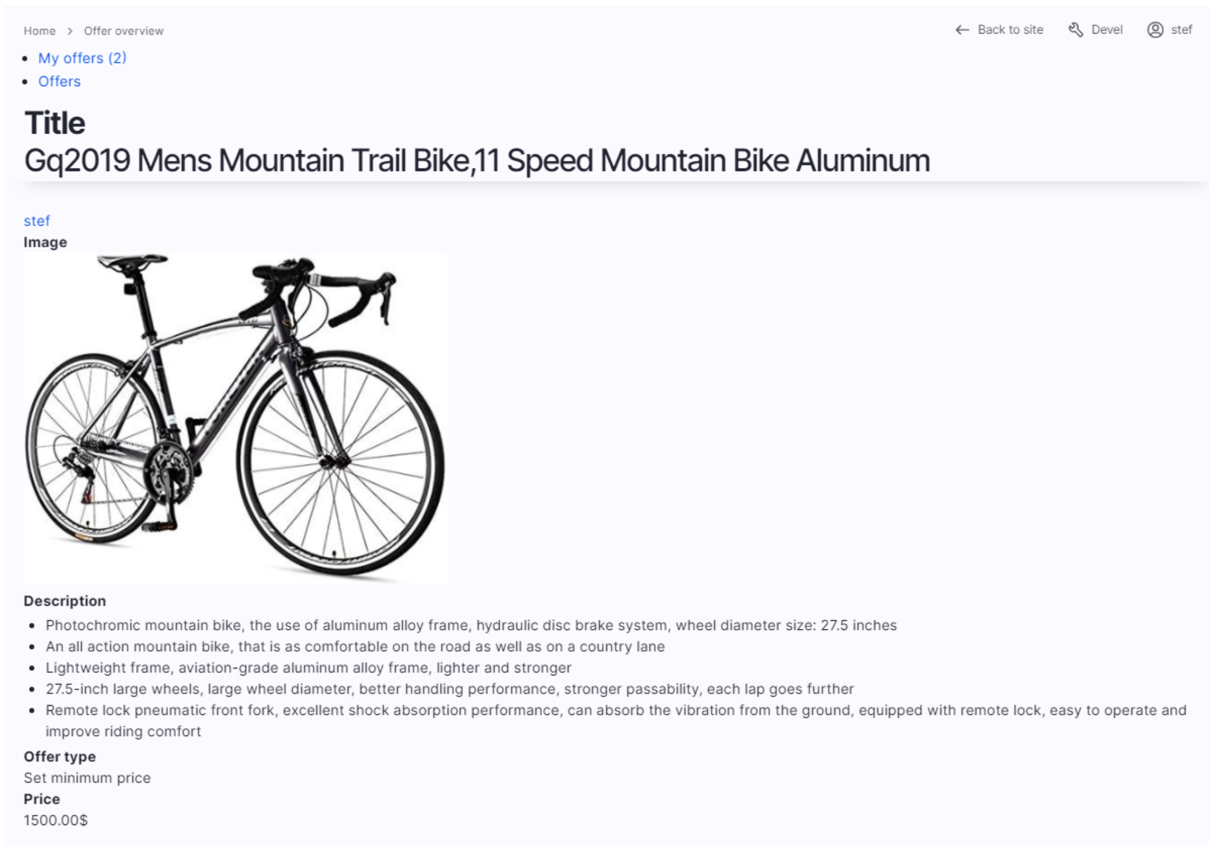
Enable to load this field as hidden. Often used to group fields, or to use as token in another field.

Formatter: Plain text

Link to the Offer: on

Style settings

Clear caches to make sure this template is rendered. Add some dummy sentence before the content statement to test it.



- My offers (2)
- Offers

Title
Gq2019 Mens Mountain Trail Bike,11 Speed Mountain Bike Aluminum

stef
Image

Description
- Photochromic mountain bike, the use of aluminum alloy frame, hydraulic disc brake system, wheel diameter size: 27.5 inches
- An all action mountain bike, that is as comfortable on the road as well as on a country lane
- Lightweight frame, aviation-grade aluminum alloy frame, lighter and stronger
- 27.5-inch large wheels, large wheel diameter, better handling performance, stronger passability, each lap goes further
- Remote lock pneumatic front fork, excellent shock absorption performance, can absorb the vibration from the ground, equipped with remote lock, easy to operate and improve riding comfort

Offer type
Set minimum price
Price
1500.00$

We can play along with the fields at admin/structure/offer/display/full to change the order of the fields and hiding or showing the label. But I prefer to be more specific when using twig, so I edit my **offer--full.html.twig** like this:

```
{{ content.field_image }}
{{ content.field_description }}
```

This will only show the image and the description.

Adding a dynamic bidding form to our page with an advanced block plugin

We want a nice and dynamic bidding form on our offer pages. We'll start by adding a form to **custom/modules/offer/src/Form** named **OfferBiddingForm.php**:

```
<?php
namespace Drupal\offer\Form;

use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\offer\Entity\Offer;
```

```
class OfferBiddingForm extends FormBase {

  /**
   * @return string
   *   The unique string identifying the form.
   */
  public function getFormId() {
    return 'offer_bid_form';
  }

  /**
   * Form constructor.
   *
   * @param array $form
   *   An associative array containing the structure of the form.
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *   The current state of the form.
   * @param \Drupal\offer\Entity\Offer $offer
   *   The offer entity we're viewing
   *
   * @return array
   *   The form structure.
   */
  public function buildForm(array $form, FormStateInterface $form_state,
$offer = NULL) {

    $form['bid'] = [
      '#type' => 'textfield',
      '#attributes' => array(
        ' type' => 'number', // this validates it as a number in
front-end
      ),
      '#title' => $this->t('Your bid'),
      '#description' => $this->t('Prices in $.'),
      '#required' => TRUE,
    ];

    // Group submit handlers in an actions element with a key of
"actions".
    $form['actions'] = [
      '#type' => 'actions',
    ];

    // Add a submit button that handles the submission of the form.
    $form['actions']['submit'] = [
      '#type' => 'submit',
      '#value' => $this->t('Submit'),
    ];

    return $form;

  }
}
```

We add the *validateForm()* and *submitForm()* functions but keep them empty for now:

```
public function validateForm(array &$form, FormStateInterface
$form_state) {
  parent::validateForm($form, $form_state);
}
```

```
public function submitForm(array &$form, FormStateInterface $form_state)
{
}
```

How do we show the form on the entity page? This is an important architectural decision. Our **entities are cached as a whole**, so we need to keep this out of our entity. A **block plugin** is the wisest decision here.

We'll use an advanced technique here. Because the form in the block needs to be dependent on the current offer, we will use the RequestStack service.

Add a file called **OfferBiddingFormBlock.php** to **custom/offer/src/Plugin/Block**:

```
<?php

namespace Drupal\offer\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Symfony\Component\HttpFoundation\RequestStack;
use Drupal\Core\Entity\EntityStorageInterface;
use Drupal\Core\Session\AccountProxyInterface;
```

```
/**
* @Block(
*   id = "offer_bidding_block",
*   admin_label = @Translation("Offer bid block"),
*   category = @Translation("Shows the bidding block to an offer"),
* )
*/

class OfferBiddingFormBlock extends BlockBase implements
ContainerFactoryPluginInterface {

 /**
  * @var $account \Drupal\Core\Session\AccountProxyInterface
  */
 protected $account;

 /**
  * The request object.
  *
  * @var \Symfony\Component\HttpFoundation\RequestStack
  */
 protected $requestStack;

 /**
  * The entity storage.
  *
  * @var \Drupal\Core\Entity\EntityStorageInterface
  */
 protected $entityStorage;

 /**
  * Constructs a new OfferBiddingBlock instance.
  *
  * @param string $plugin_id
  *   The plugin_id for the plugin instance.
  * @param mixed $plugin_definition
  *   The plugin implementation definition.
  * @param \Symfony\Component\HttpFoundation\RequestStack $request_stack
  *   The request stack object.
  * @param \Drupal\Core\Entity\EntityStorageInterface $entity_storage
  *   The entity storage.
  */
 public function __construct(array $configuration, $plugin_id,
$plugin_definition, RequestStack $request_stack, EntityStorageInterface
$entity_storage, AccountProxyInterface $account) {
   parent::__construct($configuration, $plugin_id, $plugin_definition);

   $this->requestStack = $request_stack;
   $this->entityStorage = $entity_storage;
   $this->account = $account;
 }

 /**
  * {@inheritdoc}
  */
 public static function create(ContainerInterface $container, array
$configuration, $plugin_id, $plugin_definition) {
   return new static(
     $configuration,
     $plugin_id,
     $plugin_definition,
     $container->get('request_stack'),
     $container->get('entity_type.manager')->getStorage('offer'),
     $container->get('current_user')
   );
 }

 /**
  * {@inheritdoc}
  */
 public function build() {
   // Make sure this is an offer page
   $offer = $this->requestStack->getCurrentRequest()->get('offer');
   if(!$offer) {
     return null;
   }
   // Make sure the current user is not owner of the offer
   if($this->account->id() == $offer->getOwnerId()) {
     return null;
   }

   $form =
\Drupal::formBuilder()->getForm('\Drupal\offer\Form\OfferBiddingForm',
$offer);
   return $form;
 }

 /**
  * A form for authenticated users never gets cached.
  */
 public function getCacheMaxAge() {
   return 0;
 }

}
```

A lot happens here. This is actually the best practice way to get the current entity loaded in your block. A fine piece of Object oriented programming. The more you dive in it, the more comfortable you will be with it. One advantage of it is that via the *create(), construct()* and *build()* methods you will receive objects from memory and not need to load again objects with some direct functions like:

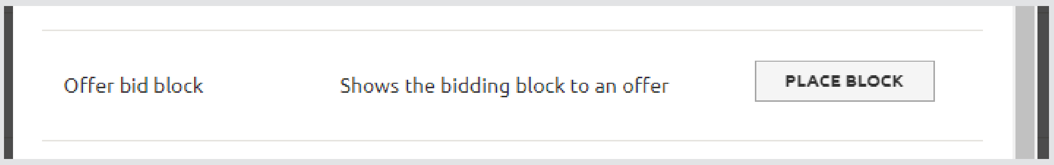
```
$offerId = \Drupal::routeMatch()->getParameter('offer');
$offer = Offer::load($offerId);
```

We pass the full offer entity object as a parameter to the form, it gets picked up by our *buildForm* method inside this piece:

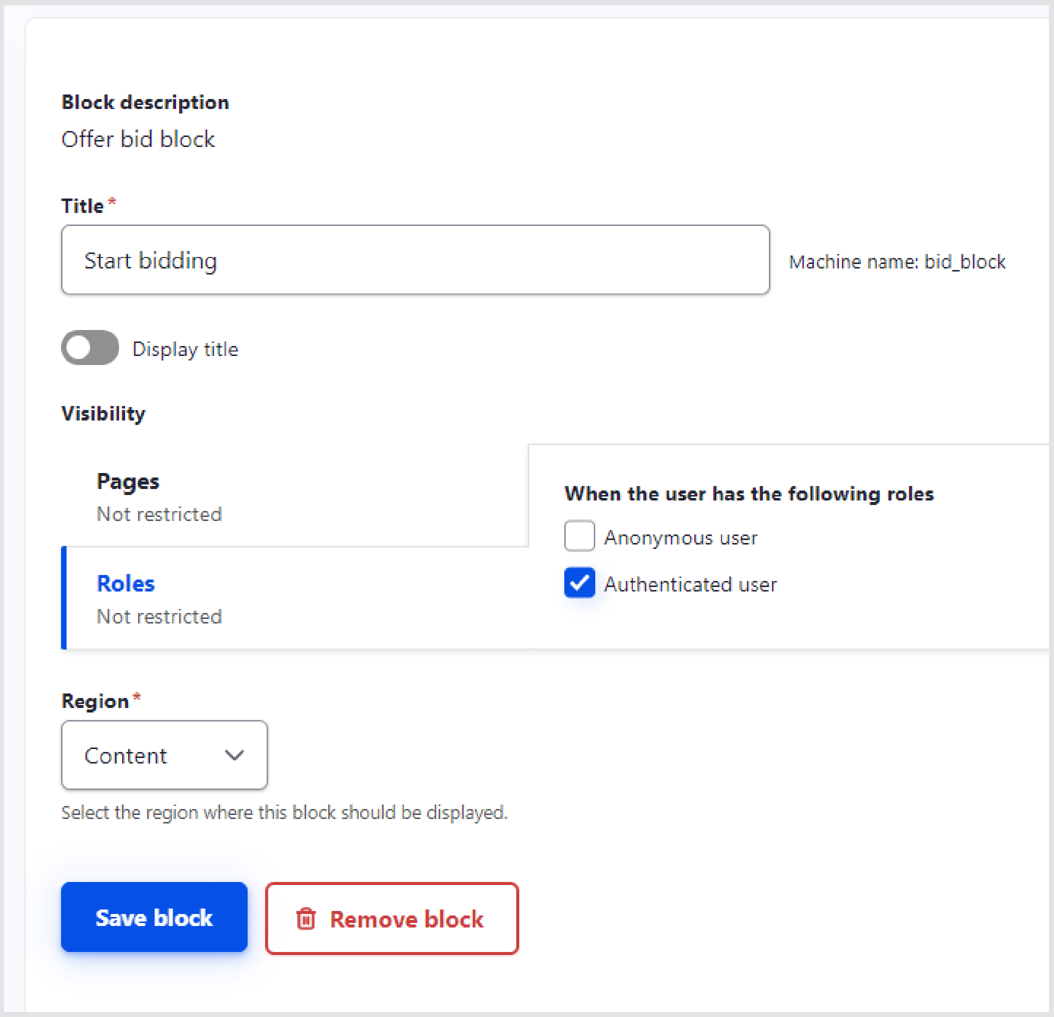
```
public function buildForm(array $form, FormStateInterface
$form_state, $offer = NULL)
```

Also check out the *getCacheMaxAge()* which is set to 0. This means the block will never be cached. More about this in the chapter about caching.

💻 Clear caches and add the block to the content region (below the main content) at admin/structure/blocks.


Offer bid block Shows the bidding block to an offer

Make sure you check this block will be only available for authenticated users:



Block description
Offer bid block

Title*
Start bidding
Machine name: bid_block

Display title

Visibility

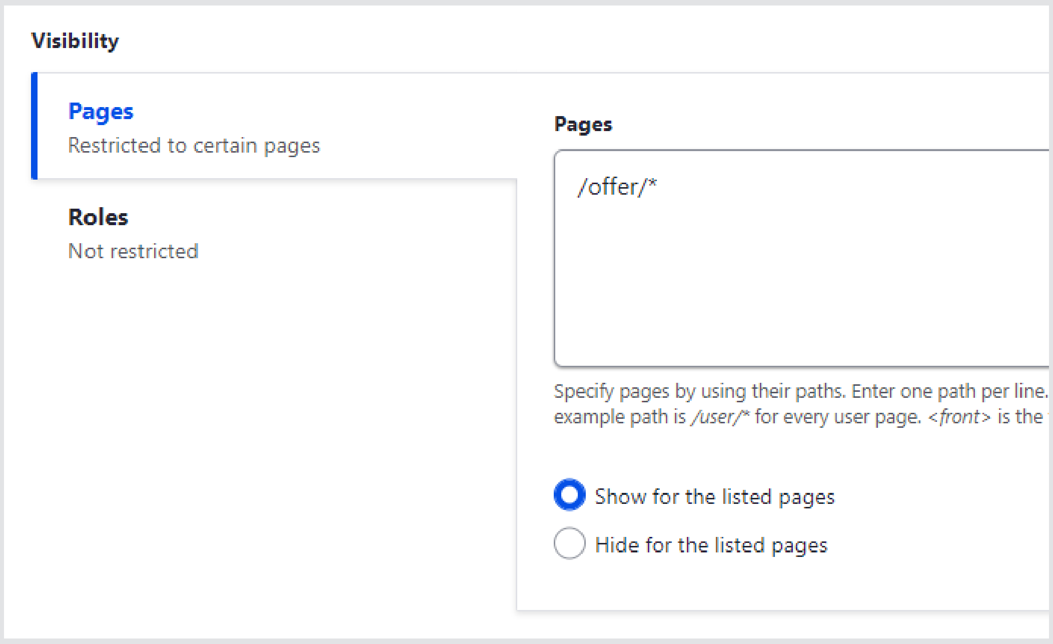
Pages
Not restricted

Roles
Not restricted

When the user has the following roles
☐ Anonymous user
☑ Authenticated user

Region*
Content
Select the region where this block should be displayed.

And restrict it to these pages only: /offer/*. This means the block will only be visible when we are on an offer page.



Visibility

Pages
Restricted to certain pages

Roles
Not restricted

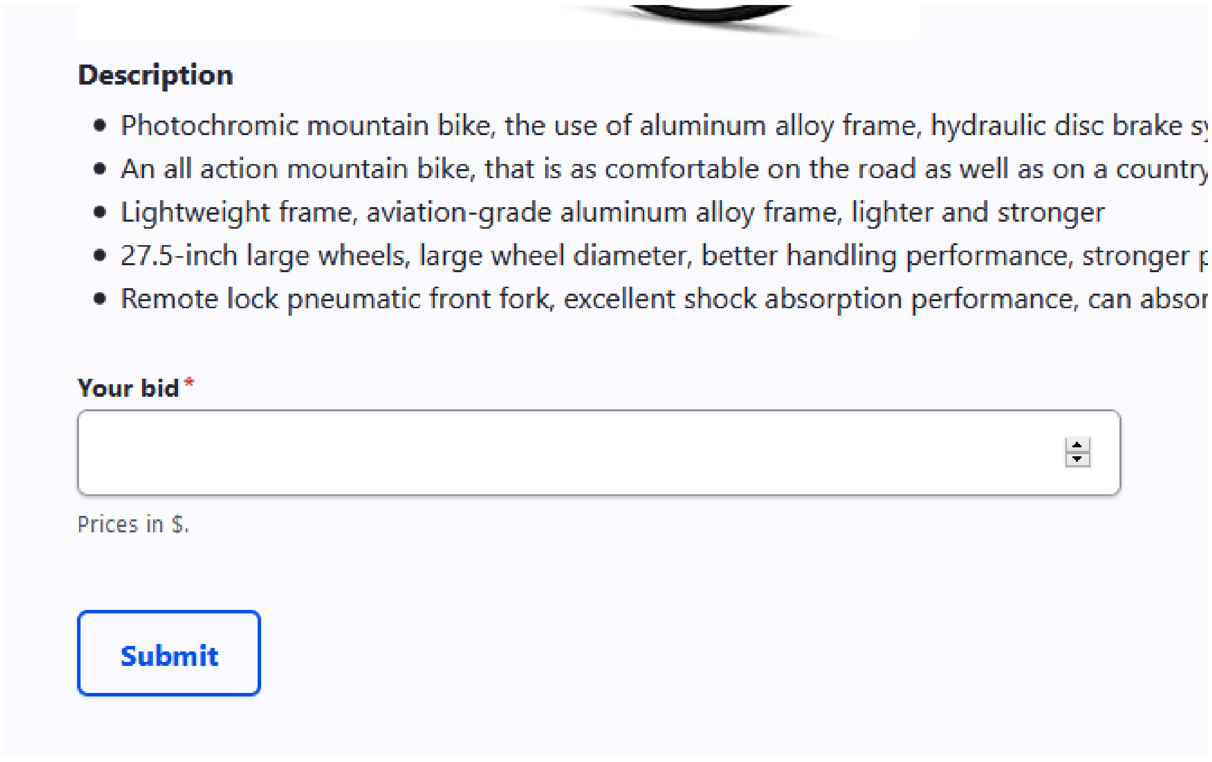
Pages

/offer/*

Specify pages by using their paths. Enter one path per line.
example path is */user/** for every user page. <*front*> is the

[✓ Show for the listed pages]
Hide for the listed pages

Clear caches and take a look.



Description

- Photochromic mountain bike, the use of aluminum alloy frame, hydraulic disc brake sy
- An all action mountain bike, that is as comfortable on the road as well as on a country
- Lightweight frame, aviation-grade aluminum alloy frame, lighter and stronger
- 27.5-inch large wheels, large wheel diameter, better handling performance, stronger p
- Remote lock pneumatic front fork, excellent shock absorption performance, can absor

Your bid *

Prices in $.

Looks good!

💻 Of course you are not able to place a bid on your own offers. Make sure you test it on an offer of a different user.

Remember from the adding fields chapter there were two options in *field_offer_type*:

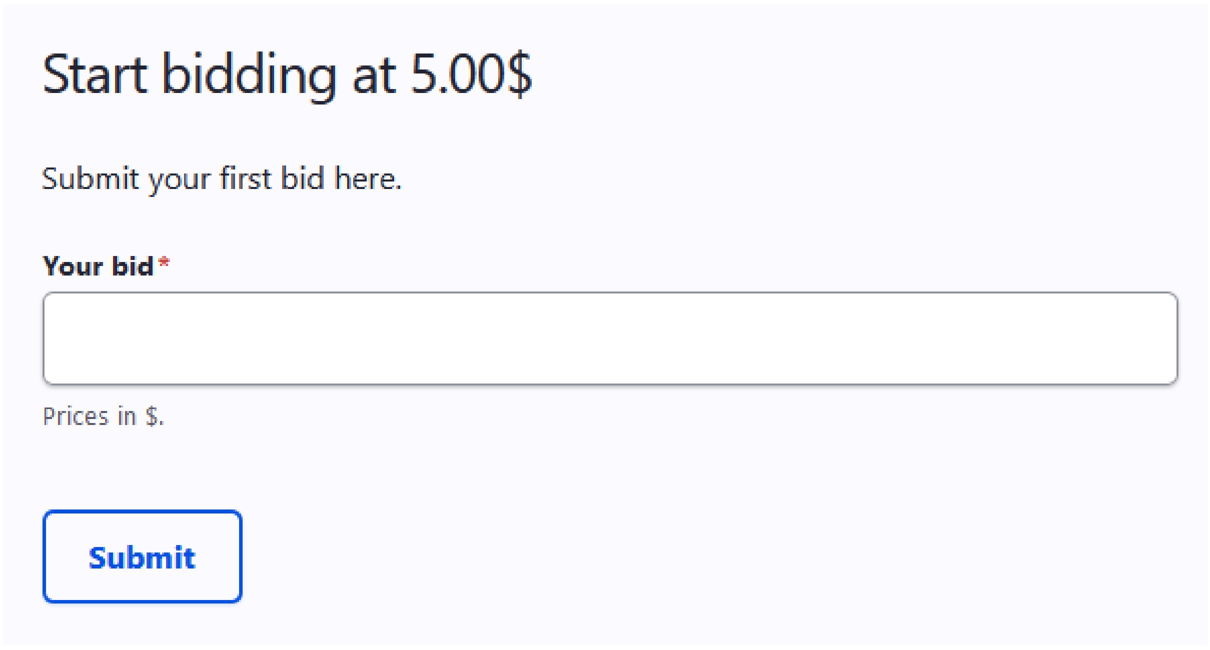
- with_minimum|Set minimum price
- no_minimum|No minimum price

We integrate it in our form in two ways:

- 1) If the form has a minimum price, display it.
- 2) if the form has a minimum price, validate the form and check for the inputted price

To our *buildForm()* function we add this before $form['description']:

```
switch($offer->get('field_offer_type')->getString()) {
 case 'with_minimum':
   $price = $offer->get('field_price')->getString();
   break;
 case 'no_minimum';
   $price = '0';
   break;
}

$form['price'] = [
 '#markup' => '<h2>' . $this->t('Start bidding at @price$', ['@price' =>
$price]) . '</h2>',
];
```


Start bidding at 5.00$

Submit your first bid here.

Your bid *

Prices in $.

Now we extend the bid textfield with two HTML5 attributes, for a minimum price and for a numeric field. This will offer front-end validation in the browser.

```
switch($offer->get('field_offer_type')->getString()) {
  case 'with_minimum':
    $price = $offer->get('field_price')->getString();
    break;
  case 'no_minimum';
    $price = '0';
    break;
}

$form['price'] = [
  '#children' => '<h2>' . $this->t('Start bidding at @price$', ['@price'
=> $price]) . '</h2>',
];

$form['bid'] = [
  '#type' => 'textfield',
  '#attributes' => [
    ' type' => 'number', // note the space before attribute key
    ' min' => $price
  ],
  '#title' => $this->t('Your bid'),
  '#description' => $this->t('Prices in $.'),
  '#required' => TRUE,
];
```

Be careful. Front-end validation is not enough. We validate the input on server-side as well with the ValidateForm() method. For now we'll validate for integers. Further along we'll check to make sure the bid exceeds the highest bid so far.:

```
public function validateForm(array &$form, FormStateInterface
$form_state) {
 parent::validateForm($form, $form_state);

 // Server side validation for numeric
 if (!is_numeric($form_state->getValue('bid'))) {
   $form_state->setErrorByName('bid', t('Bid input needs to be
numeric.'));
 }

}
```

We need to submit the form in the *submitForm()* method. For now, we just submit without saving anything. In a further chapter, we'll save the bid.

Adding site-wide css and javascript

👉 In this chapter you will learn how to add **libraries** containing css and load it on all pages At the end you will have discovered how to apply **global styling** to your project.

While in this course we will not focus (at all) on drupal theming we need to define a css file that gets used site-wide in order to do some small tweaks to make the bidding page look good. In order to add css or javascript to pages we need to define libraries. We will define this inside **custom/offer/offer.libraries.yml**:

```
platform:
 css:
   theme:
     css/platform.css: {}
```

Now we need to tell the system *when* to include the library. There are ways to add them specifically to:

- Controllers

- Forms
- Blocks
- Views

But in this case, we always want it. For this we can use *hook_page_attachments()* in **custom/offer/offer.module**:

```
/**
* Implements hook_page_attachments().
*/
function offer_page_attachments(array &$attachments) {
   $attachments['#attached']['library'][] = 'offer/platform';
}
```

Below this, we add an extra body class to the offer detail page using *hook_preprocess_html*:

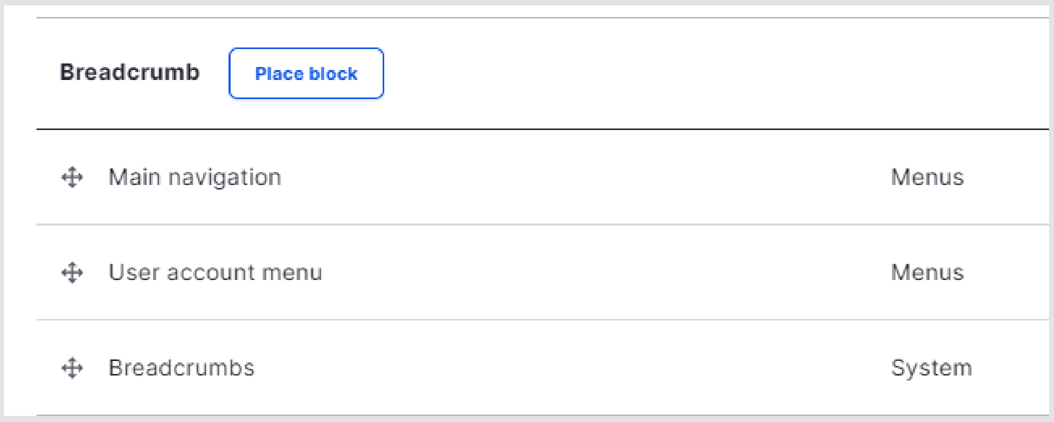
```
/**
 * Add a "offer-detail-page" class to the body on a offer detail page
 */
function offer_preprocess_html(&$variables) {
  $offer = \Drupal::routeMatch()->getParameter('offer');
  if($offer) {
    $variables['attributes']['class'][] = 'offer-detail-page';
  }
}
```

We add **custom/offer/css/platform.css**:

```
.offer-detail-page .block-system-main-block {
 width: 55%;
 float:left;
}
.offer-detail-page #block-bid-block {
 width: 30%;
 float:right;
}
```

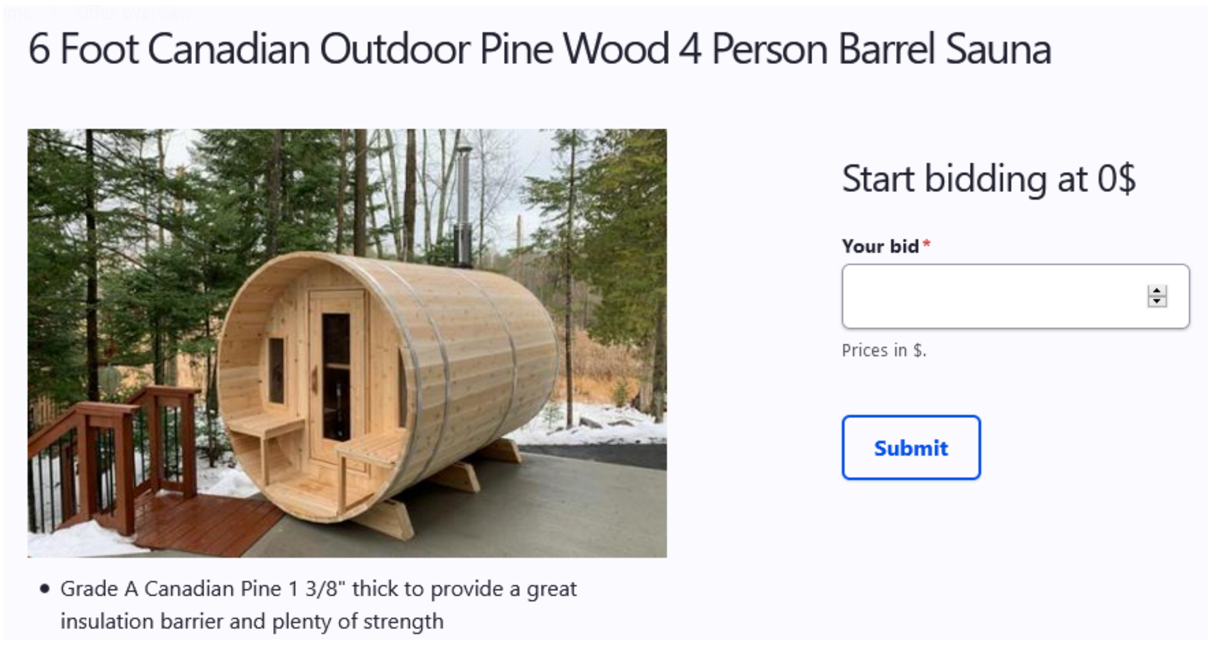
Clear caches, and you'll see the blocks showing more nicely next to each other.

💻 Add the user account menu as well to the breadcrumbs region (choose "hide title").



Main navigation	Menus
User account menu	Menus
Breadcrumbs	System

This puts the bidding block next to the offer:

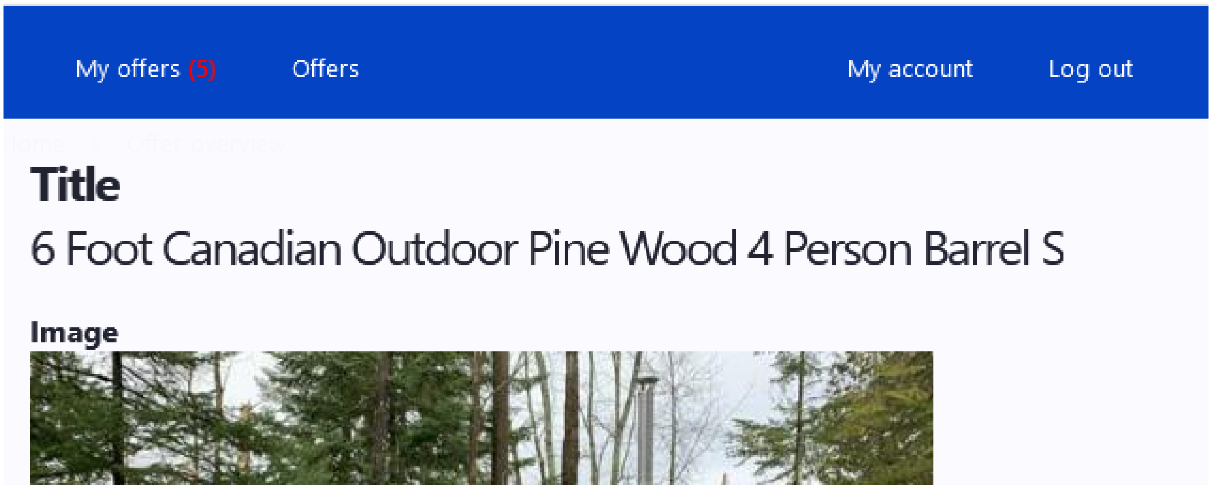

6 Foot Canadian Outdoor Pine Wood 4 Person Barrel Sauna
Start bidding at 0$
Your bid *
Prices in $.
Submit
Grade A Canadian Pine 1 3/8" thick to provide a great insulation barrier and plenty of strength

Add some extra css in **platform.css** for the menus:

```
/** Main menu + user menu **/
.navigation.menu--main ul, .navigation.menu--account ul {
 list-style:none;
 height: 50px;
```

```css
}
.navigation.menu--account {
 float:right;
}
.navigation.menu--main {
 float:left;
}
.navigation.menu--main ul li, .navigation.menu--account ul li {
 float:left;
 list-style:none;
}
.navigation.menu--main .menu, .navigation.menu--account .menu {
 margin: 0;
}
.navigation.menu--main ul li a, .navigation.menu--account ul li a
{
 color: #0444C4;
 padding: 20px;
 text-decoration:none;
 font-size: 0.8125rem;
}
.navigation.menu--main ul li a.is-active, .navigation.menu--main
ul li a:hover, .navigation.menu--account ul li a.is-active,
.navigation.menu--account ul li a:hover {
 background: #E5EDFD;
 color:black;
}
/** breadcrumbs **/
.block-system-breadcrumb-block {
 clear:both;
 margin-left: min(5vw,48px);
 margin-right: min(5vw,48px);
}
.count-badge {
 color: red;
 font-weight: 600;
}
```

We now have a more attractive menu system.



Title

6 Foot Canadian Outdoor Pine Wood 4 Person Barrel S

Image

Note again that for professional platforms we would do a more advanced theming (tailwind, bootstrap, ...), with a primary focus on mobile experience. But this is not the purpose of this course.

For more about libraries, visit this page on drupal.org.

Adding a code-only bid entity

👉 This chapter teaches you how to **define an entity** that will only live in our database and is **not visible in our UI**. After this chapter you will be able to create custom entities with fields like entity reference.

For storing our bids, we need a second entity which will have a minimum of code. We start a new module adding the following files:

- **bid.info.yml**
- **src/Entity/Bid.php**

We start with **custom/bid/bid.info.yml:**

```
name: bid
type: module
description: bid entity
core: 8.x
core_version_requirement: ^8 || ^9
```

Next, our bid entity. We'll add the following database fields:

- user_id (entity_reference): the owner of the bid
- offer_id (entity_reference): the offer the bid is for
- bid (decimal): the amount of the bid
- created (created): the time the bid was done
- changed (changed): the updated time of the bid

Once again, we include the revisioning system which allows us to show a history (e.g. “Person x has raised his bid with 4$”).

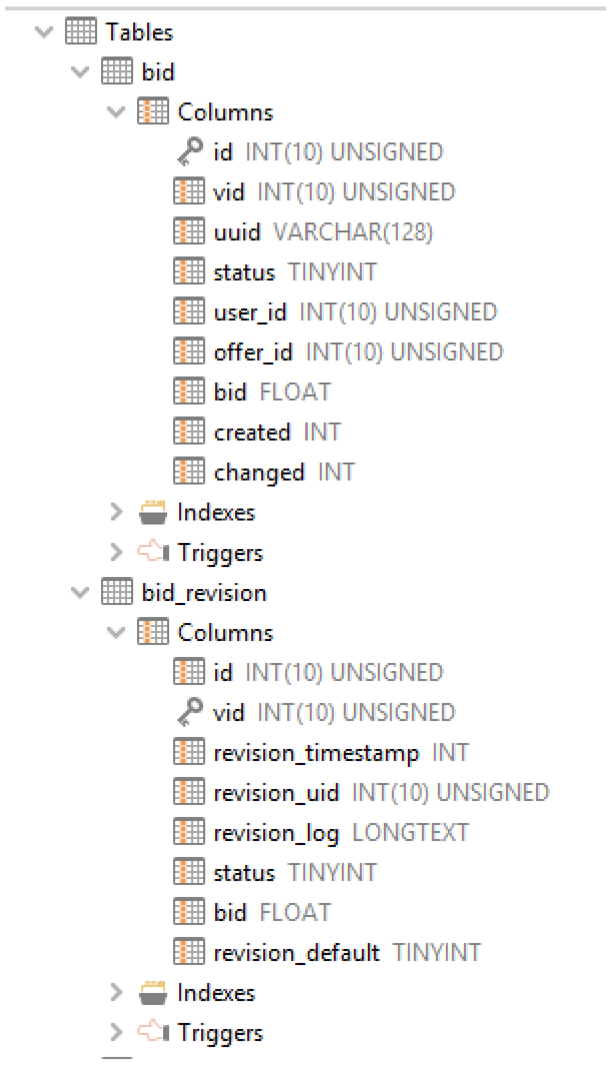
To **custom/bid/src/Entity/Bid.php**:

```
<?php
/**
 * @file
 * Contains \Drupal\bid\Entity\bid.
 */

namespace Drupal\bid\Entity;

use Drupal\Core\Entity\EditorialContentEntityBase;
use Drupal\Core\Field\BaseFieldDefinition;
use Drupal\Core\Entity\EntityTypeInterface;

/**
 * Defines the bid entity.
 *
 * @ingroup bid
 *
 * @ContentEntityType(
 *   id = "bid",
 *   label = @Translation("bid"),
 *   base_table = "bid",
 *   data_table = "bid_field_data",
 *   fieldable = TRUE,
 *   revision_table = "bid_revision",
 *   revision_data_table = "bid_field_revision",
 *   entity_keys = {
 *     "id" = "id",
 *     "uuid" = "uuid",
 *     "label" = "title",
 *     "owner" = "uid",
 *     "revision" = "vid",
 *     "published" = "status",
 *     "uid" = "uid",
 *     "owner" = "uid",
 *   },
 *   handlers = {
 *     "access" = "Drupal\bid\BidAccessControlHandler",
 *   },
 *   revision_metadata_keys = {
 *     "revision_user" = "revision_uid",
 *     "revision_created" = "revision_timestamp",
 *     "revision_log_message" = "revision_log"
 *   }
 * )
 */

class Bid extends EditorialContentEntityBase {

  public static function baseFieldDefinitions(EntityTypeInterface
$entity_type) {
    $fields = parent::baseFieldDefinitions($entity_type); // provides id
and uuid fields

    $fields['user_id'] = BaseFieldDefinition::create('entity_reference')
      ->setLabel(t('User'))
      ->setDescription(t('The user that created the bid.'))
      ->setSetting('target_type', 'user')
      ->setSetting('handler', 'default');

    $fields['offer_id'] =
BaseFieldDefinition::create('entity_reference')
      ->setLabel(t('Offer'))
      ->setDescription(t('The offer the bid is for.'))
      ->setSetting('target_type', 'offer')
      ->setSetting('handler', 'default');

    $fields['bid'] = BaseFieldDefinition::create('float')
      ->setLabel(t('Bid amount'))
      ->setRevisionable(TRUE)
      ->setDescription(t('The bid amount in $.'));

    $fields['created'] = BaseFieldDefinition::create('created')
      ->setLabel(t('Created'))
      ->setDescription(t('The time that the entity was created.'));

    $fields['changed'] = BaseFieldDefinition::create('changed')
      ->setLabel(t('Changed'))
      ->setDescription(t('The time that the entity was last edited.'));

    return $fields;
  }

  /**
   * {@inheritdoc}
   */
  public function getOwner() {
    return $this->get('user_id')->entity;
  }

  /**
   * {@inheritdoc}
   */
  public function getOwnerId() {
    return $this->get('user_id')->target_id;
  }

}
```

Time to enable the module.

```
bash$ drush en bid
 [success] Successfully enabled: bid
```

When double checking our database, we see our columns correctly created:



Tables
bid
Columns
id INT(10) UNSIGNED
vid INT(10) UNSIGNED
uuid VARCHAR(128)
status TINYINT
user_id INT(10) UNSIGNED
offer_id INT(10) UNSIGNED
bid FLOAT
created INT
changed INT
Indexes
Triggers
bid_revision
Columns
id INT(10) UNSIGNED
vid INT(10) UNSIGNED
revision_timestamp INT
revision_uid INT(10) UNSIGNED
revision_log LONGTEXT
status TINYINT
bid FLOAT
revision_default TINYINT
Indexes
Triggers

We've created our bid entity. This entity will only live in our database. It is fully powered by the drupal entity API. But because we don't make it "fieldable".

Saving the bid entities on form submission

👉 This chapter will teach you **how to save custom entity values** on a **form submission**. This is a very common use case for saving data from a custom form. At the end you will be able to save values such as an entity reference and integer values to an entity that only lives in our database.

We now have everything in place to start saving the bids of the users.

Let's change the *submitForm()* function of
custom/offer/src/Form/OfferBidForm.php.
On top we now add:

```
use Drupal\bid\Entity\Bid;
```

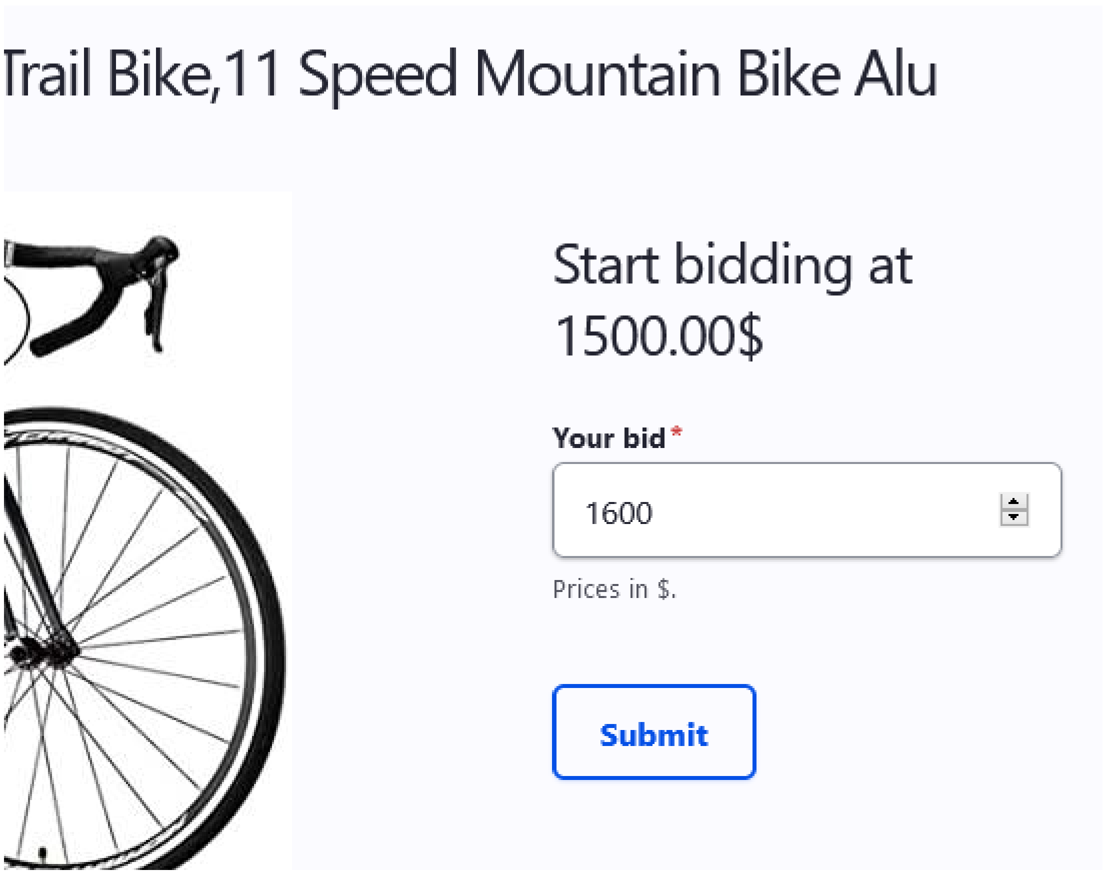
One thing we'll need for saving the bid is the current offer id. We'll send it along in the *buildForm()* function as a hidden and inaccessible field:

```
$form['offer_id'] = [
 '#type' => 'hidden',
 '#value' => $offer->id(),
 '#access' => FALSE
];
```

In the *submitForm()* function we save the offer:

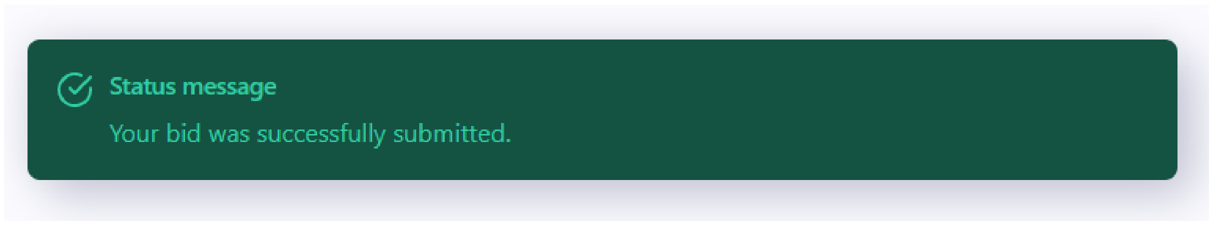
```
public function submitForm(array &$form, FormStateInterface $form_state)
{
  $bid = Bid::create([
    'bid' => $form_state->getValue('bid'),
    'user_id' => ['target_id' => \Drupal::currentUser()->id()],
    'offer_id' => ['target_id' => $form_state->getValue('offer_id')]
  ]);
  $bid->save();
  \Drupal::messenger()->addMessage($this->t('Your bid was successfully
submitted.'));
}
```

Time to test the bidding form.


Trail Bike,11 Speed Mountain Bike Alu

Start bidding at 1500.00$

After submission, we get a success message.


Status message

Your bid was successfully submitted.

We now have successfully saved our bid to the custom entity. But let us build in some security so our database stays clean.

Form validation based on highest bids

👉 In this chapter we will add **custom methods** to our entity. More specifically these are functions to **validate input** on our bidding form server-side.

There are still things left undone. The form needs to anticipate the latest bid so now offer lower or equal than the highest bid gets accepted.

To **custom/offer/src/Entity/Offer.php** we add a method for receiving the highest bids of a given offer.

On top of the file, add:

```
use Drupal\bid\Entity\Bid;
```

To the Offer() class:

```
/**
* Returns the highest bid on an offer
* @return integer $price
*   The price
*/
public function getOfferHighestBid() {
 $bids = [];
 $id = $this->id();
 $query = \Drupal::entityQuery('bid')
   ->condition('offer_id', $id)
   ->sort('bid', 'ASC')
   ->range(NULL, 1);
 $bidIds = $query->execute();
 $price = null;
 foreach($bidIds as $id) {
   $bid = Bid::load($id);
   $price = $bid->get('bid')->getString();
 }
 return $price;
}
```

Also, add a second function loading all the bids of a given offer:

```
/**
* Returns all bids of an offer
* @return array $bids
*   Array of bid entities
*/
public function getOfferBids() {
  $bids = [];
  $id = $this->id();
  $query = \Drupal::entityQuery('bid')
    ->condition('offer_id', $id)
    ->sort('bid', 'DESC');
  $bidIds = $query->execute();
  foreach($bidIds as $id) {
    $bid = Bid::load($id);
    $bids[] = $bid;
  }
  return $bids;
}
```

Inside **custom/offer/src/Form/OfferBiddingForm.php** we change our *buildForm()* function to the following:

```
...// switch statement
   $price = '0';
   break;
}

$OfferHasBid = $offer->getOfferHighestBid();
if($OfferHasBid) {
 $price = $OfferHasBid + 1;
}

$form['price'] = [
 '#children' => '<h2>' . $this->t('Start bidding at @price$', ['@price'
=> $price]) . '</h2>',
];
```

This will change the minimum price for both our title and our front-end validation. To definitely make sure we're not accepting lower bids than our highest bid, we validate server-side too in our *validateForm()* function. Always remember to do this, because front-end validation can be skipped easily.

To the top of **OfferBiddingForm.php:**

```
use Drupal\offer\Entity\Offer;
```

Inside our *validateForm():*

```
// Load the offer and make sure no higher bid was done in the meantime
$offer_id = $form_state->getValue('offer_id');
$offer = Offer::load($offer_id);
$OfferHasBid = $offer->getOfferHighestBid();

switch($offer->get('field_offer_type')->getString()) {
 case 'with_minimum':
   $minium_price = isset($OfferHasBid) ? $OfferHasBid :
$offer->get('field_price')->getString();
   break;
 case 'no_minimum';
   $minium_price = isset($OfferHasBid) ? $OfferHasBid : 0;
   break;
}
if($minium_price >= $form_state->getValue('bid')) {
 $form_state->setErrorByName('bid', t('Minimum bid needs to be @price',
['@price' => (@$minium_price + 1) . '$' ]));;
}
```

We now validate server-side that no bid can pass that is lower than the current highest bid.

Add dynamic variables to our entity teaser

👉 In this chapter we will add **dynamic variables** such as the current amount of bids to our teaser overview. At the end you will be able to **call custom entity methods** and process them into variable output of your teasers in twig.

We can now update our **offer--teaser.html** *template_preprocess_offer()* function inside **custom/offer/offer.module** with the following variables:

Variable name	Description
bid_amount	Shows the amount of bids so far
promo	Shows a “Be the first”-badge if there is no bid on the offer yet.

The function now integrates the *getOfferBids()* and *getOfferHighestBids()* methods:

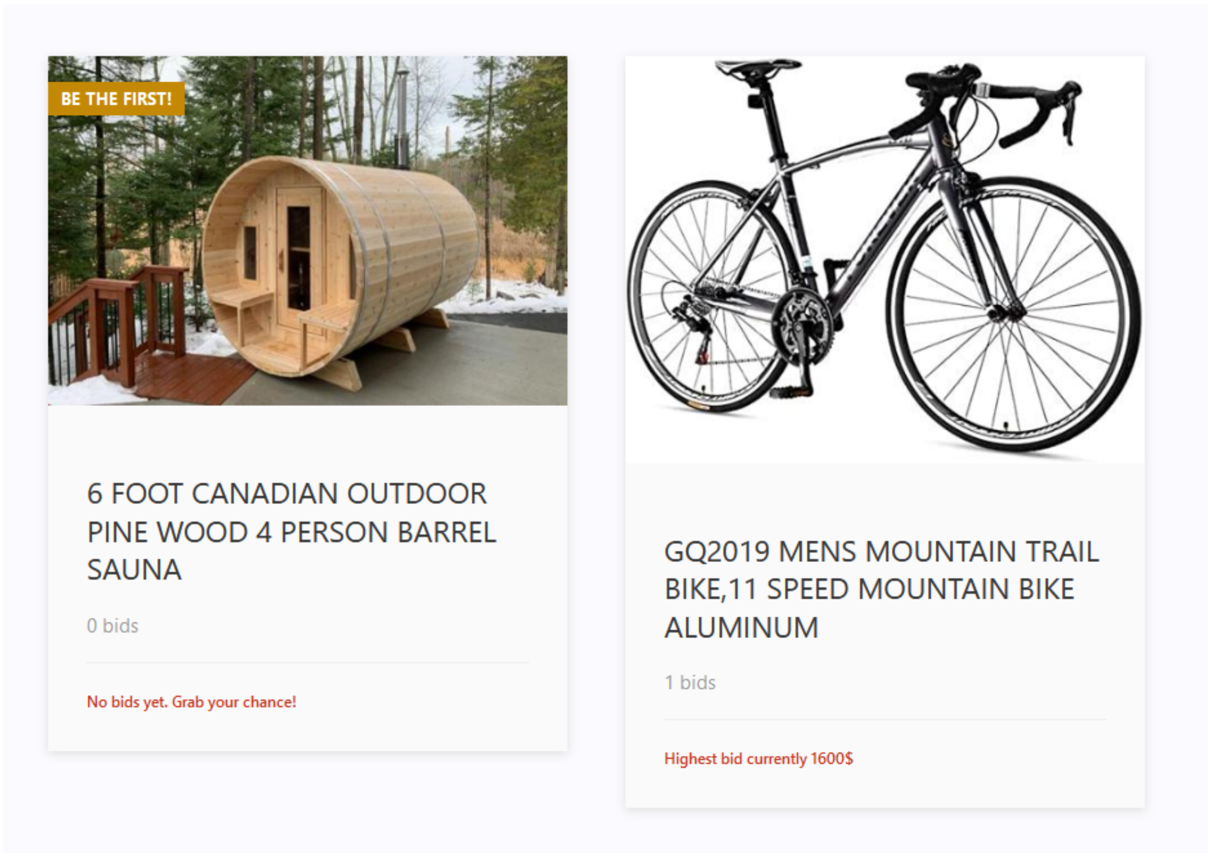
```
 /**
 * Prepares variables for templates.
 * implements hook_preprocess_HOOK()
 */
function template_preprocess_offer(array &$variables) {
  foreach (Element::children($variables['elements']) as $key) {
    $variables['content'][$key] = $variables['elements'][$key];
  }

  $offer = $variables['elements']['#offer'];
  // The full offer object
  $variables['offer'] = $offer;

  // The current price: minimum or highest bid if available
  switch($offer->get('field_offer_type')->getString()) {
    case 'with_minimum':
      $price = 'Start bidding at '.
$offer->get('field_price')->getString() . '$';
      break;
    case 'no_minimum';
      $price = 'Start bidding at 0$';
      break;
  }
  $OfferHasBid = $offer->getOfferHighestBid();
  if($OfferHasBid) {
    $price = 'Highest bid currently ' . $OfferHasBid . '$';
    $variables['price'] = $price;
  } else {
    $variables['price'] = 'No bids yet. Grab your chance!';
  }

  // The amount of bids
  $bid_amount = count($offer->getOfferBids());
  $variables['bid_amount'] = $bid_amount;

  // a promo badge
  if(($bid_amount == 0) && (\Drupal::currentUser()->id() !=
$offer->getOwnerId())) {
    $variables['promo'] = 'Be the first!';
  }
}
```

The final **offer--teaser.html** file:

```
<a href="{{ path('entity.offer.canonical', {'offer': offer.id()}) }}" {{
```

```
attributes.addClass('offer-teaser') }}>
  {% if promo %}<div class="badge">{{promo }}</div>{% endif %}
  <div class="product-tumb">
    {{ content.field_image }}
  </div>
  <div class="product-details">
    <h4>{{ offer.label() }}</h4>
    <p>{{ bid_amount }} bids</p>
    <div class="product-bottom-details">
      <div class="product-price">{{ price }}</div>
      <div class="product-user-profile">{{ user_profile }}</div>
    </div>
  </div>
</a>
```

Clear caches and take a look at */offer*:



6 FOOT CANADIAN OUTDOOR PINE WOOD 4 PERSON BARREL SAUNA

0 bids

No bids yet. Grab your chance!

GQ2019 MENS MOUNTAIN TRAIL BIKE,11 SPEED MOUNTAIN BIKE ALUMINUM

1 bids

Highest bid currently 1600$

We now have more dynamic teasers, showing the current amount of bids, the highest bid so far, and a promo badge if there are no bids yet.

You might stumble upon the fact that this overview not always get updated when bidding, we'll fix this in the chapter about caching.

Validating the entity with constraints

👉 In this chapter we explore the **Entity Validation API**. With this API we set constraints on saving entities to keep our database clean at all times.

What if something goes wrong and a bid gets saved with incorrect values?

The Entity Validation API ensures that all entities pass the validation criteria. We can add validation on an entity level, or on a field level. For our bids we will first validate to make sure all values are set and are not empty.
We could choose to do these checks in our *submitForm()* function, but using constraints is more a good practice way because this way other classes or modules can use the same validators.

We make a new map structure: **custom/bid/src/Plugin/Validation/Constraint**. First file we add is **AllFieldsRequiredConstraint.php**:

```
namespace Drupal\bid\Plugin\Validation\Constraint;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\Exception\MissingOptionsException;

/**
* Requires an offer entity to have all fields required to save as an
object.
*
* @Constraint(
*   id = "AllFieldsRequired",
*   label = @Translation("All fields required.", context =
"Validation"),
*   type = "entity:bid"
* )
*/
class AllFieldsRequiredConstraint extends Constraint {
 public $message = 'At least one field was empty and prevented saving
the bid.';

}
```

This file registers the constraint on the bid entity. It would be used to give feedback in an entity form. This means it will go over this constraint if we ask to validate input for an entire entity. This is also possible for one field only.

But when validating on custom forms, we have to make sure the constraint is also set in **custom/bid/src/Entity/Bid.php**. To the annotations, add the following:

```
 *   constraints = {
 *     "AllFieldsRequired" = {}
 *   }
```

Next file will do the actual validation:

AllFieldsRequiredConstraintValidator.php:

```
<?php

namespace Drupal\bid\Plugin\Validation\Constraint;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

/**
* Validates the AllFieldsRequired constraint.
*/
class AllFieldsRequiredConstraintValidator extends ConstraintValidator {
 /**
  * {@inheritdoc}
  */
 public function validate($entity, Constraint $constraint) {
   if ($entity->get('user_id')->isEmpty()) {
     $this->context->addViolation($constraint->message);
   }
   if ($entity->get('bid')->isEmpty()) {
     $this->context->addViolation($constraint->message);
   }
   if ($entity->get('offer_id')->isEmpty()) {
     $this->context->addViolation($constraint->message);
   }
 }

}
```

The code itself is quite readable. If one of the fields ‘bid’, ‘offer_id’, ‘user_id’ is empty, it will not validate.

A final step is to clear the caches so the constraint gets picked up.

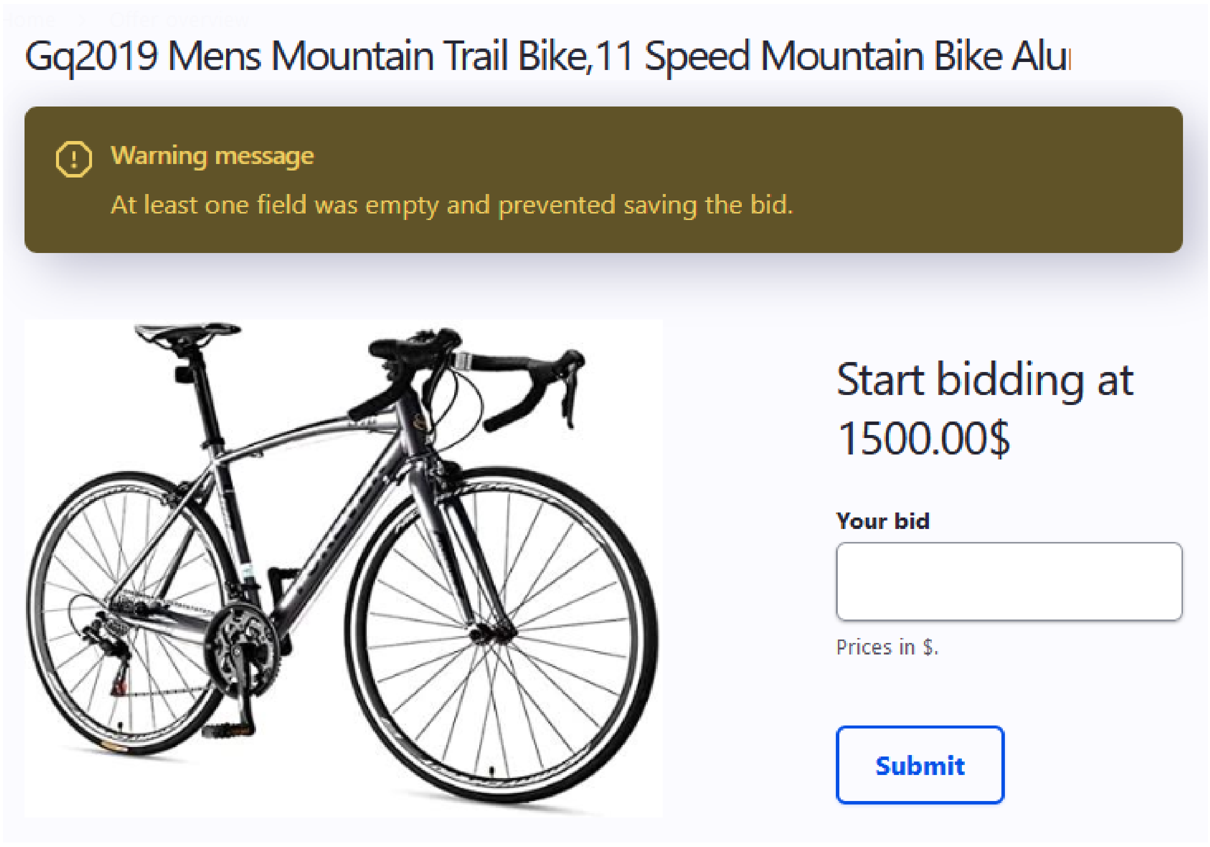
We change the creation of bids in the *submitForm()* function of the **custom/offer/src/Form/OfferBiddingForm.php** to integrate the constraint:

```
public function submitForm(array &$form, FormStateInterface
$form_state) {
  $bid = Bid::create([
    'bid' => $form_state->getValue('bid'),
    'user_id' => ['target_id' => \Drupal::currentUser()->id()],
    'offer_id' => ['target_id' =>
$form_state->getValue('offer_id')]
  ]);
  $violations = $bid->validate();
  $validation = $violations->count();

  if($validation === 0) {
    $bid->save();
    \Drupal::messenger()->addMessage($this->t('Your bid was
successfully submitted.'));
  } else {

\Drupal::messenger()->addWarning($violations[0]->getMessage());
  }
}
```

This is pretty neat. We made sure our entities will be complete on save with custom constraint validators.

💻 You can check if the constraints works like expected if you comment out the *validateForm()* and the ‘#attributes’ and ‘#required’ key in $form[‘bid’] inside the

OfferBiddingForm.php


Gq2019 Mens Mountain Trail Bike,11 Speed Mountain Bike Alu

Warning message

At least one field was empty and prevented saving the bid.

Start bidding at 1500.00$

In a way, this is adding extra security towards malicious input on your database. In addition to server-side validation in our form, we added constraints to saving our entity. The **Entity validation API** can also be used to do all kinds of other checks. For example to check if the offer is still published when the form gets submitted.

For more, check out the Entity validation API on drupal.org:
https://www.drupal.org/docs/drupal-apis/entity-api/entity-validation-api

Displaying all bids in a dynamically rendered table

👉 This chapter will teach you how to use **render arrays** for displaying **processed data**. At the end you will know how to add tables or other ways to display items to a block. By using the **RequestStack** service, you will be able to receive data of the current entity in your block.

We need an overview of previously done bids to the offer. In this section we'll be rendering a drupal table render array. The advantage of using these render arrays is that their layout will be consistent over all themes. Therefore, I like using them.

Because we'll be doing quite some preprocessing on the bids, I add an extra class for properly returning my data. This way we keep things a bit more readable.

Add this file **custom/offer/src/OfferPreprocess/OfferPreprocess.php**. To the file, add:

```
<?php

namespace Drupal\offer\OfferPreprocess;

use Drupal\bid\Entity\Bid;
use Drupal\offer\Entity\Offer;

class OfferPreprocess {

 /**
  * Returns rendered table below an offer
  * @param entity $offer
  *  The offer entity
  * @return array $table
  *  Drupal table render array
  */
 public static function offerTable(Offer $offer) {
   $bids = $offer->getOfferBids();

   $rows = [];
   foreach($bids as $bid) {
     $price = $bid->get('bid')->getString();
     $owner = $bid->getOwner();
     $ownerName = $owner->getDisplayName();
     $time =
\Drupal::service('date.formatter')->formatTimeDiffSince($bid->created->v
alue);

     $row = [
       $ownerName . ' - ' . $time . ' ago', $price . '$'
     ];
     $rows[] = $row;
   }
```

```
    $build['table'] = [
      '#type' => 'table',
      '#rows' => $rows,
      '#empty' => t('This offer has no bids yet. Grab your chance!'),
    ];
    return [
      '#type' => '#markup',
      '#markup' => render($build)
    ];
  }

}
```

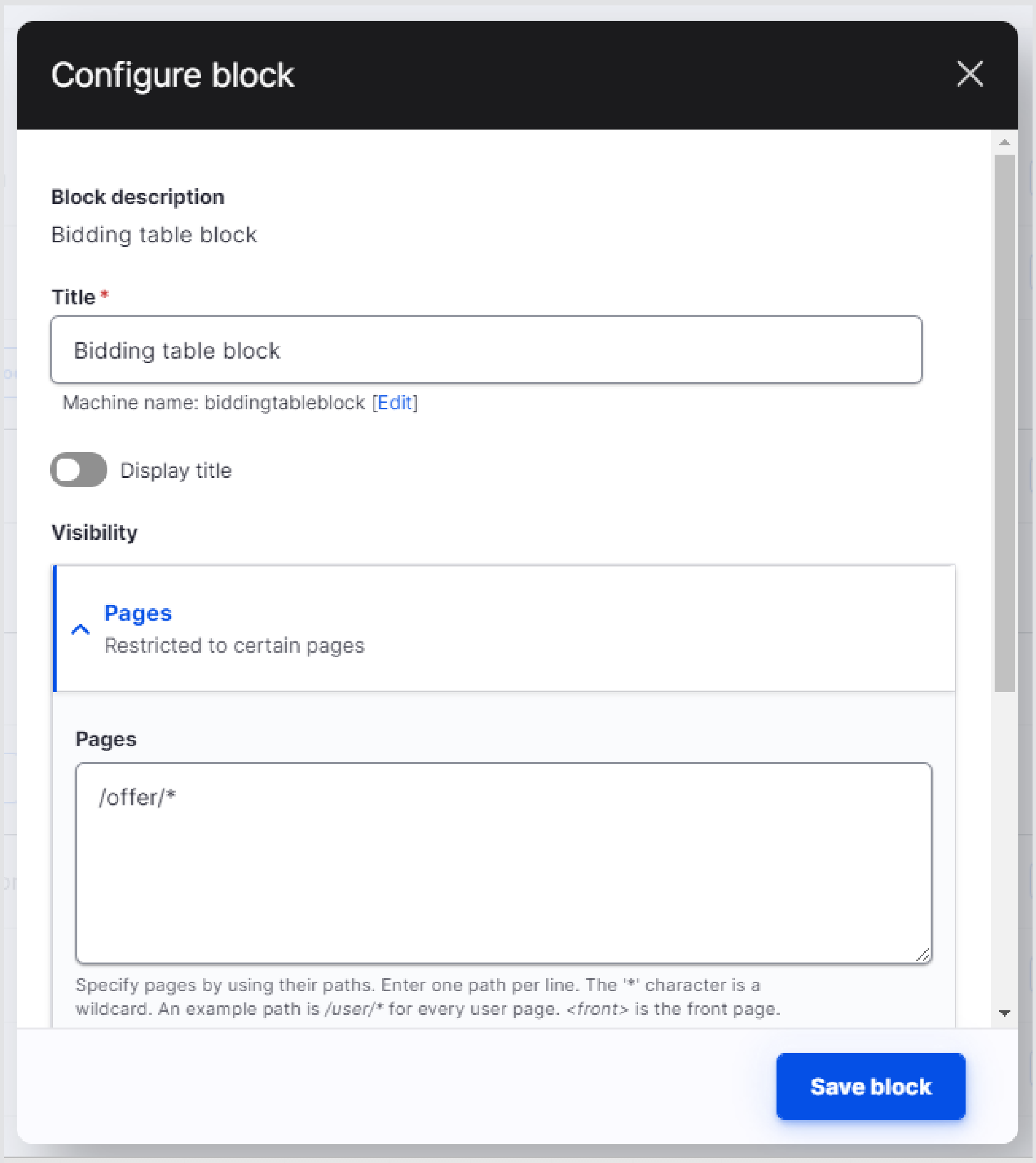
Next, in order to show our table with the latest offers we add it to a new block inside **custom/offer/src/Plugin/Block/OfferBiddingTableBlock.php:**

```
<?php

namespace Drupal\offer\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Symfony\Component\HttpFoundation\RequestStack;
use Drupal\Core\Entity\EntityStorageInterface;
use Drupal\offer\OfferPreprocess\OfferPreprocess;

/**
* @Block(
*   id = "offer_bidding_table_block",
*   admin_label = @Translation("Bidding table block"),
*   category = @Translation("Shows the bidding table to an offer"),
* )
*/

class OfferBiddingTableBlock extends BlockBase implements
ContainerFactoryPluginInterface {

 /**
   * The request object.
   *
   * @var \Symfony\Component\HttpFoundation\RequestStack
   */
 protected $requestStack;
```

```
  /**
   * The entity storage.
   *
   * @var \Drupal\Core\Entity\EntityStorageInterface
   */
  protected $entityStorage;

  /**
   * Constructs a new OfferBiddingTableBlock instance.
   *
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   * @param \Symfony\Component\HttpFoundation\RequestStack $request_stack
   *   The request stack object.
   * @param \Drupal\Core\Entity\EntityStorageInterface $entity_storage
   *   The entity storage.
   */
  public function __construct(array $configuration, $plugin_id,
$plugin_definition, RequestStack $request_stack, EntityStorageInterface
$entity_storage) {
    parent::__construct($configuration, $plugin_id, $plugin_definition);

    $this->requestStack = $request_stack;
    $this->entityStorage = $entity_storage;
  }

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container, array
$configuration, $plugin_id, $plugin_definition) {
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition,
      $container->get('request_stack'),
      $container->get('entity_type.manager')->getStorage('offer')
    );
  }

  /**
   * {@inheritdoc}
   */
  public function build() {
    $offer = $this->requestStack->getCurrentRequest()->get('offer');
    if(!$offer) {
      return null;
    }
    $bid_table = OfferPreprocess::offerTable($offer);
    return $bid_table;
  }

  /**
   * Never cache this block (for now)
   */
  public function getCacheMaxAge() {
    return 0;
  }

}
```

We get the table from our OfferPreprocess class and render it in the same way we did with our form: by making use of the request stack services to get the current offer.

💻 Clear caches and add the block to the bottom of the content region at /admin/structure/blocks. Restrict for authenticated users and set page visibility only at */offer/** like we did for the bidding form.


Configure block

Block description
Bidding table block

Title *
Bidding table block
Machine name: biddingtableblock [Edit]

Display title

Visibility

Pages
Restricted to certain pages

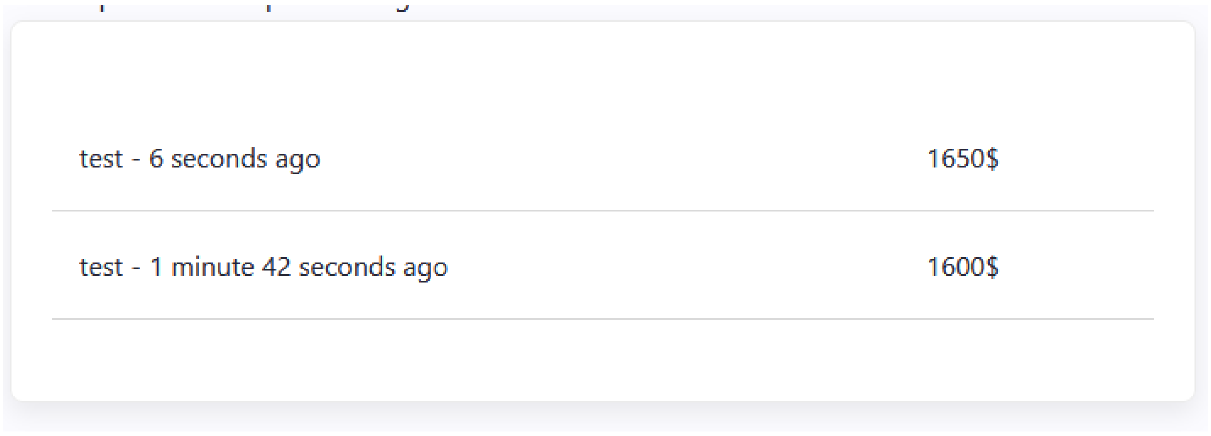
Pages
/offer/*

Specify pages by using their paths. Enter one path per line. The '*' character is a wildcard. An example path is */user/** for every user page. *<front>* is the front page.

Update the **custom/offer/css/platform.css** file to give the block a width:

```
.offer-detail-page #block-bids, .offer-detail-page
#biddingtableblock {
 float:left;
 clear:both;
 width: 55%;
}
```

You'll now see a nicely rendered table with the bids. We will add a nice looking profile picture and some layout later on.


test - 6 seconds ago 1650$

test - 1 minute 42 seconds ago 1600$

Integrating the core revision system into the bidding process to raise a bid

👉 In this important chapter we integrate the **core revisioning** system into our platform. This allows us to modify our entities while keeping a **history of changes**.

When a user raises his bid we will not save this as a new entity but as a revision of his current bid.
This allows us to nicely show a "user x raised his bid with 4$" message to the table. It also adds a variety of options when our software platforms grows larger. Because ultimately, a raising of a bid *is* a revision of a current bid and not a new bid.

First, we need a new method to check if the offer the user is looking at already has bids from him/her. To **custom/modules/offer/src/Entity/Offer.php** we add a method for it:

```
/**
* Checks if the current user has bids on the current offer
* @return bool
*  True if it has, false if it doesn't
*/
public function CurrentUserHasBids() {
 $user_id = \Drupal::currentUser()->id();
 $id = $this->id();
```

```
  $query = \Drupal::entityQuery('bid')
    ->condition('offer_id', $id)
    ->condition('user_id', $user_id);
  $count = $query->count()->execute();
  if($count > 0) {
    return true;
  } else {
    return false;
  }
}
```

Now, in our form we change the label of the submit button dependent on the previous function to from “Submit” to “Raise my bid”

Inside **custom/offer/src/Form/OfferBiddingForm**, to the *buildForm()* method we add above $form[‘actions’]:

```
// Group submit handlers in an actions element with a key of
"actions"
$currentUserHasBid = $offer->CurrentUserHasBids();
$callToAction = $currentUserHasBid ? $this->t('Raise my bid') :
$this->t('Submit');
// Add a submit button that handles the submission of the form.
$form['actions']['submit'] = [
  '#type' => 'submit',
  '#value' => $callToAction,
];
```

Now in our *submitForm*() function, dependent on the *currentUserHasBids()* result we save the offer as a revision instead of a new entity.

Once again we need a helper method on the offer entity to get the bid entity from the current user. To **custom/modules/offer/src/Entity/Offer.php** we add:

```
/**
* Returns the current users bid on the offer
* @return Drupal\bid\Entity\Bid Bid
*   The offer entity
*/
function currentUserBid() {
  $user_id = \Drupal::currentUser()->id();
  $id = $this->id();
```

```
 $query = \Drupal::entityQuery('bid')
   ->condition('offer_id', $id)
   ->condition('user_id', $user_id);
 $result = $query->execute();
 $bidId = reset($result);
 $bid = Bid::load($bidId);
 return $bid;
}
```

In our *submitForm()* function, before our validation we integrate our revisions. We save as a new revision if the user raises his bid, we save it as a new bid if he hasn’t done any bids yet:

```
// Save as new revision of existing bid if user already has bids
// Save as new bid if not
$offer = Offer::load($form_state->getValue('offer_id'));
if($offer->currentUserHasBids()) {
 $bid = $offer->currentUserBid();
 $bid->set('bid', $form_state->getValue('bid'));
 $bid->set('user_id', ['target_id' => \Drupal::currentUser()->id()]);
 $bid->set('offer_id', ['target_id' =>
$form_state->getValue('offer_id')]);
 $bid->setNewRevision();
 $bid->setRe/**
* Checks if the bid has revisions
* @return bool
*  True if it has, false if it does not
*/
public function hasRevisions() {
 $id = $this->id();
 $query = \Drupal::entityQuery('bid')
   ->condition('id', $id);
 $count = $query->allRevisions()->count()->execute();
 if($count > 1) {
   return true;
 } else {
   return false;
 }
}

/**
* Returns list of revision entity ids of the bid. Key is the revision
ID.
* @return array
```

```
*/
public function getRevisionsList() {
 $id = $this->id();
 $query = \Drupal::entityQuery('bid')
   ->condition('id', $id);
 $revisions = $query->allRevisions()->execute();
 return $revisions;
}
```

Drupal cores revision system does a great deal of the heavy lifting. We will now add a ‘raise’ icon to our bid table when a bid has been raised. Another possible feature, for example, is to add actions like sending emails to all other bidders based on this.

To include revision information to our bid table, we’ll add some helper methods to our entities first. To our **custom/bid/src/Entity/Bid.php**, we add two methods. The first is to track if a bid has revisions, the second is a list of all the revision ids:

```
/**
* Checks if the bid has revisions
* @return bool
*  True if it has, false if it does not
*/
public function hasRevisions() {
 $id = $this->id();
 $query = \Drupal::entityQuery('bid')
   ->condition('id', $id);
 $count = $query->allRevisions()->count()->execute();
 if($count > 1) {
   return true;
 } else {
   return false;
 }
}

/**
* Returns list of revision entity ids of the bid. Key is the revision
ID.
* @return array
*/
public function getRevisionsList() {
 $id = $this->id();
 $query = \Drupal::entityQuery('bid')
   ->condition('id', $id);
 $revisions = $query->allRevisions()->execute();
```

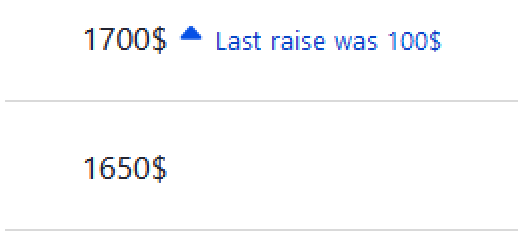
```
  return $revisions;
}
```

We’ll include these two methods in the **custom/offer/src/OfferPreprocess/OfferPreprocess.php** *offerTable()* method.

💻 `use Drupal\Core\Render\Markup; // on top of file`

```
/**
 * Returns rendered table below an offer
 * @param entity $offer
 *  The offer entity
 * @return array $table
 *  Drupal table render array
 */
public static function offerTable(Offer $offer) {
  $bids = $offer->getOfferBids();

  $rows = [];
  foreach($bids as $bid) {
    $price = $bid->get('bid')->getString();
    $owner = $bid->getOwner();
    $ownerName = $owner->getDisplayName();
    $time =
\Drupal::service('date.formatter')->formatTimeDiffSince($bid->created->v
alue);

    $updates = '';
    $link = '';
    if($bid->hasRevisions()) {
      $revisions = $bid->getRevisionsList();
      // We now have the list of revisions.
      // Let's compare the latest bid with the last revision
      $current_revision_id = $bid->getLoadedRevisionId();
      // We now know the current, we want the one before the current
      // We remove the current from the revisions list
      unset($revisions[$current_revision_id]);
      // And take the last one from the revisions list
      $last_revision_id = max(array_keys($revisions));
      $revisionBid = \Drupal::entityTypeManager()
        ->getStorage('bid')
        ->loadRevision($last_revision_id);
      $revisionAmount = $revisionBid->get('bid')->getString();
      $priceDifference = $price - $revisionAmount;
      $updates = '<svg width="24px" height="18px" viewBox="0 0 24 24"
fill="#61f70a" xmlns="http://www.w3.org/2000/svg">
      <path d="M6.1018 16.9814C5.02785 16.9814 4.45387 15.7165 5.16108
14.9083L10.6829 8.59762C11.3801 7.80079 12.6197 7.80079 13.3169
8.59762L18.8388 14.9083C19.5459 15.7165 18.972 16.9814 17.898
16.9814H6.1018Z" fill="#61f70a"/>
      </svg><small style="color:#0444C4">Last raise was ' .
$priceDifference .'$</small>';
    }

    $row = [
      Markup::create($ownerName . ' - ' . $time . ' ago'),
Markup::create($price . '$'  . $updates)
    ];
    $rows[] = $row;
  }

  $build['table'] = [
    '#type' => 'table',
    '#rows' => $rows,
    '#empty' => t('This offer has no bids yet. Grab your chance!'),
  ];
  return [
    '#type' => '#markup',
    '#markup' => render($build)
  ];
}
```

The result of all this is that in the bid table a notice appears that indicates that the bid was raised.



1700$ ▲ Last raise was 100$

1650$

Revisions are complete entities stored aside from our bid entities. We now took advantage of this quite complex system to store differences, like a raise of a bid.

Think about the possibilities of this. You could show a detailed table overview or graphic of the bidding history.

Deleting a bid with a core dialog pop-up

👉 In this chapter we explore the drupal API system further. First, we will create a **delete form** to make it possible to delete your own bids. Second, we will open the form with a **core dialog pop-up**, to make sure you stay on the page.

Users need to have the possibility to delete a bid when they want to. Again, we use the standard entity behaviours for the deletion. To **custom/bid/src/Entity/bid.php**, we add a delete link:

```
*   handlers = {
*     "access" = "Drupal\bid\BidAccessControlHandler",
*     "form" = {
*       "delete" = "Drupal\bid\Form\BidDeleteForm",
*     },
*   },
*   links = {
*     "delete-form" = "/bid/{bid}/delete",
*   },
```

We add a route inside **custom/bid/bid.routing.yml**:

```
entity.bid.delete_form:
 path: '/bid/{bid}/delete'
 defaults:
   _entity_form: bid.delete
   _title: 'Delete bid'
 requirements:
   _entity_access: 'bid.delete'
```

We add the delete form itself to **custom/bid/src/Form/BidDeleteForm.php**:

```
<?php

/**
* @file
* Contains \Drupal\bid\Form\BidDeleteForm.
*/

namespace Drupal\bid\Form;

use Drupal\Core\Entity\ContentEntityConfirmFormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Url;
use Drupal\offer\Entity\Offer;

/**
* Provides a form for deleting a content_entity_example entity.
*
* @ingroup bid
*/
class BidDeleteForm extends ContentEntityConfirmFormBase {

 /**
  * {@inheritdoc}
  */
 public function getQuestion() {
   return $this->t('Are you sure you want to delete your bid of
%price$?', array('%price' => $this->entity->get('bid')->getString()));
 }

 /**
  * {@inheritdoc}
  *
  * If the delete command is canceled, return to the bid list.
  */
 public function getCancelUrl() {
   $offer_id = $this->entity->get('offer_id')->getString();
   $url = new Url('entity.offer.canonical', ['offer' => $offer_id]);
   return $url;
 }

 /**
  * {@inheritdoc}
  */
 public function getConfirmText() {
   return $this->t('Delete');
 }

 /**
  * {@inheritdoc}
  *
  * Delete the entity and log the event. logger() replaces the watchdog.
  */
 public function submitForm(array &$form, FormStateInterface
$form_state) {
   // Redirect to offer after delete.
   $offer_id = $this->entity->get('offer_id')->getString();

   $entity = $this->getEntity();
   $entity->delete();

   $this->logger('bid')->notice('deleted bid %id.',
     array(
       '%title' => $this->entity->id(),
     ));
   $form_state->setRedirect('entity.offer.canonical', ['offer' =>
$offer_id]);
 }

}
```

This is a standard entity delete form with the exception of the getCancelUrl() and SubmitForm() methods. There we needed to get the offer ID from the bid to generate a url to redirect to after deletion or cancel action.

To do this properly, we need again to add an AccessControlHandler. To **custom/bid/src/BidAccessControlHandler.php**, add

```
<?php

namespace Drupal\bid;

use Drupal\Core\Access\AccessResult;
use Drupal\Core\Entity\EntityAccessControlHandler;
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Session\AccountInterface;

/**
 * Access controller for the bid entity. Controls create/edit/delete
access for entity and fields.
```

```
 *
 * @see \Drupal\bid\Entity\Bid.
 */
class BidAccessControlHandler extends EntityAccessControlHandler {

  protected function checkAccess(EntityInterface $entity, $operation,
AccountInterface $account) {
    $access = AccessResult::forbidden();
    switch ($operation) {
      case 'view':
        $access = AccessResult::allowed;
        break;
      case 'update':
        $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
        break;
      case 'edit':
        $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
        break;
      case 'delete':
        $access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($entity);
        break;
    }

    return $access;
  }
}

?>
```

This code determines if I am the owner of the bid. This is an important security check!

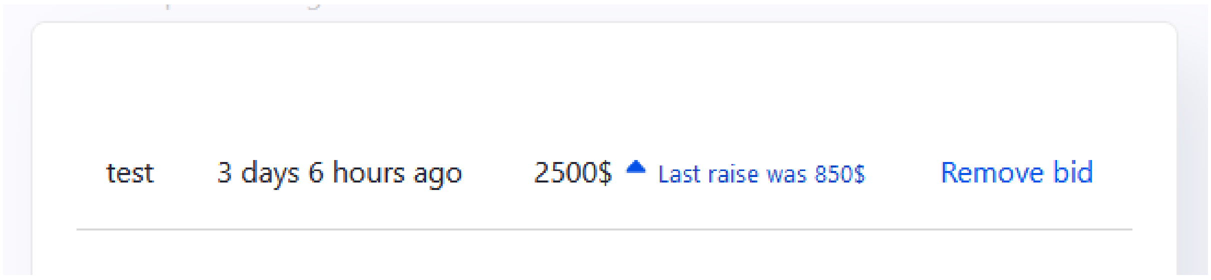
The final part of this section is to provide a delete link to the bid when a user is the owner of the bid.

To the **custom/offer/src/OfferPreprocess/OfferPreprocess.php** *OfferTable()* we can now add a delete link:

```
use Drupal\Core\Link; // on top of file
```

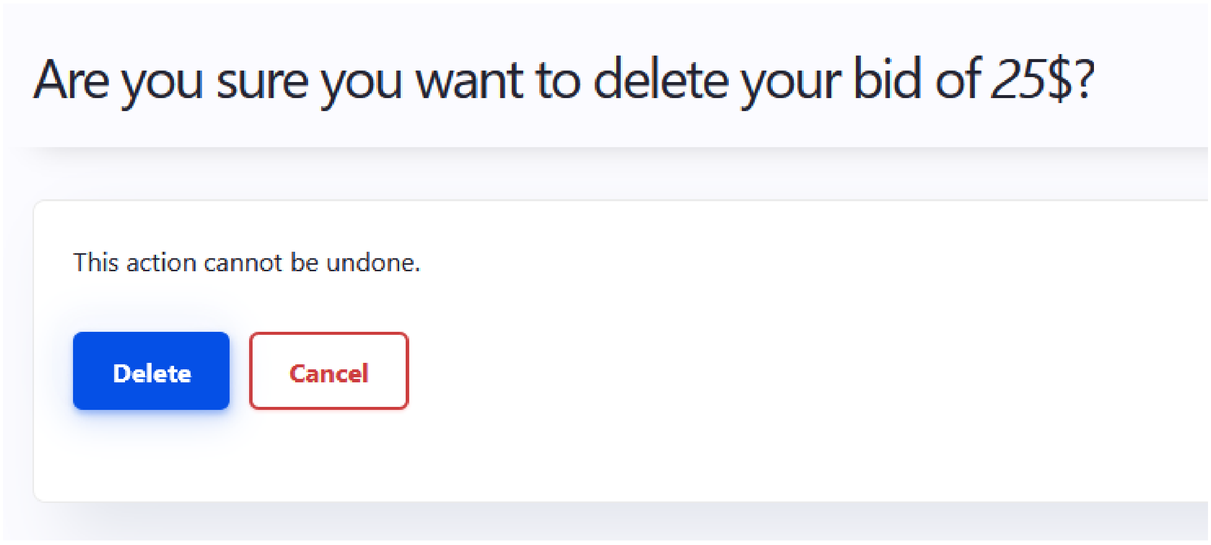
```
if($bid->access('delete')) {
  $url = $bid->toUrl('delete-form');
  $link = Link::fromTextAndUrl('Remove bid', $url)->toString();
}

$row = [
  Markup::create($ownerName), Markup::create($time.' ago'),
Markup::create($price . '$'  . $updates), Markup::create($link)
];
$rows[] = $row;
```

We clear caches and now we have a delete link on our own bids:



test 3 days 6 hours ago 2500$ Last raise was 850$ Remove bid

When we click the link, we get redirected to a delete form with cancel and delete link. We took advantage of the core entity delete form to do this. No need to to access checks on the form, Drupal core provides us these by default thanks to the Entity API!



Are you sure you want to delete your bid of 25$?

This action cannot be undone.

The confirm page is quite nice. But I prefer to stay on the same page when deleting a bid. Drupal core offers a nice option to do this: an ajax dialog to render the very same form in. Let's see how this works.

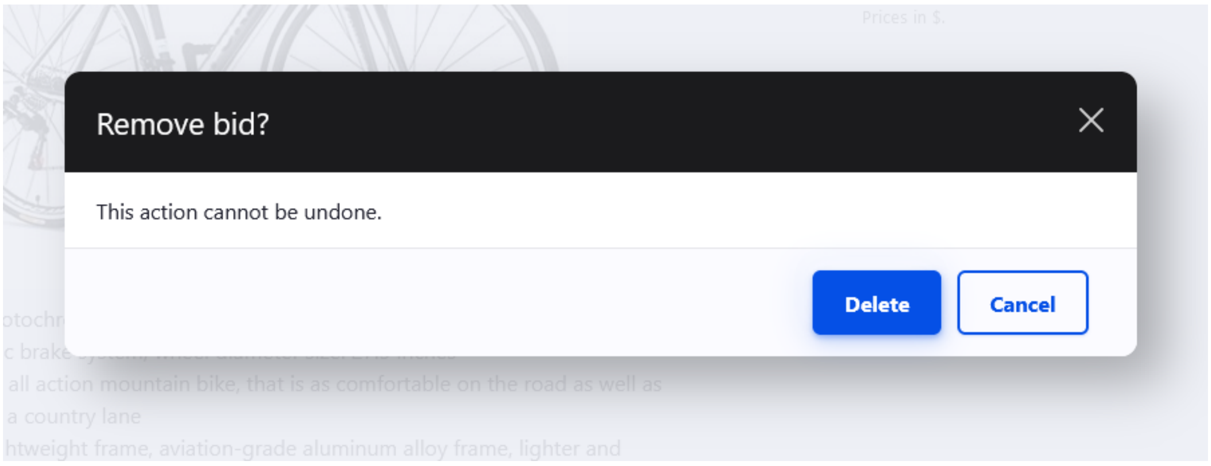
We only need to add extra attributes to the delete link. Therefore we'll use a link render array. We've already used drupal core's render arrays for tables and drop buttons. The link render array offers possibilities to add attributes we'll need to ajaxify dialogs. We change the *offerTable()* part where we define the delete link:

```
if($bid->access('delete')) {
 $url = $bid->toUrl('delete-form');
 $deleteLink = [
   '#type' => 'link',
   '#title' => 'Remove bid',
   '#url' => $url,
   '#attributes' => [
     'class' => ['use-ajax', 'button', 'button--small',
'button--danger'],
     'data-dialog-type' => 'modal',
     'data-dialog-options' =>
\Drupal\Component\Serialization\Json::encode(['title' => t('Remove
bid?'), 'width' => 800,]),
   ],
 ];
 $link = render($deleteLink);
}
```

One last thing is that we need to make sure the core drupal ajax wrappers are loaded as well. To **offer/offer.libraries.yml** we must add the following dependencies to our platform library:

```
platform:
  css:
    theme:
      css/platform.css: {}
  dependencies:
    - core/drupal.ajax
    - core/jquery
    - core/drupal
```

Now clear caches..When we now click 'Remove bid', we stay on the same page and get this very nicely looking dialog:



This action cannot be undone.

Again a nice example of where drupal does the heavy lifting for us. The dialogs underlying architecture can change in the future, but will still work because we make use of the API.

Part 5: Transitions, Events, Caching and user registration

👉 Now that we've built the main parts of our software we have the chance to start with **automated tasks** like sending emails and the expiration of a bid. But we'll also discuss **caching** and how to handle **user registration**.

User notifications on transition events

Drupal 8 has significantly decreased the number of *hooks* it uses. For a number of events it will use the Event system with Event Subscribers (more about them in the next sections).

Unfortunately there is no "Event" to subscribe to when an entity switches its state. Therefore we have to use a hook for notifying when an offer switches from "published" to "expired".

I've quickly added a third code-only entity named "notification" in a new module. It stores these values:

Key	Description
offer_id	The offer the notification is about
user_id	The user that has a bid on an offer
type	The type of notification (f.e. "expired")

I've also added a handler for access and for views. This way we can add a views page at url /notifications, to show the notifications for this user.

💻 If you are following along, copy the notification module into your project and install. Building an entity is something that was covered already!

```
bash$ drush en notification
[success] Successfully enabled: notification
```

We add a hook_entityType_update() function to **custom/offer/offer.module**:

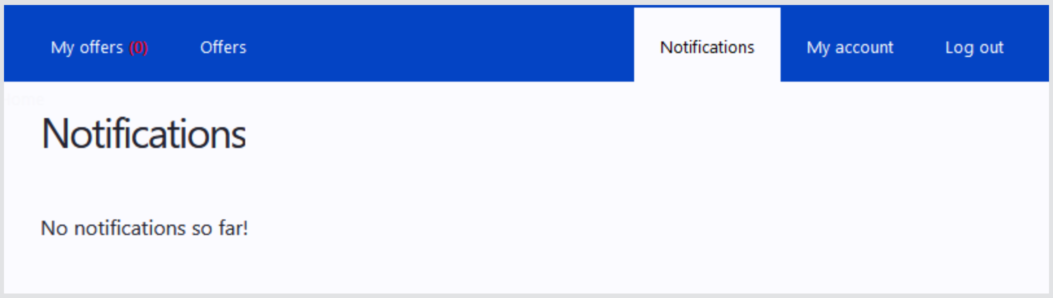
```
use Drupal\notification\Entity\Notification;

/**
* Implements hook_ENTITY_TYPE_update().
*/
function offer_offer_update(Drupal\Core\Entity\EntityInterface
$entity) {

 // Current Moderation state
 $currentState = $entity->get('moderation_state')->getString();

 // Original Moderation state
 $originalState =
$entity->original->get('moderation_state')->getString();

 // Check if transition equals "published" -> "expired"
 // Save a notification for all bidders
 if(($originalState == 'published') && ($currentState ==
'expired')) {
   $bids = $entity->getOfferBids();
   foreach($bids as $bid) {
     Notification::create([
       'type' => 'expired',
       'user_id' => ['target_id' => $bid->getOwnerId()],
       'offer_id' => ['target_id' => $entity->id()]
     ])->save();
   }
 }
}
```

What this snippet does is the following: it checks if the offer goes from a ‘published’ to a ‘expired’ state. Then it will loop over all the bids on the offer, and create notification entities for every bidder.

Next I’ve added view support for notification entities, just like we did for our offer entities.

I’ve added a view and a menu-link for the **/notifications** page, which will show the notifications, filtered on the current user.

💻 If you have installed the module from the full project code, the view should be created on install. This is because the config of this view was added via *custom/notification/config/install/views.view.notifications.yml*



Notifications

No notifications so far!

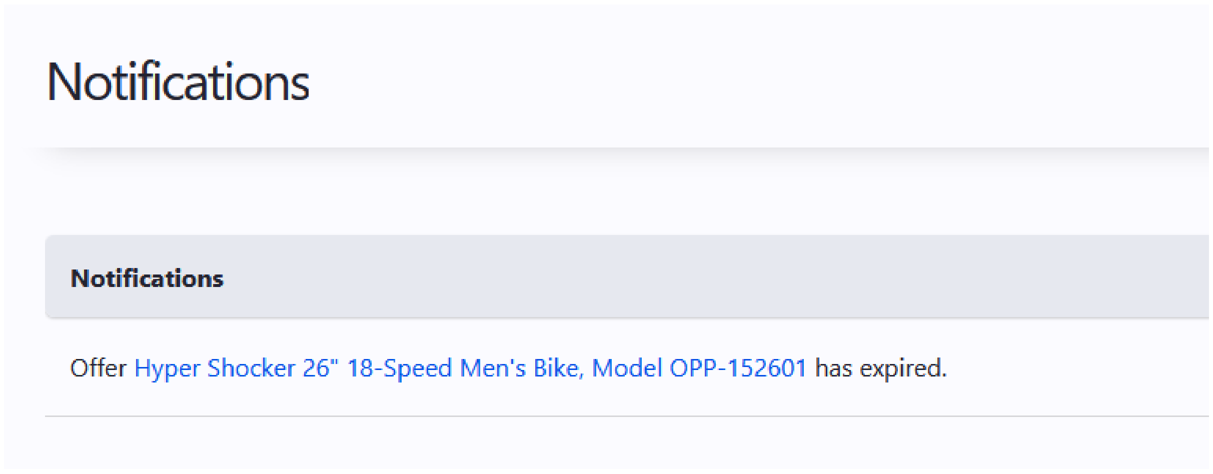
In a custom views field in **custom/notification/src/Plugin/views/field/NotificationMessage.php** the *render(ResultRow $values)* function looks like this:

// on top:

```
use Drupal\offer\Entity\Offer;
use Drupal\Core\Url;
use Drupal\Core\Link;
```

```
public function render(ResultRow $values) {
 $entity = $values->_entity;
 $type = $entity->get('type')->getString();
 $offer_id = $entity->get('offer_id')->getString();
 $offer = Offer::load($offer_id);
 if($type == 'expired') {
   $url = Url::fromRoute('entity.offer.canonical', array('offer'
=> $offer->id()));
   $link = Link::fromTextAndUrl($offer->label(),
$url)->toRenderable();
   $text = 'Offer '. render($link) .' has expired.';
 }
 return [
   '#children' => $text
 ];
}
```

This result in the following notification for the bids that I did:


Notifications

Notifications

Offer Hyper Shocker 26" 18-Speed Men's Bike, Model OPP-152601 has expired.

We’ve now shown how to create notifications based on workflow states. It’s not hard to imagine the following possible features with this:

- Send email when someone bids higher than your bid
- Send email to all bidders when offer has expired
- Send email when new offers are available
- ….

To **custom/notification/notification.links.menu.yml** this was added to get the link in the user menu:

```
offer.account.notifications:
  title: 'Notifications'
  menu_name: account
  route_name: view.notifications.page_1
  weight: -40
```

Update entire view with custom ajax callback

👉 In this chapter you will learn how to **update a view** with a **ajax callback.** At the end you will be able to delete/insert/update records, reload a view without leaving your page.

Ajax is a great technology to refresh parts of a page without actually refreshing the full page. In general this enhances user experience. The difference between just using javascript is a server-side callback is included in the call where actual database operations can be done.

For this project, we want to be able to delete notifications in the notifications view. To start, we create a controller which will be used to delete a notification. This is a regular controller, but instead of visiting it, only our ajax call will be visiting the class.

Create the routing file **custom/notification/notification.routing.yml:**

```
notification.delete:
 path: '/notification/delete/{method}/{id}'
 defaults:
   _controller:
'\Drupal\notification\Controller\NotificationDeleteController::Render'
   _title: 'Delete notification?'
 requirements:
   _access: 'TRUE'
   id: '\d+'
   method: 'nojs|ajax'
```

Check out the {method} slug. This can be either 'nojs' or 'ajax'. In the actual link, we will use 'nojs'. Drupal will use ajax if it has detected javascript is enabled. This is good for accessibility. Further, we use a regex for id to be sure it is numeric. Access checks are done in the controller itself.

To **custom/notification/src/Controller/NotificationDeleteController.php:**

```php
<?php

namespace Drupal\notification\Controller;

use Drupal\Core\Controller\ControllerBase;
use Drupal\notification\Ajax\DeleteNotificationCommand;
use Drupal\Core\Ajax\AjaxResponse;
use Drupal\notification\Entity\Notification;
use Drupal\Core\Access\AccessResult;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Drupal\Core\Url;

/**
* Class NotificationDeleteController.php.
*/
class NotificationDeleteController extends ControllerBase {

 public function Render($id, $method) {
   // Load the notification
```

```
  $notification = Notification::load($id);

  // Send back users that do not have access
  if(!$notification) {
    return AccessResult::forbidden();
  }
  if(!$notification->access('delete')) {
    return AccessResult::forbidden();
  }

  // Delete the notification
  $notification->delete();

  if($method == 'ajax') {
    $response = new AjaxResponse();
    $response->addCommand(new DeleteNotificationCommand());
  } else {
    // no javascript: send back to page
    $path = Url::fromRoute('view.notifications.page_1');
    $response = new RedirectResponse($path);
    $response->send();
  }

  return $response;
 }

}
```

What stands out is that in this controller the notification gets deleted (if owner) and our response is an AjaxResponse(). In fact a “deleteNotificationCommand”, which is a custom one. If javascript is disabled, it sends back a redirect to the view.

Now let’s register the AjaxResponse. To **custom/offer/src/Ajax/DeleteNotificationCommand.php:**

```
<?php
namespace Drupal\notification\Ajax;
use Drupal\Core\Ajax\CommandInterface;

class DeleteNotificationCommand implements CommandInterface {

 // Implements Drupal\Core\Ajax\CommandInterface:render().
 public function render() {
   return array(
     'command' => 'DeleteNotification',
     'selector' => $this->selector,
   );
 }
}
```

With this Command, the system knows what to send to javascript. Not that it is also possible to send data in an extra key. If we would need this, we would call New DeleteNoficiationCommand($extra) in **NotificationDeleteController.php.** And then add 'extra' next to 'command' and 'selector' in our **DeleteNotificationCommand.** In the javascript file we're about to add, the 'extra' would be available in the *response* argument. But we'll do another handy trick: refreshing the view after deletion.

To **custom/notification/notification.libraries.yml** we add a new library for our notification javascript file:

```
notification:
  js:
    js/notification.js: {}
  dependencies:
    - core/drupal.ajax
    - core/jquery
    - core/drupal
    - views/views.ajax
```

We add the javascript file **custom/notification/js/notification.js**:

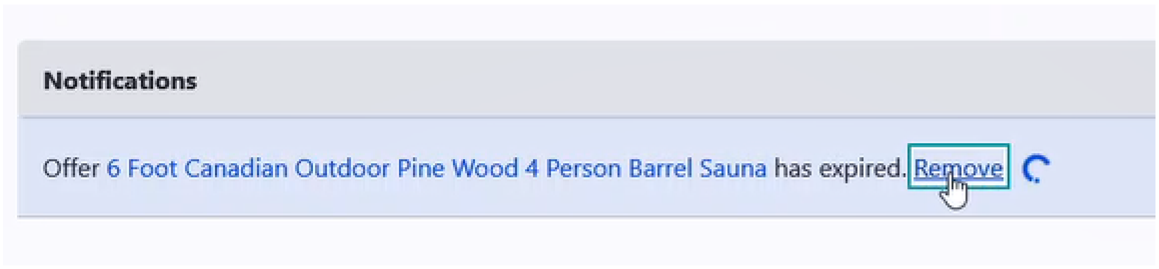
```
( function ($) {

  // Command to replace element.
  Drupal.AjaxCommands.prototype.DeleteNotification = function(ajax,
response, status) {
    if(status === 'success') {
      $('.view-id-notifications').trigger('RefreshView');
    }
  }

})(jQuery);
```

This was the final bit. The DeleteNotification javascript function gets called in the front-end to make a visible change to the users page. Here, after a successful deletion call, we refresh the view using a built-in trigger. Make sure the ‘use ajax’ option is set to ‘ON’ in your view settings.

To start using it, we extend the **custom/offer/src/Plugin/views/field/NotificationMessage.php** *render()* function with a “Remove” link:

```
/**
* {@inheritdoc}
*/
public function render(ResultRow $values) {
 $entity = $values->_entity;
 $type = $entity->get('type')->getString();
 $offer_id = $entity->get('offer_id')->getString();
 $offer = Offer::load($offer_id);
 if($type == 'expired') {
   $url = Url::fromRoute('entity.offer.canonical', array('offer' =>
$offer->id()));
   $link = Link::fromTextAndUrl($offer->label(), $url)->toRenderable();
   $text = 'Offer '. render($link) .' has expired.';
   // Add delete link for removing notifications
   $deleteUrl = Url::fromRoute('notification.delete', ['method' =>
'nojs', 'id' => $entity->id()]);
   $deleteLink = Link::fromTextAndUrl('Remove',
$deleteUrl)->toRenderable();
   $deleteLink['#attributes'] = ['class' => 'use-ajax'];
   $deleteText = render($deleteLink);
   $output = $text . ' ' . $deleteText;
 }
 return [
   '#children' => $output
 ];
}
```

Note we just have to add the ‘use-ajax’ class to trigger the ajax call. We’re done!


Notifications

Offer 6 Foot Canadian Outdoor Pine Wood 4 Person Barrel Sauna has expired. Remove

If we now remove a notification, our view gets updated automatically while staying in the page.

OO in practice: deletion of bid and notification entities when an offer gets deleted

👉 In this chapter we will teach how you can remove entities that are linked to other entities using the **preDelete()** functions on the entity.

We should think through all of the necessary use-cases of our software: what happens to bids when somebody deletes his offer? This is a relevant question. The following entities rely on an offer:

- The bids to the offer
- The notifications about the offer

We must make sure these are deleted as well, when deleting an offer.

Remember the following line in **custom/offer/src/Entity/Offer.php**:

```
class Offer extends EditorialContentEntityBase
```

Let's drill down how the inheritance goes:

- EditorialContentEntityBase extends ContentEntityBase
- ContentEntityBase extends EntityBase

In **Core/Entity/EntityBase.php** on **line 401** we see this line:

```
/**
* {@inheritdoc}
*/
public function delete() {
 if (!$this->isNew()) {

$this->entityTypeManager()->getStorage($this->entityTypeId)->delete([$th
is->id() => $this]);
 }
}
```

Thanks to these lines, we are able to use $offer->delete() to delete an offer.

On **line 463** we find this function:

```
/**
* {@inheritdoc}
*/
public static function preDelete(EntityStorageInterface $storage, array
$entities) {
}
```

An empty function that gets called right before deletion of an entity. This is the guy we need! Let's add this one to our **custom/offer/src/Entity/Offer.php**, below the *preCreate()* function:

```
/**
* {@inheritdoc}
*/
public static function preDelete(EntityStorageInterface $storage, array
$entities) {
 parent::preDelete($storage, $entities);

 // Delete all bids and notifications of the offer that will be deleted
 foreach ($entities as $entity) {
   $entity->deleteAllLinkedBids();
   $entity->deleteAllLinkedNotifications();
 }
}

/**
* Deletes all bids linked to the offer.
* @param bool $delete
* @throws \Drupal\Core\Entity\EntityStorageException
*/
public function deleteAllLinkedBids($delete = FALSE) {
 $id = $this->id();

 $query = \Drupal::entityQuery('bid')
   ->condition('offer_id', $id);
 $bidIds = $query->execute();
 foreach($bidIds as $id) {
   $bid = Bid::load($id);
   $bid->delete();
 }
}
```

```
/**
* Deletes all notifications linked to the offer.
* @param bool $delete
* @throws \Drupal\Core\Entity\EntityStorageException
*/
public function deleteAllLinkedNotifications($delete = FALSE) {
 $id = $this->id();

 $query = \Drupal::entityQuery('notification')
   ->condition('offer_id', $id);
 $notificationIds = $query->execute();
 foreach($notificationIds as $id) {
   $notification = Notification::load($id);
   $notification->delete();
 }
}
```

Now, when a user deletes an offer in the UI, all linked dataparts get removed as well. This keeps our database clean and prevents orphan entities which could lead to errors.

A good example of the power of Object Oriented programming (OO): we inherited the *preDelete()* function and use it in our offer entities. Make sure you take a look at all the other methods that can be used in *EntityBase* as well.

As an extra we could add a new notification type to inform the user that an offer he had done a bid on was deleted, but this would lead us too far away for this course.

Caching in-depth

👉 In this important chapter we dig deeper on the powerful **drupal caching API**. After this chapter you will be able to **implement complex caching mechanisms** on entities, views and controllers.

Caching is an important aspect that was improved drastically when the drupal community built drupal 8 from scratch (it was a complete rewrite of drupal 7). More specifically it added great control over *which* portions of your page you would like to have cached and *how*.

Cache API types	What	Example	Example code
Cache tags	For dependencies on data like entities and configuration	A block that needs to be refreshed when a node entity with id 68 is changed.	`public function getCacheTags() { return Cache::mergeTags(parent::getCacheTags(), array('node:68')); }`

Cache contexts	For variations, i.e. dependencies on the request context	Cache a block differently on every path.
		
		Or cache a block differently for every user.
			`public function getCacheContexts() { return ['url.path']; }`
			
			`public function getCacheContexts() { return ['user']; }`

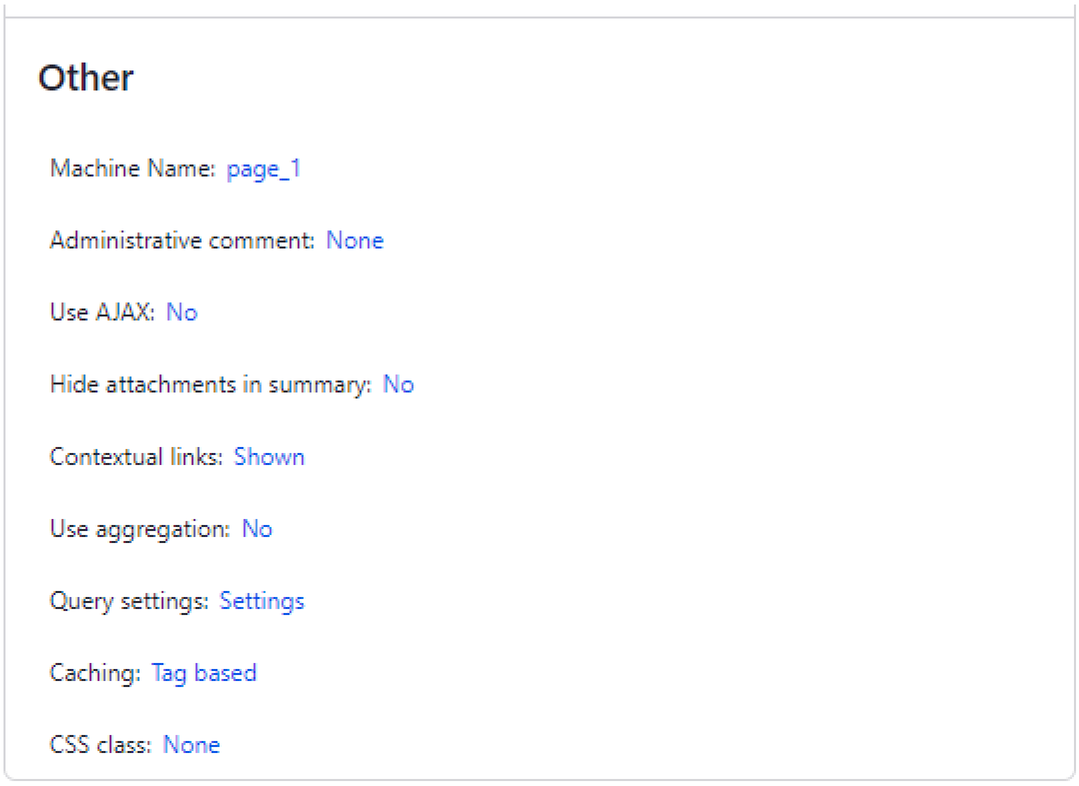
Cache max-age	For time-sensitive caching, i.e. time dependencies	Never cache this block.
		
		Cache 1 day, then renew each day
			`public function getCacheMaxAge() { return 0; }`
			
			`public function getCacheMaxAge() { return 86400; }`

Certainly interesting to check out https://www.drupal.org/docs/drupal-apis/cache-api to see all contexts, available tags etc.

Caching of views pages

👉 This short and theoretical chapter explains how caching works on view listings.

For our offering platform, let's start with the /offer page. This is a views page. It has caching support in the settings. On the right bottom corner of **/admin/structure/views/view/offer**:



Other

Machine Name: page_1

Administrative comment: None

Use AJAX: No

Hide attachments in summary: No

Contextual links: Shown

Use aggregation: No

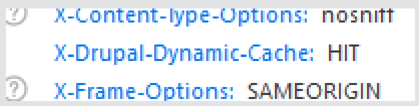
Query settings: Settings

Caching: Tag based

CSS class: None

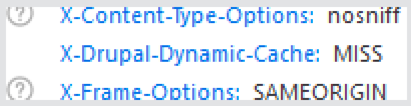
Set “Caching” to “Tag based”. Because this view only looks for entities of type ‘offer’, it will automatically invalidate when new offers are added.

💻 I can check this in the “headers” tab of the network profile in my developer tools:



```
X-Content-Type-Options: nosniff
X-Drupal-Dynamic-Cache: HIT
X-Frame-Options: SAMEORIGIN
```

X-Drupal-Dynamic-Cache: HIT means I’ve got myself a cached page. If I add a new offer, this is what I get the first time I render the page:



```
X-Content-Type-Options: nosniff
X-Drupal-Dynamic-Cache: MISS
X-Frame-Options: SAMEORIGIN
```

So, the first time this views page gets rendered it goes in the cache until a new offer is added. This way I’m sure the page will be loaded very quickly.

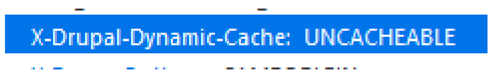
Caching of custom entity pages

👉 In this chapter we explore **the caching mechanisms** behind the **detail page** of an **entity**.

Great news for our custom entity detail pages. Drupal 8 has an automatic caching system by default that is pretty amazing, which makes it possible to provide caching without any configuration.

💻 Make sure the core module “Internal dynamic page caching” (dynamic_page_cache) is enabled to properly cache your entities.

But be sure to check this. It is why in an earlier chapter I advised to keep forms in blocks instead of trying to render them IN your entity. If I had added the form in my preprocess() functions I would have got this header:

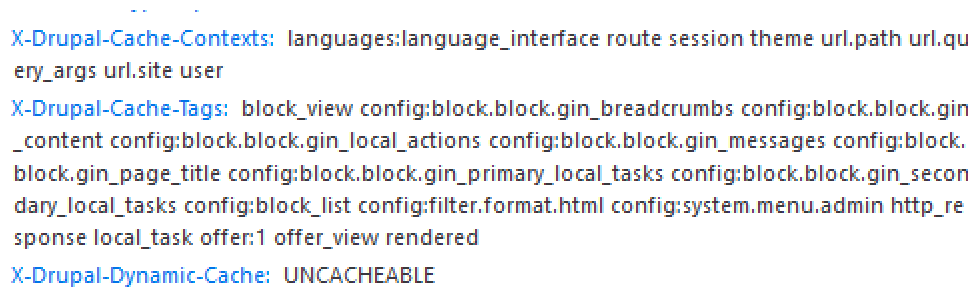


```
X-Drupal-Dynamic-Cache:  UNCACHEABLE
```

The X-Drupal-Dynamic-Cache: UNCACHEABLE response header says our page is definitely not cached. A useful help is to add the following (development) setting to your **services.yml** file:

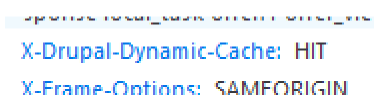
```
http.response.debug_cacheability_headers: true
```

Refreshing the page resulted in the following response headers (with my form inside my entity):



```
X-Drupal-Cache-Contexts:  languages:language_interface route session theme url.path url.qu
ery_args url.site user
X-Drupal-Cache-Tags:  block_view config:block.block.gin_breadcrumbs config:block.block.gin
_content config:block.block.gin_local_actions config:block.block.gin_messages config:block.
block.gin_page_title config:block.block.gin_primary_local_tasks config:block.block.gin_secon
dary_local_tasks config:block_list config:filter.format.html config:system.menu.admin http_re
sponse local_task offer:1 offer_view rendered
X-Drupal-Dynamic-Cache:  UNCACHEABLE
```

The nice part about this is we get more information about all the cache contexts and cache tags of our page. The drupal entity cache *should work* by default so when I use the block to show my form I get the desired result:



```
X-Drupal-Dynamic-Cache: HIT
X-Frame-Options: SAMEORIGIN
```

I have my cached entity again. This works pretty smart. Any update on the entity will clear the caches for it. For our form and bidding table, we need to use more advanced caching techniques like contexts and cache tags.

> Caching was a pain-point in drupal 7, in drupal 8 and later it is definitely one of the strengths!

Use of cache contexts and cache tags for caching custom blocks

> 👉 This chapter digs deeper on **cache tags** and **how to invalidate caches** on a custom block, when something happens like an entity update.

How can we properly cache our bid table for our users? We want two things:

- A table cached uniquely for our page (i.e. based on the entity we are looking at)
- A table that gets updated when a new bid is added

To make it uniquely for our page, remove the max-age:0 based caching that was currently set in **custom/offer/src/Plugin/Block/OfferBiddingTableBlock.php** and add:

```php
use Drupal\Core\Cache\Cache; // on top of file
```

```php
/**
* Cache per page
*/
public function getCacheContexts() {
 return ['url.path'];
}
```

This way we are sure we cache a bidding table that is different per page. But of course we need to have it changed once there are new bids.

```
/**
 * Invalidate caches when there are new bids
 */
public function getCacheTags() {
  $offer = $this->requestStack->getCurrentRequest()->get('offer');
  $offerId = $offer->id();
  return Cache::mergeTags(parent::getCacheTags(),
array('offer:'.$offerId));
}
```

While this will update the table when our offer entity gets updated, this won't update the table when there is a new bid, yet.

We add an *invalidateTags()* command on the offer whenever there is a new bid on the offer. This way we have two nice advantages:

- We automatically update offer teasers to show an update on the amount of bids
- We automatically invalidate caches of the OfferBiddingTableBlock, to show the latest bids

We go back to our **custom/offer/src/Plugin/Form/OfferBiddingForm.php**. More specifically in the *submitForm()* method we add the *invalidateCacheTags()* command on the current offer:

```
use Drupal\Core\Cache\Cache; // on top of file
```

```
if($validation === 0) {
  $bid->save();
  Cache::invalidateTags($offer->getCacheTags());
  \Drupal::messenger()->addMessage($this->t('Your bid was successfully
submitted.'));
} else {
  \Drupal::messenger()->addWarning($violations[0]->getMessage());
}
```

With rather easy commands, we are now able to have custom-tailored caching working!

Use of cacheable dependencies in render arrays

👉 This chapter will expand your knowledge on caching a little more. More specifically **Cacheable Dependencies**. With this API we can add caching *dependent* of certain situations.

To make dealing with cacheability metadata (cache tags, cache contexts and max-age) easier, Drupal 8 has CacheableDependencyInterface. It is implemented by a majority of objects you interact with while writing Drupal 8 code!

Because not only controllers or blocks need to have caching, also access results, menu links, context plugins, condition plugins, and so on.

You might remember the following from our chapter on access:

```
$access = AccessResult::allowedIf($account->id() ==
$entity->getOwnerId())->cachePerUser()->addCacheableDependency($en
tity);
```

This means that access to the entity gets cached for each user and this will be invalidated when the entity changes. That's why we get access to an offer as soon its workflow state changes from draft to published.

We'll use this in the following example. Remember our form with global configuration from the chapter on configuration management? Let's extend it with a textarea field. We want to create a "About" page with the content of that textarea field. I only want the caches to invalidate when the content of the textarea (which is config) is changed.

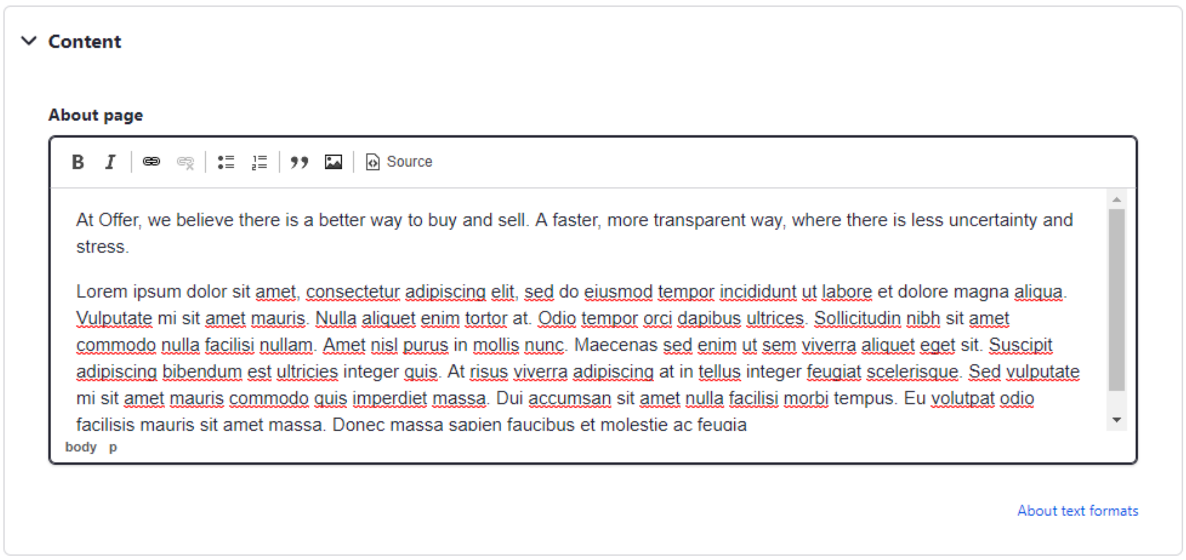
To **custom/offer/src/Form/OfferSettingsForm** we add to the *buildForm()* method:

```
$form['content'] = array(
 '#type' => 'details',
 '#title' => $this->t('Content'),
 '#open' => TRUE,
);
$form['content']['about'] = [
 '#type' => 'text_format',
 '#title' => $this->t('About page'),
 '#default_value' => $config->get('about'),
 '#maxlength' => NULL,
];
```

Extend the *submitForm* method:

```
$this->config('offer.customconfig')
  ->set('tagmanager', $form_state->getValue('tagmanager'))
  ->set('about', $form_state->getValue('about')['value'])
  ->save();
```

Notice the use of 'text_format' instead of textarea. This makes my textarea render with a ckeditor wysiwyg overlay at /admin/config/offer/adminsettings. I add some text to it and save:



At Offer, we believe there is a better way to buy and sell. A faster, more transparent way, where there is less uncertainty and stress.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Vulputate mi sit amet mauris. Nulla aliquet enim tortor at. Odio tempor orci dapibus ultrices. Sollicitudin nibh sit amet commodo nulla facilisi nullam. Amet nisl purus in mollis nunc. Maecenas sed enim ut sem viverra aliquet eget sit. Suscipit adipiscing bibendum est ultricies integer quis. At risus viverra adipiscing at in tellus integer feugiat scelerisque. Sed vulputate mi sit amet mauris commodo quis imperdiet massa. Dui accumsan sit amet nulla facilisi morbi tempus. Eu volutpat odio facilisis mauris sit amet massa. Donec massa sapien faucibus et molestie ac feugia

Okay, I move on with adding a controller.

First, the route and the access to **custom/offer/offer.routing.yml**:

```
offer.about:
  path: '/about'
  defaults:
    _controller: '\Drupal\offer\Controller\AboutPageController::Render'
    title: 'About us'
  requirements:
    _access: 'TRUE'
```

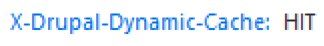
The page will be accessible for all users, hence the _access: 'True' parameter. Next, I add **custom/offer/src/Controller/AboutPageController.php**:

```php
<?php

namespace Drupal\offer\Controller;

use Drupal\Core\Controller\ControllerBase;

/**
* Class AboutPageController
*/
class AboutPageController extends ControllerBase {

 public function Render() {
   $config = \Drupal::config('offer.customconfig');

   $build = [
     '#markup' => $config->get('about'),
   ];

   $renderer = \Drupal::service('renderer');
   $renderer->addCacheableDependency($build, $config);

   return $build;
 }
}
```

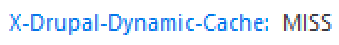
Add the menu item to the menu at **offer.links.menu.yml**:

```yaml
offer.main.about:
 title: 'About'
 menu_name: main
 route_name: offer.about
 weight: 80
```

Clear caches and visit your page. Like always, the very first time this page ever gets visited it goes into the cache. Next time, everyone gets a correctly caches page:


X-Drupal-Dynamic-Cache: HIT

But when I add a new line of text to the About configuration textarea and save:


X-Drupal-Dynamic-Cache: MISS

Very nice. Again we controlled precisely when and where the caches needed to be emptied.

💻 We've now learned that we can use cacheable dependencies for every renderable object we use in our drupal installation.

Cache menu items with a custom cache tag

👉 This chapter is about how to add **caching mechanisms** to **plugins**. At the end you will be able to provide your system with a menu item with a counter, that is cached the way you want it.

Remember the section about custom menu links. We added a menu link with a counter, to show the amount of offers this user has. This is a query for this user only. One strategy could be to never cache this. But doing a custom query on every page load is not good practice.

This is where custom cache tags come into play. Let's head back to the **custom/offer/src/Plugin/Menu/MyOffers.php** and we remove the *getCacheMaxAge()* function. Instead we add:

```
use Drupal\Core\Cache\Cache; // on top of file

/**
* {@inheritdoc}
*/
public function getCacheTags() {
 return Cache::mergeTags(parent::getCacheTags(),
array('my_offers_user_'. \Drupal::currentUser()->id()));
}
```

This is a unique cache tag for this user only. The menu item will only get queried again when this specific cache tag gets invalidated.

We must make sure to invalidate this whenever:

A) A new offer is added
B) A new offer is deleted

We need to add *postCreate()* and *postDelete()* methods in **custom/offer/src/Entity/offer.php**:

```
use Drupal\Core\Cache\Cache; // on top of file
```

```
 /**
   * {@inheritdoc}
   */
  public function postCreate(EntityStorageInterface $storage) {
    Cache::invalidateTags(['my_offers_user_'. $this->getOwnerId()]);
  }

  /**
   * {@inheritdoc}
   */
  public static function postDelete(EntityStorageInterface $storage,
array $entities) {
    parent::preDelete($storage, $entities);

    // Delete all bids and notifications of the offer that will be
deleted
    foreach ($entities as $entity) {
      $entity->deleteAllLinkedBids();
      $entity->deleteAllLinkedNotifications();
      Cache::invalidateTags(['my_offers_user_'. $entity->getOwnerId()]);
    }

  }
```

That's it! Whenever an offer is added or deleted, the custom cache tag for this user gets invalidated. Again, another example of the power of Object oriented coding and the great caching system of drupal.

Invalidate cache of another entity after saving

👉 In this chapter more about **invalidating caches of an entity dependent on other**

linked entities.

On building the platform, I stumbled upon the following. Whenever I added bids to an offer, the teaser kept saying ‘0 offers so far’. This makes sense, because my offer did not know about new bids. Bids are just entities being stored in its own table.

To solve this issue I inherit the *postSave()* and *postDelete()* method from *ContentEntityBase()* to my bid entity. To **custom/offer/src/Entity/bid.php** add the following:

```
// Add this on top of the file
use Drupal\Core\Cache\Cache;
use Drupal\offer\Entity\Offer;
```

```
  /**
   * {@inheritdoc}
   */
  public function postSave(EntityStorageInterface $storage,
$update = TRUE) {
    parent::postSave($storage, $update);
    $offer = Offer::load($this->get('offer_id')->target_id);
    Cache::invalidateTags($offer->getCacheTagsToInvalidate());
  }

  /**
   * {@inheritdoc}
   */
  public static function postDelete(EntityStorageInterface
$storage, array $entities) {
    parent::preDelete($storage, $entities);
    // Invalidate all caches of offers whenever bids are deleted
    foreach ($entities as $entity) {
      $offer = Offer::load($entity->get('offer_id')->target_id);
      if($offer) {
        Cache::invalidateTags($offer->getCacheTagsToInvalidate());
      }
    }
  }
```

With these changes, after a bid gets saved or deleted it will invalidate the caches of the offer the bid is for. This way, our offer teaser gets updated wherever it is visible in the system whether it is on the entity detail page or in a views listing.

We got an insight into the various possibilities of caching in drupal 9. I dare to say the caching system is one of the most advanced out there. Once you get comfortable, it is set up very rapidly.

Dispatch custom Events with an EventSubscriber to redirect users

👉 In this chapter more about the **Event** system in drupal. Moreover you get an insight into how to add an EventSubscriber and how to *dispatch the event*. At the end you will be able to define and catch certain events and attach actions to them.

There are still a few things left unfinished about our software. Our homepage still redirects to our users page. We set the standard homepage to our offer page at /admin/config/system/site-information. Fill in “/offer” at the default front page textfield. This way, we get the overview with the offers as our frontpage.

We’ve told earlier that drupal is moving away (slowly) from the hook system. We will learn how to redirect users on login with an EventSubscriber.

From the symfony documentation:

Symfony triggers several events related to the kernel while processing the HTTP Request. Third-party bundles may also dispatch events, and you can even dispatch custom events from your own code.

For dispatching own events, we have to add an Event and an EventSubscriber service to our module.

We start with adding an Event at **custom/offer/src/Event/UserLoginEvent.php**:

```php
<?php
```

```
namespace Drupal\offer\Event;

use Drupal\user\UserInterface;
use Symfony\Contracts\EventDispatcher\Event;

/**
 * Event fired when a user logs in.
 */
class UserLoginEvent extends Event {

  const EVENT_NAME = 'offer_user_login';

  /**
   * The user account.
   *
   * @var \Drupal\user\UserInterface
   */
  public $account;

  /**
   * Constructs the object.
   *
   * @param \Drupal\user\UserInterface $account
   *   The account of the user logged in.
   */
  public function __construct(UserInterface $account) {
    $this->account = $account;
  }

}
```

This is simply telling our system this event exists (in the same way we added files to the Ajax directory). Next we have to add a 'subscriber' which handles the aftermath of when the event is triggered, like a redirection or a log:

To a new file **offer.services.yml** we add:

```
services:
  offer_login.event_subscriber:
    class: Drupal\offer\EventSubscriber\UserLoginSubscriber
    arguments: [ '@path.matcher', '@current_user' ]
    tags:
      - { name: event_subscriber }
```

To a new folder we add
custom/offer/src/EventSubscriber/UserLoginSubscriber.php:

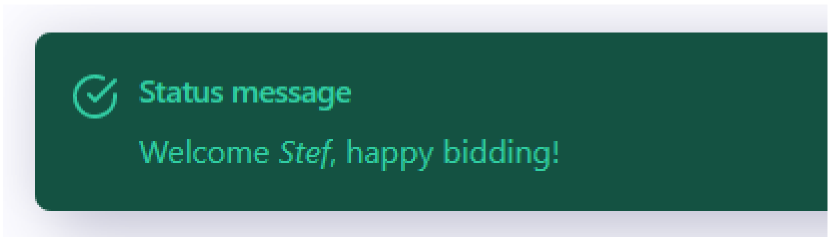
```
<?php

namespace Drupal\offer\EventSubscriber;

use Drupal\offer\Event\UserLoginEvent;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpFoundation\RedirectResponse;

/**
 * Class UserLoginSubscriber.
 *
 * @package Drupal\custom_events\EventSubscriber
 */
class UserLoginSubscriber implements EventSubscriberInterface
{
  /**
   * {@inheritdoc}
   */
  public static function getSubscribedEvents()
  {
    return [
      UserLoginEvent::EVENT_NAME => ['onUserLogin', 29]
    ];

  }

  /**
   * Subscribe to the user login event dispatched.
   *
   * @param \Drupal\custom_events\Event\UserLoginEvent $event
   *   Dat event object yo.
   */
  public function onUserLogin(UserLoginEvent $event)
  {
    $username = \Drupal::currentUser()->getDisplayName();
    \Drupal::messenger()->addStatus(t('Welcome %name, happy bidding!', [
      '%name' => $username,
    ]));
    $response = new RedirectResponse("/");
    $response->send();
  }
}
```

An event subscriber is a PHP class that's able to tell the dispatcher exactly which events it should subscribe to. It implements the Symfony\Component\EventDispatcher\EventSubscriberInterface interface, which requires a single static method called getSubscribedEvents() which shows the priority of each listener method. The higher the number, the earlier the method is called. Note that we added a higher number than dynamic page cache, otherwise the system could miss the event!

The last thing we need to do is telling the system *when* to dispatch the event. In our case it is when a user logs in. Therefore we hook into the user login form in **custom/offer/offer.module**:

```
/*
* Implements hook_user_login()
*/
function offer_user_login($account) {
  // Instantiate our event.
  $event = new UserLoginEvent($account);
  // Get the event_dispatcher service and dispatch the event to fire the
event.
  $event_dispatcher = \Drupal::service('event_dispatcher');
  $event_dispatcher->dispatch(UserLoginEvent::EVENT_NAME, $event);
}
```

Make sure to clear caches the first time to make sure everything is registered correctly. Next time you login, we get redirected to the frontpage and get a welcoming message:


Status message

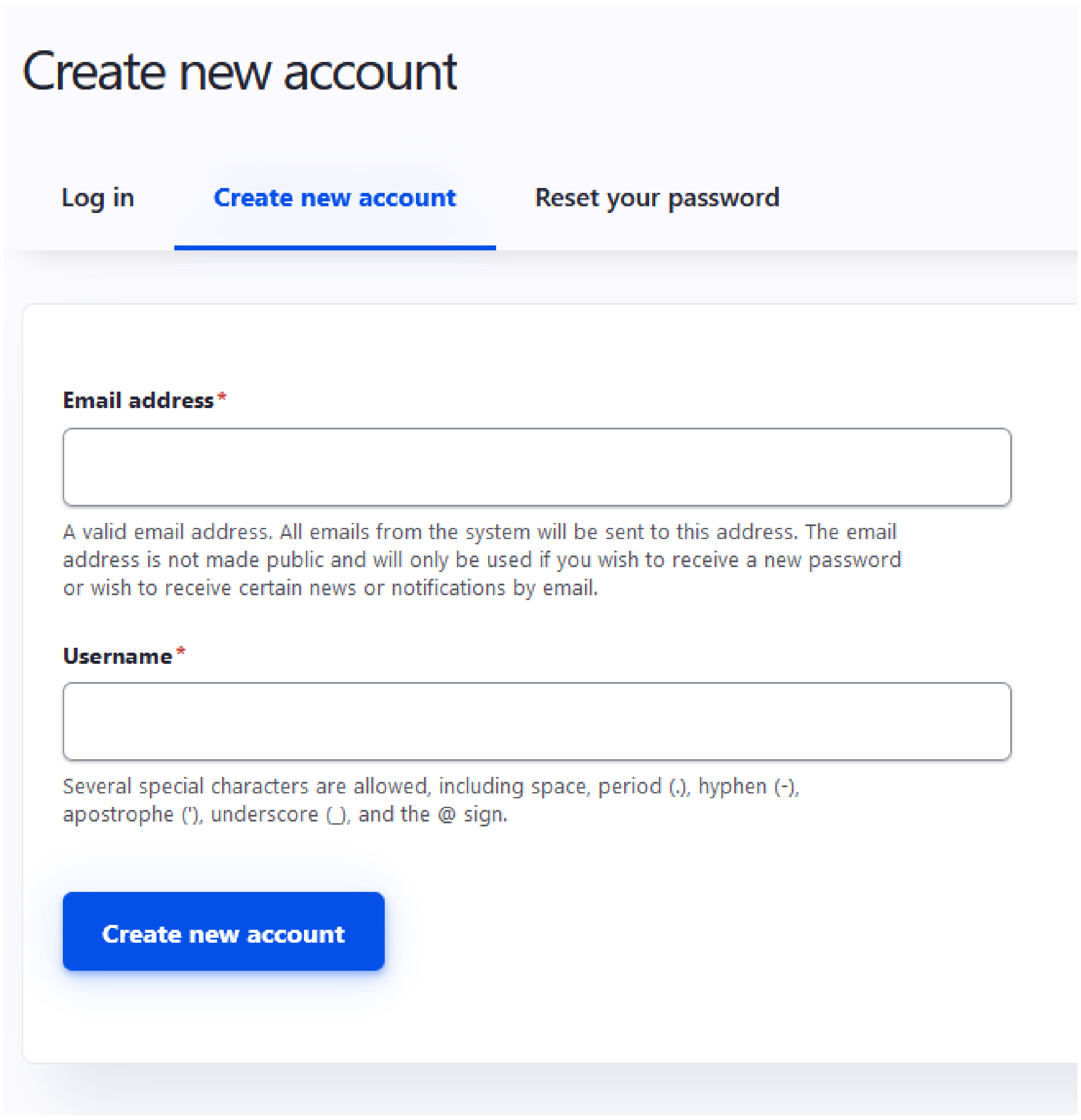
Welcome *Stef*, happy bidding!

We learned to create custom Events and how to subscribe to them. The nice thing about custom events is that we can fire them whenever we would like to. In this case an event on logging in, but we could also fire logging events or e-mail events on multiple occasions. A very powerful feature of drupal 9, with as special thanks to symfony!

Customize the user registration process with a RouteSubscriber

With all the things we learned so far, it is interesting to take a look at the core user module. You'll find out that users are nothing more than an entity. The core registration, login and password forms are entity forms with a bit of custom logic attached.

The core user register form looks like this:



Create new account

Email address*

A valid email address. All emails from the system will be sent to this address. The email address is not made public and will only be used if you wish to receive a new password or wish to receive certain news or notifications by email.

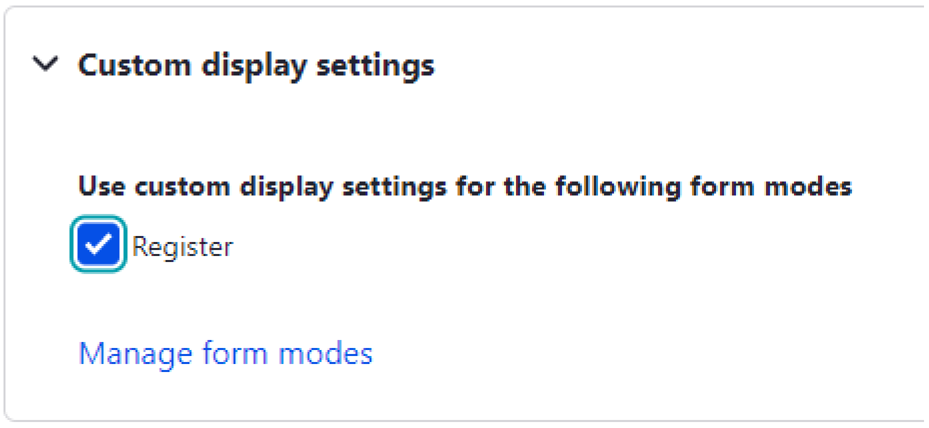
Username*

Several special characters are allowed, including space, period (.), hyphen (-), apostrophe ('), underscore (_), and the @ sign.

This form is not exactly what we wanted. We'd like a registration with name, e-mail address and password only. I want to on-board users more rapidly and worry about verification later. How do we change this?

Using a typical drupal hook *hook_form_alter()* could do the job, but is tricky in my opinion because we do not have control on how the default form will evolve in the future. Well, it's not *that* tricky I guess but I prefer to keep control in code. Thus, a safer option would be to show a different form on this page.

At /admin/config/people/accounts/form-display we can change the entity forms display. A view mode for 'register' (the registration page) is present. Check the box and save. This way, we can build the registration form.

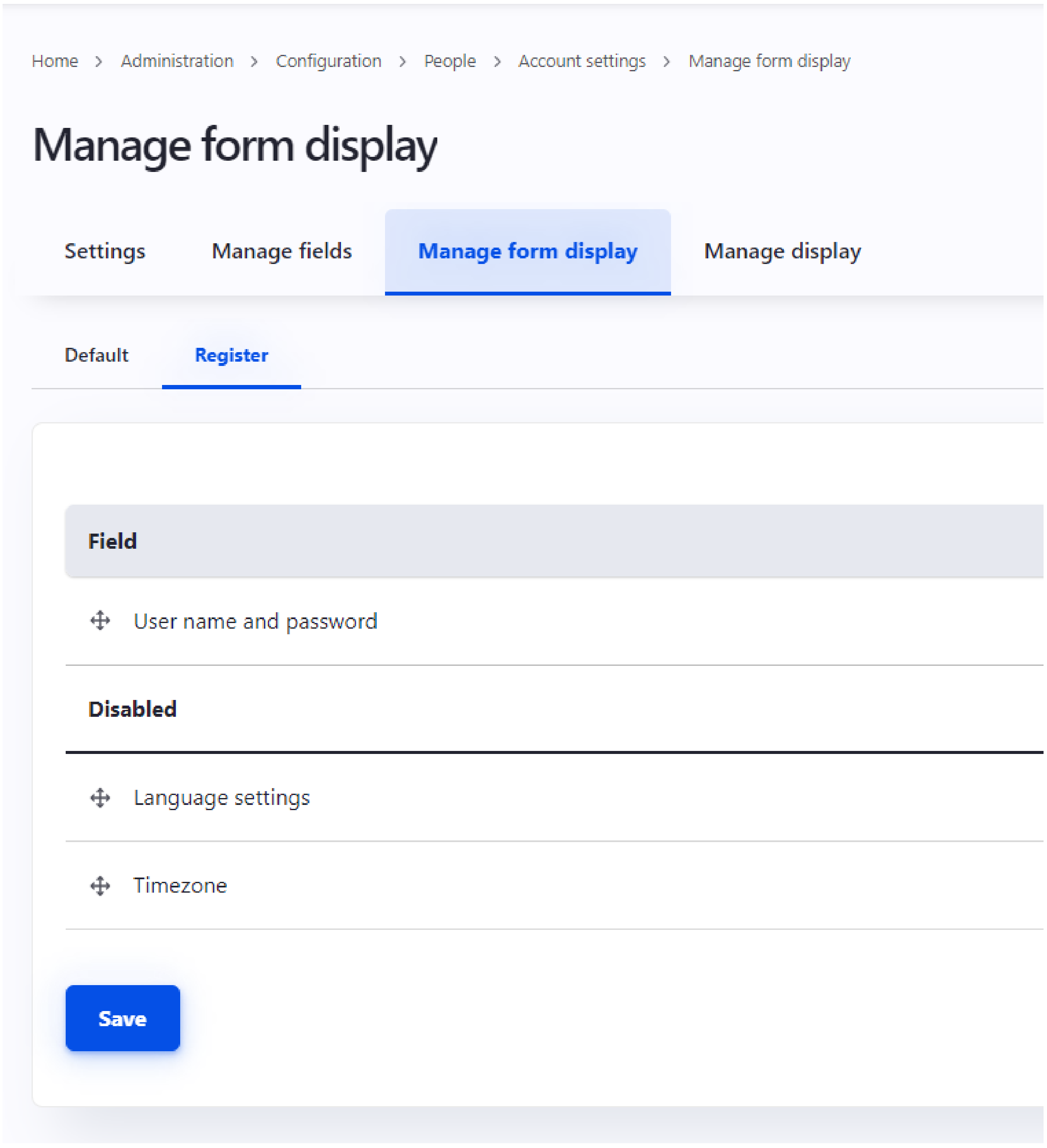


Custom display settings

Use custom display settings for the following form modes

[✓] Register

This results in the following form settings at /admin/config/people/accounts/form-display/register:


Home > Administration > Configuration > People > Account settings > Manage form display

Manage form display

Field
User name and password
Disabled
Language settings
Timezone

Drupal core combines User name and password in one form field, and what we want is a form that asks first for an e-mail and then automatically offers access to the platform. We want to grant a 1-hour access before the user has to validate it's address.

Seems like something to do with a custom form instead of an entityform. We add **custom/offer/src/Form/RegistrationForm.php:**

```php
<?php
namespace Drupal\offer\Form;
```

```php
use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
use Drupal\user\Entity\User;
use Drupal\Component\Utility\Xss;

class RegistrationForm extends FormBase {

  /**
   * @return string
   *   The unique string identifying the form.
   */
  public function getFormId() {
    return 'offer_registration_form';
  }

  /**
   * Form constructor.
   *
   * @param array $form
   *   An associative array containing the structure of the form.
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *   The current state of the form.
   * @param \Drupal\offer\Entity\Offer $offer
   *   The offer entity we're viewing
   *
   * @return array
   *   The form structure.
   */
  public function buildForm(array $form, FormStateInterface $form_state,
$offer = NULL) {

    $form['email'] = [
      '#type' => 'email',
      '#attributes' => array(
        ' type' => 'email', // note the space before attribute name
      ),
      '#title' => $this->t('Your email address'),
      '#required' => TRUE,
    ];

    $form['username'] = [
      '#type' => 'textfield',
      '#attributes' => array(
        ' minlength' => 2, // note the space before attribute name
      ),
      '#title' => $this->t('Your name'),
      '#required' => TRUE,
    ];

    $form['password'] = [
      '#type' => 'password',
      '#attributes' => array(
        ' type' => 'password', // note the space before attribute name
        ' minlength' => 8
      ),
      '#title' => $this->t('Your password'),
      '#description' => $this->t('Should be minimum 8 characters.'),
      '#required' => TRUE,
    ];

    // Group submit handlers in an actions element with a key of
"actions" so
    // that it gets styled correctly, and so that other modules may add
actions
    // to the form. This is not required, but is convention.
    $form['actions'] = [
      '#type' => 'actions',
    ];

    $form['actions']['submit'] = [
      '#type' => 'submit',
      '#value' => $this->t('Register'),
    ];

    return $form;

  }

  /**
   * Validate the input values of the form
   *
   * @param array $form
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *
   */
  public function validateForm(array &$form, FormStateInterface
$form_state)
  {
    parent::validateForm($form, $form_state);
    // Server side validation for email
    if
(!\Drupal::service('email.validator')->isValid($form_state->getValues()[
'email'])) {
      $form_state->setErrorByName('Email', $this->t('Email address is
not a valid.'));
    }

    // Check if username exists
    $user_exists =
user_load_by_name(Xss::filter($form_state->getValues()['username']));
    if(!empty($user_exists)) {
      $form_state->setErrorByName('username', $this->t('An account with
this username already exists.'));
    }

    // Check if email exists
    $ids = \Drupal::entityQuery('user')
      ->condition('mail',
Xss::filter($form_state->getValues()['email']))
      ->range(0, 1)
      ->execute();
    if(!empty($ids)){
      $form_state->setErrorByName('email', $this->t('An account with
this email address already exists.'));
    }

    // check if pass = minimum 8 characters server-side
    if(strlen($form_state->getValues()['password']) < 8) {
      $form_state->setErrorByName('password', $this->t('Minimum length
of password needs to be 8 characters.'));
    }

  }

  /**
   * Form submission handler.
   *
   * @param array $form
   *   An associative array containing the structure of the form.
   * @param \Drupal\Core\Form\FormStateInterface $form_state
   *   The current state of the form.
   */
  public function submitForm(array &$form, FormStateInterface
$form_state) {
    $user = User::create();
    $user->enforceIsNew();
    $user->setEmail($form_state->getValues()['email']);
    $user->setUsername($form_state->getValues()['email']); //This
username must be unique and accept only a-Z,0-9, - _ @ .

$user->setPassword(Xss::filter($form_state->getValues()['password']));
    $user->activate();
    $user->save();
    user_login_finalize($user); // logs a new session etc.
    // This will redirect with UserLoginEvent
  }

}
```

This form asks for a name, mailing address and password and then creates the user and automatically logs the user in.

Note: on a production website we would have to verify the email of the user and probably add a reCaptcha spam detection. We will skip this in this course.

But how do we get this form at the core /user/register place? We can dispatch a RouteSubscriber event for this. It allows custom code to listen for dynamic routing events.

Any route - whether statically defined in a YAML file, as seen in the introductory example, or a dynamic route as described in Providing dynamic routes - can be altered. You can do so by modifying a RouteCollection using an EventSubscriber triggered by the RoutingEvents::ALTER event.

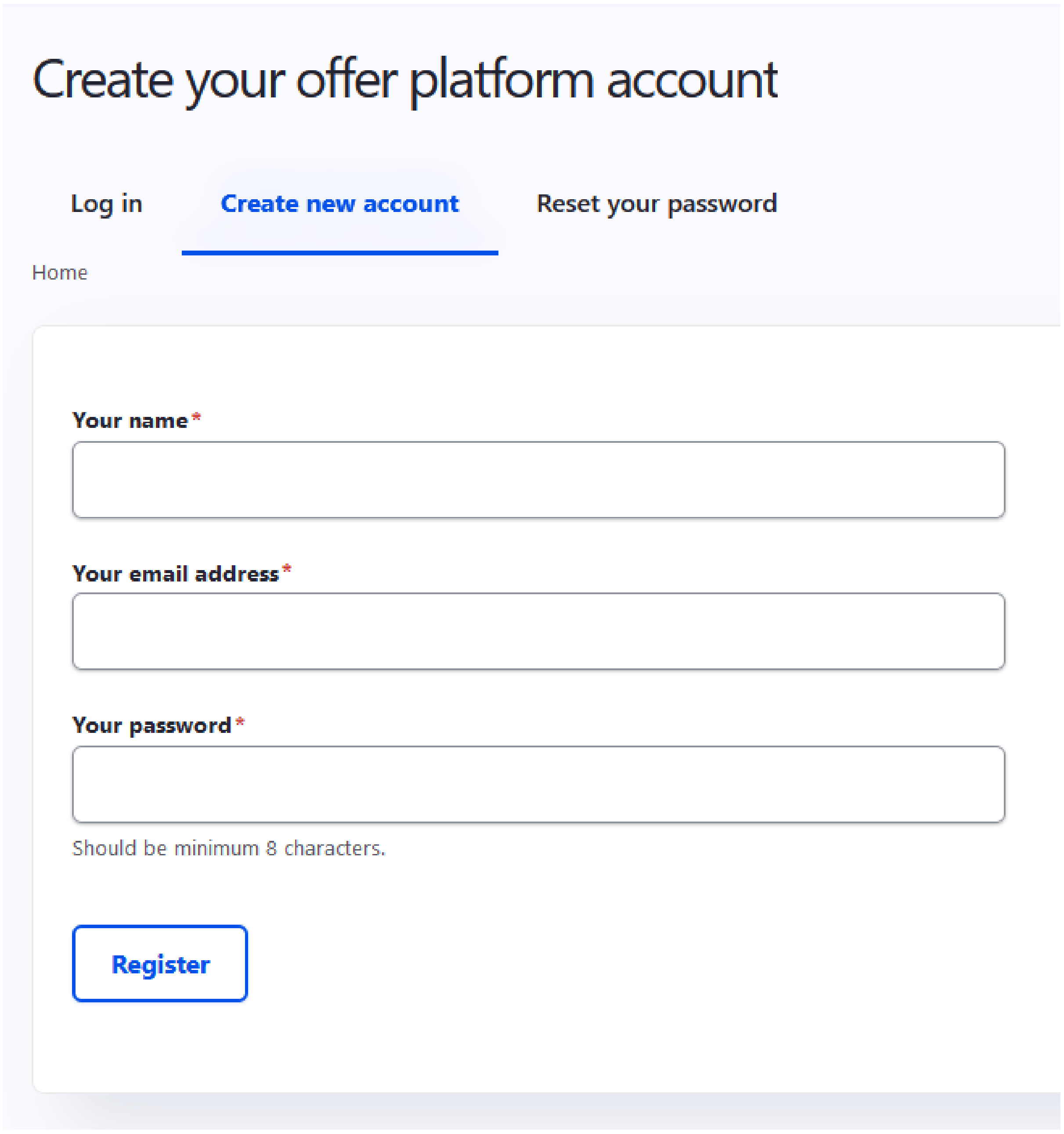
We add a new service to **custom/offer/offer.services.yml:**

```
services:
  ... // already 1 service here
  offer.route_subscriber:
    class: Drupal\offer\Routing\RouteSubscriber
    tags:
      - { name: event_subscriber }
```

Now, add the file **RouteSubscriber.php** to **custom/offer/src/Routing:**

```php
<?php
namespace Drupal\offer\Routing;

use Drupal\Core\Routing\RouteSubscriberBase;
use Symfony\Component\Routing\RouteCollection;

/**
* Listens to the dynamic route events.
*/
class RouteSubscriber extends RouteSubscriberBase {

 /**
   * {@inheritdoc}
   */
 protected function alterRoutes(RouteCollection $collection) {

   $entityUserRegisterFormRoute = $collection->get('user.register');
   if ($entityUserRegisterFormRoute) {
     $entityUserRegisterFormRoute->setDefaults([
       '_form' => '\Drupal\offer\Form\RegistrationForm',
       '_title' => 'Create your offer platform account',
     ]);
   }

 }
}
```

If we clear caches, our custom form appears on the /user/register route:



Create your offer platform account

Your name*

Your email address*

Your password*

Should be minimum 8 characters.

This is powerful stuff. With this routeSubscriber we can basically override *any* of the default routes in drupal core. We basically changed the behaviour of registered routes in drupal.

The *validateForm()* and *submitForm()* methods guard our input and make sure the user does not exist yet. Because we already listen to the login event (remember this from a previous tutorial), the new user gets redirected to the homepage automatically.

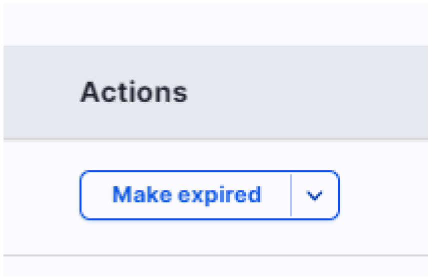
For our platform, this 1-step log in will on-board many more users than the core user register form.

We could attach an Event to send out an email for verifications and so forth. This leads us too far for this course but is something you would do on a software in production. In the meantime you could let users have a bid and put an *unverified* label to their profile.

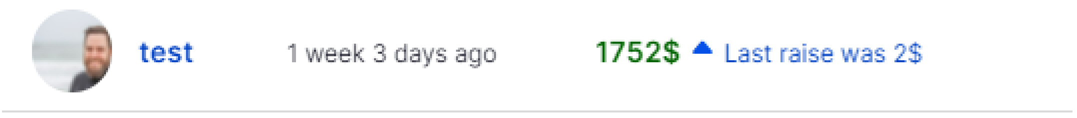
Finishing up the platform

This course was quite a ride. While this stays just a course and the software is not ready for production use I added the following stuff that isn't covered because it is either repetitive stuff from earlier chapters or it leads too far:

- I added a *RedirectProfilePageToProfileSubscriber* **Event Subscriber to redirect the profile page to the edit page of the user**. You'll find out you need this quite a lot! Note: for this platform, user profiles wouldn't be a bad thing though not the focus of this course. Tip: to make this page a bit more production-ready, remove unnecessary fields at /admin/config/people/accounts/form-display
- I added an **OfferExpiredController** just like the *OfferPublishController*. This is used to set the moderation state from published to expired. Affected files are offer.routing.yml to add a new route, OfferDynamicOperationLInks plugin to add the link. Result is this:



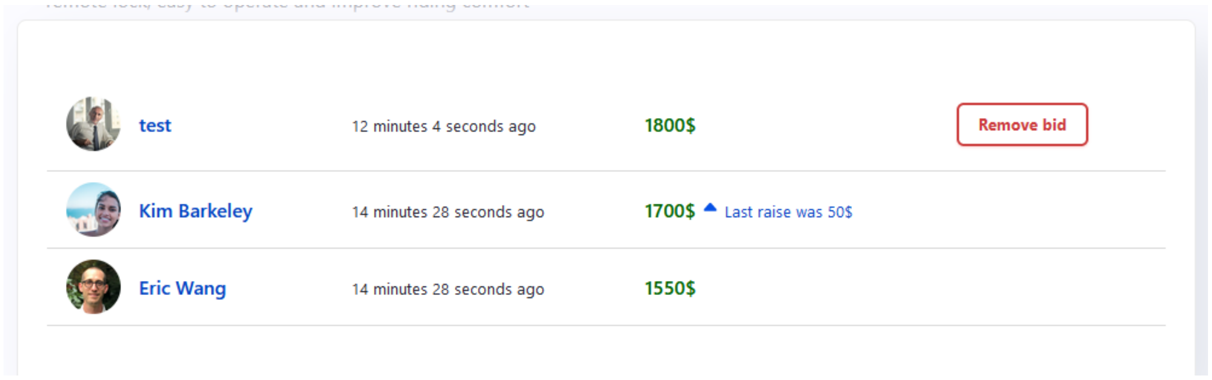
- **I've created a view mode 'compact' for the users**, added a profile image and added them to the offer teaser and full template, and to the bid table. A good example on how to render a custom entity. I spiced it up a little with css. Take a look at **offer.module** and **user--compact.html.twig** to see how I did it. The trick was to alter the template suggestion so that we could override the twig of the user module. Last step was to edit the **OfferBiddingTable** to add the user teasers. This is how the integration looks:



- Copy the **SeedDataGenerator.php** from the final platform code to yours. It is extended with some extra users and offers. Remove your own test users, offers (drush entity:delete offer) with drush and create your offers and users like in the [Seed Data chapter](#):

  ```
  drush offer-create-seeds
  ```

  Log in with username “test” and password “test” and add a profile image. Place a bid on an offer and it will look like this:

test	12 minutes 4 seconds ago	1800$	Remove bid
Kim Barkeley	14 minutes 28 seconds ago	1700$ ▲ Last raise was 50$	
Eric Wang	14 minutes 28 seconds ago	1550$	

These things were left unfinished:

- Every modern platform should have a responsive lay-out with a mobile-first approach. This was far from the purpose of this course. We stayed away from theming. Maybe I’ll create a course about this later! Personally I like building custom themes with Tailwind css.
- The registration form should contain some sort of spam prevention. Take a look at some ReCaptcha modules on drupal.org to do the trick.
- On a production platform we’d use cron jobs to do some tasks (and automated drupal cleanup tasks). hook_cron() is used for this.

💻 Download the latest project files here:
https://stefvanlooveren.me/download/8cb8b68a0242ac130003?i=g

Also, check out the changelog and subscribe to the mailing list at https://stefvanlooveren.me/courses/drupal-9-framework to stay updated about changes and new chapters in the future.