# 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!

**[THIS IS A FREE SAMPLE FROM CHAPTER 3: Custom entities 101, CRUD operations, workflow states and access]**

## 3.2 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:

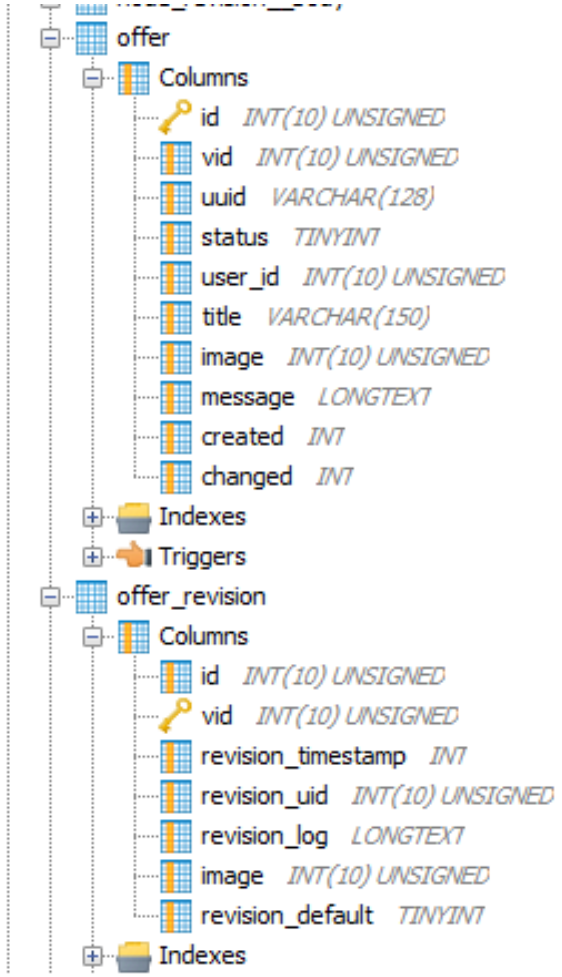
```
<?php
/**
* @file
* Contains \Drupal\offer\Entity\Offer.
*/

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:




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) {
```

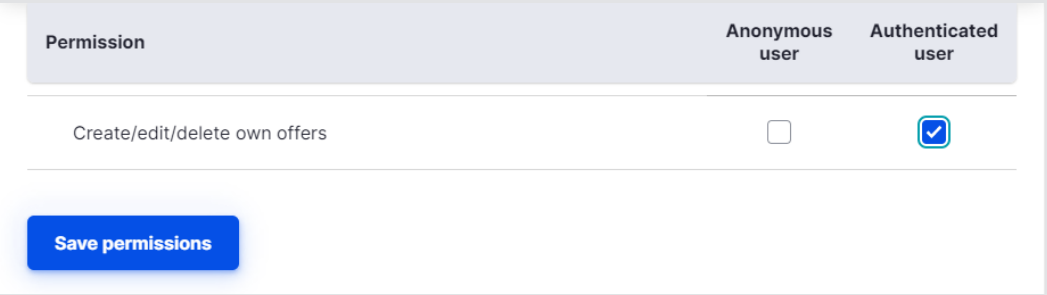
```
   $access = AccessResult::forbidden();

   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.




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!