

# 1

## Preparation

### 1.1 Ensure PHP is installed on your computer

### 1.2 Ensure the Composer PHP command line tool is installed on your computer

Type `composer` in a command line terminal. If you get an error saying no such application, then install Composer from:

- <https://getcomposer.org/doc/00-intro.md#installation-windows>

### 1.3 Ensure the SQLite PHP database extension is enabled (required for Windows)

If Symfony is working, you have PHP setup on your computer. However, the SQLite database extension may not be setup.

You can either work ahead, hoping it is setup, and fix it if you hit a problem when trying to create a database. Or you can check, and fix it now.

PHP extensions are already installed with PHP, but may not be activated. All we have to do is ensure there is no semi-colon character `;` at the beginning of the line `'extension=php_pdo_sqlite.dll'`.

Do the following:

1. Use Notepad++ to open file `C:\php\php.ini`
2. Search for the line `'extension=pdo_sqlite'`
  - use `<CTRL>-F` to search
3. If the line reads `;extension=php_pdo_sqlite.dll` remove the semi-colon `;` at the beginning of the line and save the changed file.
4. That's it - you have now enabled SQLite for PHP applications.

If you want to enable the MySQL database as well, then do the same for the line saying `'extension=php_pdo_mysql.dll'`.

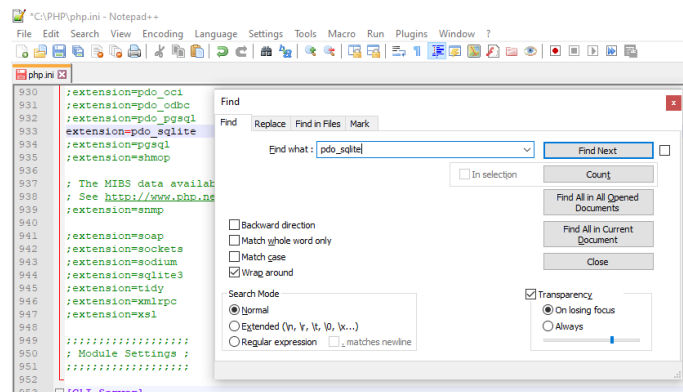


Figure 1.1: SQLite being enabled in php.ini in the Notepad++ editor.

## 1.4 Install the PHPStorm code editor

1. Get your free one-year subscription to JetBrains products using your **TU Dublin** university email address
  - <https://www.jetbrains.com/shop/eform/students>
2. Download and install the PHPStorm editor

## 1.5 Download project template and open in a code editor

1. Download to the Desktop and unzip folder `'crud1.zip'` from Moodle
2. Start your IDE editor (e.g. Notepad++ or PHPStorm)
3. Open a terminal window (either a Terminal application like `cmd`, or open a Terminal window inside your IDE)

4. In the terminal `cd` into folder `crud1`

## 1.6 Run the Symfony web sever

Let's run the web server on our machine (`localhost:8000`) by entering terminal command:

```
php bin\console server:run`.
```

Note, you might get some warnings/info messages about version of PHP etc. - just ignore them!

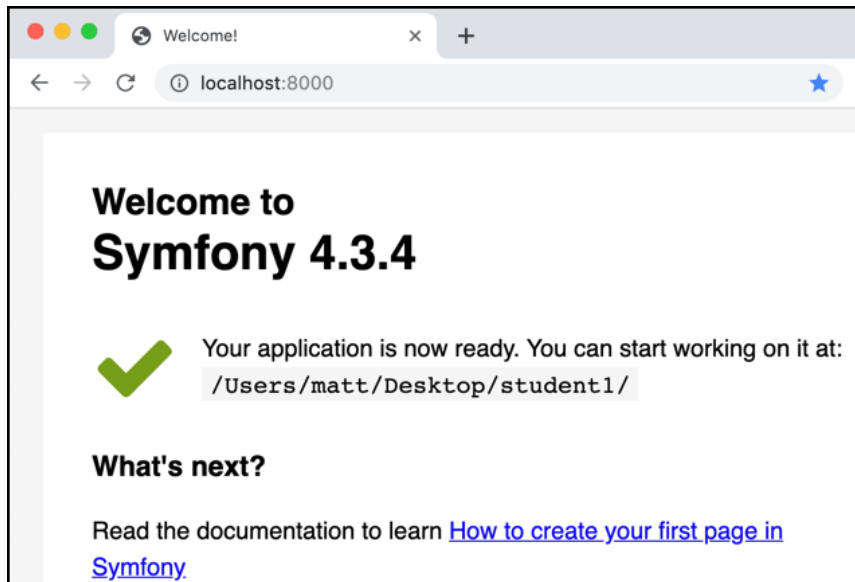
```
> php bin\console server:run
Sep 27 07:22:19 |DEBUG| PHP    Using PHP version 7.3.8 (from default version in $PATH)
Sep 27 07:22:19 |INFO | PHP    listening path="/usr/local/php5-7.3.8-20190811-205217/sbin/php-fpm"
Sep 27 07:22:19 |DEBUG| PHP    started
Sep 27 07:22:19 |INFO | PHP    ready to handle connections
```

```
[OK] Web server listening on http://127.0.0.1:8000 (PHP FPM 7.3.8)
```

## 1.7 Visit the home page localhost:8000

Open a web browser and visit our website home page at `http://localhost:8000`.

Since we didn't create a home page, we'll see a default Symfony home page. See Figure ?? shows a screenshot of PHPStorm and our new class PHP code.



# CRUD for a Student

entity class

## 1.8 Create Student class

Let's create a Student class and generate automatic CRUD web pages.

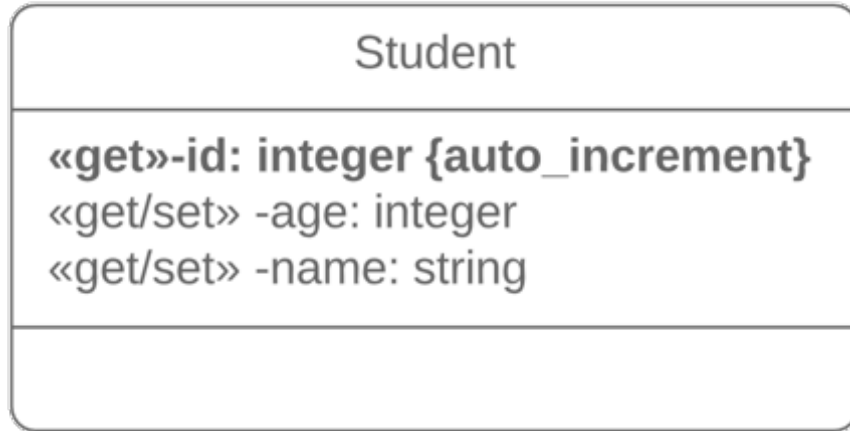


Figure 1.2: Class diagram for Student entity.

Do the following:

1. At the command line type:

```
php bin/console make:entity Student
```

You should see the following:

```
> php bin/console make:entity Student
```

```
created: src/Entity/Student.php
```

```
created: src/Repository/StudentRepository.php
```

```
Entity generated! Now let's add some fields!
```

```
You can always add more fields later manually or by re-running this command.
```

- notice that it tells us that it has created 2 new classes `src/Entity/Student.php` and `src/Repository/StudentRepository.php`
  - `Student.php` is a simple class with private properties and getters/setters, with special 'annotation' comments so these objects can map directly to rows in a database table...

2. Now we need to ask this console 'make' tool to add an integer `age` property for us:

- we need to enter the property name `age`
- we need to specify its data type `integer`

- (to keep things simple) we don't mind if our properties start as null (just press <RETURN> for this question)
- you should see the following when answering these questions at the `make` command tool prompt:

```
New property name (press <return> to stop adding fields):
```

```
> age
```

```
Field type (enter ? to see all types) [string]:
```

```
> integer
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

```
updated: src/Entity/Student.php
```

3. Now add a string `name` property:

- we need to enter the property name `name`
- we need to specify its data type `string` (since default just press <RETURN>)
- accept default string length of 255 (since default just press <RETURN>)
- can be nullable (since default just press <RETURN>)
- you should see the following when answering these questions at the `make` command tool prompt:

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
> name
```

```
Field type (enter ? to see all types) [string]:
```

```
>
```

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

```
updated: src/Entity/Student.php
```

4. That's all our fields created, so just press <RETURN> to complete creation of our entity:

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

&gt;

Success!

Next: When you're ready, create a migration with `make:migration`

## 1.9 Take a look at the created entity class

Take a look at what's been created for us: `src/Entity/Student.php`. If you ignore the comments, mostly this is a class

See Figure 1.3 shows a screenshot of PHPStorm and our new class PHP code.

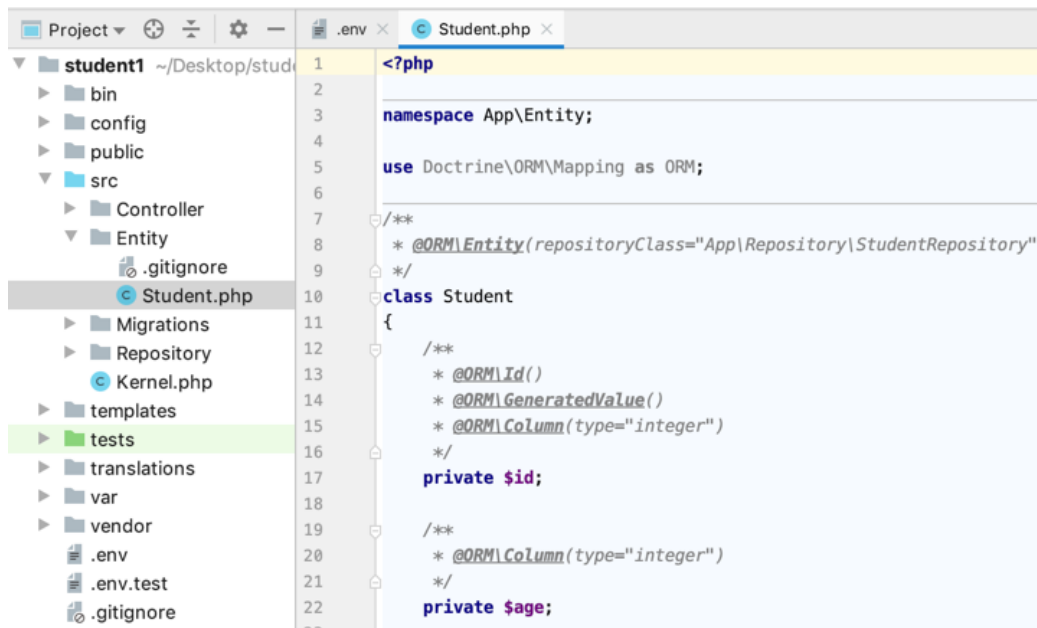


Figure 1.3: New Student.php entity.

## 1.10 Make 'migration' SQL to create database to correspond to our entity class

We can also use the 'make' command line tool to look at our classes and create the SQL commands we need to update our database create/update tables for storing the object data in tables and rows.

Enter the following at the command line `php bin/console make:migration`:

```
> php bin/console make:migration
```

Success!

Next: Review the new migration `"src/Migrations/Version20190927055812.php"`

Then: Run the migration with `php bin/console doctrine:migrations:migrate`

See <https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html>

If you look inside the newly created file you'll see a line like this showing the SQL generated to create a database table to match our `Student.php` class:

```
$this->addSql(
    'CREATE TABLE student (
        id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
        age INTEGER NOT NULL,
        name VARCHAR(255) NOT NULL
    )'
);
```

## 1.11 Execute our ‘migration’ SQL to create/update database

Now let's tell Symfony to connect to the database and execute the migration SQL - to actually **create** the new `Student` table in the database.

We need to enter the terminal command `php bin/console doctrine:migrations:migrate:`

```
> php bin/console doctrine:migrations:migrate
```

Application Migrations

```
WARNING! You are about to execute a database migration that could result in schema changes and data loss.
Are you sure you wish to continue? (y/n)
```

At this point we must enter `y` to go ahead - saying we are happy for our database structure to be changed by executing our migration SQL:

```
WARNING! You are about to execute a database migration that could result in schema changes and data loss.
Are you sure you wish to continue? (y/n)y
```

```
Migrating up to 20190927055812 from 0
```

```
++ migrating 20190927055812
--> CREATE TABLE student (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
```

```
age INTEGER NOT NULL, name VARCHAR(255) NOT NULL)

++ migrated (took 64.4ms, used 18M memory)
-----
++ finished in 70.8ms
++ used 18M memory
++ 1 migrations executed
++ 1 sql queries
```

That's it - we have now created a table in our database to match our PHP entity class.

## 1.12 Generate the CRUD web form for class Student

Let's generate some HTML and PHP code for a web form to list and create-read-update-delete data from our database.

We need to execute this command to create that code `php bin/console make:crud Student`:

```
> php bin/console make:crud Student

created: src/Controller/StudentController.php
created: src/Form/StudentType.php
created: templates/student/_delete_form.html.twig
created: templates/student/_form.html.twig
created: templates/student/edit.html.twig
created: templates/student/index.html.twig
created: templates/student/new.html.twig
created: templates/student/show.html.twig
```

Success!

Next: Check your new CRUD by going to `/student/`

## 1.13 Visit our generated Student crud pages at `/student`

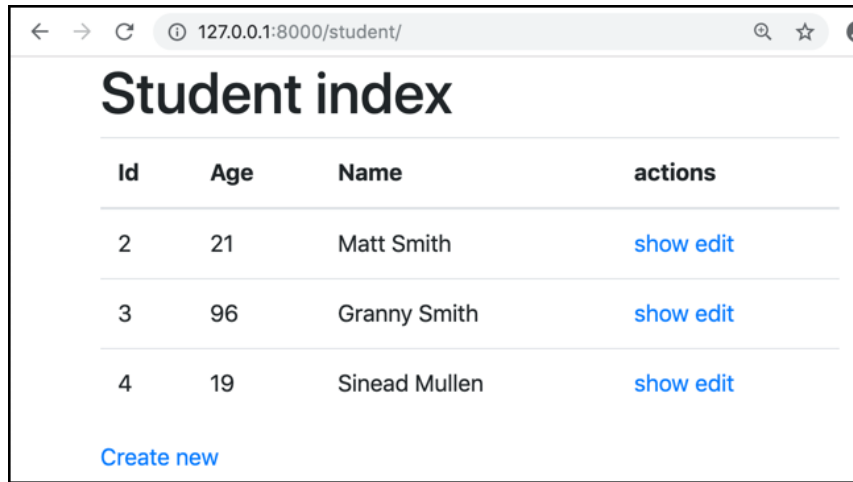
Let's visit our generated CRUD pages, these can be found by adding `/student` at the end of the URL.

Click **Create new** and add a student. Then try clicking edit, and change some values or delete it and create it again.

You should find you have a fully working web-based CRUD interface to your database.



See Figure 1.4 shows a screenshot of several students having been created (and yes, one of my grandmothers did live to 96!).



Id	Age	Name	actions
2	21	Matt Smith	<a href="#">show edit</a>
3	96	Granny Smith	<a href="#">show edit</a>
4	19	Sinead Mullen	<a href="#">show edit</a>

[Create new](#)

Figure 1.4: Screenshot of nice looking Bootstrap style admin CRUD pages.

## 1.14 Databases are persistent

Kill the Symfony web server at the command line by pressing `<CTRL>-C`. Then quit the PHPStorm IDE application.

You could also go have a cup of coffee, or perhaps shut down and restart your computer.

Then restart the PHPStorm editor, and restart the web server with `php bin\console server:run`.

Now open a web browser to URL `http://localhost:8000/student` and you should see that the students you created in your database are still there.

Any changes we make are remembered (persisted) as part of our `var/data.db` database file.



# 2

## Adding a campus entity and relating them

### 2.1 Create a new class: Campus

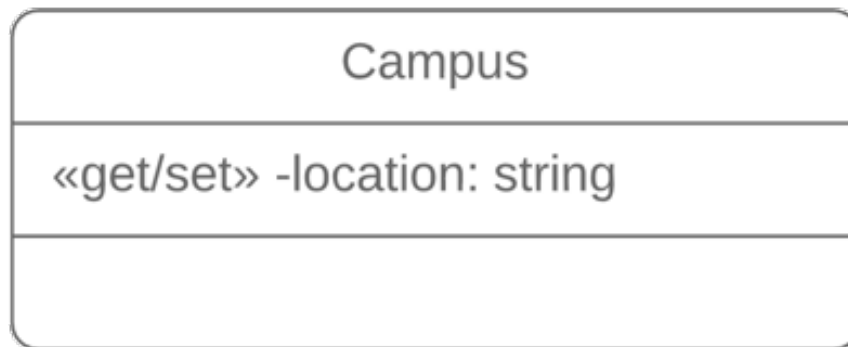


Figure 2.1: Class diagram for Campus class.

Create entity Campus with single property 'location' (string) add a `__toString()` method to Campus class (`src/Entity/Campus.php`) containing the following

```
public function __toString()
{
    return $this->location;
}
```

(we'll need this `toString` method in Campus later, so that when creating/editing Students we can choose the related Campus object from a drop down menu - which needs a string description of

each Campus)

## 2.2 Create CRUD for this Campus class

generate CRUD for Campus

```
php bin\console make:crud Campus
```

## 2.3 Create relationship between Student and Campus (each student linked to one campus)

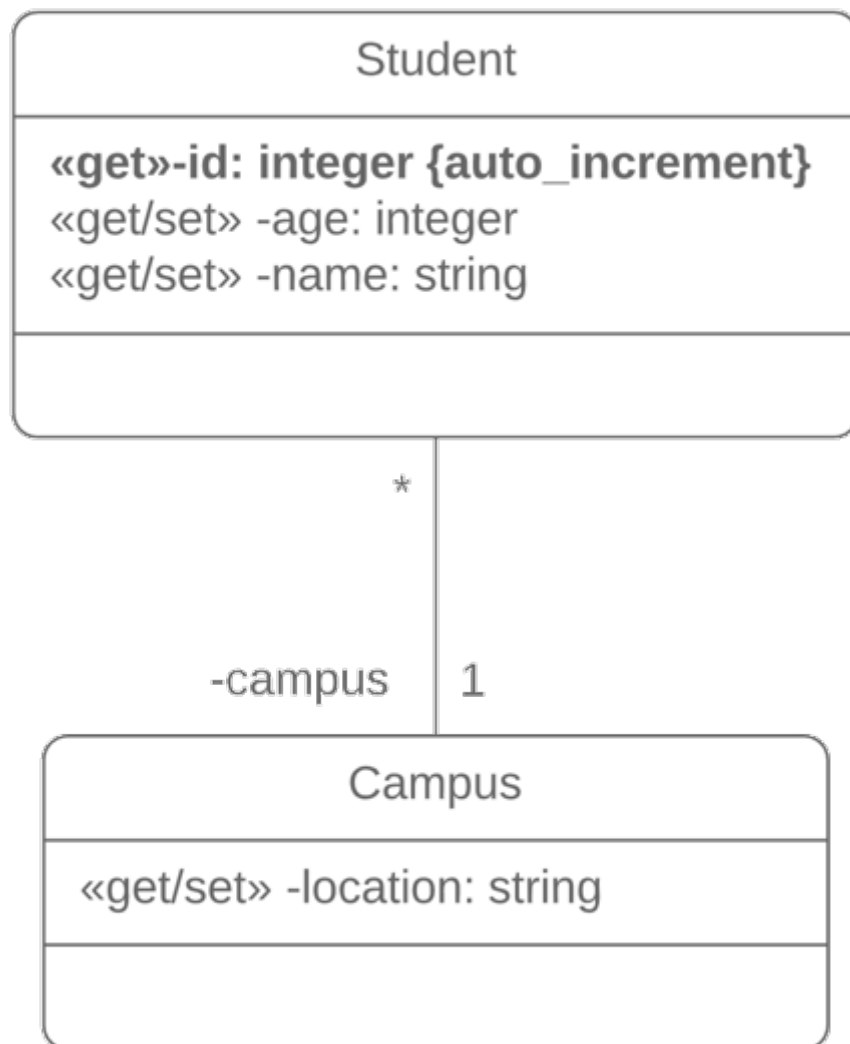


Figure 2.2: Class diagram for Student-Campus multiplicity.

To create this relationship we are going to add a ‘campus’ property to the Student class, that is a reference to a Campus object. Here is our detailed new Student class diagram:

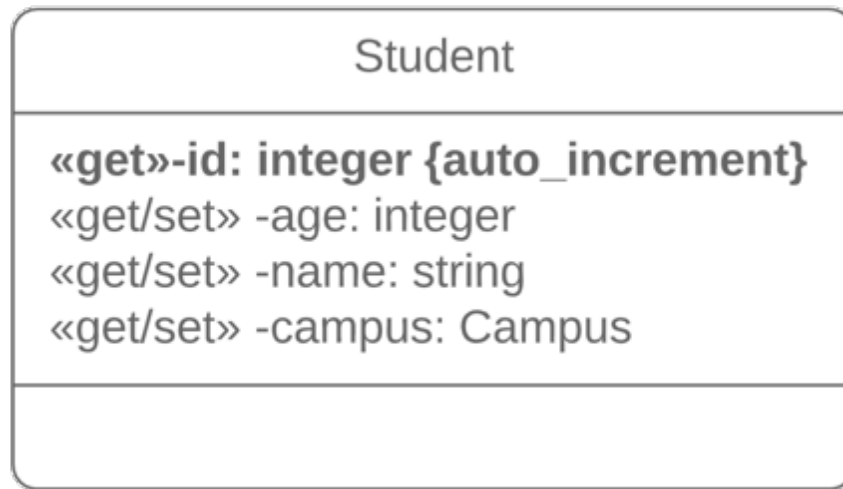


Figure 2.3: Detailed Student class diagram.

Here is how to add a related property to a class:

add a property ‘campus’ to the Student class, of type relation, that is ManyToOne to the Campus class i.e. many students linked to one campus to ADD a property to an existing class, we need to run the `make:entity` console command again:

```
php bin\console make:entity Student
```

the console should see the entity already exists, and invite us to add a new property...

## 2.4 update Database Structure (since we changed our classes)

1. Create and run new DB migration
2. Create a migration by typing:

```
php bin\console make:migration
```

3. now run the migration by typing:

```
php bin\console doctrine:migrations:migrate
```

## 2.5 Delete old CRUD and generate new CRUD for both classes

1. delete the old Student CRUD
  - FILE: `src/Controller/StudentController.php`
  - FILE: `src/Form/StudentType.php`
  - folder: `templates/student`
2. generate CRUD for Student

```
php bin\console make:crud Student
```

## 2.6 Run server add some related records:

1. now run server:
 

```
php bin\console make:crud Student
```
2. visit the Campus CRUD pages and create record for 2 campuses
  - e.g. Blanch and Tallaght)
3. visit the Student CRUD pages, and edit / create Student's related to your new Campuses

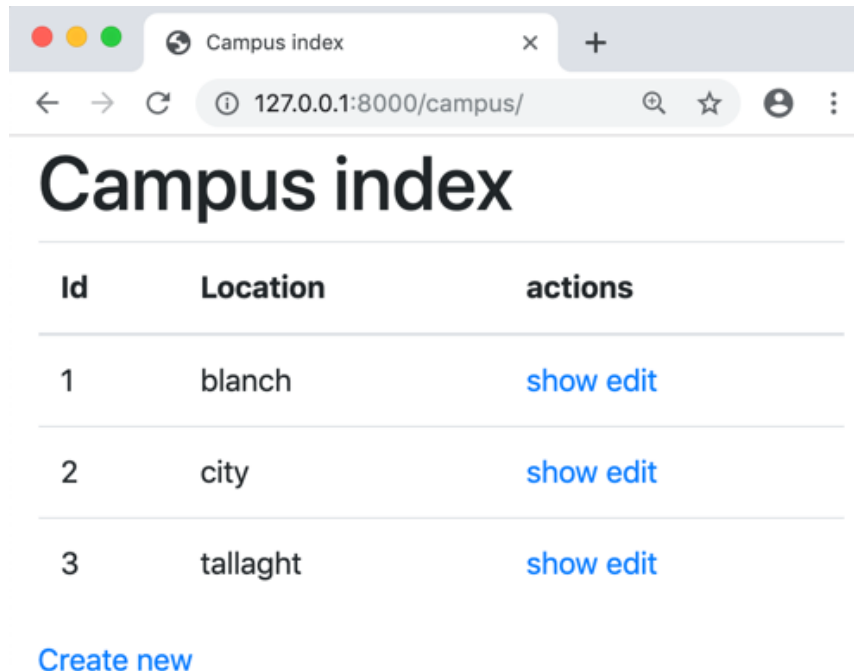
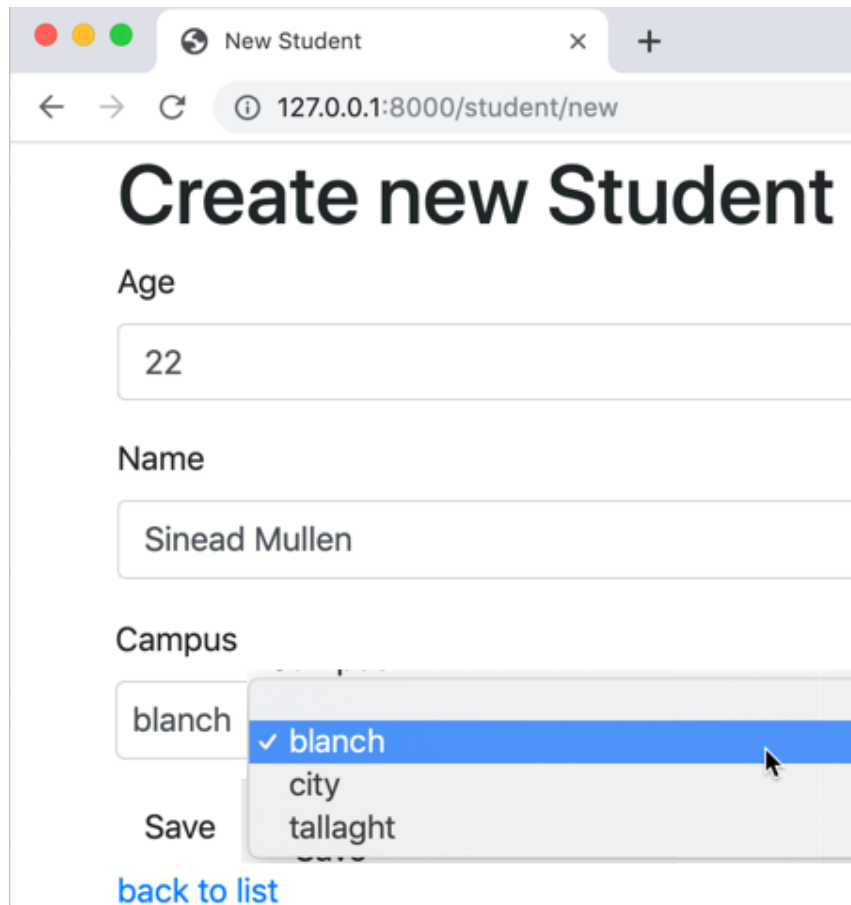


Figure 2.4: Screenshot showing list of Campus objects.

When we create/edit a student, we now get a dropdown menu of the Campus objects (the text in the dropdown menu is from the `__toString()` method we created for the Campus class).



The screenshot shows a web browser window with the title 'New Student'. The address bar displays '127.0.0.1:8000/student/new'. The main heading is 'Create new Student'. Below this, there are three form fields: 'Age' with the value '22', 'Name' with the value 'Sinead Mullen', and 'Campus' with the value 'blanch'. The 'Campus' dropdown menu is open, showing a list of options: 'blanch' (which is selected and highlighted in blue), 'city', and 'tallaght'. Below the form fields, there is a 'Save' button and a 'back to list' link.

Figure 2.5: Screenshot showing new Student form with Campus choice dropdown menu





# 3

## Customising the Twig templates

### 3.1 Let's add a new Campus column to our list of students

We need to edit file: `/templates/student/index.html.twig`

See Figure 3.1 for the location of this file in the project folder structure.

Figure 3.2 shows the 2 places where we need to edit.

Do the following:

1. Edit file: `/templates/student/index.html.twig`
2. Insert a new HTML table column header for `Campus`
3. Insert a new HTML table data item for `student.campus`

Figure 3.2 shows how the file should look now - after the lines have been inserted.

And when we visit `/student` we should see a campus column added to the student details - see Figure 3.4.

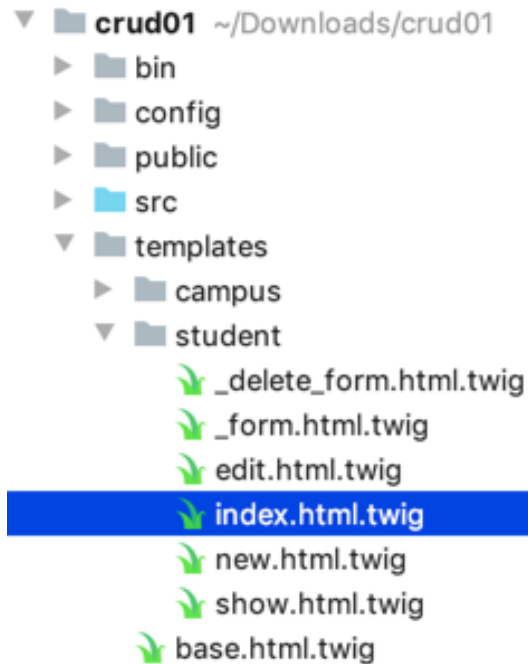


Figure 3.1: Location of Twig template files.

```
<table class="table">
  <thead>
    <tr>
      <th>Id</th>
      <th>Age</th>
      <th>Name</th>
      <th>actions</th>
    </tr>
  </thead>
  <tbody>
    {% for student in students %}
      <tr>
        <td>{{ student.id }}</td>
        <td>{{ student.age }}</td>
        <td>{{ student.name }}</td>
        <td>
          <a href="{{ path('student_show', {'id':
          <a href="{{ path('student_edit', {'id':
        </td>
      </tr>
    {% else %}
      <tr>
```

**Add column header** (arrow pointing to the `<th>Name</th>` line)

**Add campus name** (arrow pointing to the `<td>{{ student.name }}` line)

Figure 3.2: Where we will insert lines into template.

```
<table class="table">
  <thead>
    <tr>
      <th>Id</th>
      <th>Age</th>
      <th>Name</th>
      <th>Campus</th>
      <th>actions</th>
    </tr>
  </thead>
  <tbody>
    {% for student in students %}
      <tr>
        <td>{{ student.id }}</td>
        <td>{{ student.age }}</td>
        <td>{{ student.name }}</td>
        <td>{{ student.campus }}</td>
        <td>
```

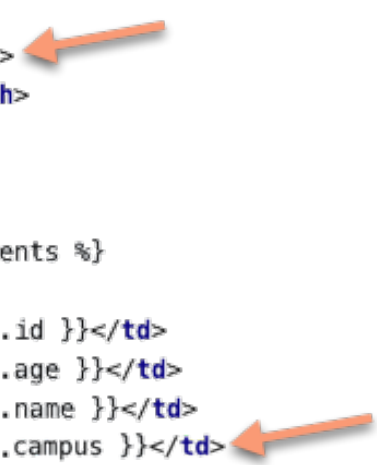


Figure 3.3: Twig template after lines inserted.

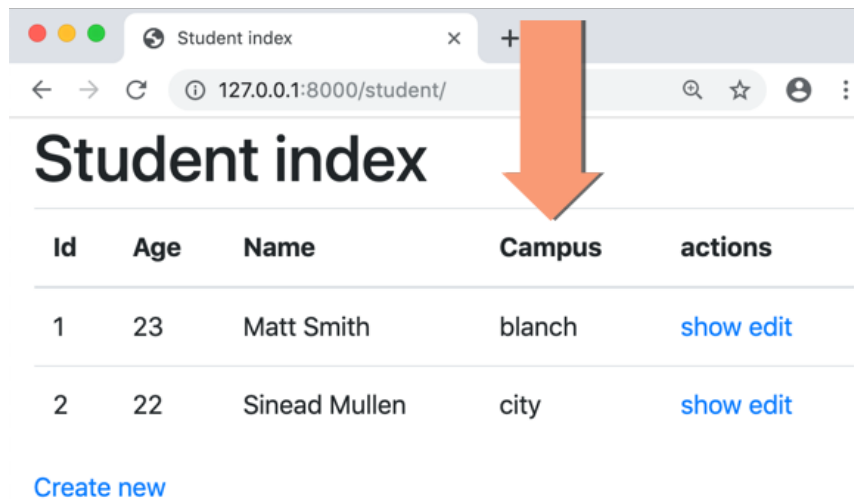


Figure 3.4: CRUD list page with extra column.

## 3.2 Turn the Campus name into a LINK to the related Campus object

We can wrap an HTML hyperlink element around the campus name, to connect our Student object to its related Campus object

Here's the edit we need to add to file: `/templates/student/index.html.twig`

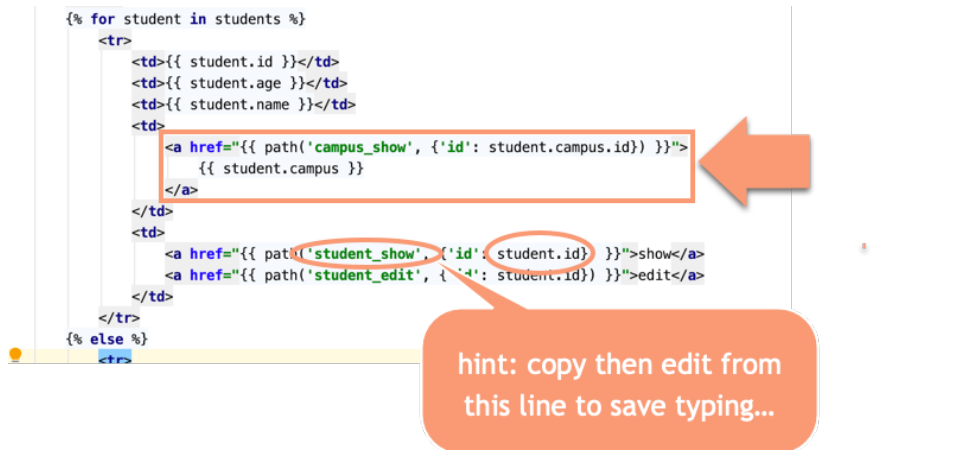


Figure 3.5: Where to edit to turn campus name into hyperlink.

When that's done, we can now click the campus and jump to the Campus objects 'show' page:

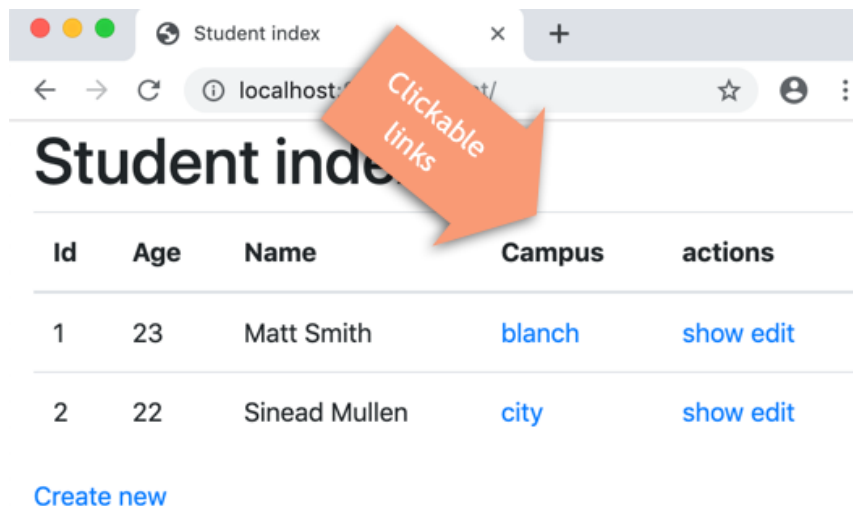


Figure 3.6: Web page where campus is clickable link to Campus object.