Progetto Symfony - Qualche *whereabouts*

PM, NTA , ABATO, a.a. 25-26

mario chiari

11 aprile 2026

SEMPRE IN BOZZA - GRATO COMMENTI E CORREZIONI REFUSI

1 Installazione

Primo obiettivo è installare Symfony sulla vostra macchina. Seguite con attenzione le istruzioni a https://symfony.com/download, per il vostro SO (ovviamente collegati a Internet, e forse disattivando momentaneamente antiVirus e simili):

ATTENZIONE: La versione corrente di Symfony è la 7.3. Consiglio di installare la 7.3. Mi sembra sia arrivata in produzione la versione 8.1, se volete provatela, ma con la 7.3 andiamo più sicuri.

ATTENZIONE: non possedendo macchine Win o Mac, non posso fare prove con quei SO (e nemmeno con MAMP, di cui su macchina Linux non ho bisogno). Ma per problemi, al 99% trovate tutte le soluzioni on line, cercando un minimo.

Per Win seguite quanto suggerito (cercate di capire i vari tool che si usano, e sopratutto chi fa che cosa):

- installazione di una terminale considerato più performante e credo recente, che dovrebbe essere già installato sulla vostra macchina Win, vedi alcune spiegazioni su i terminali di Win qui, altrimenti PowerShell dovresti trovarlo qui con istruzioni per installarlo;
- muovetevi nella directory del vostro utente (non so se necessario, ma credo meglio);

- installazione di Scoop, un tool *installatore* che aiuta a installare pacchetti su Win ("*Scoop installs programs you know and love, from the command line with a minimal amount of friction*"). Vedi le due righe di comando con cui installi Scoop, dal terminale PowerShell;

- installazione di `symfony-cli`, da terminale PowewrShell, invocando Scoop: `$ scoop install symfony-cli`. Osserva che `symfony-cli` **non** è il pacchetto Symfony con cui sviluppiamo il nostro progetto, ma uno strumento sviluppato dalla Symfony (intesa come ente/gruppo di sviluppatori), per facilitare l'installazione (da riga di comando) di Symfony: "*The Symfony CLI is a free and open source developer tool to help you build, run, and manage your Symfony applications directly from your terminal*", vedi a symfony-cli.

  Per capire i comandi che vi mette a disposizione `symfony-cli`, lanciate `symfony --help` (di fatto ne utilizzeremo solo alcuni, ma di grande aiuto);

- installazione di Symfony, da riga di comando, indicando il nome del progetto di applicativo web che vi piace (potete lasciare il nome `my_project`): `$ symfony new --webapp my_project`;

- controllate che sia stata creata una cartella `my_project` (o con il nome che avete scelto). Questa è la cartella del vostro *sito*, o applicativo web. In particolare, il codice php con cui saranno create *on the fly* le pagine del sito (quelle che si vedono lato client) si trova (sopratutto, ma non solo) nella sotto-cartella (meglio, directory) `src`, le pagine sono create mediante le *template* che si trovano nella directory `templates`;

- tra le altre cose, con `symfony-cli` avete installato anche un tool che comprende un server http, che potete attivare con il comando `symfony server:start --port=8089` (o altra porta che sapete libera). Lato client, alla *url* `localhost:8089` dovreste vedere la pagina di welcome della vostra web app Symfony. Se si, lo scheletro della web app è installato e funzionante. GOOD!!!

- a https://symfony.com/doc/current/setup.html
  avete qualche istruzione di installazione più diretta, senza appoggiarvi su `symfony-cli`. Se volete potete tentare, utile per capire meglio come funziona.

- ricordatevi che Symfony è un framework `php`, quindi dovete essere sicuri di aver installato php (meglio: l'interprete php). Alla url del punto precedente trovate alcune richieste sulla versione di php necessaria. Per chi ha installato MAMP, trovate `php`, in varie versioni:

cercate sotto credo la cartella *bin*, se dovete scegliere, scegliete (il percorso a) la versione più recente/alta;

- come per ogni tool informatico, anche `php` ha delle estensioni che possono/devono essere installate separatamente (anche se in genere suite come MAMP installano tutto quello che serve). Composer è un tool che aiuta a installare estensioni di `php`, gestendo conflitti e aggiornamenti, se ti incuriosisce leggi qui (`symfony-cli` dovrebbe aver installato Composer silenziosamente, altrimenti dovete installarlo direttamente). Inoltre con Composer installiamo estensioni (o librerie) utili per estendere le funzionalità di Symfony, in genere con un comando come
  `composer require symfony/ai-agent`. Composer crea un file che contiene le info sulle estensioni utilizzate nel tuo progetto Symfony, cerca dove è `composer.json` e capisci che informazioni contiene.

Per Mac dovrebbe essere anche più semplice. Il terminale è già performante. Come tool installatore vedo suggerito homebrew ("*Homebrew installs the stuff you need that Apple didn't*"), con questo installate `symfony-cli`: `brew install symfony-cli/tap/symfony-cli`, e quindi con `symfony-cli` installate Symfony esattamente come in ambiente Win:
`symfony new --webapp my_project`. Come per Win, avete bisogno di Composer.

2 MVC - Model, View, Controller
CRUD - Create, Read, Update, Delete

Su MVC. Il codice di Symfony è organizzato secondo il paradigma, o l'architettura (*architectural design pattern*), **MVC - Model–View–Controller**.

- per una intro generale a MVC - con anche un poco di storia - leggete la voce wikipedia Model–view–controller. In bibliografia link ad articoli classici che hanno introdotto queste idee. Capisco (ma non ho controllato le info storiche), che un informatico norvegese, Trygve Reenskaug, sia considerato il padre di MVC. Le note della pagina wikipedia a lui dedicata ha link ai suoi articoli pionieristici, di fine anni '70 secolo scorso (quindi idee in giro da oltre 45 anni). Considerate che MVC fu formulato primariamente per progettare GUI;

- On line si trovano diversi post, in genere brevi, che cercano di illustrare cosa si intenda MVC in parole semplici e con qualche diagramma, per esempio questo, di un certo Robert Sheldon, è piuttosto chiaro. In particolare lo sviluppo di Symfony mi sembra seguire quello che Sheldon indica come *Linear approach to MVC model*, la sua figura 2;

  Attenzione: Vedo molte introduzioni a MVC, ma spesso per illustrare come questo o quel tool sia stato sviluppato seguendo la metodica MVC. Per es. per il mondo Microsoft, vedo MVC ASP.NET e Overview of ASP.NET Core MVC.[1] Sconsiglio però di perdersi in letture su altri tool; quando avrete una qualche familiarità con MVC in Symfony, potrebbe allora essere interessante il confronto con altri sviluppi;

- Symfony ha una presentazione su come si riorganizza (e riscrive) un codice php di una web app secondo il paradigma MVC e quali ne sono i vantaggi, leggete con una qualche attenzione: Symfony versus Flat PHP;

- In un post del 2011, What is Symfony2?, il principale/iniziale sviluppatore di Symfony, il francese Fabien Potencier, presentava alcune sue considerazioni su Symfony, MVC e problematiche affini; scritto ai tempi della versione 2, forse superato, ma da tenere presente.

  Nota: Questo il più recente intervento di Fabien Potencier che sono riuscito a trovare. Non so bene di che cosa stia parlando, qualcosa di IA. Qui un poco di *pubblicità* francese;

- Ancora utile consultare A Gentle introduction to Symfony, anche se scritta ai tempi della versione 2, e il codice nel frattempo si è evoluto.

Symfony e MVC - Un esempio. Per un primissimo orientamento su la struttura MVC di Symfony, e di come capirla per riuscire a sviluppare una web app impiegando le funzionalità messe a disposizione dal framework, immaginiamo un progetto di massima: si voglia sviluppare un blog, in modo da permettere a degli **utenti** di pubblicare dei **post**, e inoltre ad ogni utente di scrivere un **commento** a un post.

Rispetto a un tale progetto, vediamo dove situare le componenti Model, View e Controller.

[1] ASP.NET è il framework per creare web app sviluppato da Microsoft. Un mondo a parte. Google (AI Overview) mi sostiene che sia popolare e utilizzato in circa il 33% della web app in giro per il mondo. Non ho idea di come si possa accertare l'attendibilità di tali info.

Per sviluppare il blog così come lo abbiamo progettato in linea di massima, dobbiamo predisporre il codice per gestire i dati relativi a gli **utenti**, i **post** e i **commenti**. Gli utenti saranno individuati, diciamo, da un nomecognone e da una email; i post da autore, titolo, testo, e mettiamo anche una data di scrittura; i commenti dal post di cui sono il commnento, e poi anch'essi da autore, titolo, testo e data. Si osservi a futura memoria che vi sono delle relazioni tra dati: i post hanno per autore un utente del blog, e i commenti lo sono di post già presenti. Tutto ciò costituisce, o meglio verrà gestito da la componente **Model** della nostra web app. In breve si può dire che la componente Model riguarda i dati della web app, come sono composti e archiviati, e anche le relazioni tra di loro. *The Model represents the information on which the application operates.*

Ovviamente la web app si presenta come un sito web, ovvero produce pagine - o visualizzazioni - web: per esempio una pagina con un modulo per inserire un nuovo post, e forse una anche per editare o eliminare un post, i moduli per inserire, editare e forse eliminare i commenti, ovviamente una visualizzazione di tutti i post in qualche ordine, e probabilmente una visualizzazione per ogni singolo post con i commenti relativi, ecc.. Tali visualizzazione sono gestite - non sarà una sorpresa - dalla componente **View**. *The View renders the model into a web page suitable for interaction with the user.*

Infine, meno evidente ma fondamentali, si deve considerare le funzionalità che raccolgono le richieste lato client, compreso i dati inseriti nei previsti moduli, e le gestiscono agendo per l'archiviazione dei dati e l'invio di una risposta al client. Nel gergo MVC, queste funzionalità sono i **Controller**. *The Controller responds to user actions and invokes changes on the model or view as appropriate.*

Su CRUD. Consideriamo di nuovo l'esempio del blog. Consideriamo delle operazioni basilari rispetto alla gestione dei post: (*) permettere di scrivere un post, con titolo, testo, data di creazione, ecc.; (*) permettere di visualizzare i post, forse in ordine di titolo, o di data di creazione; (*) permettere di modificare il contenuto di un post, forse per correggere dei refusi, o per aggiungere del testo; (*) in qualche caso, permettere di cancellare un post.

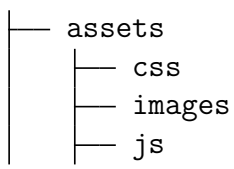
I post del blog - qui immaginati composti da titolo, testo e data di scrittura - sono i dati da archiviare. Le quattro operazioni sono quelle di creare (**C**reate), leggere (**R**ead), aggiornare (**U**pdate) e cancellare (**D**elete) tali dati. Nel gergo sono note come operazioni CRUD. A queste quattro operazioni corrispondono, nel linguaggio SQL con cui si formulano le query al db, corrispondenti query: INSERT INTO, SELECT FROM, UPDATE,

DELETE FROM (le sto scrivendo malamente, vedremo meglio la sintassi corretta quando avremo un minimo di conoscenza di SQL).

Symfony, MVC e CRUD. Anticipo alcuni elementi di come tutto ciò è realizzato in Symfony:

- componente **Model**. I dati gestiti dalla nostra web app sono archiviati in un database mySQL (si potrebbe usare altro tool, ma noi useremo mySQL, in particolare quello che avete installato con la suite AMP), e quindi, continuando con lo stesso esempio usato poc'anzi, dovremo avere un db con tabelle opportune per utenti, post e commenti. Il punto importante è che Symfony ci evita di creare direttamente tali tabelle. Piuttosto richiede che siano definite delle **Entity** che formalizzano le caratteristiche dei dati che la web app deve gestire. Sulla base di tali Entity, Symfony creerà e gestirà le tabelle del db necessarie in modo opportuno. Inoltre `symfony-cli` comprende delle funzionalità che ci permettono di creare in modo semiautomatico - semplicemente indicando quali tipo di dato desideriamo avere - gli script php che definiscono tali Entity;
- componente **View**. Per le visualizzazione, Symfony comprende un cosidetto *template engine* chiamato twig. Per capire a cosa ciò serve nella programmazione della nostra web app, e come permette di ottimizzare la produzione delle pagine html da inviare al client, bisogna capire cosa sia una *template*. Vedremo con un esempio come funziona;
- componente **Controller**. Infine Symfony è predisposto per facilitare la definizione dei controller. Di nuovo `symfony-cli` comprende delle funzionalità che permettono in particolare di definire i controller corrispondenti alle funzionalità CRUD per ciascuna Entity del progetto.

Con queste prime nozioni, potete iniziare a capire come sia organizzata la cartella del progetto `my_project` (o altro nome, se ne avete scelto un altro), che si dovrebbe presentare così (nella vostra installazioone ci potrebbero esserci delle differenze):



```
│   └──styles
├── bin
├── config
│   ├── packages
│   └── routes
├── migrations
├── public
│   └── build
├── src
│   ├── Controller
│   ├── Entity
│   ├── Form
│   ├── Repository
│   └── Security
├── templates
│   ....
├── tests
├── translations
├── var
│   ├── cache
│   └── log
└── vendor
  ....
```

La directory (cartella) a cui prestare più attenzione è `src`. Gli scripts per definire le Entity devono essere situati nella directory `Entity`, quelli per definire i controller in `Controller`. In `Form` abbiamo degli script per definire eventuali moduli di inserimento (per produrre `<form> </form>` html, e quindi di aiuto alla componente View della web app), e in `Repository` la definizione di query diverse e in genere più complesse delle semplici query CRUD, query che poi possono essere richiamate da opportuni controller (quindi di aiuto alla componente Controller della web app).

Nella directory `template` abbiamo le template. Come vedrete sono (in genere) file con estensione `.html.twig`. Osservate che se definite a Entity e create i corrispondenti controller per le funzionalità CRUD, qui verrano create le template (minimali) per le corrispondenti visualizzazioni, un file template per ciascuna visualizzazione, i.e., per ciascuna operazione CRUD.

Nelle varie sotto directory di `assets` sono salvati le risorse per la resa grafica delle varie visualizzazioni: file `.css`, `.js`, dei vari formati grafici come `.jpg`, possibilmente `.pdf`, ecc.

Sotto `vendor` trovate le numerose librerie che costituiscono Symfony, da non modificare, e che semmai potete tenere aggiornate con le funzionalità di `composer update` (ricordate che Symfony è un framwork php, e Composer è un tool per installare, ma anche aggiornare, librerie/estensioni `php`. Potete configurare come svolgere tali aggiornamenti dal file `composer.json`).

Le altre directory le presentiamo quando ne sapremo un poco di più.

3 Intermezzo

(ad oggi 23/11/2025)

- Semplice ma utile per iniziare a orientarsi, riproducete la creazione di questo primo controller (senza Model e View minimale): Create your First Page in Symfony
- Come un secondo progetto (quindi in una directory diversa) installate sulla vostra macchina il demo di Symfony - che crea un blog - come pubblicizzato a Introducing the Symfony Demo application. Le istruzioni a quella pagina sono però obsolete (del 2015). Il demo lo trovate con istruzioni che (a me) hanno funzionato - quella della opzione 1 - a demo blog a github (forse dovete crearvi un account github). Potete così curiosare nel codice e iniziare a capire come funziona una web app Symfony di media potenza.
- trovate le istruzioni per configurare `.env` (senza non funziona, sono gli elementi di config minima per permettere al progetto di collegarsi al database). Per prima cosa, in mySQL, via interfaccia myphpadmin o da linea di comando, dovete creare un db vuoto, nominatelo Symfony. In `.env`, alla riga
  `DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"`,
  dovete sostituire le credenziali per accedere a mySQL, considerando quelle che avete di default nella vostra installazione mySQL,[2] le info sulla versione di tale installazione, e il nome del db che avete creato;

[2] Attenzione, non id+pw con cui accedete al vostro OS, e neppure quelle che potremmo creare per accedere al back end del nostro progetto Symfony, ma quelle per accedere a mySQL, di default dovrebbero essere root:root, ma meriterebbe creare un utente specifico mySQL con propria pw. Se mai spostaste la web app su un server di produzione (nel mondo reale, per così dire), avremmo id+pw del db dateci dal provider.

- esplorate i comandi a vostra disposizione con `symfony console` (dalla directory del vostro progetto, o anche con comando `php bin/console`, se `php` è nel path corretto). Tra i comandi listati, considerate quelli della serie `make`, e in particolare `make:entity` e `make:crud`, che aiutano a creare gli script per definire le Entity e i Controller CRUD per la vostra web app, vedi prossima sezione;
- ricordate che per progettare la vostra web app, dovrete quasi sempre saper progettare un area Back End, riservata a redattori/amministratori. Per Back End si intende una porzione del sito web (pagine, visualizzazione) riservate a una particolare categoria di visitatori/utenti, ovvero accessibili solo con una qualche forma di autentificazione (dietro a un login). Dobbiamo capire come Symfony predispone i meccanismi per facilitare la creazione di quel che serve.

4 Come creare un web app con Symfony.

Immaginiamo di volere realizzare un web app per gestire la relazione docente / studente dell'Accademia. Vogliamo definire le due Entity di *docente* e di *studente*. Per ogni docente e ogni studente, vogliamo salvare qualche informazione di base (nome, matricola, disciplina, ecc.) e sopratutto di quale docente uno studente ha seguito uno o più corsi. Voi pensate a un esempio similare, e realizzatelo per familiarizzarvi su come programmare Symfony. Nel seguito indico solo alcuni passaggi che spero sufficienti per illustrare la procedura da seguire.

4.1 Creare una web app con `make:entity`, `make:crud`

In linea di principio, si dovrebbe scrivere del codice a mano, e in particolare del codice che rispetti le specifiche richieste dal framework, nel senso che, per esempio, il codice per definire una Entity deve essere scritto in certi modi, includendo la defizione di variabili e funzionalità secondo certi criteri. Per nostro aiuto Symfony ci mette a disposizione, come già detto, un *gadget* che produce il codice necessario in modo semi automatico: lancia `$ symfony console` per vedere le utilities a nostra disposizione, e in particolare possiamo servirci di due comandi della serie `make`: `make:entity` e `make:crud`. Vediamo quindi come usarli.

1. intanto controlliamo che le componenti (librerie php) necessarie siano installate, in particolare:

   `$ composer require symfony/orm-pack`[3]

   `$ composer require --dev symfony/maker-bundle`

   la prima installa una libreria utile per interfacciarsi con il database (mysql o un qualche alternative, ma noi useremo mysql), la seconda attiva i comandi della famiglia `make`, utili per creare del codice adatto in maniera semi-automatica;

2. teniamo presente i consigli di *Best Practices for developing web applications with Symfony* (ma delle *technicalities* preoccupatevene via via);

3. Definiamo interattivamente, da linea di comando, la Entity Studente:

```
$ symfony console make:entity

Class name of the entity to create
or update (e.g. FiercePopsicle):
> Studente

created: src/Entity/Studente.php
created: src/Repository/StudenteRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually
or by re-running this command.

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

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

[3] *orm* è acronimo per Object-relational mapping.

Abbastanza facile, dobbiamo indicare il nome della nostra Entity, nell'esempio *Studente*, e poi le proprietà per cui, per ogni studente (record), potrà essere associato un dato. Per ogni proprietà, devono essere precisati alcuni parametri: il nome (name) della proprietà, nell'esempio *Nome*; il tipo di dato dei valori assegnabili, nell'esempio *string*, scelta di default; la lunghezza dei dati, nell'esempio *255*; e infine se la proprietà *Nome* potrà rimanere, per questo o quel record, senza valore (vuota o NULL): nell'esempio *no*, non potrà.

Via via, `make:crud` crea dei file nelle opportune directory del progetto, e poi anche li potrà aggiornare:

```
created: src/Entity/Studente.php
created: src/Repository/StudenteRepository.php
```

Nel vostro esempio di prova (possibilmente diverso da questo) controllare sempre che file siano in effetti stati creati. Potremo creare altre proprietà, per esempio Scuola (di appartenenza), Anno di iscrizione, Numero matricola, data di nascita, ecc.. Quali e quante dipende da quali dati volete gestire. Per prova ne ho create un altro paio, e il file `Studente.php` contiene quindi le seguenti righe di codice, è facile vedere la corrispondenza con quanto detto. Nota che è stata inserita di default una proprietà aggiuntiva, *id*, dovresti sapere perché:

```
#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;

#[ORM\Column(length: 255)]
private ?string $Nome = null;

#[ORM\Column(length: 255)]
private ?string $Scuola = null;

#[ORM\Column(length: 255)]
private ?string $Matricola = null;

.....
```

Il file contiene, per ogni proprietà, delle funzionalità *setter* e *getter*. Per ora lasciamole così come sono.

Similarmente creo un Entity Docente (ometto i dettagli);

4. Definite le Entity, si deve creare - lato db - la struttura per salvare i dati che verranno gestiti dalla nostra web app. Abbiamo due comandi di `make` che ci aiutano:

   ```
   $ symfony console make:migration
   $ symfony console doctrine:migrations:migrate
   ```

   Il primo crea la query da eseguire, il secondo la esegue. Controllate nella directory *migrations*, il file più recente, e individuate le query preparate per creare la tabella corrispondente alla Entity che avete creato, nel mio esempio erano:

   ```
   $this->addSql('CREATE TABLE studente
   (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   nome VARCHAR(255) NOT NULL,
   scuola VARCHAR(255) NOT NULL,
   matricola VARCHAR(255) NOT NULL)');

   $this->addSql('CREATE TABLE docente
   (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
   nome VARCHAR(255) NOT NULL,
   corso VARCHAR(255) NOT NULL)');
   ```

   Controllate nel DB che le tabelle siano state create, e come. Studiate la corrispondenza tra la definizione della proprietà nei file Entity, e la definizione delle tabelle mysql. Ricordate sempre che se aggiornate le Entity, dovete poi ripetere i due comandi per aggiornare anche le tabelle del db.

5. Nell'esempio di sopra, tutte le proprietà hanno `Field type` (tipo di dati) *string*. Ma sono presenti molte più opzioni. Se alla prompt di `make:entity`, quando viene chiesto
   `Field type (enter ? to see all types) [string]:`, digitate `?`, vedrete la lista delle opzioni a disposizione. Ovviamente queste corrispondono - ma non esattamente - alle distizioni previste in mySQL per il tipo di dato dei campi di una tabella. Qui, il primo gruppo `Main Types` contiene le opzioni più comuni, e a noi note. Cosa siano le `Relationships/Associations` lo vedremo nella prossima sezione. Gli altri se ne avremo bisogno (alcuni sono per la gestione di date, in vari formati):

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

```
Main Types
* string or ascii_string
* text
* boolean
* integer or smallint or bigint
* float

Relationships/Associations
* relation a wizard  will help you build the relation
* ManyToOne
* OneToMany
* ManyToMany
* OneToOne

Array/Object Types
* json
* binary
* blob

Date/Time Types
* datetime or datetime_immutable
* datetimetz or datetimetz_immutable
* date or date_immutable
* time or time_immutable
* dateinterval

Other Types
* enum
* decimal
* number
* guid
* jsonb
* simple_array
* smallfloat
* date_point
```

6. Per concludere la creazione della nostra web app, porzione studenti, abbiamo il comando `make:crud`, che ci aiuta a creare il file *StudenteController.php* che contiene le definizione dei controller CRUD e le corrispondenti templates. Come si usa interattivamente credo sia di immediata comprensione, più facile provare che spiegare a parole

(nota: per i vari parametri si possono accettare i valori di default suggeriti):

```
$ symfony console make:crud

The class name of the entity to create CRUD (e.g. OrangeElephant):
> Studente
Choose a name for your controller class
(e.g. StudenteController) [StudenteController]:
>
Do you want to generate PHPUnit tests? [Experimental] (yes/no) [no]:
>
created: src/Controller/StudenteController.php
created: src/Form/StudenteType.php
created: templates/studente/_delete_form.html.twig
created: templates/studente/_form.html.twig
created: templates/studente/edit.html.twig
created: templates/studente/index.html.twig
created: templates/studente/new.html.twig
created: templates/studente/show.html.twig

Success!
```

Si può controllare che i file siano stati effettivamente creati. Vedremo poi meglio il codice che contengono.

7. A questo punto abbiamo programmato la web app, in forma minimale, ma completa. Controlliamo che funziona: accendete il server interno `symfony server:start --port=8089`, e poi - lato client -, con un qualsiasi browser, visualizzate la pagina principale a `localhost:8089/studente/`. Dovreste vedere anche come visualizzare le pagine degli altri casi CRUD. So far so good!

   Nota: Symfony crea delle *speaking url*,[4] che indicano in modo leggibile quel che succede nella corrispondente pagina/visualizzazione. Ciò grazie alla componente di *routing* del framework, di cui a una prossima sezione. Per ora osservate che la corrispondenza url/controller: l'opportuna speaking url la leggete (e viene letta) nel file dei controller, come annotazione alla defizione di ciascun controller ivi definito, nell'esempio

   ```
   #[Route('/studente')]
   #[Route(name: 'app_studente_index', methods: ['GET'])]
   ```

[4] *speaking urls are web page addresses that consist of content-specific keywords, and not of combinations of numbers and characters.*

```
public function index(StudenteRepository $studenteRepository):
Response
{
return $this->render('studente/index.html.twig',
[
'studentes' => $studenteRepository->findAll(),
]);
}
```

8. Se il tutto funziona (intendo con un vostro esempio) cercate ora di leggere il codice che avete creato con `make:crud`, e di capire come è strutturato. Qualche osservazione, rispetto all'esempio di sopra di `StudenteController.php`

   (a) Intanto consideriamo che in `StudenteController.php` Symfony si definisce una classe (e una sola) `StudenteController{}`. All'inizio del file sono richiamate una serie di librerie del framework:

```
namespace App\Controller;

use App\Entity\Studente; << def. della Entity
use App\Form\StudenteType;
<< def. dei moduli di inserimento per creare oggetti della Entity
use App\Repository\StudenteRepository;
<< def. delle query
use Doctrine\ORM\EntityManagerInterface;
<< libreria per interfacciare il DB
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
<< libreria di base per definire i controller
use Symfony\Component\HttpFoundation\Request;
<< libreria per gestire i dati provenienti dal client
use Symfony\Component\HttpFoundation\Response;
<< libreria per gestire i dati da inviare al client
use Symfony\Component\Routing\Attribute\Route;
<< libreria per creare le url

#[Route('/studente')]
final class StudenteController extends AbstractController
{
.....<< qui le def. dei controller relativi alla Entity Studente
}
```

   (b) i metodi della classe `StudenteController{}` realizzazano innazitutto le funzionalità CRUD per l'Entity `Studente`. Il codice

di questi metodi è molto pulito, e se letto con attenzione risulta estremamente modulare, e riutilizzabile in svariati modi. Questo che segue è la funzionalità per visualizzare tutti il contenuto della Entity (tutti gli studenti), la R di CRUD:

```
#[Route(name: 'app_studente_index', methods: ['GET'])]
public function index(StudenteRepository $studenteRepository):
Response
<< $studenteRepository è la connessione alla tabella del db
   - il repository -  corrispondente alla Entity studente
<< Response indica che index() restituisce
   un risposta da inviare al client
{
return $this->render('studente/index.html.twig', [
'studentes' => $studenteRepository->findAll(),
<< findAll() è la query per selezionare tutto,
  formalizza una query SELECT * FROM 'studente'
<< 'studentes' possiamo considerarlo l'array
    con i valori ottenuti con la query
<<  index.html.twig è la template
]);
}
```

(c) quello che segue è la funzionalità per creare un nuovo oggetto della Entity (un nuovo studente): la C di CRUD; ovvero sia per visualizzare il modulo di inserimento sia per ricevere i dati inseriti nel modulo e salvarli nella appropriata tabelle del db. Vediamo come:

```
#[Route('/new', name: 'app_studente_new', methods: ['GET', 'POST'])]
<< fisso che la url per questa visualizzazione sia /studente/new/
public function new(Request $request,
EntityManagerInterface $entityManager): Response
<< $request è la richiesta del client
<< $entityManager è l'interfaccia al  db
<< Response è la risposta al client
{
        $studente = new Studente();
<< $studente è l'oggetto da creare
        $form = $this->createForm(StudenteType::class, $studente);
<< $form è il modulo di inserimento,
<< creato secondo quanto indicato in StudenteType
        $form->handleRequest($request);
se il modulo è già stato compilato correttamente >>
        if ($form->isSubmitted() && $form->isValid()) {
```

```
        $entityManager->persist($studente);
        $entityManager->flush();
<< si salva i dati nel db
e si reindirizza al controllore index >>
        return $this->redirectToRoute('app_studente_index',
                                [], Response::HTTP_SEE_OTHER);

        }
altrimenti si valuta la template new.html.twig da inviare al client
con il modulo/form da far compilare >>
        return $this->render('studente/new.html.twig', [
        'studente' => $studente,
        'form' => $form,
        ]);

}
```

Ci sono poi le funzionalià per le altre funzionalità di CRUD (in genere quelle create di default con `make:crud` sono cinque: `index`, `show`, `new`, `edit`, `delete`). Nel vostro caso esploratele con attenzione. Prendendo ispirazione dal codice di questi metodi potete programmarne altri, per realizzare funzionalità più particolari.

4.2 Relazioni ManyToMany (e le altre).

Non abbiamo ancora programmato niente che ci aiuti a gestire che uno Studente sia in relazione con uno o più Docente/i, e di un Docentcon con uno o più Studente/i. Ci soccorre di nuovo `make:crud`, che fornisce financo una micro tutorial su come fare, alla prompt di `Field type` scegli `relation`, e poi considera l'opzione `ManyToMany`, la più complessa delle quattro. Cerca di capire cosa succede, controlla come vengono modificati i file delle Entity coinvolte, e come - dopo i comandi di migration - è modificato il database:

```
Field type (enter ? to see all types) [string]:
> relation
What class should this entity be related to?:
> Docente
What type of relationship is this?
------------ -----------------------------------
Type         Description
------------ -----------------------------------
```

```
ManyToOne
Each Studente relates to (has) one Docente.
Each Docente can relate to (can have) many Studente objects.

OneToMany
Each Studente can relate to (can have) many Docente objects.
Each Docente relates to (has) one Studente.

ManyToMany
Each Studente can relate to (can have) many Docente objects.
Each Docente can also relate to (can also have) many Studente objects.

OneToOne
Each Studente relates to (has) exactly one Docente.
Each Docente also relates to (has) exactly one Studente.
------------ ---------------------------------
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToMany

Do you want to add a new property to Docente so that you can
access/update Studente objects from it
- e.g. $docente->getStudentes()? (yes/no) [yes]:
> yes

A new property will also be added to the Docente
class so that you can access the related Studente objects from it.
New field name inside Docente [studentes]:
> Studentes

updated: src/Entity/Studente.php
updated: src/Entity/Docente.php

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

poi controlla come sono stati modificati i file delle Entity coinvolte,
nel mio esempio trovo aggiunte le seguenti linee di codice:

```
Studente.php
/**
* @var Collection<int, Docente>
*/
```

```
#[ORM\ManyToMany(targetEntity: Docente::class, inversedBy: ‘Studentes’)]
private Collection $Docentes;

public function __construct()
{
$this->Docentes = new ArrayCollection();
}

Docente.php
/**
* @var Collection<int, Studente>
*/
#[ORM\ManyToMany(targetEntity: Studente::class, mappedBy: ’‘Docentes’)]
private Collection $Studentes;

public function __construct()
{
        $this->Studentes = new ArrayCollection();
}
```

e infine esegui per aggiornare la struttura del db:

```
$ symfony console make:migration
$ symfony console doctrine:migrations:migrate
```

osserva sopratutto la creazione della tabella composita delle due Entity coinvolte, nel mio esempio `studente_docente`, e spiegati a cosa serve.

Un ultimo - almeno per ora - passaggio. Lato client vuoi fornire un modulo di inserimento tale che alla creazione di un nuovo Studente si possa subito associare uno i più Docenti. Per questo Symfony prevede che nel file `StudenteType.php` sia possibile configurare tale modulo:

```
StudenteType.php
-----
use App\Entity\Docente;
...
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
...

public function buildForm(FormBuilderInterface $builder,
                                          array $options): void
{
$builder   << il builder del modulo di inserimento
        ->add(‘Nome’)   << il campo di inserimento
```

```
                                per la proprietà Nome
....
        ->add('Docentes', EntityType::class, array(
        << il campo di inserimento per la proprietà Docentes
        'class'     => Docente::class,   << la Entity collegata
        'choice_label' => 'Nome',
<< la proprietà per identificare i record dell'Entity collegata
        'expanded'  => true,
        << per mostrare un checkbox, oppure un menu a tendina
        'multiple'  => true,
        << la possibilità di selezione multipla di più docenti
        ));
}
```

Lato client, il modulo di inserimento di un nuovo Studente mostra ora anche una lista di Docenti tra cui scegliere quello o quelli da associarli.

Per maggiori dettagli vedi a Forms, e anche EntityType Field.

4.3 Twig e gestione delle templates.

Si veda a *Twig, a modern template engine for PHP*.

Cosa significhi che *Twig* è un *template engine* è illustrato nel sito ufficiale in modo molto chiaro con un semplicissimo esempio.

In breve una template Twig è uno script in un qualche *linguaggio target* ove siano inseriti - tra specifici delimitatori: `{{ ... }}` e `{% ... %}` - dei *marker*, tali che le funzionalità Twig possano sostituire a tali *marker* opportune stringhe del linguaggio target. Con tale sostituzione la template è *evaluated* (valutata, resa) nel linguaggio target, e può essere trasmessa al client.

Nel nostro contesto, *linguaggio target* deve essere un qualche linguaggio che sia comprensibile ed eseguibile lato client, in genere quindi uno script .html (+css+js), ma potrebbe essere uno script .json, .xml, o un semplice .txt.

Nel contesto di una web app Symfony, tali markers sono formulati con riferimento alla struttura di una o altra Entity, si potrà quindi inserire un marker con la *dot notation*: `{{ studente.Nome }}`, per indicare di sostituirvi il valore assegnato alla proprietà di un oggetto della Entity.

La sintassi dei markers comprende inoltre anche un minimo di strutture di controllo; in particolare quella per eseguire un ciclo su un array di oggetti, e quindi produrre, per ogni oggetto dell'array, corrispondenti stringhe di codice del linguaggio target.

Per il nostro esempio con Studenti e Docenti, una opportuna template per la visualizzazione dell'elenco di tutti gli Studenti, con indicati i nomi dei Docenti di ciascuno Studente, potrebbe contenere il seguente codice - in rosso i marker che le funzionalità Twig andranno a sostituire con i valori che ricevono dall'opportuno Controller:

```
<table class="table">
<tbody>
{% for studente in studentes %}
<tr>
<td>{{ studente.Nome }}</td>
<td>{{ studente.Matricola }}</td>
<td>{% for docente in studente.docentes %}
  -- {{docente.Nome }}
{% endfor %}
</td>
</tr>
{% endfor %}
</tbody>
</table>
```

Per ripetere, se a `studentes` (plurale!) è assegnato un array di oggetti Studente, l'espressione `{% for studente in studentes %} ... {% endfor %}` indica a Twig di eseguire un ciclo su tale array, assegnando via via ciascun oggetto Studente a `studente` (singolare!). Per ogni Studente, i *marker* `{{ studente.Nome }}` e `{{ studente.Matricola }}` indicano di sostituirvi i valori assegnati alle proprietà Nome e risp. Matricola per quel-

lo Studente; in quanto stringhe di testo, il documento risultante sarà nel linguaggio target considerato.

Un aspetto importante è che questo meccanismo si può ripetere. Nel caso del nostro esempio con relazione ManyToMany tra le Entity, si ripete per la proprietà Docentes dell'Entity Studente:
`{% for docente in studente.docentes %} ... {% endfor %}`. Alla fine tutti gli elementi presenti tra delimitatori Twig, più o meno nidificati, saranno sostituiti con stringhe del linguaggio target, e lo script risultante potrà essere inviato al lato client.

Twig prevede una varietà di espressioni per arricchire la flessibilità ed espressività delle template, vedi a *Twig for Template Designers*. Capita l'idea alla base del meccanismo delle template e il ruolo dei markers, i vari casi e sottocasi risultano di immediata comprensione. Vedi anche questa guida italiana.

4.4 Speaking urls + Routing

In breve, ricapitoliamo come Symfony permette una trasmissione tra client e server trasparente, fondata sulla struttura delle Entity.[5] Iniziamo dall'esperienza del visitatore, lato client.

a) (lato client) url `studente\new`. Una *speaking url* che rende chiaro all'utente che cosa chiede, in questo esempio un modulo per creare un nuovo elemento della Entity Studente;

b) (lato server) la componente Routing di Symfony, avuta la richiesta, individua la corrispondente *route* nel file StudenteController.php:
`#[Route('/studente')]`
`#[Route('/new', name: 'app_studente_new', ... )]`
e l'associato *controller*
`public function new(...): ... {...}` (argomenti omessi);
(nota: le *route* si compongono)
(nota: alle *route* è associato un nome `app_studente_new`, che può essere utilizzato per creare dei link alla stessa *route*)

c) (lato server) Symfony esegue il controller `new(){}`
(questo potrebbe comprendere i varii step già commentati poc'anzi),
e `return` una template `studente/new.html.twig` e dei valori;

[5] Questo almeno per progetti di web app come quelli dei nostri esempi.

d) (lato server) Twig valuta la template, sostituendo i valori ai markers, e crea lo script da inviare al client;

e) (lato client) il client esegue lo script, ovvero visualizza la pagina web;

4.5 Registrazione e Autenticazione.

INTRO DA SCRIVERE

a) installa nel tuo progetto: `composer require symfony/security-bundle`
Controlla la creazione del file si configurazione `security.yaml` nella directory `config/packages`

b) crea la classe (Entity) User con l'utility `console make:user` (versione dedicata di `make:entity`) e migrala nel db:
`symfony console make:migration`
`symfony console doctrine:migrations:migrate`.

Attento a quando viene chiesto *Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:* Il valore che scegli, per es. `username`, deve poi essere indicato nel file di configurazione: `config/packages/security.yaml`

```
providers:
    app\_user\_provider:
        entity:
            class: App\Entity\User
                property: username  <<<<<
```

c) installo la componente che aiuta a creare il modulo di registrazioni utenti
`composer require symfonycasts/verify-email-bundle`
`symfony console make:registration-form`
(non abbiamo visto, ma potreste tentare di implementare anche la funzionalità per il controllo via email)

d) installo la componente per gestire il login
`symfony console make:security:form-login`
e imparo a indicare le sezioni della web app (le urls) protette da login, configurando `security.yaml`, per es. qualcosa del genere:

```
access_control:
    - { path: ^/studente/new, roles: ROLE_ADMIN }
    - { path: ^/studente/edit, roles: ROLE_USER }
```

e) configuro il logout, per es. qualcosa del genere:

```
logout:
   path: app_logout
   # where to redirect after logout
   target: app_studente_index
```

Fate le prove che funzioni. Le pagine protette non devono essere accessibili se non avete effettuato il login, e non più dopo il logout.

Questo è il minimo. Ci sono molte altre opzioni e varianti, vedete a Security.

5 Digressione: Come valutare se si tratta di un buon framework.

Vi copio i criteri (features) individuati da un tal Alex Ryabtsev, autore del breve post da cui traggo l'immagine di Figura 1, indicando le corrispondenti componenti Symfony:

- **Web Caching**. Che una web app debba prevedere, per essere considerata performante, un meccanismo di *cache*,[6] lo consideriamo scontato. Symfony comprende una componente cache (lato server) considerata performante, ma come esattamente sia stata programmata, e le molteplici possibilità di configurazione, sono materia oltre il nostro corso (ma se non sapete che cosa si intenda, leggete la voci wikipedia). Ricordo solo che in fase di sviluppo della web app può capitare di dover pulire la cache: si può eseguire un comando `console cache:clear`, o più direttamente rimuovendo la directory `var\cache`;
- **Scaffolding**. Per cosa si intenda per *scaffolding*, vedi e leggi Scaffold_(programming).

[6] Vedi wikipedia su cosa si intenda in generale per cache, in particolare nel nostro ambito per web chaching.

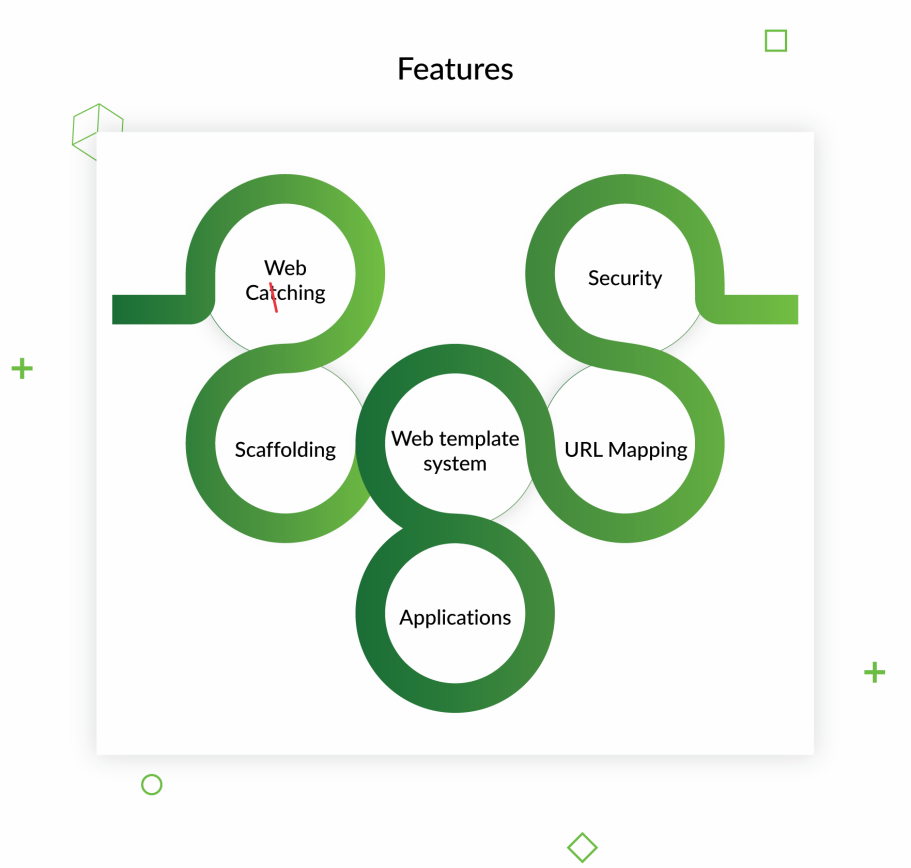


Figura 1: da https://djangostars.com/blog/what-is-a-web-framework/

In Symfony - come abbiamo visto - abbiamo la componente `console`, e in particolare i comandi della serie `make`, vedi a SymfonyMakerBundle. Se siano performanti e flessibili meno o più di analoghi strumenti di altri framework non saprei. A voi il giudizio.

- **Web template system** Abbiamo già detto come Symfony abbia sviluppato una strumento potente: twig, pubblicizzato come *the flexible, fast, and secure template engine for PHP*.
- **Security** DA SCRIVERE.
- **URL Mapping** Per Symfony, vedi la componente di Routing. Vedi la sezione sopra, su come Symfony organizza il flusso di trasmissione client/server sulla base di url e *route* costruite corrispondenti della struttura delle Entity e associati controller;

  Si trovano una infinità di post sulla importanza di tutto ciò, anche a fini di SEO, facilità di navigazione (user experience), oltre che di

facilita di sviluppo della web app. Tra i tanti, segnalo A Guide to URL Mapping for Developers e Using URL Maps to Redirect URLs.

- **Applications** Un framework va valutato anche per la varietà e ricchezza di tiplogie di possibili applicazioni che permette di sviluppare. Noi siamo interessati principalmente a applicazioni web, e vediamo solo in parte la forza di Symfony.

PER ORA È TUTTO.