**Git**

Repository: <https://github.com/myouri-code/lismoica>

**Symfony**

Create a Symfony project:

$ composer create-project symfony/website-skeleton my\_project\_name

If no my\_project\_name, the folder create will be named website-skeleton

**What I learned through Tuto #1**

Tuto: <https://youtu.be/UTusmVpwJXo>

Visual Studio Code:

* Show terminal: View > Terminal
* Delete complete line: control + u

Command-line:

* Install a personalized server to launch our Symfony app that will be used only when I dev (aka no need to install it when app goes in prod): $ composer require server --dev
* Install a library: composer require [xxx]
* Launch server on localhost:8000 (http://127.0.0.1:8000): $ php bin/console server:run
* Create a controller: $ php bin/console make:controller
* Create a docker-compose.yml file: $ nano docker-compose.yml
* Build the services written on the docker-compose.yml file (the d will return the bash prompt): $ docker-compose up –d
* Create the database (accordingly to the DATABASE\_URL value in .env): $ php bin/console doctrine:database:create
* Create an Entity (PHP class) which represents a table: $ php bin/console make:entity
* Create a SQL migration: $ php bin/console make:migration
* Launch the SQL migration script to update the database: $ php bin/console doctrine:migration:migrate
* Install Fixtures component needed only for dev (will create AppFixtures): $ composer require orm-fixtures
* Create Fixture: $ php bin/console make:fixtures
* Load fixtures into database: $ php bin/console doctrine:fixtures:load
* Update the migration file if changes in Entites (you need to run the migrate command after to update the database): $ php bin/console doctrine:migration:diff
* xxx

How Symfony works:

* Controllers: to manage treatments
* Doctrine: to access data
* Twig: to manage display
* {{ xxx }} is called an interpolation where xxx is a value from another file than the twig we are writing in

Controllers:

* When one goes to a web application, the browser (Lilo, Safari…) which called the web application needs a response from it.
* The browser calls a Route (an address) 🡪 the server receives the Request 🡪 the server sends a Response (can be a HTML page or redirection) 🡪 the Browser uses the Response.
* AKA Controllers listen to an address (Route in Symfony) 🡪 analyze HTTP request sent by the browser to the Route 🡪 make an HTTP answer 🡪 send answer to browser in order to have either a display, redirection, download of an image or file…
* Controller will contain only 1 class.
* There will be many controllers in this web application (one for the Blog, one for the Article…)

Doctrine ORM (Database):

* ORM means Objet Relational Mapping and links an app (PHP, Java…) with a database
* We manage data inside our app through PHP classes and objects; what we do will be mapped to the database thanks to the ORM
* Symfony ORM is called Doctrine
* Doctrine gives 3 tools without writing any SQL code:
  1. Entity: represents a table
  2. Manager: manages the lines (write, update, delete)
  3. Repository: does data selection (SQL search and selection)

Repository (+ Dependency Injection):

* At the beginning of the tutorial, in a function (let’s say index() from ArticlesController), to create a repository we type the following: $repo = $this->getDoctrine()->getRepository(Article::class)
* At the end of the tutorial, we replace the previous code by adding a parameter to the index() function as this: public function index(ArticleRepository $repo) and then delete the $repo line to directly write $articles = $repo->findAll();
* Do not forget to add use App\Repository\ArticleRepository;
* Dependency Injection is when we ask Symfony to give us what we need instead of us to create it (the case for the $repo variable)

Twig:

* Twig filters (written |): formats a variable before displaying it (useful to display date time); for example {{ article.createdAt | date(‘d/m/Y’) }}

Migration (Database):

* Symfony privileges the use of files: the files should let us understand how the database looks like
* A migration is a script which make a database from state A to state B
* One will run all the files to have the same database

Fixtures (Database):

* Dataset which can be re-usable by anyone

Annotations (Route):

* Will begin with @ in comment
* @Route: when browser calls this Route, Symfony needs to call the function related (there are 2 ways to call this route: by its exact path or by its name)

Route:

* Route Parameters: includes parameters which can change inside a Route; therefore, you can call a route the following way: {{ path(‘name\_of\_the\_route’, { ‘parameter’ : argument.property}) }}
* Quick memo: parameters are the variables when you create the function whilst arguments are the variables passed to the function. In the function show($id); $id is the parameter and article.id would be an argument.

Symfony folders:

* src: all php code
* templates: manages display

Bootstrap theme (Front):

* Go to Bootswatch to find Bootstrap 4 pre-manufactured themes
* Download the Flatly theme (<https://bootswatch.com/flatly/)>
* Click right on bootstrap.min.css (Flatly dropdown)
* Add <https://bootswatch.com/4/flatly/bootstrap.min.css> on the base twig (so it’ll deploy on all web pages) 🡪 add a link tag above stylesheets block (nothing is in the blocks in the base twig to let one customizes the pages at her hear content)

Docker compose:

* Docker compose is a tool to handle multiple containers at once
* 2 containers (so 2 images) will be created to contains:
  1. database
  2. phpmyadmin
* We will set all the rules in a single docker-composer.yml configuration file (3306:3306 are the default MySQL ports)
* We change the DATABASE\_URL on the .env file and run the docker-compose command to build the services.

Step by step:

1. Creation of Symfony project (website-skeleton), appears on localhost:8000
2. Installation of personalized server
3. Run of server (cannot write on terminal #1 anymore)
4. Opening new terminal to create controller (BlogController) – *controller purpose is to listen to browser request on web application and create an answer which will be displayed by browser; so, we create a controller for the Blog*.
5. Chose a Bootstrap theme (<https://bootswatch.com/flatly/)> and change base twig accordingly
6. Creation of a docker-compose.yml file and setting rules for database and phpmyadmin services; we will secure the database by creating a partition on our computer (~/database:/var/lib/mysql follows host:docker model). We change the DATABASE\_URL in the .env file and build the services.
7. Creation of database lismoica
8. Creation of Article entity (which creates the Entity and the Repository) that will automatically ask us the properties (fields table)
9. Create migration where Doctrine will check the entities and the and the database; if any differences, Doctrine will create and run a SQL script
10. Launch of the migration script to update the database
11. Install fixture component in project to populate the database with my fake data (old data from blogspot) – AppFixtures was create but I deleted it (no use)
12. Create ArticleFixtures
13. Create a set of 10 articles and ask the Manager to **make them persist** through time (the articles are still not created before this persist action nor after); what creates the articles is the flush action
14. As I have made a mistake in the properties of Article Entity (image in array), I have made modification (change array to string) and run the doctrine:migration:diff command (that will update the migration file if any changes) then the doctrine:migration:migrate
15. Create a Repository in the ArticlesController to access the database
16. Change the index twig to show the articles thanks to the Repository in ArticlesController
17. Create show twig which will display a specific article (route parameter was needed for the index twig)
18. Improve code by doing dependency injection (erase lines to create an ArticleRepository variable and pass it directly as parameters, in order for Symfony to give us what we need instead of us creating it) – do it for index() and show() functions
19. Further improve code by deleting ArticleRepository (parameter and line code where it exists) and directly put an Article variable as parameter AND delete also $id parameter. Only put a variable Article as parameter. **Magic happens** as Symfony will look at the show() function and understand that he needs to give it an Article variable; then Symfony checks the Route (“/articles/{id}”) and look for the article with the id given.

**What I learned through Tuto #2**

Tuto: <https://youtu.be/_cgZheTv-FQ>

Command line:

* Install form feature: $ composer require symphony/form
* Create form: $ php bin/console make:form

Symfony page:

* There are 3 pillars to create a page in Symfony:

1. A Function
2. A route
3. An answer (what to display)

Route confusion:

* When Route patterns are similar, Symfony might be confused on which function to call. For example, the functions show() and create() have similar Route patterns, resp. /articles/{id} and /articles/new).
* If one writes the Route /articles/new, Symfony will look at the functions written in order. Therefore, if show() (with /articles/{id}) is written before create(), Symfony might call the function show() instead of the function create() we intented to have we the Route /articles/new (aka Symfony will think that ‘new’ is the value for {id}).
* To bypass this confusion, you can change the functions order (put create() before show())

Function:

* A function can have multiple routes (for example, one routing to create a new article and to edit an article)
* Tuto changed create() function to form() function (purpose was to make 2 actions from it: create a new article and edit one); for the sake of understanding, I have made 2 separate functions (imo, 1 action = 1 function)

HttpFoundation\Request:

* Is a class which can analyze and manage HTTP request
* To put as parameters in a function (dependency injection)

Form:

* Install the form feature with composer
* Create an empty instance of what we want to make of form from
* Create a FormBuilder using $this->createFormBuilder($article), you will have a form binded to an article.
* Then set the form with the properties we want to have in the form (using add()) and then get the result with getForm().
* It’s not HTML, it’s a complex object: we cannot render $form but something else easier to render for twig which is $form->createView()
* In twig, use the form() function to render the form

Form’s properties:

* As we’ve seen earlier, in create(), to add properties to the form we use add (e.g. $this->add(‘title’))
* You can add more arguments, like what’s the property type you want (in Entity, property is a string but you can decide it to be an array in the form) and the field’s options (e.g. placeholder aka grey text in the form to indicate what we seek) by using a tab ([])
* ‘attr’ is a tab of HTML attributes
* Instead of having a complex create() function, you can instead keep it simple in terms of arguments and complete the twig by using form\_row (and adding arguments inside: becareful the arguments are in JSON so a table is {} and not [] !)

Form’s visual:

* In twig, by calling form(formArticle), twig decides how the form looks like: you can modify the look to your taste
* In twig, to begin a form, please us form\_start (in an interpolation) and use form\_end to end a form
* To have a bootstrap render, you can use form\_widget
* Use form templates by modifying twig.yaml and adding how the form should looks like
* Re-tell twig what bootstrap them we should use (not necessary as I could experience but the Tuto said to do it)
* At this point, you can even take out form\_start and form\_end but it’s better to keep it in case you want to add something between the beginning/the render and the render/ending

Request:

* Request sends data from form

Step by step:

* 1. Creation of a create() function that will allow us to create an article through a formular – be careful of Route confusion
  2. Do a dependency injection by adding parameters to the create() function. Adds a Request variable
  3. /articles didn’t want to appear (An exception occurred in driver: SQLSTATE[HY000][2002] Connection refused): $ docker-compose down then $ docker-compose up-d and the webpage will appear.
  4. In function create(Request, ObjectManager), create an empty article then a form with FormBuilder
  5. Set the FormBuilder with the properties you want (properties are common to the ones of an Article entity) then get the result (to have the final form)
  6. Render the form (not directly because the form is a complex object which cannot be easily render in twig)
  7. In create(), complete the properties with the HTML attributes you want
  8. Add a Bootstrap theme in twig.yaml
  9. Tell twig that we are using the template in twig.yaml
  10. Add a button which will change depending on situation in twig
  11. Add a Request Handler (asks Symfony to handle the HTTP request of the form)
  12. Create an edit() function to edit article
  13. Create a form from terminal to get an ArticleType which will be called to create a form (instead of using formBuilder and add properties)

**What I learned through Tuto #3**

Tuto:

Command line:

* Create entity: $ php bin/console make:entity
* Install Faker: $ composer require fzaninotto/faker --dev (if you added it without dev, no problem, just redo it: it will change composer.json, from require to require-dev)
* Undo a migration: $ php bin/console doctrine:migrations:execute [number of the migration] --down (to get the number, see Version in the function title)
* xxx

Relation:

* Symfony allows relation between Entities
* When creating a relation, Symfony will ask if we want to automatically delete orphaned objects. For example, if you want to change the Category of an Article, when you delete the category (to change it) should Symfony suppress the Category (considered as orphaned)? (NO. JUST SAY NO – but yes for a comment, because a comment is under one article and if you ever want to move it: it means you have to delete it)

Migration (/!\ Good learning):

Migration does the diff between PHP and the DB. When I did my 1st migration (to create Category and the foreign key between category and article), an error occurred because I wanted to add a new column (category\_id which cannot be null) in my DB (in Article table), but there was already data inside; migration exploded the DB because category\_id was indeed added but all its values were null which is not possible. Then (thinking it was OK because the category\_id was created – but foolish me didn’t think about the foreign key creation) I did a 2nd migration (to create Comment and foreign key between comment and article) and again, explosion. Thanks to Cheong, here’s a scheme to understand the problem and how to solve it:

* + Migration 1: SQL1 succeed (Category table creation); SQL2 failed (foreign key)
  + Migration 2: SQL3 succeed (Comment table creation); SQL2 failed again

🡺 When you migrate a migration, if all the SQL requests went well, the Migration # will be saved by Symfony and it will never be ran again (exception for new dev which seek for the same DB)

🡺 When a migration hasn’t been successful (one or more SQL requests failed), Doctrine will call again the missing SQL requests. That is why in Migration 2, the SQL2 was called again (because failed in Migration 1)

* Solution? To make in comment or to delete some SQL requests.
  + Migration 1: comment SQL1 and run 🡪 Migration 1 completed
  + Migration 2: delete SQL2 and run 🡪 Migration 2 completed

Faker Librarie:

* Creates fake random data better than what we would insert

Step by step:

* Create an Entity called Category; make a relation OneToMany with the Article entity
* Create an Entity called Comment; make a relation ManyToOne with the Article entity
* Please look at the migration above, where I had an interesting error
* Add fake data thanks to Faker with the command line

**What I learned through Tuto #4**

Tuto: <https://youtu.be/_GjHWa9hQic>

Command line:

* Create a controller: $ php bin/console make:controller
* xxx

Questions regarding protection:

* How do we want to secure the app?
* How secure the data? (hash password)
* Where are our users? (on the database? From multiple places? In a text file? LDAP?)

🡺 Sum up in Security component

Security component:

* Gives firewalls: how to protect entry points
* Providers: where are the users’ data? 🡪 you can have multiple providers
* Encoders: how to secure (which algorithm hash)

🡺 Security.yaml file in config>packages>test

User entity:

Step by Step:

* Create a User entity, create migration file and migrate
* Create a RegistrationType form and add a property in the buildForm() called confirm\_password. If you do so, you need to add it to the Entity related (User) 🡪 in User, add a public variable called confirm\_password. We don’t add a comment (ORM) to conform that this variable has nothing to do with the DB.
* Create a SecurityController, you can delete the index() fuction and instead do a registration() function
* Create the registration twig
* Add labels to the forms (it’s another parameter like attr), we can see that the password can still be seen 🡪 we need to add a PasswordType to the passwords (in RegistrationType) so the password will appear like dot
* Use handleRequest to analyze the request given
* If the form is submitted and valid, persist the user and use flush (aka “do it” to Symfony)
* Add validation to make sure password is same than confirmation password 🡪 in User entity, add constraints in namespace (please add the related library)
* Constraints to add: length, EqualTo (in property path, add the value which we should match with)
* Hash passwords in the DB: in the security.yaml file, add encoders to User entity, using algorithm BCRYPT. It means that every time encoders encounter a User, they will use the BCRYP algorithm. You need to hash password, in SecurityController, before persisting the user 🡪 by using encorePassword you can use any hash algorithm
* There will be an error as we need to implement method that User doesn’t have yet and that UserpasswordEncoder needs it 🡪 in User, you need to implement UserInterface and add corresponding metho (eraseCredentials, getSalt and getRoles with nothing inside except for the last one that returns an array)
* Make unique users, add constraint in the Entity itself (@UniqueEntity)
* Create vue to render the user created
* Add a provider called in\_database which is based on the User Entity (App\Entity\User), to check where are the User, are the passwords and users OK; add a property based on the email
* Add this provider to the firewalls