1 Introduction

1.1. Tools and requirements

LAMP (Short form of Linux, Apache, MySQL, and PHP) Stack is the most popular environment in PHP website development and web hosting. Where Linux is an operating system, Apache is the popular web server developed by Apache Foundation, MySQL is relational database management system used for storing data and PHP is the widely used programming language.

First of all, you need to check that your computer has a friendly working environment for web development. We will use Ubuntu 16.04 LTS installed on your computer. At a minimum, you need a web server (Apache, for instance), a database engine (MySQL) and PHP 7 or later. PHP 7 is the default available packages in Ubuntu 16.04 repositories. Simply use the following commands to update apt cache and install PHP packages on your system.

- **Add the main PHP repository to your system.**

  In a nutshell, “apt-get update” doesn’t actually install new versions of software. Instead, it updates the package lists for upgrades for packages that need upgrading, as well as new packages that have just come to the repositories.

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:ondrej/php
sudo apt update
sudo apt-get install php7.2
```

You may also need to install some additional PHP modules for supporting various tasks

```
sudo apt-get install php7.2-curl php7.2-gd php7.2-json php7.2-mbstring php7.2-mcrypt
php7.2-intl php7.2-cli php7.2-xml
```

- **Install Apache, your web server:**

```
sudo apt-get install apache2 libapache2-mod-php7.2
```

And enable Apache mod-rewrite:

```
sudo a2enmod rewrite
```

- **Install the MySQL Server:**

Install mysql-server packages for MySQL database. Also, install php-mysql package to use MySQL support using php. Use the following command to install it.

```
sudo apt-get install mysql-server php7.2-mysql
```

Now, you need to restart Apache service:

```
sudo service apache2 restart
```

- **Install phpMyAdmin**
  You can also install phpMyAdmin for the administration of MySQL using web interface

```
sudo apt-get install phpmyadmin
```

Exercise
Installation and setup

2 Introduction to Symfony

2.1. Installing & Setting up the Symfony Framework

- **Installing Composer:**

Composer is the package manager used by modern PHP applications. Use Composer to manage dependencies in your Symfony applications and to install Symfony Components in your PHP projects. There are two ways to install Composer. Locally as part of your project, or globally as a system wide executable.

If you don't have "curl" extension installed, you can install it using this command:

```
sudo apt-get install curl
```

- ✓ **Local installation of composer**

Curl downloads file from a given URL, oprions below are -s= silent(don't show any output), -S=don't show errors. Read more on how to install composer locally, https://getcomposer.org/download/

```
curl -sS https://getcomposer.org/installer -o composer-setup.php
php composer-setup.php
rm composer-setup.php
```

This installer script will simply check some php.ini settings, warn you if they are set incorrectly, and then download the latest composer.phar in the current directory. The 4 lines above will, in order:

- Download the installer to the current directory and rename it to ***composer-setup.php***
- Run the installer, will generate a ***composer.phar*** file in your current directory, which can be executed with ***"php composer.phar"*** command
- Remove the installer

- ✓ ***Global installation of Composer***

```
sudo mv composer.phar /usr/local/bin/composer
```

If you like to install it only for your user and avoid requiring root permissions, you can use ~/.local/bin instead which is available by default on some Linux distributions.

Creating Symfony Applications

Symfony provides a dedicated application called the **Symfony Installer** to ease the creation of Symfony applications. This installer is a PHP 5.4 compatible executable that needs to be installed on your system only once:

```
sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
sudo chmod a+x /usr/local/bin/symfony
```

Once the Symfony Installer is installed, create your first Symfony application with the new command:

```
symfony new hrm 3.4
```

This command creates a new directory called hrm**/** that contains an empty project based on the most recent stable Symfony version available. In addition, the installer checks if your system meets the technical requirements to execute Symfony applications. If not, you'll see the list of changes needed to meet those requirements.
If you specify version number to be downloaded, it will not download the latest version rather it will download the specified version of symphony. In this example we are specifying version to be 3.4.

Creating Symfony Applications with Composer

If you can't use the Symfony installer for any reason, you can create Symfony applications with Composer, the dependency manager used by modern PHP applications.

```
composer create-project symfony/framework-standard-edition hrm "3.4"
```

In order to add packes to our symfony application, use composer.json file

Edit ***composer.json*** with your favorite editor. Put the following content for simple example wich tells the composer to install additional package called ***monolog.***

```
{
   "require": {
      "monolog/monolog": "1.0.*"
   }
}
```

Then, execute the following command in order to install the required packages with specified version number.

```
composer install
```

Read more on basic usage https://getcomposer.org/doc/01-basic-usage.md

> TIP: If your Internet connection is slow, you may think that Composer is not doing anything. If that's your case, add the **-vvv** flag to the previous command to display a detailed output of everything that Composer is doing.

2.2. Built-in web server

On production servers, Symfony applications use web servers such as Apache or Nginx (see configuring a web server to run Symfony). However, on your local development machine you can also use the web server provided by Symfony, which in turn uses the built-in web server provided by PHP.

First, install the Symfony Web Server and then, execute this command:

```
cd hrm/
php bin/console server:run
```

Open your browser, and access the ***http://localhost:8000/*** URL to see Welcome Page of Symfony.

If you see a blank page or an error page instead of the Welcome Page, there is a directory permission misconfiguration. The solution to this problem is the following commands

```
sudo chmod -R 777 var/cache var/logs var/sessions
sudo setfacl -dR -m u::rwX var/cache var/logs var/sessions
```

When you are finished working on your Symfony application, stop the server by pressing Ctrl+C from the terminal or command console.

2.3. Web Server Configuration

A good web practice is to put under the web root directory only the files that need to be accessed by a web browser, like stylesheets, JavaScripts and images. By default, it's recommended to store these files under the web/ sub-directory of a symfony project.

To configure Apache for your new project, you will create a virtual host. In order to do that, go to your terminal and type in the next command:

```
sudo nano /etc/apache2/sites-available/hrm.conf
```

Now, a file named hrm***.conf*** is created. Put the following inside that file, then hit Control – O and Enter to save it, then Control – X to exit the editor.

```
<VirtualHost *:80>
    ServerName hrm.local
    DocumentRoot /home/user/sfProjects/hrm/web
    DirectoryIndex app.php
    ErrorLog /var/log/apache2/hrm-error.log
    CustomLog /var/log/apache2/hrm-access.log combined
    <Directory "/home/user/sfProjects/hrm/web">
        AllowOverride All
        Require all granted
        Allow from All
     </Directory>
</VirtualHost>
```

The domain name hrm***.local*** used in the Apache configuration has to be declared locally. If you run a Linux system, it has to be done in the ***/etc/hosts*** file. If you run Windows, this file is located in the ***C:\Windows\System32\drivers\etc*** directory. Add the following line:

```
127.0.0.1       hrm.local
```

You need to enable the newly created virtual host and restart your Apache. So go to your terminal and type

```
sudo a2ensite hrm.conf
sudo service apache2 restart
```

2.4. Checking your apache configurations

Symfony comes with a visual server configuration tester to help make sure your Web server and PHP are correctly configured to use Symfony. Use the following URL to check your configuration: http://hrm.local/config.php

Probably, you will get all kind of requirements when you go to config.php. Below, is a list of things to do for not getting all those ***"warnings"***

```
sudo chmod -R 777 var/cache var/logs var/sessions
sudo setfacl -dR -m u::rwX var/cache var/logs var/sessions
```

Edit your php.ini file in ***/etc/php/7.2/apache2/php.ini***

```
sudo nano /etc/php/7.2/apache2/php.ini
```

Search for date.timezone. You can use Ctr+W for search and edit it like below

```
date.timezone = Africa/Addis_Ababa
```

Enable a PHP Accelerator (Opcache recommended). To enable the OPcache, change to the following lines of your php.ini file

```
opcache.enable=1
opcache.memory_consumption=128
opcache.max_accelerated_files=4000
sudo phpenmod opcache
sudo service apache2 restart
```

2.5. Directory structure

After creating the application, enter the hrm**/** directory and you'll see a number of files and directories generated automatically

```
hrm/
├─ app/
│  ├─ config/
│  └─ Resources/
├─ bin
│  └─ console
├─ src/
│  └─ AppBundle/
├─ var/
│  ├─ cache/
│  ├─ logs/
│  └─ sessions/
├─ tests/
│  └─ AppBundle/
├─ vendor/
└─ web/
   ├─ app.php
   └─ app_dev.php
```

This file and directory hierarchy is the convention proposed by Symfony to structure your applications. The recommended purpose of each directory is the following:

- ***app/config/***, stores all the configuration defined for any environment;
- ***app/Resources/***, stores all the templates and the translation files for the application;
- ***src/AppBundle/***, stores the Symfony specific code (controllers and routes), your domain code (e.g. Doctrine classes) and all your business logic;
- ***var/cache/***, stores all the cache files generated by the application;
- ***var/logs/***, stores all the log files generated by the application;
- ***var/sessions/***, stores all the session files generated by the application;
- ***tests/AppBundle/***, stores the automatic tests (e.g. Unit tests) of the application.
- ***vendor/***, this is the directory where Composer installs the application's dependencies and you should never modify any of its contents;
    - ***web/***, stores all the front controller files and all the web assets, such as stylesheets, JavaScript files and images.

2.6. Generating a New Bundle

The ***generate:bundle*** generates a new bundle structure and automatically activates it in the application.

By default the command is run in the interactive mode and asks questions to determine the bundle name, location, configuration format and default structure

To deactivate the interactive mode, use the *--no-interaction* option but don't forget to pass all needed options, you can read https://symfony.com/doc/3.4/bundles.html.

```
php bin/console generate:bundle –namespace=BuleHora\HRMBundle --no-interaction
```

If the application fails to add the bundle in autoload section automatically, you will have an error that tells you to edit ***composer.json*** file.

```
The command was not able to configure everything automatically.
 You'll need to make the following changes manually.
```

Open composer.json, and check if ***autoload*** is as follows
Before:

```
"psr-4": {
            "AppBundle\\": "src/AppBundle"
        },
```

After changing:

```
"psr-4": {
            "": "src/"
        },
```

And finally, run:

```
composer dump-autoload
```

2.7. Application environments

In Symfony, the idea of "***environments***" is the idea that the same codebase can be run using multiple different configurations. Symfony defines two environments by default: ***dev*** (suited for when developing the application locally) and ***prod*** (optimized for when executing the application on production). The main difference between environments is that ***dev*** is optimized to provide lots of information to the developer, which means worse application performance. Meanwhile, ***prod*** is optimized to get the best performance, which means that debug information is disabled, as well as the web debug toolbar.

The other difference between environments is the configuration options used to execute the application, each environment simply represents a way to execute the same codebase with different configuration. It should be no surprise then that each environment loads its own individual configuration file. If you're using the YAML configuration format, the following files are used:

- for the ***dev*** environment: `app/config/config_dev.yml`
- for the ***prod*** environment: `app/config/config_prod.yml`
- for the ***test*** environment: `app/config/config_test.yml`

➢ Executing an Application in different Environments

To execute the application in each environment, load up the application using either app.php (for the prod environment) or app_dev.php (for the dev environment) front controller:

```
http://localhost/app.php        ==> *prod* environment
http://localhost/app_dev.php  ==> *dev* environment
```

If you don't have *either* filename in your URL, then it's up to your web server to decide *which* file to execute behind the scenes. If you're using the built-in PHP web server, it knows to use the ***app_dev.php*** file. On production, you'll configure your web server to use ***app.php***. Either way: one of these two files is always executed.

2.8. Symfony console

The Symfony framework provides lots of commands through the bin/console script (e.g. the well-known bin/console cache:clear command). These commands are created with the *Console component*. You can also use it to create your own commands.
You can get all the list of available commands to use, by typing the following command on the terminal window. It will list commands with their description

```
php bin/console list
```

2.9. Symfony basic concepts

The Symfony Framework is well-known for being *really* flexible and is used to build micro-sites, enterprise applications that handle billions of connections and even as the basis for other frameworks. Since its release in July 2011, the community has learned a lot about what's possible and how to do things *best*.

One of the main goals of a framework is to keep your code organized and to allow your application to evolve easily over time by avoiding the mixing of database calls, HTML tags and other PHP code in the same script. To achieve this goal with Symfony, you'll first need to learn a few fundamental concepts.

When developing a Symfony application, your responsibility as a developer is to write the code that maps the user's *request* (e.g. ***http://localhost:8000/***) to the *resource* associated with it (the Homepage HTML page).

The code to execute is defined as methods of PHP classes. The methods are called ***actions*** and the classes are ***controllers***, but in practice most developers use '**controllers**' to refer to both of them. The mapping between user's requests and that code is defined via the ***routing*** configuration. And the contents displayed in the browser are usually rendered using ***templates***.

When you go to ***http://localhost:8000/app/example***, Symfony will execute the controller in ***src/AppBundle/Controller/DefaultController.php*** and render the ***app/Resources/views/default/index.html.twig*** template.

- **Bundle**

A bundle is similar to a plugin in other software, but even better. The key difference is that *everything* is a bundle in Symfony, including both the core framework functionality and the code written for your application. Bundles are first-class citizens in Symfony. This gives you the flexibility to use pre-built features packaged in third-party bundles or to distribute your own bundles. It makes it easy to pick and choose which features to enable in your application and to optimize them the way you want.

A bundle is simply a structured set of files within a directory that implement a single feature. You might create a BlogBundle, a ForumBundle or a bundle for user management

(many of these exist already as open source bundles). Each directory contains everything related to that feature, including PHP files, templates, stylesheets, JavaScript files, tests and anything else. Every aspect of a feature exists in a bundle and every feature lives in a bundle.

Exercise

Generate HRM bundle

3. Databases and Doctrine

One of the most common and challenging tasks for any application involves persisting and reading information to and from a database. Although the Symfony Framework doesn't integrate any component to work with databases, it provides tight integration with a third-party library called Doctrine. Doctrine's sole goal is to give you powerful tools to make database interactions easy and flexible.

Doctrine is totally decoupled from Symfony and using it is optional. This chapter is all about the Doctrine ORM, which aims to let you map objects to a relational database (such as MySQL, PostgreSQL or Microsoft SQL). If you prefer to use raw database queries, this is easy.

3.1. Configure and create database

The easiest way to understand how Doctrine works is to see it in action. So, first we'll configure our database, then we will define the HRM data model, use an ORM to interact with the database and persist objects to the database and fetch it back out.

- **Configuring the Database**

To define the database connection parameters, you have to edit the ***app/config/parameters.yml*** file

```
parameters:
    database_driver: pdo_mysql
    database_host: localhost
    database_name: hrm
    database_user: root
    database_password: password
```

- **Create the Database**

Now Doctrine knows about your database, you can have it create the database for you:

```
php var/console doctrine:database:create
```

- **Add Mapping Information**

Doctrine allows you to work with databases in a much more interesting way than just

fetching rows of a column-based table into an array. Instead, Doctrine allows you to persist entire objects to the database and fetch entire objects out of the database. This works by mapping a PHP class to a database table, and the properties of that PHP class to columns on the table.

For Doctrine to be able to do this, you just have to create "metadata", or configuration that tells Doctrine exactly how the Product class and its properties should be mapped to the database. This metadata can be specified in a number of different formats including ***YAML***, ***XML*** or directly inside the Entity class via ***annotations***.

In this training we use YAML to configure our mapping information. So, before all create folder '***doctrine***' inside '***src/BuleHora/HRMBundle/Resources/config'***
Create the first object called ***ItemMeasuringUnit***, orm yaml configuration file

src/BuleHora/HRMBundle/Resources/config/doctrine/ItemMeasuringUnit.orm.yml

```
BuleHora\HRMBundle\Entity\ItemMeasuringUnit:
  type: entity
  table: item_measuring_unit
  id:
    id:
      type: integer
      generator: { strategy: AUTO }
  fields:
    name:
      type: string
      length: 255
      unique: true
    symbol:
      type: string
      length: 20
      nullable: true
    description:
      type: string
      length: 255
      nullable: true
    created_at:
      type: datetime
```

```
  updated_at:
    type: datetime


lifecycleCallbacks:
  prePersist: [ setCreatedAtValue , setUpdatedAtValue, checkSomething]
  preUpdate: [ setUpdatedAtValue]
```

3.2. Entity

- **What is an Entity?**

The class - often called an "entity", meaning a basic class that holds data - is simple and helps fulfill the business requirement of needing objects in your application.
Since you're building an application where objects need to be displayed, without even thinking about Doctrine or databases, you already know that you need getter and setter functions for your objects. These classes are inside the ***Entity*** directory of your Bundle

- **Generate entity**

Make sure that you have ***Entity*** directory in your bundle when you write type the following command unless it will show you an error:

```
In DisconnectedMetadataFactory.php line 42:
Bundle "BuleHoraHRMBundle" does not contain any mapped entities.
```

Once you learn the concepts behind Doctrine, you can have Doctrine create simple entity classes for you. This will ask you interactive questions to help you build any entity. This command creates Entity folder in your bundle and Entity classes for your objects.

```
php bin/console doctrine:generate:entity
```

If you want to create entity once for all mapped entities, type the following command

```
php bin/console doctrine:generate:entities BuleHoraHRMBundle
```

3.3. Creating the Database Tables/Schema

You now have a usable Entity classes with mapping information so that Doctrine knows exactly how to persist it. Of course, you don't yet have the corresponding tables in your database. Fortunately, Doctrine can automatically create all the database tables needed for every known entity in your application. To do this, run

```
php bin/console doctrine:schema:update --force
```

✓ Tip: Actually, this command is incredibly powerful. It compares what your database should look like (based on the mapping information of your entities) with how it actually looks, and executes the SQL statements needed to update the database

schema to where it should be. In other words, if you add a new property with mapping metadata to Product and run this command, it will execute the "ALTER TABLE" statement needed to add that new column to the existing product table.

3.4. What is a Repository

Repositories are the only classes that interacts with a DB. You use the default ones in most cases but do need one if you have some special queries. Business logic should not be here.

- **Persist Object /Insert data to database**

Open Default Controller and Change indexAction to:

```
public function indexAction()
   {
        $em = $this->getDoctrine()->getManager();
        $item = new Item();
        $item->setName('Laptop');
        $item->setDescription('Description for Laptp');
        $item->setCode('L001');
        $item->setSerialNumber('L001-S001');
        $item->setQuantity(10);
        $em->persist($item);
        $em->flush();

      return $this->render('default/index.html.twig',array(
        'item'=>$item,
      ));
   }
```

- **Get some data from database**

We can get list of Objects from database, by accessing repositories. For example, to access list of Items in the database:

```
public function getItemsAction()
   {
        …
        $em = $this->getDoctrine()->getManager();
        $items = $em->getRepository('BuleHoraHRMBundle:Item')->findAll();

     return $this->render('default/index.html.twig',array(
        'items'=>$items,
      )
        );
   }
```

Edit your app/Resources/view/default/index.html.twig

```
{% if item is defined %}
        Id={{item.id}}
        Name={{item.name}}
        Code={{item.code}}
        Serial Number = {{item.serialnumber}}
{% endif %}

{% for item in items %}
        Id={{item.id}}
        Name={{item.name}}
        Code={{item.code}}
        Serial Number = {{item.serialnumber}}
        <br/>
{% endfor %}
```

- **Extend and update existing entity**
  Add ***User.orm.yml*** to your mapping objects, generate ***Entities***, and Update your ***schema***. Create user table With the following list of attributes:

User
✓ Fname
✓ Mname
✓ Gender
✓ Office //relation with office
✓ item //relation to item

After adding your Mapping, type the following two commands.

```
php bin/console doctrine:generate:entities BuleHoraHRMBundle
php bin/console doctrine:schema:update –force
```

The first command will generate PHP classes to access objects of database, which are called ***Entity classes***, and the second command update our database so that we can have latest information about our tables in database.

persist()**:** you are telling the entity manager to track changes to the object.
flush()**:** the em will push the changes of the entity objects the em tracks to the database in single transaction. Most often em have to manage multiple objects.

3.5. Lazy loading and Proxy objects

Doctrine is an Object Relational Mapper (ORM) that sits on top of a powerful Database Abstraction Layer (DBAL). When you've been working with Symfony (+ Doctrine) for a

while, you're bound to have come across Doctrine's Proxy Objects.

```
public function setItemFamily(\BuleHoraHRMBundle\Entity\ItemFamily $itemFamily =
null)
{
      $this->itemFamily = $itemFamily;
      return $this;
}
```

If you've ever generated an entity which is associated with another entity, and have taken a look at the arguments for the function associated with that entity in the generated entity file, you might have noticed the extra backslash before the type declaration for the function argument. In this case, '****' before ***'BuleHoraHRMBundle\Entity\ItemFamily'***.

On a general day-to-day basis, a developer shouldn't be bothered by a ***proxy object***, they should be transparent to your code. And Doctrine accomplishes that perfectly.
A proxy object is an object that is put in place or used instead of the "real" object. A proxy object can add behavior to the object being proxied without that object being aware of it. In Doctrine 2, proxy objects are used to realize several features but mainly for transparent lazy-loading.
To put it in simple term, a proxy object is simply a wrapper object that extends the functionality of the base entity class and provides it with ***lazy loading*** abilities. When you fetch an entity from a database, the base entity is fully initialized, except the entities that are associated with it. These entities are then partially loaded and wrapped into a proxy object. At this point, only the ***id*** of the associated entity is known. Then when we further access a method or property of this proxied object, Doctrine will make a request to the database to load that property if it's not already loaded.

Exercise

HRM Entities and schema

4. Lets code!

4.1. Handle CRUD operations for entity

If you run the command below, it will create a new controller ***src/Ju/GebeyaBundle/Controllers/ItemController.php*** with actions for listing, creating, editing and deleting Items (and their corresponding templates, form and routes):

```
php bin/console doctrine:generate:crud
```

After running this command, you will need to do some configurations the prompter requires you to. So just select the default answers for them.

To view this in the browser, we must import the new routes that were created in ***src/Ju/GebeyaBundle/Resources/config/routing/routing/item.yml*** into our bundle main routing file, which is ***app/config/routing.yml***:

```
bule_hora_hrm_item:
   resource: "@BuleHoraHRMBundle/Resources/config/routing/item.yml"
```

```
    prefix:   /item
...
```

CRUD: generates the following abilities on entities

- ✓ Create
- ✓ Retrieve
- ✓ Update
- ✓ Delete

Generating crud for Item entity, generates different files in our bundle

- ✓ It adds the following routing configuration in ***app/config/routing.yml***

```
bule_hora_hrm_item:
    resource: "@BuleHoraHRMBundle/Resources/config/routing/item.yml"
    prefix:   /item
```

- ✓ Create yaml file, routing configurations in

```
src/BuleHoraHRMBundle/Resource/config/routing/item.yml
```

- ✓ Creates Form directory in src/BuleHoraHRMBundle and create

```
src/BuleHoraHRMBundle/Form/ItemType.php
```

- ✓ Create ***ItemController*** in src/BuleHoraHRMBundle/Controller directory

We will also need to add a _***toString()*** method to our ***Entity*** classes ItemMeasuringUnit, ItemGroup, ItemFamily, ItemType,Store and similar classes to be used by the ***Item*** drop down from the edit and create forms:

```
//…
public function __toString()
{
    return $this->getName();
}
//…
```

Clear the cache:

```
php bin/console cache:clear --env=dev
php bin/console cache:clear --env=prod
```

You can now test the Item controller in a browser: ***http://127.0.0.1:8000/Item/*** or, in development environment, ***http://127.0.0.1:8000/app_dev.php/Item/***.

You can now create and edit Items. Try to leave a required field blank, or try to enter invalid data. That's right, Symfony has created basic validation rules by introspecting the database schema.

In order to create form, controller, routes for all of your entities you have to generate crud for all of your entities one by one or you can also create custom controller/form/routes.

4.2. Twig

- ✓ **Displaying message**

```
{{ }}
{{ variable }}
{{ 'message here' }}
```

- ✓ **do something**

```
{% %}
Twig uses this format to do something
```

- ✓ **if**

```
     {% if i==0 %}

     {% endif %}


{% if somecondition %}
   {{ 'show something here or do' }}
{%elseif anothercondition %}
   {{ 'show something here or do' }}
{%else %}
  {{ 'show something here or do' }}
{% endif %}
```

- ✓ **for**

```
{% for item in items %}
  {{item.attribute }}
  Or any other thing
{% endfor %}
```

- ✓ **set**

```
Set twig variable
{% set i=0 %}
{% set i,j,k = 0,3,10 %}
```

- ✓ **Loop**

```
loop.index
    You can use it only inside loop
```

Starts counting from 1

Loop.index0

You can use it only inside loop

Starts counting from 0

- ✓ **Array**

  items|length = shows number of items in items array

  item|first = shows the first item in an array

  item|last = shows the last item in an array

- ✓ **Concatenation**

  ~ (tilt) is used to concatenate string

  {{ user.fname ~’ ’~user.midname }}

4.3. Template inheritance

We are going to customize the basic ***controllers*** we created earlier. It already has most of the code we need for HRM:

- A page to **list** all Items
- A page to **create** a new Item
- A page to **update** an existing Item
- A page to **delete** an Item
- And many pages ...

Although the code is ready to be used as is, we will refactor the templates to look great using bootstrap.
Twig enable us to use one template in another template by different methods.

- ✓ **extends**

- One template can only extends one template
- The extended template is the parent of the extending template
- Unless overridden by the child template, all blocks of the parent blocks are visible
- Child template will have all blocks of the parent template

- ✓ **Include**

- Template can include other template in a block or anywhere in the template
- Included template is not a parent of the including template

- ✓ **Embed**

- Template can also embed other template in a block or anywhere in the template
- Included template is not a parent of the including template

4.4. View, layout and blocks

For web development, the most common solution for organizing your code nowadays is

the MVC design pattern. In short, the MVC design pattern defines a way to organize your code according to its nature. This pattern separates the code into **three layers**:

- The **Model** layer defines the business logic (the database belongs to this layer). You already know that Symfony stores all the classes and files related to the Model in the Entity/ directory of your bundles.

- The ***View*** is what the user interacts with (a template engine is part of this layer). In Symfony 3.4, the View layer is mainly made of Twig templates. They are stored in various Resources/views/ directories as we will see later in these lines.

- The ***Controller*** is a piece of code that calls the Model to get some data that it passes to the View for rendering to the client. When we installed Symfony at the beginning of this tutorial, we saw that all requests are managed by front controllers (***app.php*** and ***app_dev.php***). These front controllers delegate the real work to **actions**.

The Stylesheets, Images and JavaScripts

As this is not about web design, we have already prepared all the needed assets we will use for HRM.

You can download any free bootstrap template to use in our project or the trainer will give you link to download the template. Put the image files into the ***src/BuleHoraHRMBundle/Resources/public/images/*** directory; put the stylesheet files into the ***src/BuleHoraHRMBundle/Resources/public/css/*** directory; and put the javascript files into ***src/BuleHoraHRMBundle/Resources/public/js***.

Now run the following command to make these files publicly available. In other word, to copy the shortcut into web folder of our Symfony application.

```
php bin/console assets:install web --symlink
```

The Layout and Bootstrap

If you take a look at many professional applcations, you will notice that much of each page looks the same. You already know that code duplication is **bad**, whether we are talking about HTML or PHP code, so we need to find a way to prevent these common view elements from resulting in code duplication.

One way to solve the problem is to define a header and a footer and include them in each template. A better way is to use another design pattern to solve this problem: check decorator design pattern. The decorator design pattern resolves the problem the other way around: the template is decorated after the content is rendered by a global template, called a ***layout***.

Symfony does not came with a default layout, so we will create one and use it to decorate our application pages.

Create a new file ***layout.html.twig*** in the ***app/Resources/views/*** directory and put in the following code:

app/Resources/views/layout.html.twig

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta name="description" content="">
   <meta name="author" content="">
   <title>{% block title %} Home | HRM {% endblock %} </title>
   {% block stylesheets %}
      <link rel="stylesheet" href="{{ asset('bundles/bulehorahrm/css/bootstrap.min.css') }}"
type="text/css" media="all" />
   {% endblock %}

</head><!--/head-->

<body>
{% block header %}
<header id="header"><!--header-->

</header><!--/header-->
{% endblock %}
{% block content %}

{% endblock %}

{% block footer %}
<footer id="footer"><!--Footer-->
</footer><!--/Footer-->
{% endblock %}

{% block javascripts %}
   <script type="text/javascript"
         src="{{ asset('bundles/bulehorahrm/js/jquery.js') }}">
   </script>
{% endblock %}

</body>
</html>
```

- **Twig Blocks**

In Twig, the default Symfony template engine, you can define **blocks** as we did above. A twig block can have a default content (look at the title block, for example) that can be replaced or extended in the child template as you will see in a moment.

Now, to make use of the layout we created, we will need to edit all the item templates (***index, edit, new and show*** from ***app/Resources/views/item/***) to extend the parent template (the layout) and to overwrite the content block we defined with the body block content from the original template.

app/Resources/views/item/index.html.twig

```
{% extends 'layout.html.twig' %}

{% block content %}
```

```
    <!-- original body block code goes here -->

{% endblock %}
```

To add a new css file in a template, we will overwrite the stylesheet block, but call the parent before adding the new css file (so we would have the main.css and the additional css files we need).

```
{% extends 'layout.html.twig' %}

{% block stylesheets %}
    {{ parent() }}
    <link rel="stylesheet" href="{{ asset('bundles/bulehorahrm/css/jobs.css') }}"
type="text/css" media="all" />
{% endblock %}
```

Exercise

CRUD for all HRM Entities

5. Routing

5.1. Routing

If you click on show item on the HRM items list, the URL looks like this: /item/1/show. If you have already developed PHP websites, you are probably more accustomed to URLs like /item.php?id=1. How does Symfony make it work? How does Symfony determine the action to call based on this URL? Why is the ***id*** of the item retrieved with the $id parameter in the action? Here, we will answer all these questions.

You have already seen the following code in the ***src/BuleHoraHRMBundle/Resources/views/item/index.html.twig*** template:

```
{{ path('item_show', { 'id': entity.id }) }}
```

This uses the path template helper function to generate the url for the item which has the id 1. The ***item_show*** is the name of the route used, defined in the configuration as you will see below.

```
item_show:
    path:     /{id}/show
    defaults: { _controller: "BuleHoraHRMBundle:Item:show" }
    methods: GET
```

5.1.1. Routing Configuration

In Symfony 3, routing configuration is usually done in the ***app/config/routing.yml***. This imports specific bundle routing configuration. In our case, the ***src/BuleHoraHRMBundle/Resources/config/routing.yml*** file is imported:

```
bule_hora_hrm:
    resource: "@BuleHoraHRMBundle/Resources/config/routing.yml"
    prefix:   /
```

Now, if you look in the ***src/BuleHoraHRMBundle/Resources/config/routing.yml*** you will see that it imports another routing file, the one for the item controller:

```
bule_hora_hrm_item:
    resource: "@BuleHoraHRMBundle/Resources/config/routing/item.yml"
    prefix:   /store
```

Open and look at ***src/BuleHoraHRMBundle/Resources/config/routing/item.yml***

```
item_index:
    path:     /
    defaults: { _controller: "BuleHoraHRMBundle:Item:index" }
    methods:  GET

item_show:
    path:     /{id}/show
    defaults: { _controller: "BuleHoraHRMBundle:Item:show" }
    methods:  GET

item_new:
    path:     /new
    defaults: { _controller: "BuleHoraHRMBundle:Item:new" }
    methods:  [GET, POST]

item_edit:
    path:     /{id}/edit
    defaults: { _controller: "BuleHoraHRMBundle:Item:edit" }
    methods:  [GET, POST]

item_delete:
    path:     /{id}/delete
    defaults: { _controller: "BuleHoraHRMBundle:Item:delete" }
    methods:  DELETE
```

Let’s have a closer look to the ***item_show*** route. The pattern defined by the ***item_show***

route acts like /*/show where the wildcard is given the name id. For the URL /1/show, the ***id*** variable gets a value of 1, which is available for you to use in your controller. The _controller parameter is a special key that tells Symfony which controller/action should be executed when a URL matches this route, in our case it should execute the ***showAction*** from the ***ItemController*** in the ***BuleHoraHRMBundle***.
The route parameters (e.g. {id}) are especially important because each is made available as an argument to the controller method.

5.1.2. Route Requirements

The routing system has a built-in validation feature. Each pattern variable can be validated by a regular expression defined using the requirements entry of a route definition:

```
item_show:
   pattern:  /{id}/show
   defaults: { _controller: " BuleHoraHRMBundle:Item:show" }
   requirements:
      id:  \d+
```

The above requirements entry forces the id to be a numeric value. If not, the route won't match.

5.1.3. Route Debugging

While adding and customizing routes, it's helpful to be able to visualize and get detailed information about your routes. A great way to see every route in your application is via the ***router:debug*** console command. Execute the command by running the following from the root of your project:

```
php bin/console router:debug
```

The command will print a helpful list of all the configured routes in your application. You can also get very specific information on a single route by including the route name after the command:

```
php bin/console router:debug item_show
```

5.2. Flash Messages

Flash messages are small messages you can store on the user's session for exactly one additional request. This is useful when processing a form: you want to redirect and have a special message shown on the next request.
When users add/edit item to the system, they want to know if the operation was successful or not. So, we use flash messages to notify the user about the result. Once the message is shown, the message will be removed from the page when the page s reloaded/refreshed.

src/BuleHoraHRMBundle/Controller/DefaultController.php

```
public function indexAction(Request $request)
{
  $em = $this->getDoctrine()->getManager();
  $items = $em->getRepository('BuleHoraHRMBundle:Item')->findAll();
```

```
  if(!$items)
   {
       $this->get('session')->getFlashBag()->add('notify', 'Warning: No Item Found!!');
   }
   else
     $this->get('session')->getFlashBag()->add('success', 'Success: Items were Retrieved');
}
```

At last, you will add a div in layout.html.twig where we can display the flash message:

src/Ibw/JobeetBundle/Resources/views/layout.html.twig

```
<div class="row">
    <div class="col-sm-12">
        {% for flashMessage in app.session.flashbag.get('error') %}
            <div  class="alert alert-danger">
                {{ flashMessage }}
            </div>
        {% endfor %}
        {% for flashMessage in app.session.flashbag.get('success') %}
            <div class="alert alert-success">
                {{ flashMessage }}
            </div>
        {% endfor %}
    </div>
</div>
```

Exercise

Add flash message in all HRM actions

6. Bundles

Third party bundle

Exercise

full installation of fosBundle

7. Form basics

Create simple Form

Submit and validate a form

Generate form for entity

Customize form and add Validation rules

Exercise

add validation and UI classes in HRM forms

8. Service Container

How to define a service
How to pass a configuration parameter to the service
How to depend on another service

Exercise
Login History Service

9. Console commands

Generate console command
Access service container in console command
Display progress of the command
How to define console command as a service

Exercise
Console command to import Item from exl. document

10. Testing your application

Introduction to PHPUnit
Environment configuration
Code coverage report
Unit testing
Functional testing

Exercise
write test for all HRM Functionalities

11. Doctrine fixtures

Configure fixtures
Load data
Sharing objects between fixtures

Exercise
fixture for all entities in HRM

12. Security

Symfony security

fosbundle

13. Deployment

Configure fixtures

Load data
Sharing objects between fixtures

14. Language

Configure fixtures
Load data
Sharing objects between fixtures

Exercise
Add Amharic support HRM

15. Summary

Topics of interest