Università di Pisa

Corso di Laurea in Informatica

RELAZIONE DI TIROCINIO

Pier

Docker Orchestrator per Web Hosting

Candidato:

Aldo D'Aquino

Tutore accademico:

Prof. Antonio Brogi

Anno Accademico 2016/17

Table of Contents

1 Introduction

1.1 Context

In recent years, the focus on containerization has grown. Containers allow the developer to package an application with all the necessary parts (libraries and other related resources) and deliver it in the form of a single object that is easy to execute.

The purpose of Docker, as they say on their site, is to make it easier to create, deploy and run applications using container (Docker, 2018). However, even the use of Docker requires different skills and can be facilitated with the use of orchestrators that automatically arrange, coordinate, and manage Docker containers. They can be released as an OS program that interfaces with Docker or in the form of a container itself and most of them feature a graphical interface with which the user can conveniently organize and manage running applications.

1.2 Aim of the internship

The aim of the internship was to automatize the creation of web spaces through Docker containers by providing a specific orchestrator for site hosting.

The orchestrator must have a simple and useful interface through which the user will choose the configuration he wants and it will have to take charge of this request automatically. This means that it will have to create and manage all the containers necessary for that website, from FTP to the Database, to the website itself.

In this way the user will not have to worry about creating the infrastructure but will only choose the features he wants to have.

1.3 Why objective is important

More and more developers are starting to use Docker so much so that in the last year according to Datadog its use has grown by 40% and is now used by 15% of hosts (8 surprising facts about real Docker adoption, 2018). Most of these companies use it for hosting: among the most used images of the Docker Hub, Nginx, Httpd (Apache) and several databases stand out (Explore Docker Hub, 2018).

The advantage of using Docker instead of creating your own tool that suits your needs are clear: it avoids writing a specific software and grants the security and reliability thanks to

continuous updates of an efficient team, relieving the hosting provider from this task. Moreover, Docker is characterized by:

- *safety*: each container is isolated from the others and from the host system, preventing security risks;
- *scalability*: each image can be run in multiple containers at the same time;
- *reusability*: you can save a container snapshot to resume running from that point;
- *extensibility*: each container can be connected to others; it can also mount volumes and expose ports;
- *customizability*: Docker lets you create custom images to suit your needs.

However, although Docker makes things much easier, the implementer must develop an orchestrator to completely automatize all the process.

Open source Docker orchestrators exist but none of them is specific for hosting. Every hosting provider has developed its own specific software which is not available for anyone. There are some more powerful, generic and open source orchestrators, but they must be configured for the specific case and often their configuration is complex for an inexperienced user. We want our orchestrator to be able to perform the following operations automatically and with a minimum configuration:

- Apache or Nginx configuration;
- creation of a volume for the container;
- running of a MySQL container and the linking to the website hosting container;
- configuration of an FTP access for the file transfer;
- smashing of the traffic coming from different domains to respective containers by the using of a proxy.

All these operations can be performed in other orchestrator by hand through a control panel, but this needs a great amount of time. The solution that we are proposing offers an automation of these tasks to simplify the website creation up to reduce it to the compilation of an intuitive form.

This project can be seen as a layer that rests above Docker. Unlike other orchestrators that basically provide a graphical interface for container management, that adds an additional level

of abstraction: some containers are put into execution automatically and many of the operations between containers (e.g. linking) are done by the orchestrator without requiring user intervention.

Indeed, we want to create an orchestrator not of containers but of websites, relieving the user from any unnecessary operation.

1.4 Achieved results

During the internship we successfully developed the requested orchestrator, and we called it Pier.

Pier has achieved the required objective: it creates and manages all the containers necessary to implement the configuration requested by the user and it does so completely automatically. Among these containers we find in particular the FTP service and the MySQL database as required. We have also added phpMyAdmin to make it easier for the user to manage the database.

The interaction with the user takes place both with APIs, to facilitate developers, and through a web GUI that makes the application ready to use for the end user.

Particular attention has been paid to usability, leaving the user the right freedom in choosing the optimal configuration without requiring technical knowledge.

In order to help the developers we have made some choices that allow the application to adapt to different configurations: for example we have installed a proxy to allow the use of Pier even with only one IP address and we used during development Docker CE to guarantee maximum compatibility even without the Enterprise version.

Finally, we tried to make the code as light as possible by eliminating any unnecessary components in order to get a more responsive application.

1.5 Document structure

The rest of this document is organized as follows.

In Chapter 2 we will recall some background information on operating-system-level virtualization and how Docker works.

Chapter 3 describes the design of the orchestrator that we have created, including the analysis of the requirements and the design and architectural choices.

Chapter 4 illustrates the implementation and structure of Pier, analyzing the modules that compose it and how they interoperate with each other. The main implementation choices are also explained.

Chapter 5 illustrates Pier at work, showing some examples of use.

Finally, Chapter 6 is devoted to conclusions, providing a critical look also to the limits of the project and to the possible future implementations and extensions of Pier.

2 Background: Docker containers

2.1 Containerization

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances. Such instances, called containers, partitions or virtualization engines, may look like real computers from the point of view of programs running in them.

A computer program running on an ordinary person's computer's operating system can see all resources (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside a container can only see the container's contents and devices assigned to the container.

Therefore, from a certain point of view it is not far from a virtual machine. But unlike a virtual machine, instead of creating an entire virtual operating system, containerization allows applications to use the same Linux kernel as the system they are running on.

On Unix-like operating systems, this feature can be seen as an advanced implementation of the standard chroot mechanism, which changes the apparent root folder for the current running process and its children. The operating-system-level virtualization is not new: the chroot mechanism is available since 1979 in most UNIX-like operating systems (Aqua Security Software Ltd., 2016). However, it is back in vogue recently, first in 2001 with Linux-VServer (Linux-VServer Project, 2011) also called security context and then booming in 2013 with Docker containers (About Docker, 2018).

2.2 About Docker

Docker is a Linux-based platform for developing, shipping, and running applications through container-based virtualization.

As we said container-based virtualization exploits the kernel of the operating system of a host to run multiple isolated user-space instances, that in Docker are called containers. Each Docker container packages the applications to run, along with whatever software support they need (e.g., libraries, binaries, etc.).

Containers are built by instantiating so-called Docker images, which can be seen as read-only templates providing all instructions needed for creating and configuring a container. Existing Docker images are distributed through so-called Docker registries (e.g. Docker Hub), and new images can be built by extending existing ones.

Docker containers are volatile, and the data produced by a container is (by default) lost when the container is stopped. This is why Docker introduces volumes, which are specially-designated directories (within one or more containers) whose purpose is to persist data, independently of the lifecycle of the containers mounting them. Docker never automatically deletes volumes when a container is removed, nor it removes volumes that are no longer referenced by any container.

Docker also allows containers to intercommunicate. It indeed permits creating virtual networks, which span from bridge networks (for single hosts), to complex overlay networks (for clusters of hosts).

3 Design of Pier

3.1 Requirements analysis

The orchestrator is required to manage the creation and dismantling of containers and the volumes associated with them. Therefore, it will need to communicate with the daemon and execute commands that interact with Docker. No implementation requirements are specified, so we think these two hypotheses are reasonable: a bash script able to work with the Docker CLI or another script (PHP or other) that will necessarily have to interface with the daemon but can be inserted into a container. The first option is easier to make, the second is more complex but more flexible.

Not knowing if we have the availability of dedicated IP addresses in order to assign one to each container, we consider the case in which the whole server has only one IP address. Since the containers hosting the websites will not have their own IP address, every domain associated with a website will be made to point to a single address. We will therefore need to install a proxy to sort incoming traffic on the right container. This proxy can run on a container.

To manage the upload of files on the site we will need an FTP connection. This service can be placed in a dedicated container or in the container of the site itself. It can also be installed on the host server and work out of the Docker environment, but although there are no precise indications about it we find it inconsistent with the goal of the job. The configuration of FTP accounts is required to be borne by the orchestrator, so it must be able to communicate with the chosen solution.

As for the volumes, there is a constraint that they are all visible to the orchestrator and the FTP service. Furthermore, each site will have to mount its container.

The use and configuration of a database to be associated with websites is also required. We think the choice of MySQL is reasonable as it is one of the most used in this area (DB-Engines Ranking, 2018). In this case it is possible to insert the database directly in the container of the website, in an associated container or to create a single container for all the databases. Once again, we consider inopportune the choice to run MySQL natively on the system.

The orchestrator must be able to communicate with MySQL in order to manage accounts and databases. For that reason, the choice of associating an external container to each website could make things more complex.

Since we want the application to do things in an easier way, it is necessary that the interaction with the orchestrator is through simple APIs or a graphical interface.

There are no requirements on the system to use but we think it is appropriate to choose the ubuntu x86 as demo environment to maximize compatibility and Docker CE to be sure that everyone can use it.

3.2 Design choices and architecture

We chose to package the Docker orchestrator in a container to ensure the greatest compatibility and better security. This also makes deployment easier. However, this involves a more complex way to interact with Docker. To get around this, it was necessary to mount on the container the socket used by the Docker to communicate with the daemon (`/var/run/docker.sock`). Communication with the socket inside the container is provided by docker.io.

We called this main container Builder.

We have chosen to request registration of users on the Builder container in order to make the project more complete and applicable to the public. Those who will extend it can, for example, request the payment of a fee for the creation of a site. In this way we also allow each user to manage only his sites.

To sort the incoming traffic on the container associated with the domain we used Jason Wilder's nginx-proxy container (Wilder, 2018).

For the FTP service we have chosen to create a special container to carry out this work. The possibility of inserting it into Builder has been discarded because we thought it was necessary to keep the two separate, to keep the Builder's performance high. We also preferred not to include this functionality in the container hosting the website because it would not be possible to use the default images for Apache and Nginx and because it was difficult for the Builder to manage users. Between the possibility to create an FTP container associated to each site and to create a single one for all we preferred the second one. This way you do not create too

many containers keeping the system's performance high. In case of need it is possible at any time to scale up the FTP service by putting in place several instances of it, that can coexist with each other.

User accounts on the FTP container are created by the Builder upon registration.

The problem of the volumes has been solved in the following way: at the start the site folder that will host all the volumes is prepared; when registering a user, the relative folder is created in sites. When the user creates a site, a folder will be created in the user folder. For example, we will have a result like:

```
sites/
  user1/
    site215/
    site543/
  user3/
    site376/
```

The sites folder will be mounted both by the Builder and by FTP, so that the first one can create the volumes for each site and the second one can manage them. Each user will only have access to his folder (e.g. user1) where he will find all his sites.

Similarly, to the FTP service MySQL also resides on its own container. indeed, by inserting it in Builder we lost the scalability while in the site's container the efficiency.

However, only one container for all databases may not be sufficient, and being a database it is not possible to have multiple instances due to problems of synchronous data consistency. We could optimize by dividing sites into buckets and allocating an instance of MySQL to every bucket, however this solution is more complex to implement and is left to future developments.

Needing all these auxiliary containers the deployment of the system is a bit complex. For this reason we have created a bash script that takes care of running all the necessary containers, without the webmaster having to deal with them.

On the interface side, we decided to develop both APIs and a web-based interface. We thought it important to provide APIs as it allows you to write your own GUI on all platforms without having to change anything in the code. However, in order to provide the application

already ready for use, the APIs have been accompanied by a minimal and unpretentious web interface which, however, allows its use. The interface obviously uses APIs and can therefore also be used as a basis for writing one's own.

3.3 Containers overview

Giving a wider view, Pier once running consists of 2 main containers, Builder and its database Builder DB, and 4 auxiliary containers: Proxy, FTP, MySQL and phpMyAdmin. To these are added the containers containing the sites.

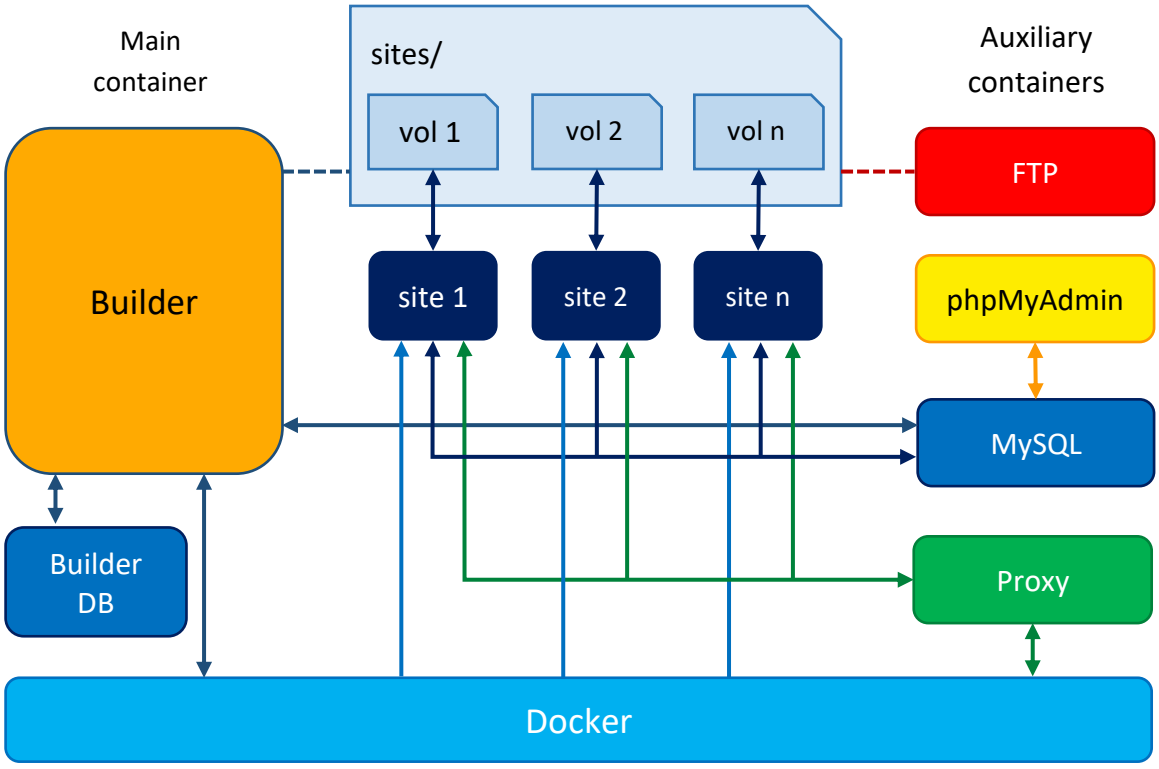

Fig. 1The Pier containers scheme

This configuration is visible with the docker ps command that shows the running containers. After starting Pier the result is the following:

```
daquinoaldo@DWB-Paris1:~$ sudo docker ps
CONTAINER ID        IMAGE                     COMMAND                  CREATED             STATUS
    PORTS                                                NAMES
2d774c41f4e8        daquinoaldo/builder       "/bin/sh -c '/usr/..."   2 hours ago         Up 2 hours
    0.0.0.0:8080->80/tcp                                 builder
da474eb0b1f7        mysql                     "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:8000->3306/tcp                               builder-mysql
58ef9e35aff7        phpmyadmin/phpmyadmin     "/run.sh phpmyadmin"     2 hours ago         Up 2 hours
    0.0.0.0:8888->80/tcp                                 phpmyadmin
cab321727506        mysql                     "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:3306->3306/tcp                               mysql
1411057778db        daquinoaldo/ftp           "/bin/sh -c '/run...."   2 hours ago         Up 2 hours
    0.0.0.0:21->21/tcp, 0.0.0.0:30000-30009->30000-30009/tcp   ftp
56b09b6b6300        jwilder/nginx-proxy       "/app/docker-entry..."   2 hours ago         Up 2 hours
    0.0.0.0:80->80/tcp                                   nginx-proxy
```

Fig. 2 The result of docker ps after starting Pier

3.3.1 Builder

Builder is the main container: the core of the orchestrator is here. It also contains the graphical interface and the APIs and it is with this container that the user interacts. It is the Builder's responsibility to manage all the other containers.

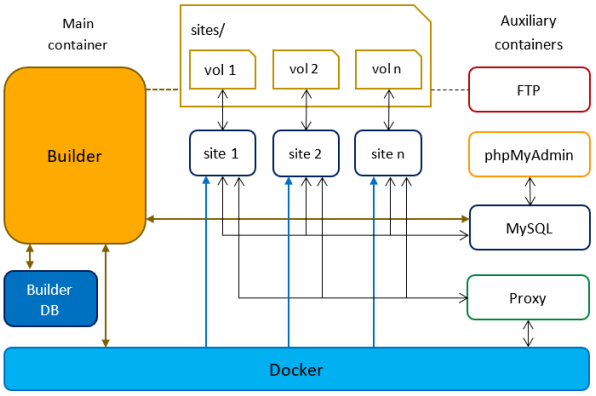

Fig. 3 Builder diagram

The image of the container Builder is generated by the Dockerfile.builder we wrote. It runs on port 8080 because the 80 as we will see later is reserved by the proxy but can be reached directly with its own domain (currently builder.wwharf.aldodaquino.com).

Builder needs to store certain information about users in a persistent manner and to do so uses a MySQL database that resides in the container builder-mysql.

3.3.2 Builder DB

This container contains tables with users and their respective websites. It runs the mysql image of the standard library on Docker Hub (library/mysql). It is visible on port 8000 from all the containers that are on localhost, including the Builder. The Builder logs in as root and has complete control over it.

It should not be considered as an auxiliary component but as a part of the Builder although it is implemented in a separate container for the reasons described above.

3.3.3 Proxy

The proxy container runs the nginx-proxy image of Jason Wilder. Port 80 is dedicated to it and it filters all incoming traffic. If the domain that points to the IP of the machine is associated to a container, it will serve to the user the content of that container.

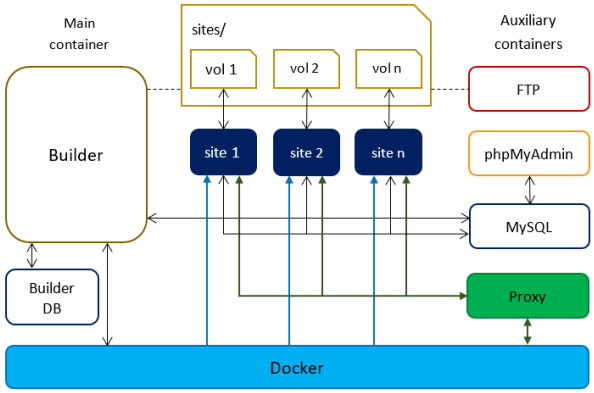

Fig. 4 Proxy diagram

To do this it needs to dialogue with the Docker daemon mounting the socket to see the environment variables. Each container will be associated with one or more domains by setting the VIRTUAL_HOST environment variable.

3.3.4 FTP

The FTP container mounts the sites folder containing all the sites. Every time a user registers to the service the Builder creates a user on this container too. Furthermore, for each registered user, the user folder in sites is also created by the Builder.

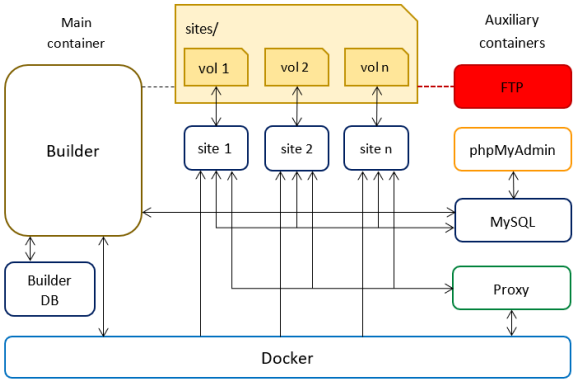

Fig. 5 FTP diagram

In order to do these operations, the Builder calls the exec on the Docker daemon and executes a bash script contained in this container.

The FTP can be accessed at ftp.pier.aldodaquino.com (or at the address specified by the system implementer) and use the standard port 21.

Username and password are the same that you have used to register on the Builder. Once logged in, the user will see his home containing all his sites.

To create the image of this container we started from Docker Pure-ftpd Server by Andrew Stilliard (Stilliard, 2018) and we entered the script necessary for user management.

3.3.5 MySQL

The MySQL database uses the Docker Hub library image (library/mysql), the standard port 3306 and is directly accessible with any domain points to the shared IP address, including the domains of the websites hosting the machine. Furthermore, all websites can connect via PHP on localhost.

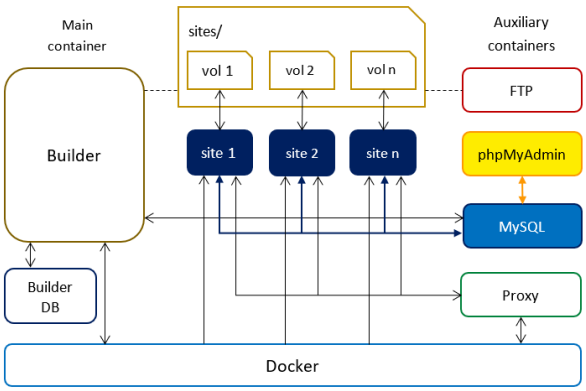

Fig. 6 MySQL and phpMyAdmin diagram

When registering a user, the Builder creates the user also on this container. In the same way, it creates a database every time you create a site that requires one.

To do this it connects with the root user to port 3306 on localhost and makes SQL calls via PHP behaving like a normal website.

3.3.6 phpMyAdmin

We found appropriate to dedicate a container also to phpMyAdmin to allow site owners to have a more convenient interface for the database. It runs on an additional container that sees MySQL.

It runs in a container that mounts the official image (phpmyadmin/phpmyadmin on Docker Hub) on port 8888 but is accessible with its subdomain (phpmyadmin.pier.aldodaquino.com).

Like other containers you can see MySQL because it is on the same machine and on the standard port.

4 Implementation of Pier

4.1 Inside containers: a closer look to the Builder

As previously stated, some auxiliary containers are already ready and unmodified images. For this reason, in this section we will focus only on the Builder, which is the main container containing the Orchestrator, taking a look also at the script that creates the FTP users and deepening a bit its DB Builder database.

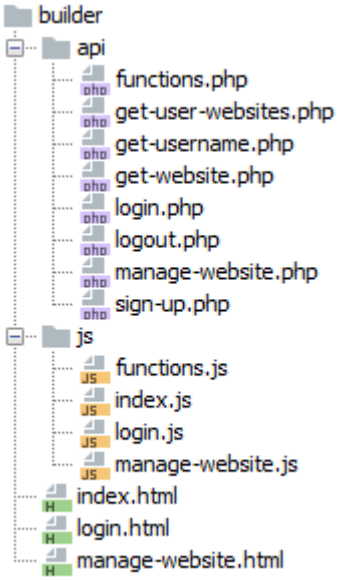

Fig. 7 Builder folder tree

The Builder is divided into 2 modules: the frontend and the backend.

The **backend** is contained in the API folder and the main part is in functions.php. The remaining files expose the functions of functions.php through the API, making operations easier.

The **frontend** consists of a few html pages and the js folder containing javascript scripts that interact with the APIs.

4.1.1 Backend: functions.php

The entire backend is written in PHP. The main file *functions.php* contains 3 main blocks of functions: those for the login, those for managing the database Builder DB and those for interacting with Docker.

Access is managed via the PHP session that solve the stateless problem of HTTP by storing user information to be used across multiple pages. By default, session variables last until the user closes the browser. Session variables hold information about one single user and are available to all pages he visits.

The login function searches for the user's data in the database, encrypts the given password and compare it with the encrypted one present in the database and if they match opens the session by saving the user's username.

The *getUsername* function is also provided, which returns the user's username if it is logged in or null if it has not been accessed. This function is particularly used by APIs that do not allow you to do any action unless you have logged in before.

The registration provided by *signUp* function saves the user's data (username, password and email) in the DB Builder database, creates the user on the MySQL database dedicated to the sites and the FTP container. Finally, it saves the user's username in the session variables.

The creation of the user on the **FTP container** is accomplished through the *ftpAddUser* function which just do *docker exec* to execute a bash script on the FTP container.

The bash script in turn automates two operations: the user's creation on ubuntu and the `pure-pw useradd` command for the user's association to the FTP service.

The creation of the user in the **MySQL database** as previously mentioned is instead realized through a direct connection. The *mySqlAddUser* function connects to the MySQL database and executes an SQL statement that creates the user.

The second block of functions takes care of **managing the Builder DB database**: every website created is saved in the database with all the necessary information.

The *addWebsiteToDatabase* function takes care of saving all the data of a website in a database. These include: username of the user who registers the site, the domain associated with it, the port number (from 8001 to 8999, excluding 8080 and 8888 reserved by the Builder and phpMyAdmin), the type of web server (Apache or Nginx) and whether or not the user has requested the use of PHP (only for Apache) and a MySQL database (only with PHP active).

The port number is calculated with the auxiliary *getPort* function which takes the highest port number from the database and calculates the next in the range of those supplied excluding the reserved port numbers.

updateWebsiteInDatabase allows you to modify the configuration of an existing website. In particular it requires the site id and allows to change the domain, the type of web server and to request or relinquish the functionality of PHP and MySQL. It does not allow you to change the user who created it and keeps the port number and the previous id.

Finally, there are the functions *getWebsitesList*, which allows you to see the list of sites of a given user, and *getWebsite*, which retrieves the data of a website given its id. This last function is used by the next block of functions to get information on a container to be put into execution.

The last block deals with **interacting with Docker**.

The main function is *builderRun* which takes the id of the website as a parameter, retrieves the information in the database and if them are correct it executes the requested configuration. To do this it uses some auxiliary functions.

First of all, set up the volume and copy us into an example website. To do this we have written the *recursive_copy* function that copies the entire contents of a *src* folder to the destination folder *dst*.

Then if a database is required it creates it with the *mySqlAddDb* function. This behaves similarly to the *mySqlAddUser* function previously shown. It connects to the MySQL container, creates the database and grants the necessary permissions to the user to manage the database destined to his site.

Finally, prepare the necessary parameters and run the container with the ***dockerRun*** function that takes care of taking the parameters passed and building the command to be executed:

```
shell_exec("sudo docker run -d --name $name -e VIRTUAL_HOST=$domain -
p $port:80 $volume $options $image");
```

Where *$name* is "site[id]", *$domain* is the one specified by the user, *$port* number is between 8001 and 8999, *$volume* is "[sites_path]: /var/www/site/", *$options* is almost even empty and *$image* could be *nginx*, *apache* or *php:apache* depending on the chosen configuration.

builderRun is also used to modify the configuration of an already created container. Indeed, using the *isRunning* auxiliary function, *builderRun* checks whether a container with that name is already running. In this case it will not create the database again and will not touch the volume that already exists. It will just stop (with *stopContainer*) and remove (with *rmContainer*) the previously created container and execute (always with *dockerRun*) a new container with the new configuration and the previous volume and port number.

The decision to write data into databases and immediately re-read them may seem out of place and without any doubt it is not for the benefit of performance. However, we have made this choice to have the security that no container is put into execution without there being a trace in the database; otherwise we would have lost all control over that container and it

would not have been possible to manage it automatically, which would have created performance problems and possible conflicts with subsequent configurations.

4.1.2 API

As already mentioned, all the other files contained in the API folder are used to display the features implemented in *functions.php* safely and with the necessary security checks.

The parsing of the parameters passed by GET or POST is done via the *htmlentities* function with the ENT_QUOTES option.

The API response message is constructed with the *newMessage* function in *functions.php*. The function accepts two parameters: the code, a number equal to 0 if the operation was successful and negative in the event of an error, and the message associated with that code that can be returned to the user or used for the logs. The result of the function is a JSON object with the code and text properties that assume values passed as parameters to the function. For example:

```
{
  code: 0,
  text: "Logged in succesffully."
}
```

The APIs are therefore REST-like but not REST since the answer is not given by an HTTP code, which in our case is always 200 OK, but through a JSON.

login.php simply checks that all the parameters are present, parsing and passing them to the *login* function. Then prepare the result with code 0 if the login was successful, -1 if the username or password is missing, -2 if username and password are wrong. This page must be called with a POST request.

logout.php takes care of disconnecting the user. This function is not present in *functions.php* and is executed directly from this file. To do this, it simply deletes the session variable username. In this way the function *getUsername* will report the user as disconnected at the next action. This page can be called either with a GET request (suggested method) or with a POST, no parameters are required.

The disconnection is not however required: since the login is done through session variables, the user will be disconnected automatically when the browser is closed.

get-username.php only calls the *getUsername* function and encapsulates the result by returning code 0 and username if the user is logged in and -1 with an error message if not yet logged in. This page can be called either with a GET request (suggested method) or with a POST, no parameters are required.

sign-up.php checks whether username, password and email parameters are present or not, parses and it passes them to the *signUp* function. Then, it prepares the result with code 0 if the signup was successful, -1 if the username or password is missing, -2 if username or email are already in database. This page must be called with a POST request.

manage-website.php is the main endpoint because it takes care of creating new sites and modifying existing sites.

First of all, check that the user is logged in, then parsing and checks the parameters: for example, check that the type of webserver is present in the list of available ones. In addition, because the boolean values are represented in databases as TINYINT, it turns the *php* and *mysql* parameters into 0 or 1.

It also takes care of calling the *getPort* function to get an available port number.

When it has finished parsing and checking the parameters, it saves all the data in the database with the *addWebsiteToDatabase* function, obtaining the configuration id. Finally call *builderRun* with the id obtained previously.

The modification of an existing website is done through some checks. First of all, it is checked if the site id is already present in the request: that means the site is already existing and the variable *update* is set to true.

Then the *update* variable is checked before requesting the port number, because if it is an update the existing one is kept, and again before doing *addWebsiteToDatabase*, calling *updateWebsiteInDatabase* instead.

This page must be called with a POST request.

get-user-websites.php checks that the user is logged in and otherwise returns code -1. Then it calls *getWebsitesList* with the user's username and returns a JSON with code 0 and text

containing the list of sites created by the user. This page can be called either with a GET request (suggested method) or with a POST, no parameters are required.

get-website.php checks that the user is logged in and otherwise returns code -1. It checks also the parameter id and if it is not correct returns the code -2. Then, it calls *getWebsite* with the id passed and returns a JSON with 0 code and text containing the requested site information. This page can be called either with a GET request (suggested method) or with a POST, and the id is required.

4.1.3 GUI

The GUI is divided into two parts: html files that describe the page structure and the js files that call the APIs. In this section we will see both, but we will pay more attention to the JavaScript code.

A JavaScript file is associated with each HTML page and implements its actions. In addition, some useful and shared features are present in *functions.js*. We chose to partition the files instead of enclosing all the scripts in one file to speed up the loading of the web page.

Each HTML page will have to load 2 scripts: the one related to its functionality and *functions.js*. That is to the detriment of performance on the first load, but avoids having to re-download the shared code when visiting the other pages.

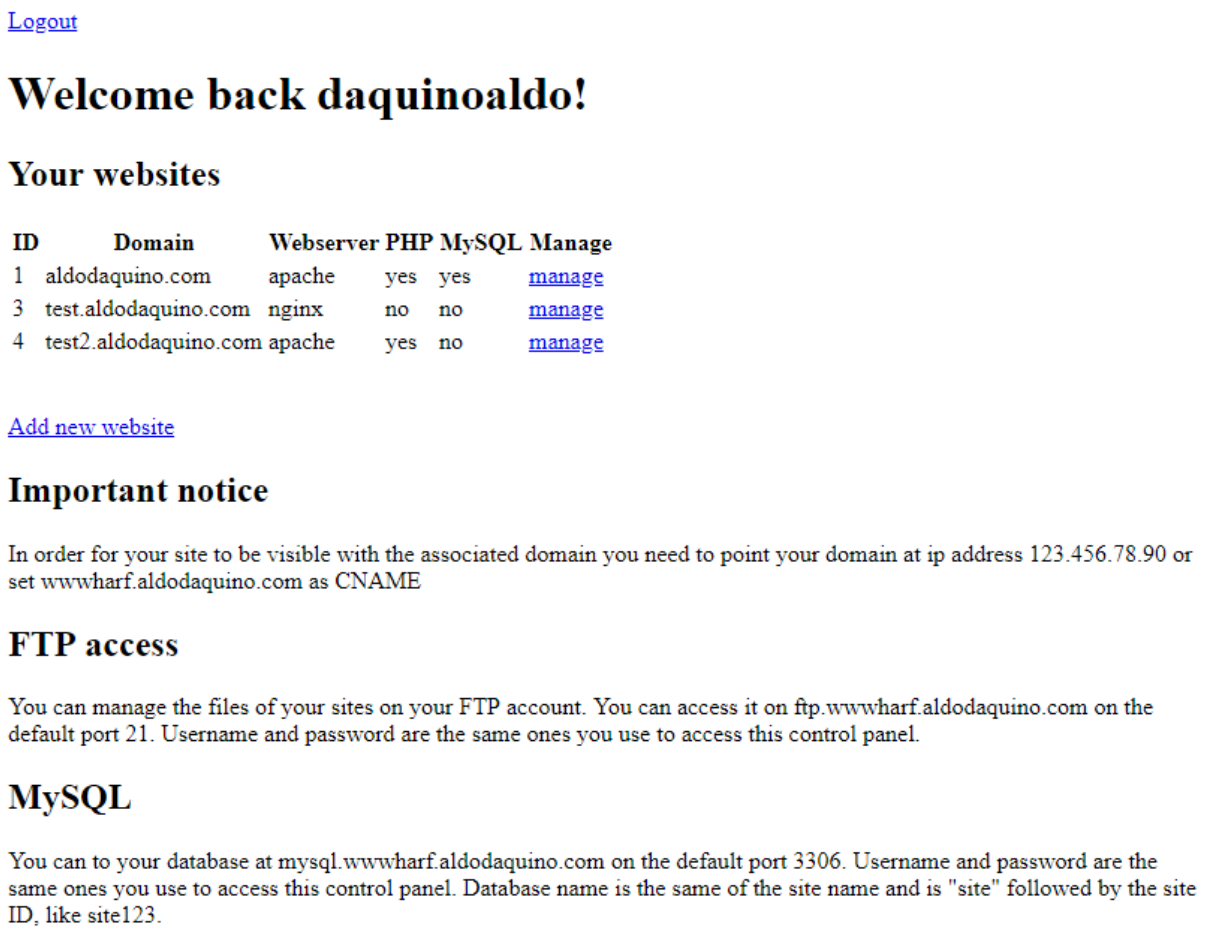

Fig. 8 Screenshot of the index.html page

The ***index.html*** file contains a table with a list of the sites of a user and some useful information to connect to FTP, MySQL database and to point the domain to your space.

It also contains links to disconnect and to create a new site. The first one calls the JavaScript *logMeOut* function that use the *logout.php* API. The second one point to the *manage-website.html* page.

The existing sites can be managed by clicking on the manage link in the site table. The user will then be redirected to the page *manage-website.html?id=[site id]*.

The list of sites is loaded by the *loadWebsitesList* function in *index.js*. It calls the endpoint *api/get-user-websites.php* and insert the result in the table, turning the values of php and mysql from 1/0 to yes/no.

Before loading any other data, however, the script checks that the user has logged in using the *checkLogin* function and if not, the user is immediately redirected to the login page.

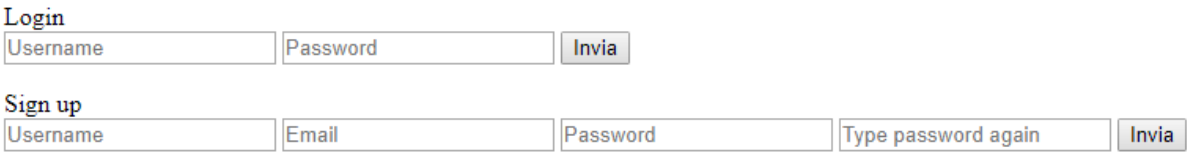

Fig. 9 Screenshot of the login.html page

The ***login.html*** page has two forms: one for login and one for registration.

For the registration the email is also required, and the password must be repeated twice for security.

login.js contains two pairs of functions: the first ones for login and the second for signup.

logMeIn is fired when the user clicks on the submit button or press enter in a text area of the login form. Its task is to make a POST request to the *login.php* API adding username and password in the request payload.

When the answer arrives, it is passed to the *onLoggedIn* function which checks that everything has been successful. If the code in the JSON is negative, then the data was incorrect and a popup is shown with the error message returned by the server. Otherwise the user is redirected to index.html.

Similarly, *signMeUp* sends the signup form data to the *sign-up.php* API. The answer is processed by *onSignedUp* which shows the error message in case something went wrong, otherwise notifies the user that the registration has been successful and redirects it to the index.html page.

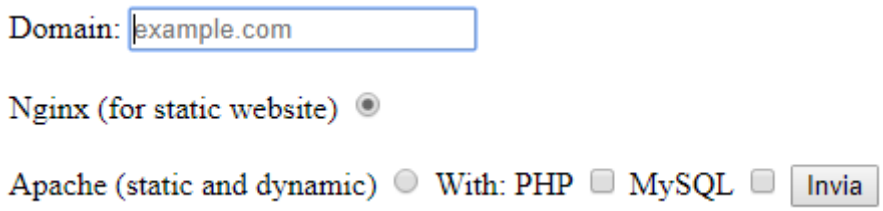

Fig. 10 Screenshot of the add-website.html page

manage-website.html is used to create new sites and modify existing ones.

It basically contains a simple form with the following fields: a text input for the domain, two radio buttons to choose between apache and nginx and two checkboxes for PHP and MySQL.

Stop. Everything that is convenient for an average user in a simplicity understandable even to those who are stranger to the web.

The form is managed by manage-website.js. The first function of this file is checkLogin again, which ensures that the user is logged in.

Then there are three functions that control the radio buttons and checkboxes: *radioHandler*, *mysqlHandler* and *phpHandler*. Indeed, PHP and MySQL are available only on Apache, then selecting Nginx are deselected, while selecting one of these is automatically selected Apache. In addition, the use of the database requires that both active PHP, therefore selecting MySQL is also activated PHP and deselecting the latter is also deactivated MySQL.

Further on we find two functions: *pushWebsite* and *onWebsiteCreated*. As you must realize by now their operation is similar to the functions that manage the forms in *login.js*: the first check the absence of errors in the form and send the data to *manage-website.php*, the second is the callback that indicates the presence of errors or redirects to the home if everything is successful.

The file also contains another important function, *retrieveWebsiteInfo*, which is called by *checkLogin* if the user is logged in. Its task is to check if the id of the website is present in the URL: in this case the user is not creating a new site but modifying one created previously.

The function adds the site id as hidden input to the form and requests the server for information on the site by calling the endpoint *get-website.php?id=[site_id]*. If the id is correct and the site has been created by the current user, the server responds with the current configuration of the website and the form is updated to match the current values, otherwise it will be left blank and the user will be able to create a new site from scratch.

The last file is ***functions.js*** which we said contain the auxiliary functions shared by the other files. The main function is *xhr* that accept as parameter the http request *method*, the endpoint *url* to be called, the payload *data* that can be null, and the *callback* to which the result of the request must have passed. *xhr* will take care of parsing the payload and creating the request with the appropriate parameters.

To speed up the use of this function, we prepared also two auxiliary functions, *get* and *post*, which take care of calling *xhr* passing as *method* respectively GET and POST.

4.1.4 Builder DB

The following is the structure of the Builder DB database in which website information is saved.

The database is initialized at the start of Pier with the content of the init.sql file, which is shown below.

```
CREATE DATABASE builder;
USE builder;
CREATE TABLE `users` (
  `username` VARCHAR(255) NOT NULL PRIMARY KEY,
  `password` CHAR(32) NOT NULL,
  `email` VARCHAR(255) NOT NULL UNIQUE
)
CREATE TABLE `websites` (
  `id` int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `username` VARCHAR(255) NOT NULL,
  `domain` VARCHAR(255) NOT NULL UNIQUE,
  `port` INT NOT NULL UNIQUE,
  `webserver` VARCHAR(6) NOT NULL,
  `php` BOOLEAN,
  `mysql` BOOLEAN,
  FOREIGN KEY (username) REFERENCES users(username)
)
```

As we can see there are 2 tables: one for users and one for websites.

The user table contains the username, the encrypted password and the email. Username and email can not exceed 255 characters and have VARCHAR type to not occupy unnecessary space. The password instead is exactly 32 characters since this is the length of the output of md5. Note that being of fixed length it is of CHAR type that is more performing than VARCHAR.

The PRIMARY_KEY is the username but there is a constraint also on the email that obviously must be unique.

The website table has the following columns: site id, username of the user who created it, the associated domain, the container port on localhost, the type of server and the PHP and MySQL features.

The id is the PRIMARY_KEY of this table. The username must be present in the users table, so the two columns have the same properties.

The domain also has a limit of 255 characters and must be unique because one domain can not redirect two containers. For the same reason, the port number also has the UNIQUE constraint.

The type of webserver is a 6-character VARCHAR, which is the length of Apache.

Finally, php and mysql have type BOOLEAN, which is a synonym of TINYINT. Therefore, they can have a value of 1 or 0.

4.1.5 init.sh

The bash script starts with some environment variables. Among these, in particular, we find *sites_folder*, that contains the path in which the files of the websites are saved on the machine, and the domain associated to the machine. The containers will then be accessible to the addresses builder.[domain], ftp.[domain] and mysql.[domain].

There is also the variable images containing the list of images to be pulled. The script takes care to pull all the necessary images before starting the system, to not slow down the user in the creation of his site. The images are ubuntu, jwilder/nginx-proxy, stilliard/pure-ftpd:hardened, mysql, phpmyadmin/phpmyadmin, httpd, php:apache and nginx.

Then there is the section that deals with the parsing of the options.

The *-s* or *--skip* option allows you not to pull images, which is useful for a quicker start if you are sure they are all up-to-date.

The *-f* or *--site_folder* option allows you to change the path of the folder. The *-d* or *--domain* option specifies the domain.

Sites folder is prepared copying an example directory in the specified path, and then the log folder: all the logs will be placed in it both for the creation and implementation of the system and for the individual containers that compose it.

The pull of the images is implemented as follows:

```
if [ ${SKIP} == false ]
then
```

```
  # Pull the images
  echo "Pulling images..."
  mkdir log/images
  for var in "${images[@]}"
  do
    printf " - Pulling ${var}... "
    plain_var=${var//\//_}
    docker pull ${var} 1>log/images/${plain_var}.log
2>log/images/${plain_var}.error
    check
  done
fi
```

Also note the printing of the strings to make the webmaster participate in what is happening and the use of the log files inserted in the folder mentioned above.

The check function checks the exit status of the last command and prints done if it is 0 and an error message otherwise.

The last preparation is the images building.

```
docker build -t daquinoaldo/ftp -f Dockerfile.ftp .
1>log/dockerfile.ftp.log 2>log/dockerfile.ftp.error

docker build -t daquinoaldo/builder -f Dockerfile.builder .
1>log/dockerfile.builder.log 2>log/dockerfile.builder.error
```

Finally, all containers are put into operation.

```
docker run -d --name nginx-proxy -p 80:80 -v
/var/run/docker.sock:/tmp/docker.sock:ro jwilder/nginx-proxy
1>log/nginx-proxy.log 2>log/nginx-proxy.error
```

Port 80 is assigned to the proxy and the Docker socket is mounted as volume to read the environment variables of the other containers.

```
docker run -d --name ftp -p 21:21 -p 30000-30009:30000-30009 -e
"PUBLICHOST=localhost" -e VIRTUAL_HOST="ftp.$domain" -v
${sites_folder}:${sites_folder} daquinoaldo/ftp 1>log/ftp.log
2>log/ftp.error
```

The FTP uses port 21 and the high passive ports 30000-30009 and mounts the site folder. The environment variable VIRTUAL_HOST specifies the domain. "PUBLICHOST=localhost" is requested by the image (Stilliard, 2018).

```
docker run -d --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=${rootpw} mysql 1>log/mysql.log 2>log/mysql.error
```

MySQL container uses the MySQL standard port 3306 and has the root password of the database as environment variable.

After running MySQL we wait for the container to finish the configuration and the database to be online.

```
until nc -z -v -w30 127.0.0.1 3306 1>/dev/null 2>/dev/null
do
  printf "Waiting for database connection... "
  sleep 5
done
echo "database online."
```

```
docker run -d --name phpmyadmin --link mysql:db -p 8888:80 -e VIRTUAL_HOST="phpmyadmin.$domain" phpmyadmin/phpmyadmin 1>log/phpmyadmin.log 2>log/phpmyadmin.error
```

phpMyAdmin uses the 8888 port and is linked to the MySQL database. Again, the VIRTUAL_HOST variable specify the domain.

```
docker run -d --name builder-mysql -p 8000:3306 -e MYSQL_ROOT_PASSWORD=${rootpw} -v `pwd`/sql-initdb.d:/docker-entrypoint-initdb.d -d mysql 1>log/builder-mysql.log 2>log/builder-mysql.error
```

The Builder DB command is similar to the MySQL container, but mounts the *sql-initdb.d* folder containing the file that initialize the database, as described in the MySQL repository (MySQL repository, 2017).

```
docker run -d --name builder -p 8080:80 -v /var/run/docker.sock:/var/run/docker.sock -v `pwd`/builder:/var/www/site -v ${sites_folder}:${sites_folder} -e VIRTUAL_HOST="builder.$domain" daquinoaldo/builder 1>log/builder.log 2>log/builder.error
```

Builder uses port 8080 and is available at the builder subdomain. Mount the Docker socket to manage containers and the site folder to creates volumes for the websites.

4.2 Languages

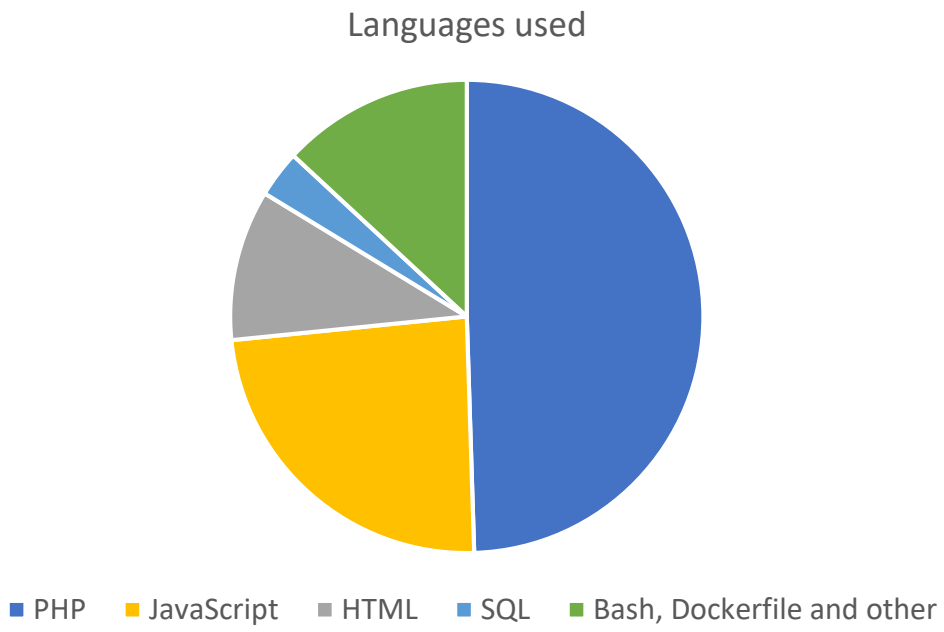

Fig. 11 Pie chart with the languages used for the project

As we have seen among the most used languages for the project there are PHP, in which the backend was written, and JavaScript and HTML for the frontend, as well as the SQL for the creation of the database.

To this we can add the shell command language of the add-user.sh script, that creates users on the FTP container, and of the init.sh.

Then there is the xml language of *apache-config.conf* and that of the Dockerfile for the Builder and our custom FTP container.

5 Pier at work

5.1 Sign-up

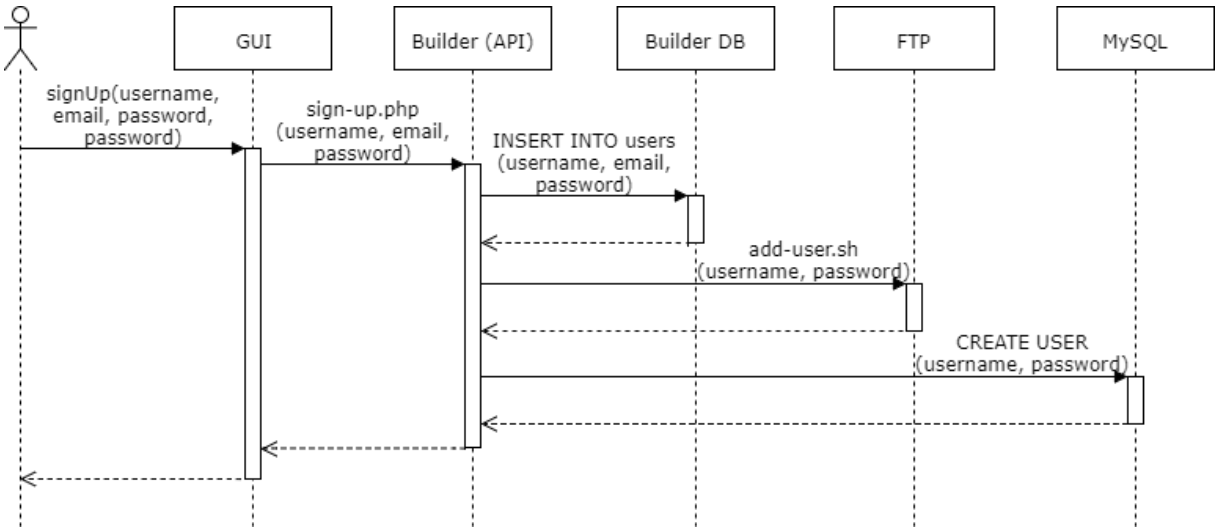

Fig. 12 UML Sequence Diagram of sign-up

The user registers by entering username, email and password repeated twice for security.

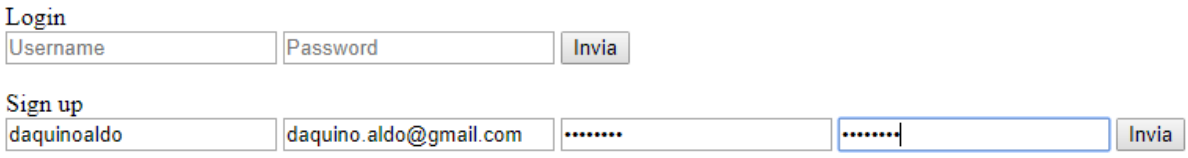

Fig. 13 Screenshot of the sign-up form

JavaScript checks that the two passwords match and sends the data to *sign-up.php*.

Then, the Builder saves the user in the database, creates the user on the FTP container and in the MySQL database.

Finally, it sends the response to the JavaScript script that redirects the user to index.html.

5.2 Login

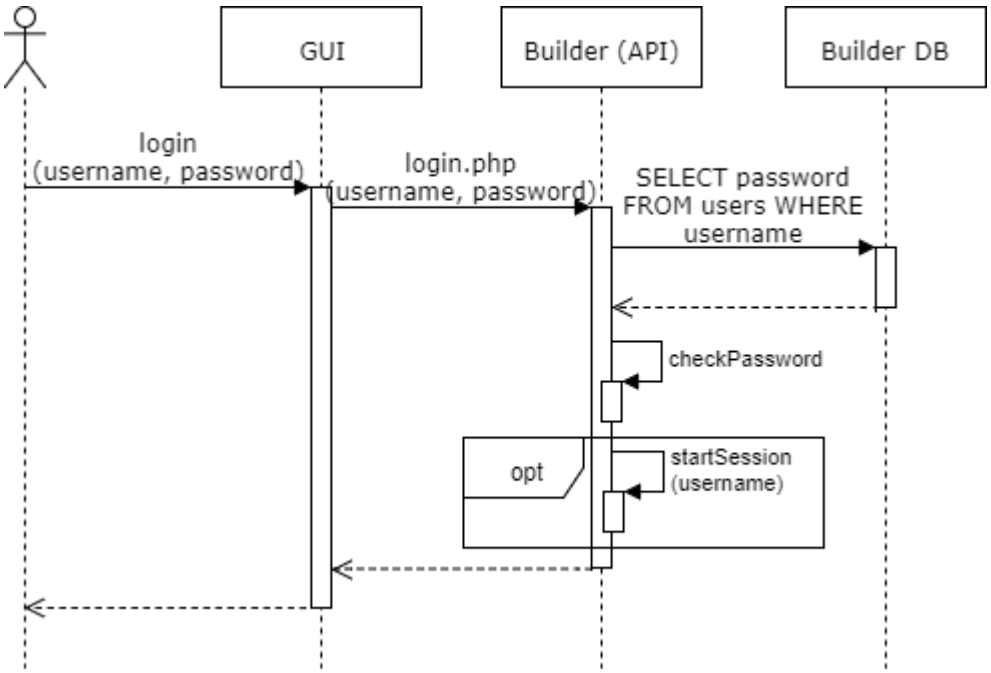

Fig. 14 UML Sequence Diagram of login

The user login by entering username and password.

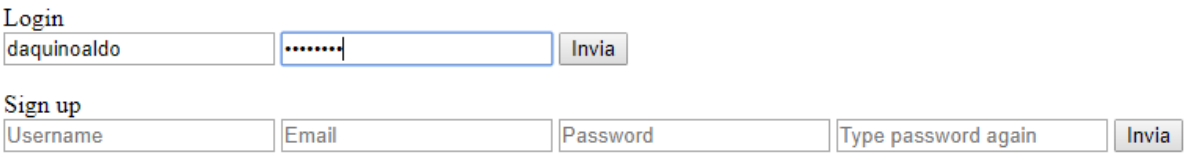

Fig. 15 Screenshot of the login form

When user submits the form, JavaScript sends the data to *login.php*.

The Builder checks that the user exists, and that the password is correct and if everything is okay starts the session.

Finally, send the response to the JavaScript script that redirects the user to index.html.

5.3 Add new website

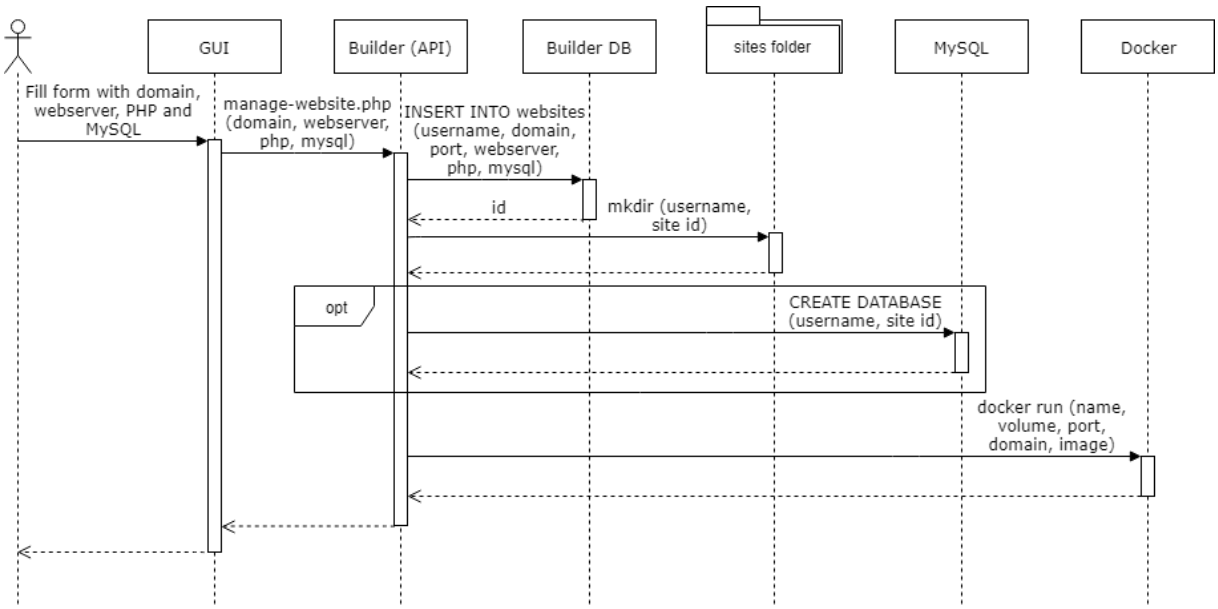

Fig. 16 UML Sequence Diagram of a website creation

The picture shows the workflow to create a website. The user interacts with the web interface in which he must enter the domain and choose the webserver type between Nginx and Apache. If he chooses Apache can also add PHP and MySQL.

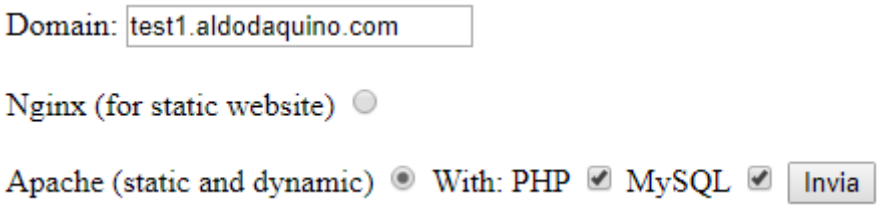

Fig. 17 Screenshot of the form to add new website

The GUI makes the right API request to the Builder that first inserts the site configuration in the Builder DB database, as following.

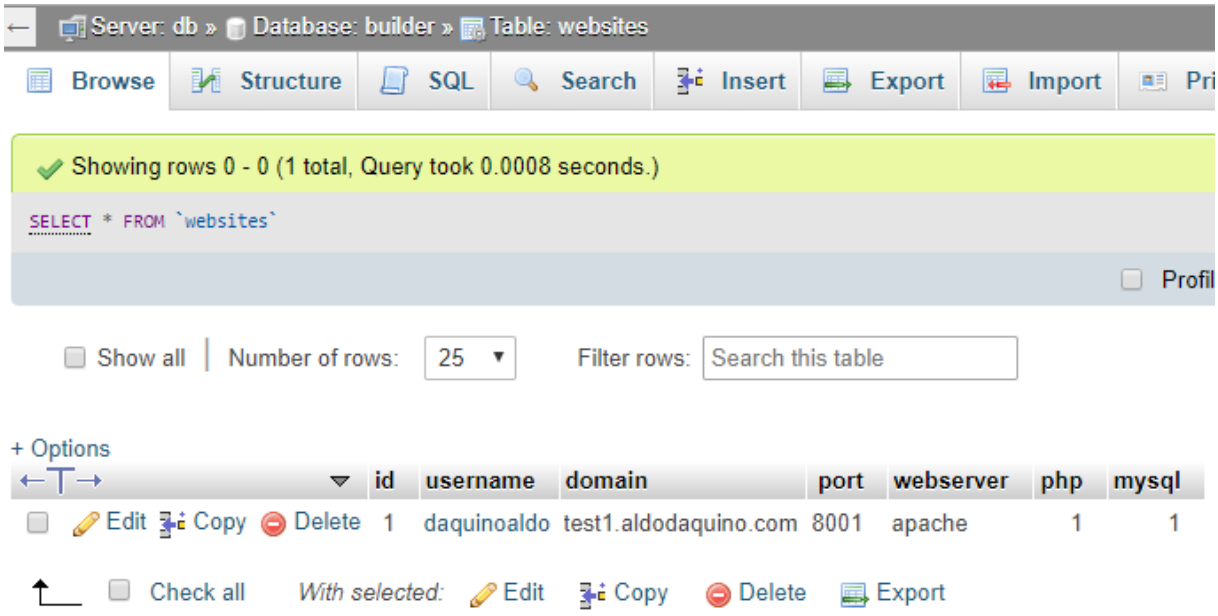

Fig. 18 Screenshot of the database Builder DB after the site creation

Then the Builder creates the site folder in sites and if the configuration includes the use of the database, one is created.

Finally, the Builder executes the run command asking the Docker daemon to execute a container with the image and the required options. The communication with Docker is via socket.

```
daquinoaldo@DWB-Paris1:~$ sudo docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
    PORTS                                                 NAMES
db17c4a20852        php:apache             "docker-php-entryp..."   6 minutes ago       Up 6 minutes
    0.0.0.0:8001->80/tcp                                  site1
2d774c41f4e8        daquinoaldo/builder    "/bin/sh -c '/usr/..."   2 hours ago         Up 2 hours
    0.0.0.0:8080->80/tcp                                  builder
da474eb0b1f7        mysql                  "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:8000->3306/tcp                                builder-mysql
58ef9e35aff7        phpmyadmin/phpmyadmin  "/run.sh phpmyadmin"     2 hours ago         Up 2 hours
    0.0.0.0:8888->80/tcp                                  phpmyadmin
cab321727506        mysql                  "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:3306->3306/tcp                                mysql
1411057778db        daquinoaldo/ftp        "/bin/sh -c '/run...."   2 hours ago         Up 2 hours
    0.0.0.0:21->21/tcp, 0.0.0.0:30000-30009->30000-30009/tcp   ftp
56b09b6b6300        jwilder/nginx-proxy    "/app/docker-entry..."   2 hours ago         Up 2 hours
    0.0.0.0:80->80/tcp                                    nginx-proxy
```

Fig. 19 The result of docker ps after creating a container

The image displays the result of docker ps. As we can see the first container is site1 where 1 is the id of the website, and the image is *php:apache* because php has been requested. The other containers represent the docker infrastructure.

Your websites

ID	Domain	Webserver	PHP	MySQL	Manage
1	test1.aldodaquino.com	apache	yes	yes	manage

Fig. 20 The website list in index.php updated with the created website

When the Builder has finished the operations, the API communicates the result to the web interface. The user is redirect to *index.html* where can see the website created.

5.4 Edit existent website

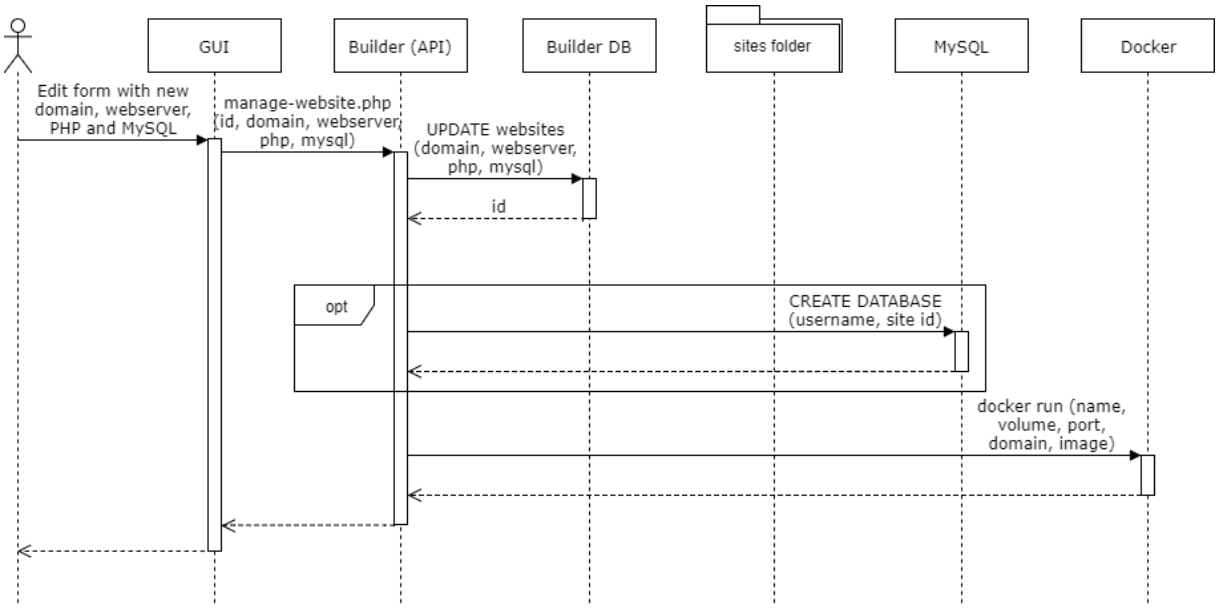

Fig. 21 UML Sequence Diagram of a website update

The workflow to update a website is very similar: in this case the web interface will also pass the site id to the API, in this way the Builder will be able to distinguish that it is an update.

Builder edits the site configuration in the Builder DB database to match changes. Here we can see the result.

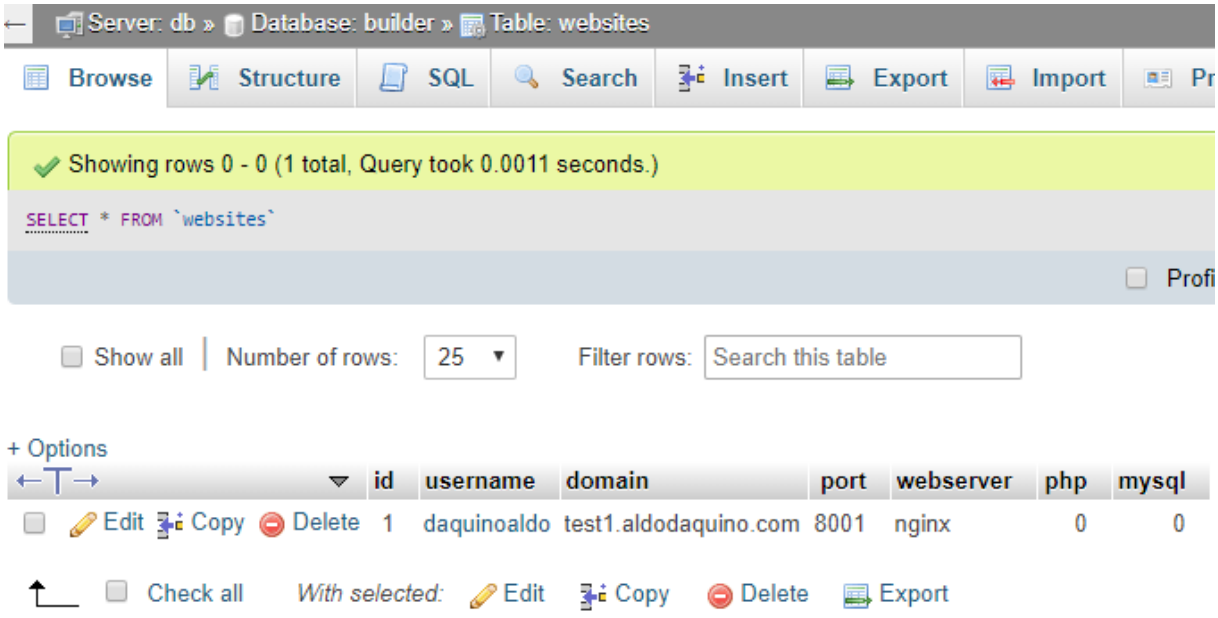

Fig. 22 Screenshot of the database Builder DB after the site creation

Note that the site id and the port number are not changed because is the same website.

This time the Builder does not create the site folder in sites because already exists, but if the configuration now requires the use of a database and had not already been created, one is now created.

Finally, the Builder executes the run command asking the Docker daemon to execute a container with the image and the required options. Before running the new container, the old one is removed.

```
daquinoaldo@DWB-Paris1:~$ sudo docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
    PORTS                                                NAMES
18c5866e77db        nginx                  "nginx -g 'daemon ..."   39 seconds ago      Up 38 second
    0.0.0.0:8001->80/tcp                                 site1
2d774c41f4e8        daquinoaldo/builder    "/bin/sh -c '/usr/..."   2 hours ago         Up 2 hours
    0.0.0.0:8080->80/tcp                                 builder
da474eb0b1f7        mysql                  "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:8000->3306/tcp                               builder-mysql
58ef9e35aff7        phpmyadmin/phpmyadmin  "/run.sh phpmyadmin"     2 hours ago         Up 2 hours
    0.0.0.0:8888->80/tcp                                 phpmyadmin
cab321727506        mysql                  "docker-entrypoint..."   2 hours ago         Up 2 hours
    0.0.0.0:3306->3306/tcp                               mysql
1411057778db        daquinoaldo/ftp        "/bin/sh -c '/run...."   2 hours ago         Up 2 hours
    0.0.0.0:21->21/tcp, 0.0.0.0:30000-30009->30000-30009/tcp   ftp
56b09b6b6300        jwilder/nginx-proxy    "/app/docker-entry..."   2 hours ago         Up 2 hours
    0.0.0.0:80->80/tcp                                   nginx-proxy
```

Fig. 23 The result of docker ps after editing a container

If we check the result of docker ps after a site editing, we can see that the container id is changed because is another container and the image and the command are different, but the name and the port are the same.

When the Builder has finished the operations, the API communicates the result to the web interface.

6 Conclusions

The aim of the internship was to develop a Docker orchestrator to automatize the creation of web spaces through containers.

The orchestrator had to have a simple and useful interface through which the user will choose the configuration he wants. Moreover, it had to take responsibility for this request automatically, creating and managing all the containers necessary for that website, including FTP and Database.

The realized orchestrator meets the requirements: it manages the FTP connection and the database and creates the sites completely automatically. It also provides APIs as required to which we have attached a demo graphical interface. Finally, user interaction with the database is facilitated by phpMyAdmin.

Among the objectives achieved in particular there is the ease of use both by the user and especially by those who will install and customize it for their server and the lightness of the code that aims for performance and effectiveness, without weighing of unsolicited features or unnecessary third-party libraries.

In particular, the ease of using is achieved through the minimalist graphical interface and thanks to APIs that allow you to write from scratch your personalized interface in an extremely simple way, avoiding having to understand the operation of complex web interface.

To achieve the lightness of the code we have instead eliminated any unnecessary component, to improve performance and make sure that every user has only what he needs, without the program being overloaded with features not necessary for him.

The software created can be taken both as a finished application for those who want to manage their server more comfortably and as a starting point easy to be extend by those who want a more specific product without having to write it from scratch.

All the code developed during the internship is Open Source and available on a public repository on my GitHub profile. See the bibliography (D'Aquino, 2018).

6.1 Lessons learned

The choice to package the application in a container made at the beginning of the training was a bit limiting and made us struggle with some problems.

The main problem was getting the PHP code to communicate inside the container with the Docker daemon standing out of the container. Docker creates isolated containers, which for us is good and that led us to choose it to ensure the safety and isolation of each site. However, wanting to put the Builder in a container was necessary to circumvent the isolation and open a way to dialogue with the demon. To do this it was necessary to understand how Docker works and how the CLI commands reach the daemon. In addition, we had to look for a tool that allows the container to interface via the CLI with the daemon without installing the whole Docker package.

Furthermore, it was not possible to package the whole project in a single container. The application itself is in fact in the main container, the Builder, but it needs other auxiliary containers for its operation. During the design we thought of combining all the necessary functionalities inside the Builder, but the performances would be drastically reduced and we would have lost the scalability. The FTP container can in fact be scaled to multiple instances as desired, as well as the proxy. MySQL is not currently scalable, as there would be synchronization problems between the various database copies, but it could still be duplicated by dividing sites into buckets and assigning one bucket to each database instance.

Another thing that we have encountered is that in reality the system needs a minimal configuration, among which it is required that Docker is obviously installed. Furthermore, the various containers must be put into operation. To solve the problem, we have created three bash scripts to facilitate the work: one of them is responsible for verifying that Docker and the other dependencies are installed and provides to make the automatic setup, the other two are dedicated to start and stop the containers needed for the infrastructure.

A different solution could be to pack everything into a Debian package, so you can put Docker as a dependency and have a daemon that automatically starts the containers at boot time. However, this option clashed with the idea of creating an open source system that was easily understandable and extensible.

6.2 Usability

Although the prototype is fully functional, usability can be considerably improved because we wanted to give a generic base that can be extended for specific cases and not the specific case directly.

For example, the domain must be registered and redirected to the IP specified by the application and can not be managed by the application itself.

In addition, the web spaces are not paid and offer unlimited CPU, space and bandwidth, which is rather unlikely for a hosting service.

6.3 Current limitations and future developments

This work was intended to provide a working open source prototype, so that developers could extend it to meet their needs. To do this it was necessary to create something very simple and basic, so as not to overload it with anything that was not necessary.

The goal has been achieved, and the prototype already provides a good starting point for developing your own application by providing same widely used tools and generalizing particular cases in a light and fast application.

However, this project could be extended to include two security aspects that can not be underestimated today: encrypted connections and data backup.

6.3.1 SSL

Although encrypted connections are not strictly necessary and connected for the purpose of the project, more and more the web is pushing towards the use of the https protocol. Recently, some of the most used browsers have started report that some websites that still use the non-secure http protocol as dangerous, especially when it comes to login pages. Even search engines prefer to give more visibility to secure sites in order to push everyone to switch to https to make navigation safer.

The https encrypted connections can be made with Let's Encrypt which provides free SSL certificates (Let's Encrypt, 2018). It’s possible to expand the Builder so that it creates one automatically after the container creation.

6.3.2 Backup

Another thing not necessary for the operation of this orchestrator but nevertheless very important is the data backup.

In fact, it may happen that users make mistakes and overwrite their work, whether they are files or strings in the database. A rescue of these could provide tools for restoring the site to a previous state.

Even more serious would be if there were any damage to the system (problems with Docker, an error in creating a site, IO problems) that could corrupt some of the data. It is always good to keep a constantly updated copy of the data to avoid problems of some kind, better if on another machine.

In our case it may be important to schedule a backup of the databases and volumes associated with each site.

6.3.3 Webserver and databases

Another important aspect to consider is the range of webserver types and database that we offer.

At the moment we only support Nginx and Apache and the only scripting language available is PHP, but the panorama is larger. For example, a less experienced user might want a ready-made installation of WordPress without having to manually install it, which many hosting providers already offer. A more advanced user may instead be interested in less common but more specific alternatives, such as Ruby or the increasingly popular Node.js (Sparks, 2018).

In the future we could extend the support to other languages, while keeping the configuration simple and mostly automatic.

As regards the database engines, at the moment we only support MySQL which is good for most sites but could be limiting for those looking for something more advanced. For example, lately there is a tendency to use NoSQL databases.

NoSQL databases are more scalable and provide superior performance. Therefore, this would avoid the division into buckets and would allow to scale better.

Currently the most used are MongoDB, Redis and Cassandra (EverSQL, 2018) (DB-Engines Ranking, 2018).

Another lack of database configuration in Pier is the little flexibility left to the user. Indeed, at the moment the user can choose only if he need a database or not, while a more advanced user may want more than one database or maybe choose the name.

For this purpose, an advanced section of the form could be reserved in which the expert user can choose the type of engine and the number and the name of the databases, while the base user can leave the suggested configuration unchanged.

7 References

8 surprising facts about real Docker adoption. (2018, January 19). Retrieved from Datadog: https://web.archive.org/web/20180119120830/https://www.datadoghq.com/docker-adoption/

About Docker. (2018, february 15). Retrieved from Docker website: https://web.archive.org/web/20180215171614/https://www.docker.com/company

Aqua Security Software Ltd. (2016, May 5). *A Brief History of Containers: From 1970s chroot to Docker 2016.* Retrieved from Aqua Sec: https://web.archive.org/web/20180215174337/https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

D'Aquino, A. (2018, February). *Pier repository.* Retrieved from GitHub: https://github.com/daquinoaldo/Pier

DB-Engines Ranking. (2018, January 21). Retrieved from DB-Engines: https://web.archive.org/web/20180121014623/https://db-engines.com/en/ranking

Docker. (2018, February 17). Retrieved from Docker: https://www.docker.com/

EverSQL. (2018, February 19). Retrieved from Most popular databases in 2017 according to StackOverflow survey: https://web.archive.org/web/20180219212049/https://www.eversql.com/most-popular-databases-in-2017-according-to-stackoverflow-survey/

Explore Docker Hub. (2018, January 05). Retrieved from Docker Hub: https://web.archive.org/web/20180105183617/https://hub.docker.com/explore/

Let's Encrypt. (2018, February 17). Retrieved from Let's Encrypt: https://letsencrypt.org/

Linux-VServer Project. (2011, October 21). *Hall of Fame.* Retrieved from Linux-VServer.org: https://web.archive.org/web/20111122024434/http://linux-vserver.org:80/Hall_of_Fame

MySQL repository. (2017, 12 12). Retrieved from Docker Hub: https://web.archive.org/web/20171212010707/https://hub.docker.com/r/_/mysql/

Sparks, C. (2018, February 20). *Which Web Programming Languages To Learn Todaty [2018].* Retrieved from Web Hosting Design Post: https://web.archive.org/web/20180220104135/https://www.webhostdesignpost.com/website/webdevelopmentcodinglanguages.html

Stilliard, A. (2018, January 26). *pure-ftpd repository.* Retrieved from Docker Hub: https://web.archive.org/web/20180126064618/https://hub.docker.com/r/stilliard/pure-ftpd/

Wilder, J. (2018, February 17). *GitHub.* Retrieved from nginx-proxy repository: https://web.archive.org/web/20180217140042/https://github.com/jwilder/nginx-proxy