Άσκηση 1

Διαδικασία εγκατάστασης – Στοιχεία σύνδεσης – Απαιτήσεις

Αφού τοποθετηθούν τα αρχεία της εφαρμογής σε κάποιον φάκελο και δημιουργηθεί η βάση δεδομένων, πρέπει να εισαχθούν τα credentials για την βάση, στο αρχείο **.env**, που βρίσκεται στο root της εφαρμογής. Θα πρέπει να έχει κάτι τέτοιο δηλαδή:

```
DB_HOST = "db"
DB_NAME = "kyranis_db"
DB_USER = "root"
DB_PASS = "root"
```

Για να δημιουργηθεί η δομή της **βάσης δεδομένων**, μαζί με αρχικά δεδομένα, πρέπει να εισαχθεί το αρχείο **sql/kyranis_db.sql** (βρίσκεται στα αρχεία της εφαρμογής), στο **phpmyadmin**. Δημιουργούμε μία νέα βάση με όνομα **kyranis_db** και μετά κάνουμε εισαγωγή το παραπάνω αρχείο.

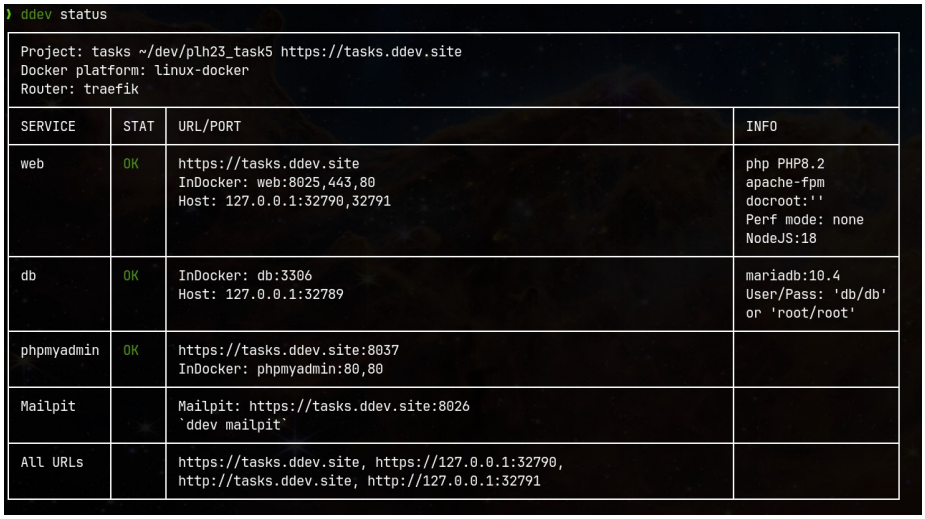
Η βάση δεδομένων έχει ήδη τους παρακάτω χρήστες, μαζί με κάποια αρχικά δοκιμαστικά δεδομένα, σε όλους τους πίνακες της βάσης.

Credentials χρηστών		
Username	**Role**	**Password**
admin	Admin	QXns62Gk
zenith	User	jlYA5O0r
quasar	User	3xT8ysAR
lumina	User	Ba9G1Eoi
eclipse	User	ByA7azZW
infinity	User	bd2vIoD3
orion	User	cA3rhBEF

Η εφαρμογή απαιτεί **PHP 8.2.** και για **βάση δεδομένων** έχει δοκιμαστεί με **MariaDB 10.4.** Επίσης έχει δοκιμαστεί με **Apache**.

Για τοπικό περιβάλλον ανάπτυξης προτιμήθηκε το εργαλείο **ddev**[1] που δημιουργεί **docker containers**, για κάθε service της εφαρμογής. Ενώ για editor προτιμήθηκε το **PhpStorm.**

[1] https://ddev.com



```
❯ ddev status

Project: tasks ~/dev/plh23_task5 https://tasks.ddev.site
Docker platform: linux-docker
Router: traefik

| SERVICE     | STAT | URL/PORT                                                                                                  | INFO                                                                                                  |
|-------------|------|-----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| web         | OK   | https://tasks.ddev.site<br>InDocker: web:8025,443,80<br>Host: 127.0.0.1:32790,32791                       | php PHP8.2<br>apache-fpm<br>docroot:''<br>Perf mode: none<br>NodeJS:18                                |
| db          | OK   | InDocker: db:3306<br>Host: 127.0.0.1:32789                                                                | mariadb:10.4<br>User/Pass: 'db/db'<br>or 'root/root'                                                  |
| phpmyadmin  | OK   | https://tasks.ddev.site:8037<br>InDocker: phpmyadmin:80,80                                                |                                                                                                       |
| Mailpit     |      | Mailpit: https://tasks.ddev.site:8026<br>`ddev mailpit`                                                   |                                                                                                       |
| All URLs    |      | https://tasks.ddev.site, https://127.0.0.1:32790,<br>http://tasks.ddev.site, http://127.0.0.1:32791       |                                                                                                       |
```

Εικόνα 1: Screenshot από την εκτέλεση του ddev

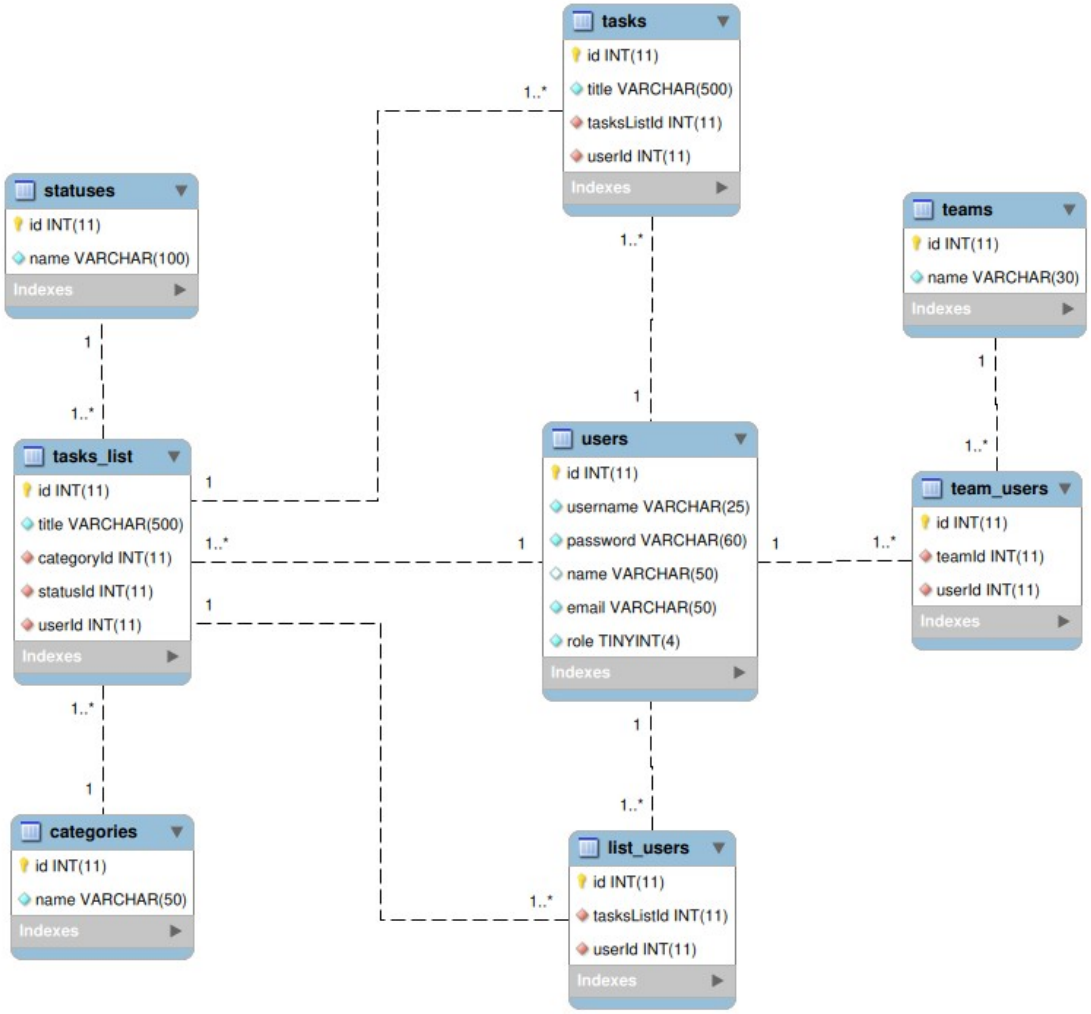
Τα σχόλια μέσα στον κώδικα της εφαρμογής, είναι στα αγγλικά, από προσωπική συνήθεια.

Video σύντομης παρουσίασης της εφαρμογής, υπάρχει στο link:

https://www.youtube.com/watch?v=6WRod4U7rDE

ER διάγραμμα της βάσης δεδομένων

Το **ER διάγραμμα** της **βάσης δεδομένων** είναι το παρακάτω και δημιουργήθηκε με το εργαλείο **Workbench**[2].



Η βάση δεδομένων έχει 4 βασικά tables (**users**, **teams**, **tasks_list**, **tasks**). Για τις **κατηγορίες** και **καταστάσεις** των λιστών εργασιών, προτιμήθηκε να είναι κι αυτά tables στην βάση, αντί να υπάρχουν σαν static arrays μέσα στην εφαρμογή.

Για τις **συσχετίσεις** (**many to many**) μεταξύ **χρηστών** και **ομάδων**, όπως και μεταξύ **χρηστών** και **λιστών**, υπάρχουν τα **pivot tables team_users** και **list_users**. Κάθε λίστα εργασιών, αλλά και εργασία, έχουν και **userId**, ώστε να γνωρίζουμε ποιος τα έχει δημιουργήσει. Έτσι μπορούν να υπάρχουν δικαιώματα ή όχι επεξεργασίας τους, μόνο από τον δημιουργό.

[2] https://www.mysql.com/products/workbench/

SQL εντολές κατασκευής της βάσης δεδομένων

Οι SQL εντολές κατασκευής της βάσης δεδομένων, είναι οι παρακάτω:

```
-- phpMyAdmin SQL Dump
-- version 5.2.0
-- https://www.phpmyadmin.net/
--
-- Host: db:3306
-- Generation Time: Mar 25, 2024 at 07:23 PM
-- Server version: 10.4.32-MariaDB-1:10.4.32+maria~ubu2004-log
-- PHP Version: 8.0.25

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";



/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `kyranis_db`
--

-- --------------------------------------------------------

--
-- Table structure for table `categories`
--

CREATE TABLE `categories` (
  `id` int(11) NOT NULL,
  `name` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `categories`
--

INSERT INTO `categories` (`id`, `name`) VALUES
(1, 'Εργασία'),
(2, 'Προσωπικά'),
(3, 'Εκπαίδευση'),
(4, 'Χόμπι'),
(5, 'Κοινωνικά');

-- --------------------------------------------------------

--
-- Table structure for table `list_users`
--

CREATE TABLE `list_users` (
  `id` int(11) NOT NULL,
  `tasksListId` int(11) NOT NULL,
  `userId` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `list_users`
--

INSERT INTO `list_users` (`id`, `tasksListId`, `userId`) VALUES
(34, 60, 42),
(35, 61, 42),
```

```
(36, 62, 42),
(37, 63, 42),
(38, 61, 44),
(39, 61, 43),
(40, 61, 48),
(41, 60, 45),
(42, 60, 47),
(43, 63, 46),
(44, 63, 44),
(45, 64, 43),
(46, 65, 44),
(47, 66, 45),
(48, 67, 46),
(49, 68, 47),
(50, 69, 48);

-- --------------------------------------------------------

--
-- Table structure for table `statuses`
--

CREATE TABLE `statuses` (
  `id` int(11) NOT NULL,
  `name` varchar(100) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `statuses`
--

INSERT INTO `statuses` (`id`, `name`) VALUES
(1, 'Νέα'),
(2, 'Σε εξέλιξη'),
(3, 'Ολοκληρωμένη');

-- --------------------------------------------------------

--
-- Table structure for table `tasks`
--

CREATE TABLE `tasks` (
  `id` int(11) NOT NULL,
  `title` varchar(500) NOT NULL,
  `tasksListId` int(11) NOT NULL,
  `userId` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `tasks`
--

INSERT INTO `tasks` (`id`, `title`, `tasksListId`, `userId`) VALUES
(117, 'Ανάλυση Δεδομένων', 63, 42),
(118, 'Σύνταξη Εκθέσεων', 63, 42),
(119, 'Αξιολόγηση Εκπαιδευτικών Προγραμμάτων', 63, 42),
(120, 'Αναφορές', 62, 42),
(121, 'Απάντηση σε σημαντικά emails', 62, 42),
(122, 'Διαχείριση Χρόνου', 62, 42),
(123, 'Ανάπτυξη νέων χαρακτηριστικών εφαρμογής', 61, 42),
(124, 'Unit testing', 61, 42),
(125, 'Code review', 61, 42),
(126, 'Documentation', 61, 42),
(127, 'Αναβάθμιση Βιβλιοθηκών', 60, 42),
(128, 'Διόρθωση Σφαλμάτων', 60, 42),
(129, 'Βελτιστοποίηση Κώδικα', 60, 42),
(130, 'Ασφάλεια', 60, 42),
(131, 'Τεκμηρίωση', 60, 42),
(132, 'Ανάλυση απαιτήσεων', 64, 43),
(133, 'Οργάνωση των δεδομένων σε πίνακες', 64, 43),
```

```
(134, 'Κανονικοποίηση', 64, 43),
(135, 'Καταγραφή διαδρομών', 65, 44),
(136, 'Ψάξιμο τοπικών μονοπατιών', 65, 44),
(137, 'Εξερεύνηση νέων περιοχών', 65, 44),
(138, 'Ανάλυση Αποτελεσμάτων Εκπαιδευτικών Προγραμμάτων', 63, 44),
(139, 'Συμμετοχή σε Καθαρισμούς Περιβάλλοντος', 66, 45),
(140, 'Διοργάνωση Εκδηλώσεων Κοινότητας', 66, 45),
(141, 'Οργάνωση Προσωπικών Αντικειμένων', 67, 46),
(142, 'Φροντίδα Κατοικιδίων', 67, 46),
(143, 'Συντήρηση Κουζίνας', 67, 46),
(144, 'Ορισμός Στόχων Ποιότητας', 68, 47),
(145, 'Ανάλυση Δεδομένων Ποιότητας', 68, 47),
(146, 'Σχεδιασμός και Εφαρμογή Βελτιώσεων', 68, 47),
(147, 'Παρακολούθηση και Αξιολόγηση', 68, 47),
(148, 'Εκπαίδευση Προσωπικών Δεξιοτήτων', 69, 48),
(149, 'Στόχοι και Σχέδια', 69, 48);

-- --------------------------------------------------------


--
-- Table structure for table `tasks_list`
--

CREATE TABLE `tasks_list` (
  `id` int(11) NOT NULL,
  `title` varchar(500) NOT NULL,
  `categoryId` int(11) NOT NULL,
  `statusId` int(11) NOT NULL,
  `userId` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `tasks_list`
--

INSERT INTO `tasks_list` (`id`, `title`, `categoryId`, `statusId`, `userId`) VALUES
(60, 'Εργασίες Συντήρησης', 1, 1, 42),
(61, 'Στόχοι Εβδομάδας', 1, 1, 42),
(62, 'Επείγουσες Εργασίες', 1, 1, 42),
(63, 'Αναφορές και Αναλύσεις', 3, 2, 42),
(64, 'Σχεδίαση Βάσης Δεδομένων', 1, 2, 43),
(65, 'Εξερεύνηση νέων διαδρομών για ποδηλασία', 4, 1, 44),
(66, 'Συμμετοχή σε Κοινωνικές Εκδηλώσεις και Πρωτοβουλίες', 5, 2, 45),
(67, 'Καθημερινές Εργασίες', 2, 1, 46),
(68, 'Βελτίωση Διαδικασιών Ποιότητας', 1, 1, 47),
(69, 'Προσωπική Ανάπτυξη', 2, 1, 48);

-- --------------------------------------------------------


--
-- Table structure for table `teams`
--

CREATE TABLE `teams` (
  `id` int(11) NOT NULL,
  `name` varchar(30) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `teams`
--

INSERT INTO `teams` (`id`, `name`) VALUES
(22, 'Ομάδα Ανάπτυξης'),
(23, 'Ομάδα Ποιότητας'),
(24, 'Ομάδα Υποστήριξης');

-- --------------------------------------------------------


--
```

```
-- Table structure for table `team_users`
--

CREATE TABLE `team_users` (
  `id` int(11) NOT NULL,
  `teamId` int(11) NOT NULL,
  `userId` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `team_users`
--

INSERT INTO `team_users` (`id`, `teamId`, `userId`) VALUES
(42, 22, 42),
(43, 22, 44),
(44, 22, 43),
(45, 23, 45),
(46, 23, 42),
(47, 23, 47),
(48, 22, 48),
(49, 24, 42),
(50, 24, 46),
(51, 24, 44);

-- --------------------------------------------------------

--
-- Table structure for table `users`
--

CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(25) NOT NULL,
  `password` varchar(60) NOT NULL,
  `name` varchar(50) DEFAULT NULL,
  `email` varchar(50) NOT NULL,
  `role` tinyint(4) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Dumping data for table `users`
--

INSERT INTO `users` (`id`, `username`, `password`, `name`, `email`, `role`) VALUES
(42,  'admin',  '$2y$10$YgY/Xxer6uQ.orEmpM8acu6cBzlzd1bBKYQyBP0.8uSqE4Deih5hq',  '',
'admin@mail.com', 0),
(43,  'zenith',  '$2y$10$RolNYv2Cxgo4HN3SPZdSCO27qw1Qp4zU6kw/wkX3QjubgUOUGYwZ.',  '',
'zenith@mail.com', 1),
(44,  'quasar',  '$2y$10$WZzOLZxsAOI98G5PaO1OLuGX/WwsceNx0DP0TBvFqVgeGIq0fF1wC',  '',
'quasar@mail.com', 1),
(45,  'lumina',  '$2y$10$MySu7faaWmixMkUNs3f.HOUNS60KlWt9vL44DO/ds.2Dgrv5.MO1W',  '',
'lumina@mail.com', 1),
(46,  'eclipse',  '$2y$10$gLnghAsGs9vuGHYqjRGt3OwnMce4APdzVGuTXzsgbU6Mss1OWl5Wy',  '',
'eclipse@mail.com', 1),
(47,  'infinity',  '$2y$10$rjKkavXpyuJQ7S4TXjBOGuZmtyk.YUXrg/yuEshVx6k56vnYsrb..',  '',
'infinity@mail.com', 1),
(48,  'orion',  '$2y$10$CPP2FcUACyqfwCtgVNRyxedzO2e8.Dz2R8Rn/5H9QOiv4bvQ7aPMm',  '',
'orion@gmail.com', 1);

--
-- Indexes for dumped tables
--

--
-- Indexes for table `categories`
--
ALTER TABLE `categories`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`);
```

```
--
-- Indexes for table `list_users`
--
ALTER TABLE `list_users`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`),
  ADD KEY `tasksListId` (`tasksListId`),
  ADD KEY `userId` (`userId`);


--
-- Indexes for table `statuses`
--
ALTER TABLE `statuses`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`);


--
-- Indexes for table `tasks`
--
ALTER TABLE `tasks`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`),
  ADD KEY `tasksListId` (`tasksListId`),
  ADD KEY `userId` (`userId`);


--
-- Indexes for table `tasks_list`
--
ALTER TABLE `tasks_list`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`),
  ADD KEY `categoryId` (`categoryId`),
  ADD KEY `statusId` (`statusId`),
  ADD KEY `userId` (`userId`);


--
-- Indexes for table `teams`
--
ALTER TABLE `teams`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`);


--
-- Indexes for table `team_users`
--
ALTER TABLE `team_users`
  ADD PRIMARY KEY (`id`),
  ADD KEY `id` (`id`),
  ADD KEY `teamId` (`teamId`),
  ADD KEY `userId` (`userId`);


--
-- Indexes for table `users`
--
ALTER TABLE `users`
  ADD PRIMARY KEY (`id`),
  ADD UNIQUE KEY `username` (`username`),
  ADD KEY `id` (`id`);


--
-- AUTO_INCREMENT for dumped tables
--


--
-- AUTO_INCREMENT for table `categories`
--
ALTER TABLE `categories`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=6;


--
-- AUTO_INCREMENT for table `list_users`
--
```

```
ALTER TABLE `list_users`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=51;


--
-- AUTO_INCREMENT for table `statuses`
--
ALTER TABLE `statuses`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=8;


--
-- AUTO_INCREMENT for table `tasks`
--
ALTER TABLE `tasks`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=150;


--
-- AUTO_INCREMENT for table `tasks_list`
--
ALTER TABLE `tasks_list`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=70;


--
-- AUTO_INCREMENT for table `teams`
--
ALTER TABLE `teams`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=25;


--
-- AUTO_INCREMENT for table `team_users`
--
ALTER TABLE `team_users`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=52;


--
-- AUTO_INCREMENT for table `users`
--
ALTER TABLE `users`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=49;


--
-- Constraints for dumped tables
--


--
-- Constraints for table `list_users`
--
ALTER TABLE `list_users`
     ADD  CONSTRAINT  `list_users_ibfk_1`  FOREIGN  KEY  (`tasksListId`)  REFERENCES
`tasks_list` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `list_users_ibfk_2` FOREIGN KEY (`userId`) REFERENCES `users` (`id`)
ON DELETE CASCADE ON UPDATE CASCADE;


--
-- Constraints for table `tasks`
--
ALTER TABLE `tasks`
   ADD CONSTRAINT `tasks_ibfk_1` FOREIGN KEY (`tasksListId`) REFERENCES `tasks_list`
(`id`) ON DELETE CASCADE ON UPDATE CASCADE,
   ADD CONSTRAINT `tasks_ibfk_2` FOREIGN KEY (`userId`) REFERENCES `users` (`id`) ON
DELETE CASCADE ON UPDATE CASCADE;


--
-- Constraints for table `tasks_list`
--
ALTER TABLE `tasks_list`
     ADD  CONSTRAINT  `tasks_list_ibfk_1`  FOREIGN  KEY  (`categoryId`)  REFERENCES
`categories` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
   ADD CONSTRAINT `tasks_list_ibfk_2` FOREIGN KEY (`statusId`) REFERENCES `statuses`
(`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `tasks_list_ibfk_3` FOREIGN KEY (`userId`) REFERENCES `users` (`id`)
ON DELETE CASCADE ON UPDATE CASCADE;
```

```
--
-- Constraints for table `team_users`
--
ALTER TABLE `team_users`
  ADD CONSTRAINT `team_users_ibfk_1` FOREIGN KEY (`teamId`) REFERENCES `teams` (`id`)
ON DELETE CASCADE ON UPDATE CASCADE,
  ADD CONSTRAINT `team_users_ibfk_2` FOREIGN KEY (`userId`) REFERENCES `users` (`id`)
ON DELETE CASCADE ON UPDATE CASCADE;
COMMIT;


/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

Έχει γίνει προσπάθεια, η εφαρμογή να έχει την λογική του **MVC pattern**[3] και είναι εμπνευσμένο από το πως περίπου λειτουργεί (επιφανειακά) το framework **Laravel**[4]. Στην ουσία, έχω φτιάξει ένα microframework, με τις πολύ βασικές λειτουργίες για **routing**, **autoloading** για τις κλάσεις (με την χρήση του **Composer**[5]) και είναι οργανωμένο με την αρχιτεκτονική του **MVC**.

Autoload

Με την χρήση του **Composer** για το **autoloading**, οι κλάσεις της εφαρμογής μπορούν να καλούνται από οποιοδήποτε σημείο χρησιμοποιώντας την εντολή **use**. Κάθε κλάση έχει δηλωμένο το **namespace** της. Με βάση αυτό γίνεται η κλήση της με την **use**. Για παράδειγμα, η κλάση **TasksListService** έχει δηλωμένο σαν **namespace**:

```
namespace apps4net\tasks\services;
```

Από οποιοδήποτε άλλο σημείο που θα χρειαστεί αυτή η κλάση, θα καλεστεί με την χρήση της **use**:

```
use apps4net\tasks\services\TasksListService;
```

Έτσι αποφεύγουμε το χάος να κάνουμε **require_once** και είναι πιο ευανάγνωστος κι εύχρηστος ο κώδικας μας.

Για να χρησιμοποιήσουμε την τεχνική αυτή, δημιουργούμε ένα αρχείο **composer.json**, στο οποίο περιγράφουμε την εφαρμογή μας και κυρίως το παρακάτω κομμάτι που είναι χρήσιμο για το **autoload**.

```
"autoload": {
   "psr-4": {
       "apps4net\\tasks\\": "app/"
   }
}
```

Εδώ περιγράφουμε ότι το **namespace apps4net\tasks**, αντιστοιχεί στον φάκελο **/app** της εφαρμογής. Όλος ο κώδικας που βρίσκεται σε αυτόν τον φάκελο, είναι στο **namespace apps4net\tasks**. Γι' αυτό και η παραπάνω κλάση βρίσκεται στο **namespace apps4net\tasks\services**. Το extra κομμάτι (**\services**) είναι φάκελος που βρίσκεται μέσα στον κεντρικό φάκελο **/app**.

Μετά την δημιουργία του **composer.json**, τρέχουμε την εντολή[6]:

```
composer dump-autoload
```

Έτσι δημιουργείται ένας φάκελος **vendor**, με κάποια βασικά αρχεία για την λειτουργία του **autoload**. Σε αυτόν τον φάκελο κατεβαίνουν και όποια **third party libraries** και **dependencies** εγκαθιστούμε στην εφαρμογή μας.

[3] https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/

[4] https://laravel.com

[5] https://www.phptutorial.net/php-oop/php-composer-autoload/

[6] Χρειάζεται να έχει εγκατασταθεί το εργαλείο composer στο pc μας (https://getcomposer.org)

Ταυτόχρονα μέσα στον φάκελο **/app** δημιουργείται και το αρχείο **bootstrap.php**, και είναι ο αρχικός κώδικας που τρέχει η εφαρμογή μας, όπου μπορούμε να προσθέσουμε και ότι άλλο θέλουμε να τρέξει με την εκκίνηση.

Routing

Με την χρήση του **routing** μπορεί να γίνεται αυτόματη ανακατεύθυνση σε **κλάσεις** και **μεθόδους**, με βάση το **url** που καλούμε. π.χ. αν έχουμε ένα **url** του τύπου https://myapp.gr/login, η εφαρμογή θα αναγνωρίσει αυτόματα ότι αυτό που θέλουμε να εκτελεστεί, είναι η μέθοδος **login()** της κλάσης **UserController**. Αντί ο χρήστης της εφαρμογής να καλεί συγκεκριμένα αρχεία .**php**, χρησιμοποιεί "**shortcuts**" που αποκρύπτουν και το πραγματικό file structure της εφαρμογής.

Όλη αυτή η λειτουργία, υλοποιείται μέσα στην κλάση **libraries/Route.php**. Αυτό που πρέπει να κάνουμε στην εφαρμογή τώρα, είναι να δηλώσουμε το κάθε **route** που θα χρησιμοποιηθεί. Είτε για συγκεκριμένες σελίδες, είτε για **API routes**. Η δήλωση αυτών, γίνεται στο αρχείο **routes/web.php**. Σε αυτό υπάρχουν δηλώσεις του τύπου:

```
Route::get('login', [UserController::class, 'login']);
```

Στην συγκεκριμένη, δηλώνεται ότι έχουμε ένα **route** (με **GET HTTP method**) με "**shortcut**" το **login**, το οποίο καλεί την μέθοδο **login()** της κλάσης **UserController**.

Στην συγκεκριμένη υλοποίηση μπορούν να χρησιμοποιηθούν μόνο **GET** και **POST**, σαν **HTTP methods**.

Αντίστοιχα, υπάρχουν και δηλώσεις για **API routes** του τύπου:

```
Route::post('api/addTask', [TasksListController::class, 'addTask']);
```

Αυτά τα **routes**, καλούνται στην εφαρμογή κυρίως μέσα από **JavaScript**.

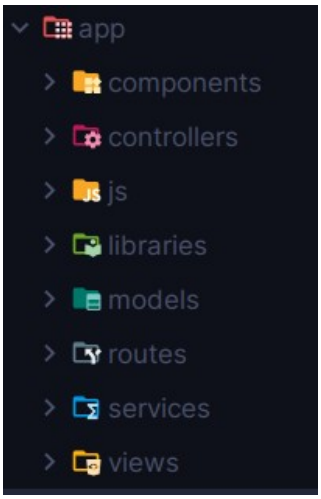
Συμπληρωματικά, χρειάζεται ένα αρχείο **.htaccess** στο **root** της εφαρμογής, που περιγράφει για τον **Apache** πως πρέπει να χειριστεί το κάθε request, ώστε να μπορέσει να λειτουργήσει ο παραπάνω κώδικας.

```
# Set routing rules for apache
RewriteEngine On

# Redirect Trailing Slashes If Not A Folder...
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_URI} (.+)/$
RewriteRule ^ %1 [L,R=301]

# Handle Front Controller...
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

File structure

Η εφαρμογή υλοποιεί το **MVC pattern**. Αυτό σημαίνει ότι έχει και σχετικό **file structure**.

Υπάρχουν οι βασικοί φάκελοι **controllers**, **views**, **models**. Υπάρχουν και οι "βοηθητικοί" **components**, **js**, **libraries**, **routes**, **services**.



Εικόνα 2: Το file structure της εφαρμογής

Αναλυτικά η περιγραφή των φακέλων:

Controllers: Οι **controllers** στο **MVC pattern**, είναι αυτοί που αναλαμβάνουν να επεξεργαστούν το **request** του χρήστη και να επιστρέψουν το τελικό αποτέλεσμα. Κάνουν τις απαραίτητες κλήσεις στην **βάση δεδομένων** και επιστρέφουν στον χρήστη, είτε ένα **view** (μια σελίδα), είτε ένα **JSON object** (αν επεξεργάζονται κάποιο **API call**). Συνήθως αποφεύγουμε οι **controllers** να έχουν πολύ **business logic**, αλλά χρησιμοποιούν άλλες κλάσεις, που θα κάνουν την περισσότερη δουλειά.

Views: Στα **views** έχουμε ουσιαστικά το **frontend** της εφαρμογής. Είναι οι ξεχωριστές σελίδες της εφαρμογής και αποτελούνται κυρίως από **HTML** κώδικα. Στα **views**, ο **controller** στέλνει συνήθως και τα **data** (αυτά που επεξεργάστηκε νωρίτερα) που θα εμφανίσει το κάθε **view**. Τα **data** αυτά μπορούν να είναι είτε κάποιες μεμονωμένες **PHP μεταβλητές**, είτε **arrays**. Έτσι μέσα στην **HTML** προσθέτουμε και κάποια κομμάτια **PHP**, ώστε να εμφανιστούν τα **data**. Καλή πρακτική εδώ, είναι να αποφεύγουμε οποιοδήποτε **business logic** και επεξεργασία δεδομένων. Να μένουμε δηλαδή κυρίως στην εμφάνιση των δεδομένων.

Models: Στα **models** περιγράφουμε την κάθε οντότητα στην **βάση δεδομένων**. Με τον τρόπο αυτό έχουμε μια αντικειμενοστραφή προσέγγιση των δεδομένων μας. Το κάθε μοντέλο μπορεί να έχει τους απαραίτητους **getters/setters**, αλλά και **συσχετίσεις** με άλλες οντότητες. Μπορεί να έχει οποιαδήποτε άλλη επεξεργασία δεδομένων της οντότητας, που μπορεί να ζητήσουμε.

Services: Όπως αναφέρθηκε παραπάνω, οι **controllers** θέλουμε να έχουν όσο γίνεται λιγότερο **business logic**. Για τον λόγο αυτό χρησιμοποιούμε βοηθητικές κλάσεις **services**. Συνήθως έχουν και το ίδιο όνομα με τον **controller** που τις χρησιμοποιεί.

π.χ. ο **UserController** μπορεί να έχει την αντίστοιχη του **UserService**. Μέσα σε μια **service** κλάση θα γίνει τελικά η κλήση στην **βάση δεδομένων**, θα γίνει η απαραίτητη επεξεργασία και θα επιστρέψει συχνά τα δεδομένα (συνήθως τα αντίστοιχα **models**) που χρειάζεται ο **controller** για να επιστρέψει στον χρήστη.

Components: Τα **views** είναι πολύ πιθανό να φτάνουν να είναι τεράστια σε πλήθος γραμμών και άρα πολύ δύσκολα στην ανάγνωση και συντήρηση. Για τον λόγο αυτό, προσπαθούμε να σπάμε τον κώδικα σε μικρά **components**. π.χ. μία φόρμα μπορεί να είναι ένα **component**, το ίδιο ένα table που εμφανίζει κάποια δεδομένα, το header, το footer κοκ. Κάθε ξεχωριστό κομμάτι της σελίδας, μπορεί να είναι ξεχωριστό **component**. Όπως και στα **views**, μπορούμε να περνάμε και το απαραίτητο set δεδομένων που χρειάζεται το **component**. Έτσι, ο φάκελος **components** περιέχει μικρότερα κομμάτια των **views**. Σε άλλες υλοποιήσεις ο φάκελος **components** μπορεί να βρίσκεται και μέσα στον φάκελο **views**, μαζί και με την **JavaScript**.

JS: Σε αυτόν τον φάκελο, υπάρχει ο κώδικας **JavaScript** που χρησιμοποιείται μέσα από τα **views** ή τα **components**. Αντίστοιχα με τα **components**, "απελευθερώνουμε" τα **views** και από τον κώδικα **JavaScript** που χρησιμοποιείται.

Libraries: Στον φάκελο αυτόν τοποθετούμε κώδικα που μπορεί να χρησιμοποιείται από όλη την εφαρμογή. Είναι **reusable** μέθοδοι που χρησιμεύουν σαν **εργαλεία**. Στην κλάση **App** για παράδειγμα, έχουμε τις μεθόδους που εμφανίζουν ένα **view**, ένα **component**, ένα **script** και άλλες βοηθητικές. Η κλάση **DB** περιέχει τις μεθόδους για **σύνδεση** και **αποσύνδεση** στην **βάση δεδομένων**. Με την **Permission**, γίνεται ο έλεγχος για τα δικαιώματα που έχει ο τρέχον χρήστης. Τέλος, με την **Route**, όπως αναφέρθηκε και παραπάνω, γίνεται η υλοποίηση του **routing**.

Routes: Σε αυτόν τον φάκελο γίνεται η δήλωση των **routes** που χρησιμοποιούνται στην εφαρμογή. Στην συγκεκριμένη περίπτωση υπάρχει μόνο το αρχείο **web.php**, αλλά θα μπορούσε να υπάρχει και ξεχωριστό **api.php** για τα **API**.

Environment variables

Για τα στοιχεία σύνδεσης στην βάση, χρησιμοποιούνται **environment variables** που δηλώνονται στο αρχείο **.env**, στο **root** της εφαρμογής. Για να υπάρχει πρόσβαση σε αυτές τις μεταβλητές χρησιμοποιείται μια **third party library**, η **PHP dotenv**[7]. Η **library** αυτή εγκαταστάθηκε με την εντολή του **composer**:

```
composer require vlucas/phpdotenv
```

Τρέχοντας αυτή την εντολή, δημιουργείται η κατάλληλη εγγραφή στο αρχείο **composer.json** και με την εγκατάσταση, κατεβαίνουν τα απαραίτητα αρχεία στον φάκελο **vendor**.

Χρησιμοποιώντας αυτή την **library**, μπορούμε σε οποιοδήποτε σημείο της εφαρμογής να γράψουμε κάτι της μορφής **$_ENV['DB_HOST']**, για να πάρουμε το **DB_HOST**, που υπάρχει στο αρχείο **.env**.

Η **library** απαιτεί να έχουμε αυτό το κομμάτι κώδικα, στο αρχείο **bootstrap.php**, ώστε να λειτουργήσει

[7] https://github.com/vlucas/phpdotenv

```
$dotenv = Dotenv\Dotenv::createImmutable(__DIR__ . '/../');
$dotenv->load();
```

Αναλυτική περιγραφή της εφαρμογής

Η εφαρμογή ξεκινάει με το αρχείο **index.php**, που βρίσκεται στον **root** φάκελο. Αυτό με την σειρά του καλεί το **bootstrap.php** και τα **routes**. Από εδώ και πέρα, αναλόγως το **url** που ζητείται, γίνεται ανακατεύθυνση στις αντίστοιχες **μεθόδους**, των **controllers**.

Η εφαρμογή έχει **5 βασικά routes** (στο αρχείο **routes/web.php**) για τις αντίστοιχες βασικές σελίδες. Αυτά είναι, το "**κενό**" (για την αρχική σελίδα), το **teams** (σελίδα ομάδων), το **tasks** (σελίδα εργασιών), το **login** (σελίδα εισόδου χρήστη) και το **register** (σελίδα εγγραφής χρήστη). Επιπρόσθετα έχει και δύο routes για τις σελίδες των **όρων χρήσης** και **πολιτικής απορρήτου**.

```
Route::get('', [MainController::class, 'index']);
Route::get('teams', [TeamsController::class, 'index']);
Route::get('login', [UserController::class, 'login']);
Route::get('register', [UserController::class, 'register']);
Route::get('tasks', [TasksListController::class, 'index']);
Route::get('termsOfUse', [MainController::class, 'termsOfUse']);
Route::get('privacyPolicy', [MainController::class, 'privacyPolicy']);
```

Όπως παρατηρούμε, κάθε **route** καλεί συγκεκριμένη μέθοδο, από κάποια κλάση **controller**.

Τα υπόλοιπα **routes** είναι κάποια βοηθητικά και κυρίως πολλά **API routes**, που πάντα έχουν και το **api/** σαν **πρόθεμα**.

Για την **αρχική σελίδα**, αν ακολουθήσουμε την μέθοδο **index()** που καλείται, φορτώνει το **view index**

```
App::view('index');
```

Η μέθοδος **view()** της βοηθητικής κλάσης **App**, θα φορτώσει ουσιαστικά το αρχείο **/views/index.php**. Αυτό το αρχείο περιέχει κυρίως κώδικα **HTML** (ο οποίος είναι περίπου αυτός που γράφτηκε για την **ΓΕ2**). Αυτό που έχει γίνει όμως τώρα, είναι ότι πολλά στοιχεία της σελίδας, έχουν "σπάσει" σε **components**. Κάθε **component** φορτώνεται με αντίστοιχη μέθοδο, παρόμοια με την **view()**, της κλάσης **App**. Για παράδειγμα το **header** φορτώνεται με την εντολή:

```
<?php App::component('header'); ?>
```

Η μέθοδος **component()** φορτώνει το σχετικό αρχείο **header.php**, που βρίσκεται στον φάκελο **components**.

Παρακάτω στον κώδικα, βλέπουμε ότι μπορούν να υπάρχουν και εντολές ελέγχου σε **PHP** για να φορτωθεί ή όχι ένα **component**.

```
if (!isset($_SESSION['username'])) {
    App::component('loginButtons');
}
```

Αντίστοιχα καλούνται και οι υπόλοιπες οθόνες της εφαρμογής. Ενώ, σε περίπτωση που ο χρήστης δεν έχει δικαιώματα να δει μια οθόνη, εμφανίζεται η σελίδα **404.php**.

Ο έλεγχος αυτός γίνεται στον **constuctor** της κλάσης **Controller** (την κληρονομούν όλοι οι **controllers**).

```
if (!Permission::getPermissionFor(App::getCurrentPage())) {
   App::view('404');
   exit();
}
```

Η συγκεκριμένη μέθοδος **getPermissionFor()** ελέγχει για τα δικαιώματα στην τρέχουσα σελίδα (δηλαδή στο συγκεκριμένο **route**), π.χ. **teams**. Για να ελεγχθούν τα δικαιώματα, χρησιμοποιείται ένα **array**, στο οποίο για κάθε **route** περιγράφεται ποιοι έχουν δικαιώματα.

```
private static array $permissions = [
        'index' => [
            'all'
        ],
        'login' => [
            'all'
        ],
        'register' => [
            'all'
        ],
        'tasks' => [
            'login', 1
        ],
        'teams' => [
            'login', 0
        ],
    ];
```

Για παράδειγμα, το **index** περιέχει το string "**all**", το οποίο σημαίνει ότι όλοι έχουν πρόσβαση σε αυτό. Το ίδιο συμβαίνει και για τα **routes login** και **register**. Τα **routes tasks** και **teams** όμως, έχουν το string "**login**". Αυτό σημαίνει ότι θα πρέπει να έχει γίνει **logged in**. Επιπρόσθετα έχουν τον ρόλο που απαιτείται σαν ελάχιστος. **1 για απλό χρήστη** και **0 για διαχειριστή**. Άρα για τα **tasks** μπορεί να έχει πρόσβαση, είτε κάποιος με ρόλο 1, είτε 0. Ενώ για τα **teams** πρέπει να έχει οπωσδήποτε ρόλο 0 (δηλαδή διαχειριστής). Με τα if statements της **getPermissionFor()**, ελέγχονται όλες αυτές οι περιπτώσεις.

Διαδικασία εγγραφής χρήστη και εισόδου

Η σελίδα εγγραφής χρήστη, είναι η **/register**. Σε αυτήν, εμφανίζεται μια φόρμα (**component registerForm**), με όλα τα απαιτούμενα στοιχεία. Επειδή πρέπει να γίνει και **validation**, σε κάθε **input element** υπάρχουν και οι σχετικοί κανόνες για το **validation**. Για παράδειγμα στο **input** του **password:**

```
<div class="mb-3">
   <label for="password" class="form-label">Κωδικός Πρόσβασης</label>
   <input type="password" class="form-control" name="password" id="password" required
       pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
       title="Ο κωδικός πρέπει να αποτελείται από τουλάχιστον 8 χαρακτήρες, εκ
       των οποίων τουλάχιστον ένας αριθμητικός και ένα κεφαλαίο γράμμα">
       <span id="passwordError" class="text-danger small d-none"></span>
</div>
```

Χρησιμοποιείται το **pattern** (είναι σε **regex** μορφή), για να περιγραφεί ότι απαιτείται **password** τουλάχιστον **8 χαρακτήρων**, με ένα **αριθμητικό στοιχείο** και ένα **κεφαλαίο**. Επίσης απαιτείται (**required**) να συμπληρωθεί.

Κάτω από το **input element** υπάρχει κι ένα **span element**, το οποίο προορίζεται για να εμφανίζονται τα **validation errors**.

Για την συγκεκριμένη φόρμα υπάρχει και ο αντίστοιχος κώδικας σε **JavaScript**. Καλείται από το αρχείο **register.php**, με την σχετική μέθοδο **script()** που ενσωματώνει το αντίστοιχο αρχείο **register.js**, από τον φάκελο **js**

```
App::script('register');
```

Στο **register.js**, υπάρχει ένα **event listener** για το **submit**. Οπότε, μόλις πατηθεί το κουμπί "**Εγγραφή**" τρέχει ο σχετικός κώδικας **JavaScript**. Συγκεκριμένα, καλείται η function **validateForm()**, στην οποία ελέγχεται κάθε **input element**, με την μέθοδο **checkValidity()**[8] της JavaScript. Η μέθοδος αυτή, αναλόγως τι κανόνες έχουν δηλωθεί στο **input element**, επιστρέφει **true/false**. Αν για κάποιο **input** επιστρέφεται **false**, εμφανίζεται σχετικό μήνυμα λάθους, κάτω από το **input** (το αντίστοιχο **span**).

```
if (!password.checkValidity()) {
    const passwordError = document.getElementById('passwordError');
    passwordError.classList.remove('d-none');
    passwordError.innerText = 'Παρακαλώ εισάγετε έναν έγκυρο κωδικό';

    valid = false;
}
```

Από την στιγμή που υπάρχει έστω κι ένα **false**, η **validateForm()**, θα επιστρέψει **false** και άρα η εγγραφή του χρήστη δεν μπορεί να προχωρήσει, εμφανίζοντας κι ένα γενικό **error**.

Ιδιαίτερη περίπτωση είναι αυτή του **username**, που γίνεται έλεγχος ύπαρξης του ήδη στην **βάση δεδομένων**. Στην περίπτωση αυτή χρησιμοποιείται η function **checkForUsernameExistence()** για να γίνει ο έλεγχος.

```
const exists = await checkForUsernameExistence(username.value)
if (exists) {
   const usernameError = document.getElementById('usernameError');
   usernameError.classList.remove('d-none');
   usernameError.innerText = 'Το όνομα χρήστη υπάρχει ήδη';

   valid = false;
}
```

Η function **checkForUsernameExistence()**, καλεί ένα σχετικό **API** (**api/checkUsername**), χρησιμοποιώντας το **fetch API**[9].

```
return fetch('api/checkUsername?username=' + username, {
        method: 'GET'
    })
        .then(response => {
            if (!response.ok) {
                return response.json().then(err => {
                    throw new Error(err.message);
                });
            }

            return response.json();
        })
        .then(data => data.exists)
        .catch(error => {
            displayMessage(error, 'error');
        });
```

[8] https://www.w3schools.com/js/js_validation_api.asp

[9] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Αναλόγως τι θα επιστραφεί στο **data.exists** (**true/false**), αποτυγχάνει ή όχι και ο έλεγχος.

Το συγκεκριμένο **API**, εκτελεί την μέθοδο **checkUsername()** του **UserController**. Με την σειρά της εκτελεί την αντίστοιχη μέθοδο της κλάσης **UserService**, όπου και εκτελείται τελικά ένα **SQL query** για να βρεθεί το συγκεκριμένο **username** στην **βάση δεδομένων**.

```
$sql = "SELECT * FROM users WHERE username = :username";
```

Αφού τελικά ολοκληρωθεί επιτυχημένα το **validation**, η φόρμα γίνεται **submit**, σύμφωνα και με το πως έχει δηλωθεί.

```
<form id="registerUserForm" action="registerUser" method="POST">
```

Το **action** δηλαδή που ζητείται είναι το **route registerUser**, που έχει δηλωθεί έτσι:

```
Route::post('registerUser', [UserController::class, 'registerUser']);
```

Η συγκεκριμένη μέθοδος, αφού πάρει τα δεδομένα της φόρμας με την χρήση της **$_POST**, εκτελεί την **registerUser()** της κλάσης **UserService**, για να εισάγει την εγγραφή στην βάση δεδομένων.

Το σημαντικό εδώ, είναι η μετατροπή του **password** σε κρυπτογραφημένο **hash**. Είναι γενική καλή πρακτική να μην σώζονται **passwords** στην βάση, σαν απλό κείμενο

```
$hashedPassword = password_hash($password, PASSWORD_DEFAULT);
```

Εδώ χρησιμοποιείται η πιο απλή μέθοδος, αλλά σε πραγματικές εφαρμογές χρησιμοποιείται πιο πολύπλοκη μέθοδος κρυπτογράφησης.

Στην διαδικασία εισόδου, έχουμε πάλι μια σχετική φόρμα (**loginForm.php**). Αυτή την φορά δεν χρειάζεται να γίνει κάποιο **validation** και στο **submit** εκτελείται το route **checkLogin**. Η διαδικασία καταλήγει στην μέθοδο **loginUser** της **UserService**, όπου αφού πρώτα βρεθεί (ή όχι) ο χρήστης με το συγκεκριμένο **username**, γίνεται σύγκριση του **password** με το **hashed password**, με την εντολή **password_verify()**.

```
if (password_verify($password, $user->getPassword())) {
     // Set the session variables for the logged in user
     $_SESSION['username'] = $username;
     $_SESSION['role'] = $user->getRole();
     $_SESSION['userId'] = $user->getId();

     return true;
}
```

Αν το **password** είναι σωστό, τότε χρησιμοποιείται η **$_SESSION**, για να αποθηκευτούν τα **username, role, userId**. Αυτά μπορούν να χρησιμοποιηθούν από κάθε σημείο πλέον της εφαρμογής, για να ελεγχθεί αν υπάρχει χρήστης **logged in** και ποιος είναι αυτός. Αυτό γίνεται στην μέθοδο **getPermissionFor()** που περιγράφηκε πιο πάνω. Αν το login είναι σωστό, ο χρήστης ανακατευθύνεται στην αρχική οθόνη, αλλά πλέον με δικαιώματα να δει κι άλλες οθόνες.

Αν το **login** αποτύχει, εμφανίζεται πάλι η οθόνη του login, για να ξαναδώσει ο χρήστης στοιχεία (δεν εμφανίζει μήνυμα αποτυχίας)

Εμφάνιση λαθών

Σε όλα τα σημεία της εφαρμογής που υπάρχει δραστηριότητα από τον χρήστη, μπορούν να συμβούν λάθη, αλλά και χρειάζεται να εμφανιστούν διάφορα μηνύματα επιτυχίας. Για τον λόγο αυτό υπάρχει ένα **component**, το **error.php**. Αυτό τοποθετείται σε κάθε **view**, στο τέλος περίπου της σελίδας, που χρειάζεται. π.χ. στο **tasks**, γίνεται η ενσωμάτωση, στο τέλος του **section element**.

```
<?php App::component('error'); ?>
```

Μέσα στο **element**, εκτός του **HTML** μέρους, γίνεται και η ενσωμάτωση της **JavaScript**, που αφορά το **component**. Στην **JavaScript** περιέχει την function **displayMessage()**, η οποία μπορεί να κληθεί από οποιοδήποτε άλλο **JavaScript** κομμάτι της εφαρμογής. Η function, αναλόγως την παράμετρο που του δίνουμε (**success/error**), εμφανίζει το **error component** και αλλάζει το κείμενο του με βάση αυτό που του δίνουμε. Για τα **errors** εμφανίζει το μήνυμα σε **κόκκινο φόντο**, ενώ αν είναι **success** έχει **πράσινο φόντο**. Στην περίπτωση του **success,** το μήνυμα γίνεται και **hide** αυτόματα, μετά από **5 δευτερόλεπτα**. Στο **error** πρέπει ο χρήστης να το κλείσει αυτός, **πατώντας στο x**. Για παράδειγμα, μέσα από την **JavaScript** στο **tasks.js**, έχουμε κλήσεις της **displayMessage()** με αυτούς τους τρόπους:

```
displayMessage("Η λίστα δημιουργήθηκε επιτυχώς", 'success');
```

```
displayMessage(error, 'error');
```

SQL queries στην βάση δεδομένων

Για τα **SQL queries** χρησιμοποιείται το **PHP Data Objects (PDO) extension**[10]. Τα **queries** γίνονται συνήθως μέσα από τις **services** κλάσεις, που έχουν όλο το σχετικό **business logic**. Κάποια **queries** γίνονται και μέσα από τα **models**, όταν υπάρχει **one to many relationship**. π.χ. στο **model Team**, υπάρχει η μέθοδος **getUsers()** που επιστρέφει όλους τους χρήστες της ομάδας.

Μία μέθοδος στην εφαρμογή, που κάνει κάποιο **query**, έχει την παρακάτω μορφή (π.χ. για να πάρουμε τον χρήστη που έχει συγκεκριμένο **userId**)

```
public function getUserById(int $userId): User
    {
        DB::connect();

        $sql = "SELECT * FROM users WHERE id = :userId";

        try {
            $stmt = DB::$conn->prepare($sql);
            $stmt->bindParam(':userId', $userId);
            $stmt->execute();

            $stmt->setFetchMode(\PDO::FETCH_CLASS, '\apps4net\tasks\models\User');

            $user = $stmt->fetch();
        } catch (\PDOException $e) {
            throw new \Exception($e->getMessage());
        }

        return $user;
    }
```

Καταρχήν χρησιμοποιείται η **DB::connect()** για να γίνει η αρχική σύνδεση στη **βάση δεδομένων**. Ύστερα υπάρχει ένα **prepare statement** για το **SQL query** που θέλουμε να κάνουμε. Στη συνέχεια περνιούνται οι όποιες παράμετροι με την **bindParam()**. Τέλος, γίνεται η εκτέλεση.

Επειδή χρησιμοποιείται **αντικειμενοστραφής προσέγγιση**, το **fetch** των δεδομένων γίνεται σε **αντικείμενο** και **όχι σε array**. Για παράδειγμα, με την χρήση της **setFetchMode**, δηλώνεται ότι θα επιστραφεί αντικείμενο της κλάσης **User**. Η κλάση **User** έχει δηλωμένα όλα τα **properties**, με το ίδιο όνομα που είναι δηλωμένα και στην βάση δεδομένων. Επίσης έχει όλους τους απαραίτητους getters/setters.

```
$stmt->setFetchMode(\PDO::FETCH_CLASS, '\apps4net\tasks\models\User');

$user = $stmt->fetch();
```

Έτσι, τελικά η μέθοδος **getUserById()** επιστρέφει ένα **User** αντικείμενο. Όπου γίνεται η χρήση αυτού, μπορούμε να πάρουμε όποιο property θέλουμε, με την χρήση των **getters**. π.χ.

```
$user->getUsername()
```

Σε όλα τα **SQL queries** (όπως και σε άλλα σημεία της εφαρμογής), γίνεται η χρήση της **try/catch**, ώστε να επιστρέφονται **Exceptions**, για όλα τα πιθανά λάθη που μπορεί να προκύψουν.

[10] https://www.php.net/manual/en/intro.pdo.php

Τα αποτελέσματα από κάθε **SQL query**, επιστρέφουν τελικά στον σχετικό **controller** κι αυτός με την σειρά του, τα περνάει στο σχετικό **view** για την χρήση τους.

Η οθόνη Λίστας Εργασιών

Στην οθόνη αυτή, υπάρχει μία βασική φόρμα, **δημιουργίας λίστας εργασιών**. Με το **submit**, καλείται το σχετικό **API** (**api/createTasksList**) και γίνεται η δημιουργία της λίστας. Το σημαντικό εδώ είναι ότι έχουμε ένα **AJAX call**[11]. Δηλαδή έχουμε ένα **POST request** που **δεν θα κάνει reload** την σελίδα. Πρέπει η σελίδα να ενημερωθεί με την νέα λίστα που προστέθηκε. Για τον σκοπό αυτό, από τον σχετικό **controller** (**TasksListController**) δεν επιστρέφεται μόνο το αντικείμενο της κλάσης **TasksList**, αλλά και το **HTML component** (ένα **string** ουσιαστικά με την σχετική **HTML**), το οποίο μπορούμε να εμφανίσουμε με **JavaScript**.

```
$HTMLComponent = App::componentHTML('tasksList', [
                'list' => $tasksList,
                'categories' => $categories,
                'statuses' => $statuses
            ]);
```

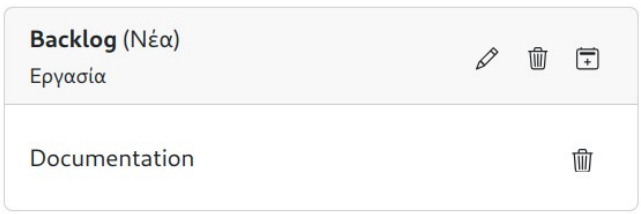
Χρησιμοποιούμε την μέθοδο **componentHTML()** που λειτουργεί όπως η απλή **component()**, αλλά αυτή την φορά αντί να ενσωματώσει τον κώδικα του **component** στην σελίδα μας, επιστρέφει ένα **string** με την **HTML**. Αυτή **HTML** θα εισαχθεί στην σελίδα μέσω **JavaScript**, με κώδικα σαν τον παρακάτω:

```
const tasksList = document.querySelector('.tasksListsContainer');
tasksList.insertAdjacentHTML('afterbegin', data.HTMLComponent);
```

Συγκεκριμένα, αφού πάρει το γενικό **container element** που βρίσκονται μέσα όλες οι λίστες, θα προσθέσει το **HTMLComponent** (με χρήση της **insertAdjacentHTML**), στην αρχή (**afterbegin**) του **container**. Δηλαδή θα το εμφανίσει στην πρώτη θέση.

Με παρόμοιο τρόπο λειτουργεί και σε όλα τα άλλα σημεία της εφαρμογής που γίνονται **AJAX calls**.

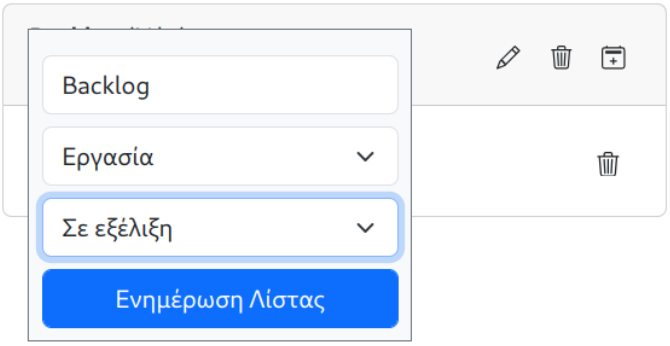
Κάθε λίστα που προσθέτουμε εμφανίζεται σαν ένα **card**, το οποίο έχει το **όνομα** της λίστας, στην παρένθεση την **κατάσταση** και από κάτω την **κατηγορία**. Στο δεξί μέρος έχει κουμπιά για **επεξεργασία** της λίστας, **διαγραφή** της και **προσθήκη εργασίας**. Στο κυρίως μέρος του **card** έχει την **λίστα των εργασιών**, με δυνατότητα **διαγραφής** της κάθε μίας.



Εικόνα 3: Κάρτα λίστας εργασιών

Πατώντας για **επεξεργασία**, ανοίγει **popup** παραθυράκι για επιλογή επεξεργασίας των στοιχείων της λίστας (**όνομα**, **κατηγορία**, **κατάσταση**).

[11] https://www.freecodecamp.org/news/ajax-tutorial/

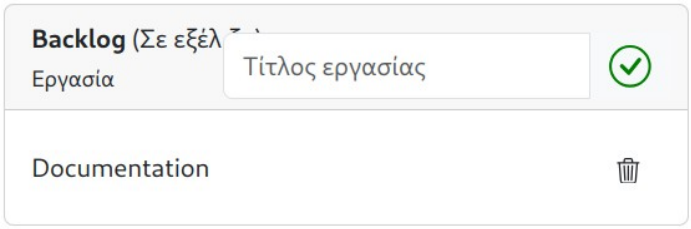


Εικόνα 4: Επεξεργασία λίστας εργασιών

Με την **διαγραφή** εμφανίζει **μήνυμα επιβεβαίωσης** και μετά γίνεται η **διαγραφή**.

Και στις δύο περιπτώσεις (**επεξεργασία, διαγραφή**), ενημερώνεται η σελίδα με την αλλαγή, χωρίς να γίνει reload.

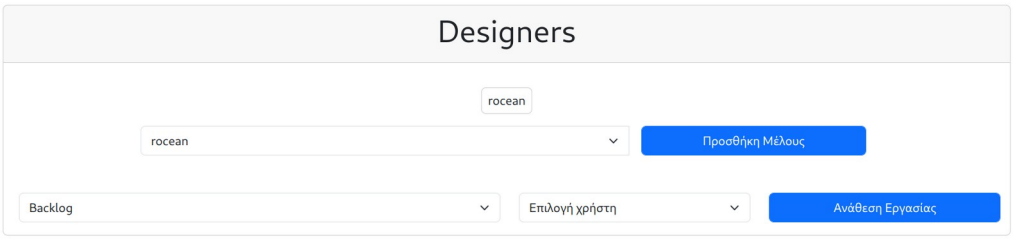
Πατώντας για **προσθήκη εργασίας**, εμφανίζει σχετικό **input text box** στο ίδιο σημείο, για εισαγωγή του **κειμένου της εργασίας**. Πατώντας **enter** (ή στο σχετικό **εικονίδιο** δίπλα) εμφανίζεται άμεσα η εργασία στη λίστα.



Εικόνα 5: Εισαγωγή νέας εργασίας στη λίστα

Η οθόνη Ομάδων

Και στην οθόνη αυτή υπάρχει μια βασική **φόρμα δημιουργίας ομάδας**. Δημιουργώντας μία, αυτή εμφανίζεται άμεσα χωρίς refresh.



Εικόνα 6: Κάρτα ομάδας

Η κάθε ομάδα είναι μια **card** που αποτελείται από την **λίστα μελών** της, **φόρμα προσθήκης μέλους στην ομάδα** και **φόρμα ανάθεση λίστας εργασιών σε χρήστη**. Για την **προσθήκη μέλους**, υπάρχει ένα **select**, στο οποίο υπάρχουν όλοι οι χρήστες της βάσης. Ο διαχειριστής επιλέγει χρήστη και πατάει "**Προσθήκη Μέλους**". Με την προσθήκη ενημερώνεται άμεσα η **λίστα των μελών**, αλλά και το **σχετικό select** για την **ανάθεση λίστας εργασιών σε χρήστη**. Αν ο χρήστης είναι μέλος ήδη στην ομάδα, θα εμφανιστεί σχετικό μήνυμα λάθους.

Στην **ανάθεση λίστας εργασιών**, υπάρχει ένα **select** με τις **λίστες εργασιών** που έχει δημιουργήσει ο **διαχειριστής** (δεν μπορεί να δει λίστες που έχουν δημιουργήσει άλλοι χρήστες). Στο δεύτερο **select** υπάρχουν **μόνο οι χρήστες που είναι μέλη της συγκεκριμένης ομάδας** και όχι όλοι. Όπως αναφέρθηκε και πιο πάνω, το **select** αυτό ενημερώνεται άμεσα με την προσθήκη νέου μέλους στην ομάδα.

Αν σε κάποιο μέλος έχει ανατεθεί ήδη μια λίστα εργασιών, θα εμφανίσει σχετικό λάθος. Εδώ θα μπορούσε να υπάρχει και μια πρόσθετη λειτουργία, που θα πατούσες, για παράδειγμα, πάνω σε συγκεκριμένο χρήστη και θα εμφάνιζε τις λίστες εργασιών που του έχουν ανατεθεί, ώστε να έχει άποψη ο διαχειριστής με τις αναθέσεις που έχει κάνει.

Ο JavaScript κώδικας

Ο **JavaScript** κώδικας βρίσκεται κυρίως σε 2 αρχεία (**tasks.js**, **teams.js**) και αφορά ενέργειες που γίνονται στα αντίστοιχα **views**. Σε αυτά, είτε προσθέτουμε **event listeners** σε κουμπιά:

```
createListForm.addEventListener('submit', function (e) {
    …
})
```

Είτε έχουμε functions που καλούνται από κουμπιά με **onclick events**:

```
<a title="Επεξεργασία Λίστας" onclick="editTasksList(<?= $listId ?>)">
```

Τα **API calls** γίνονται με την χρήση του **fetch API** και σε γενικές γραμμές επαναλαμβάνεται κώδικας παρόμοιος με τον παρακάτω:

```
fetch(this.action, {
    method: 'POST',
    body: new FormData(this)
})
    .then(response => {
        // Get the response and check if it's ok
        if (!response.ok) {
            return response.json().then(err => {
                throw new Error(err.message);
            });
        }

        // Return the success response
        return response.json();
    })
    .then(data => {
         // Do this on success

         // Clean form data
         this.reset();

        // Display new tasks list component with the new data, at the top of the page
        const tasksList = document.querySelector('.tasksListsContainer');
        tasksList.insertAdjacentHTML('afterbegin', data.HTMLComponent);

        displayMessage("Η λίστα δημιουργήθηκε επιτυχώς", 'success');
    })
    .catch(error => {
        // Do this on error
        displayMessage(error, 'error');
     });
```

Στο **fetch** περνάμε το **action**, που είτε το παίρνει από την φόρμα με **this.action**, είτε το τοποθετούμε με το "χέρι" (π.χ. **api/deleteTask**). Επίσης δηλώνουμε την **μέθοδο** (**POST, GET**) και περνάμε τα **data** στο **body**. Παίρνουμε τα δεδομένα της φόρμας απευθείας, με **new FormData(this)**.

Στην συνέχεια, στο πρώτο **.then** γίνεται ο αρχικός έλεγχος αν το **response** έχει έρθει σωστά και είτε επιστρέφουμε **error** (**new FormData(this)**), είτε επιστρέφουμε το **JSON object** που επιστρέφει το **API** (**return response.json()**). Στο επόμενο **.then** παίρνει το **response.json()** και κάνει οποιαδήποτε επεξεργασία των δεδομένων χρειάζεται. Αλλιώς στο **catch** πιάνει το **error** που κάναμε **throw** προηγουμένως και κάνει τις απαραίτητες ενέργειες (συνήθως να εμφανίσει το μήνυμα λάθους).

Η λειτουργία των API

Τα διάφορα **API** της εφαρμογής, αφού κάνουν την όποια επεξεργασία, επιστρέφουν ένα **JSON object,** με τα όποια data θέλουμε. Είτε σε περίπτωση επιτυχίας, είτε αποτυχίας. Για παράδειγμα, στον παρακάτω κώδικα, στο τέλος της μεθόδου **createTeam()** του **TeamsController**.

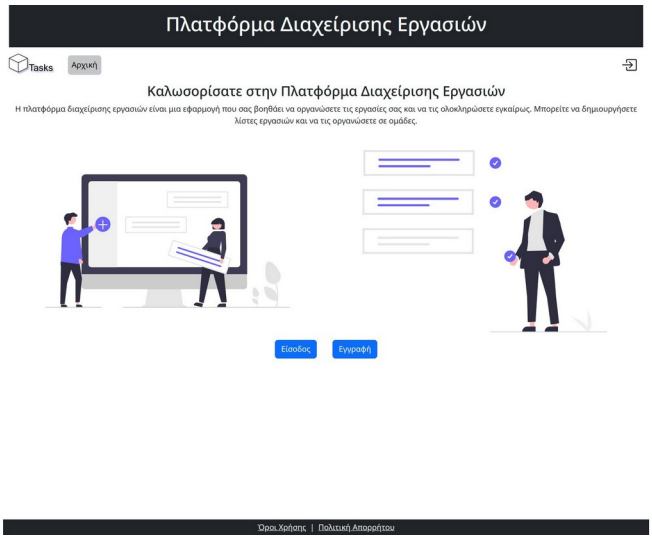
```
try {
    $team = $this->teamsService->createTeam($name);

    // Get the HTML of the tasks list component, to add it to the page, without
refreshing
     $HTMLComponent = App::componentHTML('team', [
          'team' => $team,
          'users' => $users,
          'tasksLists' => $tasksLists
     ]);

     // Return success json response
     $this->returnSuccess(['team' => $team, 'HTMLComponent' => $HTMLComponent]);
} catch (\Exception $e) {
    // Return error message
    $this->returnError(400, $e->getMessage());
}
```

Παρατηρούμε ότι χρησιμοποιούνται 2 μέθοδοι. Οι **ReturnSuccess()** και **returnError()**. Αυτές οι μέθοδοι κληρονομούνται από την κλάση **Controller**. Η **returnSuccess()** επιστρέφει μαζί με τα σχετικά data και το **HTTP code 200**. Που είναι το προκαθορισμένο για την επιτυχή εκτέλεση. Στην **returnError(),** εκτός από το μήνυμα, περνάμε και το **HTTP code** που θέλουμε να επιστραφεί. Τα **error HTTP codes** μπορούν να είναι αρκετά πιο αναλυτικά (αναλόγως την περίπτωση)[12], αλλά στην εφαρμογή χρησιμοποιείται μόνο το **HTTP code 400**. Αναλόγως το **HTTP code**, μπορεί μετά η **εφαρμογή – πελάτης** που χρησιμοποιεί το **API**, να χειριστεί αναλόγως τα λάθη.

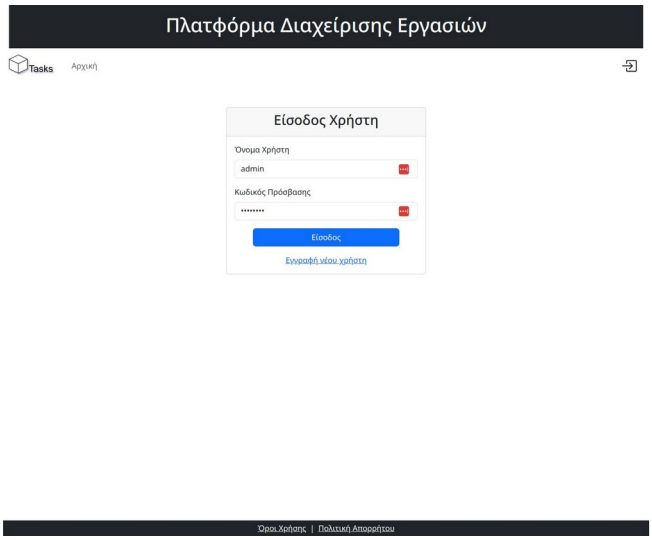
[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Screenshots της εφαρμογής

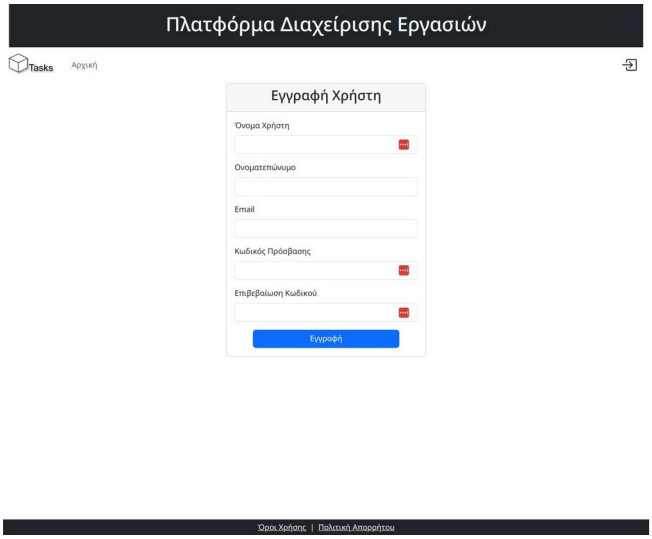
Παρακάτω είναι κάποια **screenshots** παρουσίασης της εφαρμογής



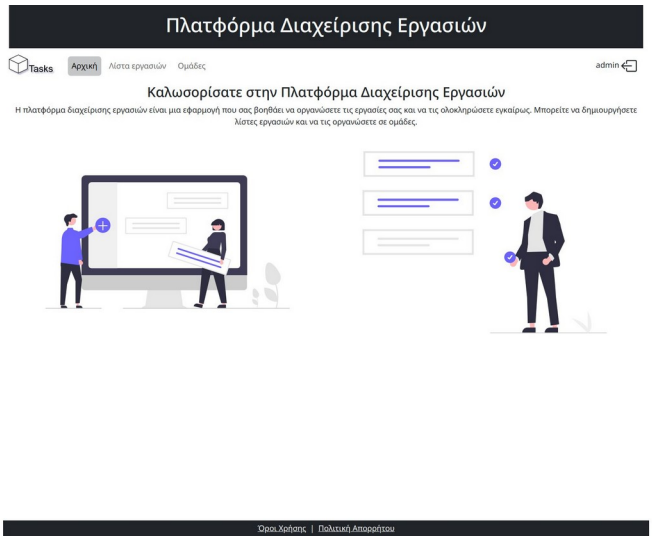
Στην **αρχική οθόνη**, υπάρχει ένα μικρό καλωσόρισμα και κουμπιά εισόδου ενός χρήστη ή δημιουργίας νέου. Είσοδος στην εφαρμογή μπορεί να γίνει και με το εικονίδιο πάνω δεξιά. Στο μενού δεν εμφανίζονται άλλες επιλογές, αφού δεν έχει κάνει login ακόμη ο χρήστης.



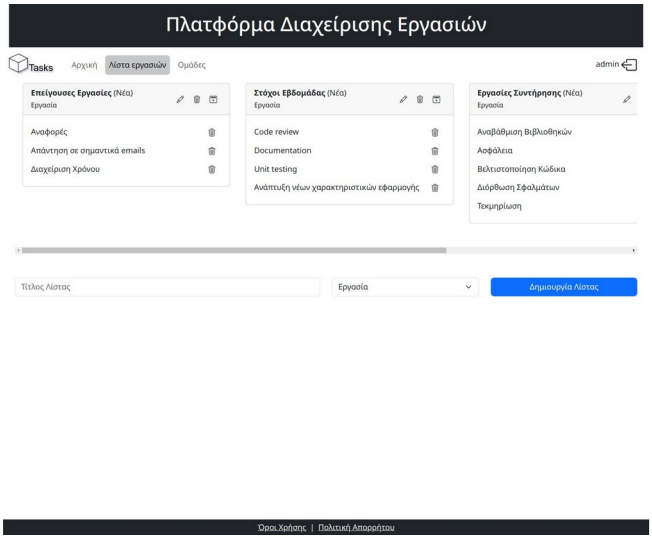
Στην **οθόνη εισόδου**, ο χρήστης γράφει το **username** του και το **password**. Από κάτω υπάρχει link για **δημιουργία νέου χρήστη**.



Στην **οθόνη εγγραφής**, ο χρήστης συμπληρώνει όλα τα στοιχεία του. Όλα είναι **απαραίτητα**, εκτός από το **ονοματεπώνυμο**. Η εφαρμογή ελέγχει ότι το **username** δεν υπάρχει ήδη στην **βάση δεδομένων**, το **email** έχει το σωστό format και η **επιβεβαίωση του password** είναι σωστή. Αν υπάρχει πρόβλημα εμφανίζεται σχετικό λάθος.

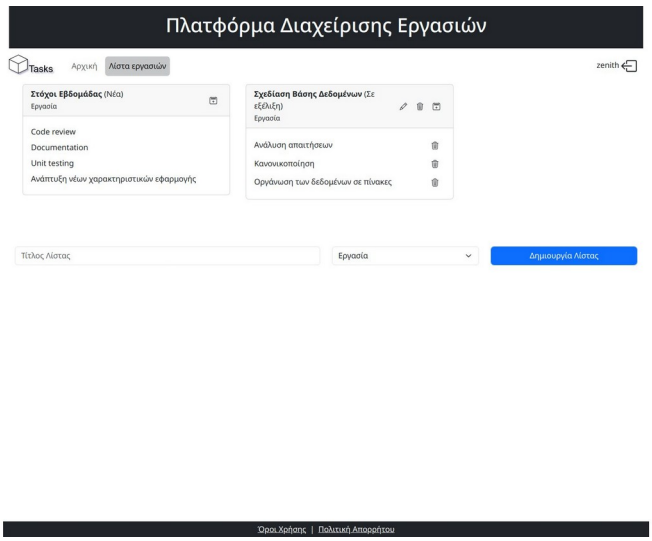


Αφού κάνει **login** ο χρήστης, η **αρχική οθόνη** διαμορφώνεται μερικώς. Πλέον υπάρχουν κι άλλες επιλογές στο **μενού**, ενώ εμφανίζεται και το **όνομα του χρήστη** πάνω δεξιά. Δίπλα υπάρχει και εικονίδιο για το **logout**. Τα κουμπιά για είσοδο και εγγραφή, δεν χρειάζεται να εμφανίζονται πλέον.



Στην **οθόνη λίστας εργασιών**, ο χρήστης βλέπει την κάθε λίστα, σαν card. Κάθε λίστα έχει τα στοιχεία της, καθώς και τις εργασίες που έχουν προστεθεί. Υπάρχουν εικονίδια για **επεξεργασία** των στοιχείων της λίστας, **διαγραφή** της, αλλά και **προσθήκης εργασίας**. Οι λίστες εμφανίζονται με οριζόντιο scrolling και είναι ταξινομημένες με βάση την κατάστασή της ("**Νέα**", "**Σε εξέλιξη**", "**Ολοκληρωμένη**")

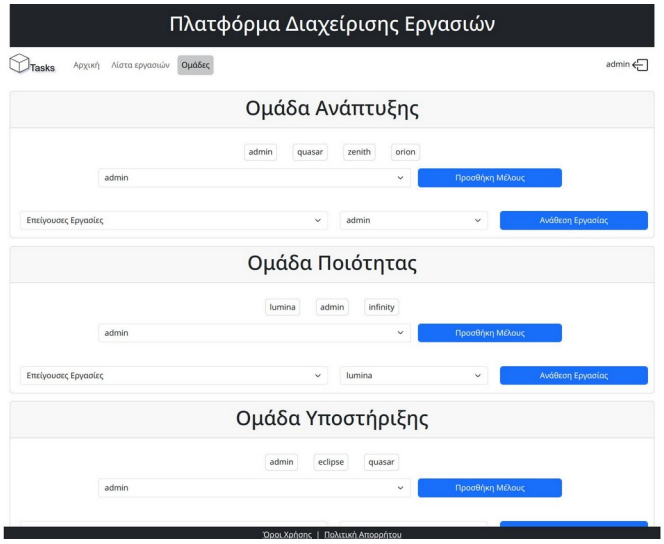
Από κάτω υπάρχει φόρμα **δημιουργίας νέας λίστας εργασιών**.



Εδώ βλέπουμε την ίδια οθόνη, αλλά έχει κάνει **login** ένας **απλός χρήστης**. Παρατηρούμε ότι στην λίστα "**Στόχοι Εβδομάδας**", δεν έχει δικαίωμα επεξεργασίας και διαγραφής, παρά μόνο εισαγωγής εργασίας. Αυτό γίνεται, γιατί δεν έχει δημιουργήσει ο ίδιος της λίστα αυτή, αλλά ο **διαχειριστής**. Επίσης δεν εμφανίζεται εικονίδιο διαγραφής εργασίας, για εργασίες που δεν έχει δημιουργήσει αυτός.

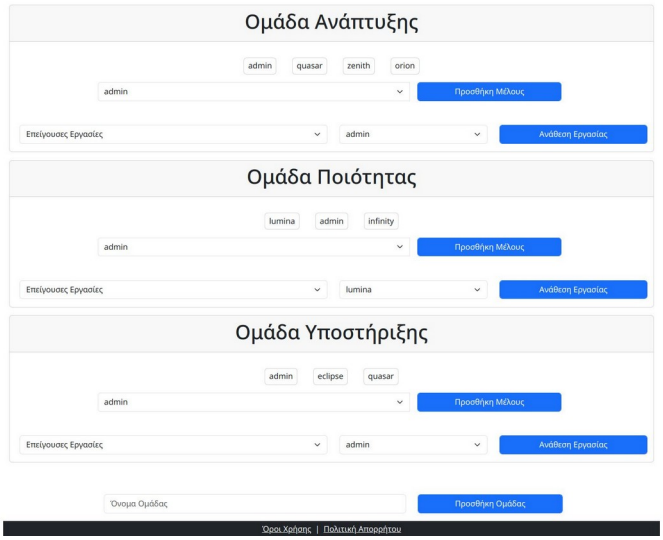
Αντίθετα, στην λίστα "**Σχεδίαση Βάσης Δεδομένων**", παρατηρούμε ότι έχει δικαιώματα επεξεργασίας και διαγραφής. Ενώ μπορεί να διαγράψει και εργασίες που δημιούργησε αυτός.

Σε κάθε δημιουργία λίστας, επεξεργασία κάποιας, προσθήκη νέας εργασίας, διαγραφή κτλ, η σελίδα **δεν γίνεται refresh**, αλλά ενημερώνεται άμεσα.

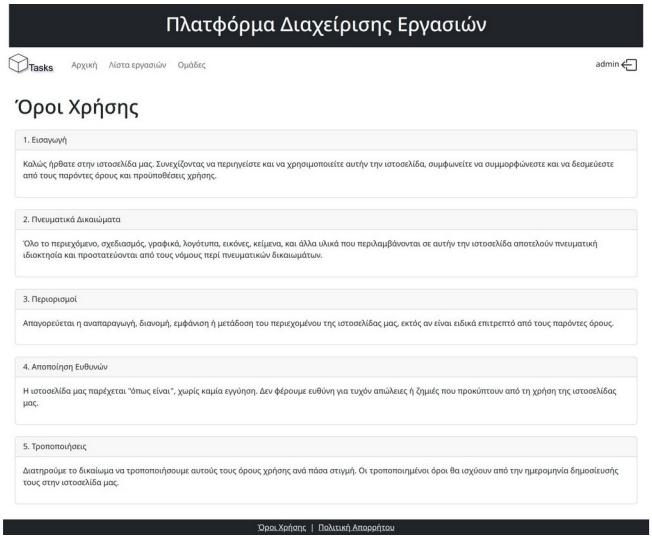


Στην **οθόνη των ομάδων** (έχει πρόσβαση σε αυτή μόνο ο **διαχειριστής**), εμφανίζονται οι ομάδες που έχει δημιουργήσει ο **διαχειριστής**. Κάθε **ομάδα** αποτελείται από την **λίστα μελών**, **φόρμα προσθήκης μέλους** στην ομάδα και **φόρμα ανάθεσης λίστας εργασιών** σε μέλος της ομάδας.

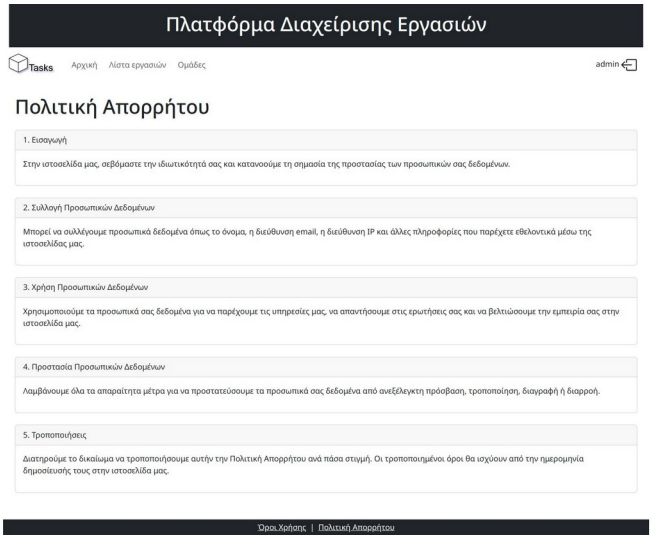
Η λίστα των χρηστών και το **select** επιλογής χρήστη για ανάθεση, ενημερώνονται άμεσα (χωρίς refresh) με την προσθήκη του νέου μέλους. Στο **select** επιλογής χρήστη για ανάθεση, εμφανίζονται μόνο τα μέλη της ομάδας.



Στο κάτω μέρος της σελίδας, υπάρχει **φόρμα για δημιουργία νέας ομάδας**. Με την δημιουργία, η σελίδα ενημερώνεται άμεσα (χωρίς refresh) με την νέα ομάδα.



Η **οθόνη των όρων χρήσης**, έχει απλά ένα γενικό κείμενο όρων χρήσης.



Παρόμοια η **οθόνη πολιτικής απορρήτου**.