**Entitások**

Alapvetően rendben vannak, de a három validációs függvény és a hibákat kezelő függvények minden entitásban kötelezők:

* public function checkIsValidInsert(): bool {}
* public function checkIsValidUpdate(): bool {}
* public function checkIsValidSave() :bool {}
* protected array $errors = [];
* public function getErrors(): array {}
* public function getErrorsAsString($separator = '<br>'): string {}

A checkIsValidInsert-tel volt egy olyan probléma, hogy a kulcsok nem követték az entitás tulajdonságainak nevét. Illetve hiba volt az is, hogy azonos kulcsokat használtál, így az egyes hibaüzenetek felülírták egymást.

Kötelező az adatbázis-adatokat egy művelettel átvevő fill() függvény is, amit a konstrukborban is meg kell hívni.

A getFields() függvényre és arra ami miatt ezt bevezetted, nincs szükség. A view táblázatait egyedileg szerkesztjük meg, ahogy az userek összefoglaló táblázatánál tegnap megcsináltuk. (Ilyesmire a query buildereknek lehet esetleg szüksége, de az nincs a Te rendszeredben.)

**Elnevezések**

Az adatbázisban a camel\_case jellegű elnevezést kéne használnod. Ez nem egy általános szabály, egyszerűen csak azért, mert az ügyviteli rendszerben mi is ezt használjuk. Így könnyeb lesz talán átállni majd az ügyviteli rendszerre.

Az entitások tulajdonságait úgy kell elnevezni, ahogy az adatbázis oszlopokat, ezért az entitások tulajdonságait is camel\_case stílusban kell használni.

A formokról érkező adatoknak az entitás tulajdonságainak megfelelő kulcs alatt kell érkezniük, ezért a „name” ott is camel\_case stílusú.

A hibaüzenetek tömbjében a kulcsoknak szintén az entitás tulajdonságaval kell megegyezniük, azaz a checkIsValidInsert-ben keletkező hibák is camel\_case kulcs alatt jönnek létre.

A HTML css classok kötőjelesek – ez sem egy általános szabály, inkább csak „így szokták”.

Minden mást, pl. függvényeket, változókat camelCase stílusban szoktuk, de ez is Rád van bízva. Az a lényeg hogy egységes legyen a rendszeren belül.

**Model**

A modellfüggvények esetén gyakori hiba volt, hogy az adatbázisból származó nyers adatokat adták vissza, ami a PDO:FETCH\_ASSOC miatt egy asszociatív tömb. Ezzel a tömbbel példányosítani kellett volna az entitást, de ez elmaradt, helyette a kontrollerben és a view-ban is a tömbökkel dolgoztál.

Objektumorientáltan programozunk, ezért nem tömbökkel dolgozunk, hanem objektumokkal. Az adatbázisból származó adatokból objektumokat készítünk (példányosítjuk a User entitást vagy a Car entitást).

Újdonság a kódodban a BaseModel. Ez egy Model szülőosztály, minden modelednek ki kell terjeszteni a BaseModelt. Ide kerülnek azok a függvények, amit minden model elérhet.

* Átkerült ide a model konstruktora benne a PDO kapcsolattal
* Létrejött a modelhibákat gyűjtő errors tömb és a két gettere (getErrors és getErrorsAsString)
* Létrejött az errorHandling, ami az exception hibákat dolgozza fel és kezeli egységesen

Az errorHandling-ot tanulmányozd át és ha szeretnéd, akkor szabd testre. Mindenhol úgy kellene alkalmazni a jövőben, ahogy az userModel-ben megcsináltuk. Ezzel a megoldással fejleszői módban a képernyőre tesszük az információt, éles üzemben viszont csak egy általános hibaüzenetet kap a felhasználó, a részletek a logban lesznek.

*Az adatbázissal kapcsolatos hibaüzenetek képernyőre kerülése fontos biztonsági probléma, ezért az éles rendszerben ez soha nem történhet meg!*

**Kontrollerek**

A BaseControllerbe bekerült a response és a request object. Ezek most már példányosítás nélkül minden kontrollerfunkcióban könnyen elérhetők.

checkAjax: új funkció a Base kontrollerben. Azt vizsgálja hogy a kérés ajax request-e vagy sem. Ha azt várunk, de nem azt kapunk, akkor elutasítjuk a kérést. Ezt a függvényt minden kontrollerfunkcióban le kell futtatni, ahol ajax requestet vársz.

A BaseControllerbe bekerült három jogosítást vizsgáló függvény, amik validálják a belépett usert, ellenőrzik hogy admin-e vagy csak vendég, és hiba esetén átirányítják a login oldalra a kérést.

A login form számára rögzítenek egy üzenetet a sessionba (ezt a login form renderelése előtt a kontrollernek kell kiolvasnia).

Az első két függvény végzi az ellenőrzést, de private lett mindkettő. A harmadik függvény hasonlít a codeigniterben használt eljáráshoz. Azért hoztam létre, hogy ezt a megnevezést szokd meg, így talán könnyebb lesz átállni majd a codeigniterre. Ott is minden kontrollerfunkció a checkPermission/checkAjax vizsgálattal kezdődik.

* authenticateUser
* authenticateAdmin
* checkPermission

**Feladatok:**

A kontrollerfunkcióból ki kell gyomlálni a new Request() és new Response() kifejezéseket, és helyettük $this->request és $this->response kifejezést kell használni, hiszen ez a két fontos rendszerobjektum minden kontroller számára elérhető a BaseController által.

Az ajax request kontrollerfunkciókban le kell futtatni a checkAjax függvényt. A bejelentkezett userhez kötött kontrollerfunkciókban le kell futtatni a checkPermission függvényt. A kontrollerfüggvények általában ezzel a két vizsgálattal kezdődnek és csak utánuk jöhet az érdemi rész (nyilván csak ott, ahol ez szükséges).

A Márkkal készített ajax requestet kiszolgáló kontrollerfunkciókban a response ne echo legyen, hanem a $this->response megfelelő függvénye! A kontrollerből soha nem lépünk ki a programból. A programot mindig a response objecten keresztül hagyjuk el!

A $\_POST és $\_GET szavak kizárólag kommentben fodulhatnak elő! *Az innen átvett nyers adatok az egyik legsúlyosabb biztonsági kockázatot jelentik a programod számára.* A kontrollerek adatokat kizárólag a $this->request objektumtól kaphatnak, és mindig valamilyen filterrel vesszük át tőlük az adatot!

**Core**

Itt csak a Request osztály került kiegészítésre az ip és néhány header függvénnyel. Ezeket felhasználjuk a kérés előzetes, gyors validációjában, amit az index.php-ban futtatunk (ezzel kiszűrjük a gyengén megírt rossz robotokat), de felhasználjuk a belépett user validációjában is az Authenticator osztályban. A belépett user kérését csak akkor fogadjuk el, ha

* a rendszer használatára engedélyezett idő nem telt el
* ha a kérés ugyanarról az ip címről érkezik, amiről az user belépett
* ha a kérés ugyanabból a böngészőből érkezik, amivel az user belépett

**View**

A formok első betöltéskor ne legyen valid formátumban (tűntesd el a zöld keretet és pipát).

A formok input mezői bootstrap szabványosak legyenek (autók szerkesztése).

A fromok labeljeiben magyar felirat legyen (új autó).

A gombok is magyarul legyenek (close pl. ne legyen sehol)

A formok küldés után mindig írja ki hogy „sikeres művelet” vagy hogy „sikertelen művelet” – utóbbi esetén a hiba okai legyenek láthatók. A minimum, hogy egyben valahol megjeleníted a hiba okait, de igazán az a megoldás profi, ha közvetlenül az alatt a mező alatt jeleníted meg, ahol a hiba keletkezett.

A táblázatban magyarul írjuk ki a dolgokat és csak olyan információt jelenítünk meg, amit az user meg tud változtatni. A created\_at updated\_at és deleted\_at nem kell, de nem kell a jelszó hash és hasonlók sem.

**Feladatok**

Minden usernek, így az adminnak is legyen profiloldala. A profiloldal ne egy táblázat legyen, hanem szépen egymás alatt felsorolva legyen a személyes adat (később lesz profilkép is, az nem fér bele a táblázatos fomába).

A profiloldalon mindenki módosítani tudja a saját adatait. Ugyanitt mindenki módosítani tudja a saját jelszavát. Az utolsó jelszó módosításának idejét tároljuk le és a profiloldalon jelenítsük is meg.

Az admin ne tudja módosítani egyelőre mások jelszavát (ezt majd akkor csináljuk meg, ha a rendszer tud emailt küldeni).

Legyen egy táblázat, amit minden bejelentkezett user láthat. Itt nincs szerkesztés, csak a belépett userek osztják meg egymással a fontosabb adataikat. Ezek legyenek az oszlopok:

Név, Életkor, Város, email, telefon, weboldal, autók (az email cím, a weboldal és az autók kattintható legyen!)

A kijelentkezésre kattintva a login form töltődik be. Ez jó így. De a login formon legyen egy zöld hátterű felirat, hogy „Sikeresen kijelentkezett”. Ilyesmi formátumban:

<https://getbootstrap.com/docs/4.5/components/alerts/>

A kontrollerfunkciókat próbáld meg egy esetleg két kontrollerbe csoportosítani. Egyelőre két entitásod van. Ez azt jelenti, hogy igazából kb. 2 kontrollerre van szükséged.

* Users: az userekeket kezelő kontroller, ide kerül a regisztráció, a módosítás, törlés és az ezekkel kapcsolatos view-k megjelenítése
* Cars: az autótípusokkal kapcsolatos összes művelet

És a két technikai kontroller:

* Login: bejelentkezés, kijelentkezés
* Index: ez a domain felkeresése miatt szükséges. Mivel ez elég soványka kontroller, csak 1 db index funkciója van, ezért átpakoltam ide az összes olyan infót, amit oktatóanyagként készítettünk a legutóbbi alkalmakkor: fájlkezelés, gyorsbillentyűk, adatbázis példa.

A többi kontrollerre nincs szükség. A kontrollerek neve általában ahhoz az entitáshoz hasonló, amit kezelnek, csak itt többesszámban kell írni az entitás nevét. Pl. az Usereket kezelő kontroller az Users. A Car entitásokat kezelő kontroller a Cars.

A Car entitással és a modellel és adatbázistábla-névvel is gondok lesznek, mert ezek nem autók, hanem autó típusok. Most lesz a feladat a konkrét autókra vonatkozóan. Szerencsésebb lenne a mostani Car-t CarType-nak nevezni, és a most kapott feladattal jöjjön létre a Car – hiszen most majd konkrét autókkal foglalkozunk majd.

**Feladat:**

Jelenleg az admin felvehet autótípusokat a rendszerbe. Az ügyfél azt szeretné, hogy a rendszerbe regisztrált userek a saját autóikat is felvehessék a rendszerbe. Ezekkel a tulajdonságokkal:

* **autó típusa**: itt azok közül lehet választani, amit az adminisztrátorok felvettek a rendszerbe
* **autó színe**: szabadon kitölthető
* **futott km**: szabadon kitölthető számszerű mező
* **üzemanyag fajtája**: „benzin” vagy „dízel” közül lehessen választani
* **állapot**: „leharcolt”, „megkímélt”, „felújított” közül lehessen választani
* **gyártás éve**: szabadon kitölthető mező, de az autó típusával összhangban kell lennie

A user a saját autóit tudja szerkeszteni és törölni. Az admin az összes user autóját tudja szerkeszteni és törölni.

A korábban említett user táblázatban az „autók”-ra kattintva töltődjön be az adott user autóinak táblázata. Ezzel minden belépett felhasználó láthatja az összes belépett felhasználó autóját a rendszerben.

Második feladat:

Az index kontrollerben létrehoztam egy függvényt, amit ezen az útvonalon fel is kereshetsz:

<https://marci.dev/index/feladat>

Nézd meg a függvényt és próbáld meg megoldani a feladatot. Arról van szó, hogy írni kell két olyan függvényt, ami a kötőjellel tagolt stringből snake case és camel case stílusú stringet készít. Egyszerű stringmanipulációs feladat arra az esetre ha nagyon unatkoznál a jövőben. De a fenti, autós feladat a fontosabb.