

Békéscsabai SZC Nemes Tihamér Technikum és Kollégium

Szakképesítés megnevezése: Szoftverfejlesztő és -tesztelő

Azonosító száma: 5 0613 12 03

VIZSGAREMEK

Flexistore.hu

Készítették:

Tanuló1 Neve: Gombkötő Gábor Osztály: 14/FE Oktatási azonosító: 73588965856

Tanuló2 Neve: Szirony Balázs Gábor Osztály: 14/FE Oktatási azonosító: 71609639101

Békéscsaba, 2024/2025



- **Dokumentum Kiemelt részei**
 - **Bevezetés:** [Bevezetés](#)
 - **Telepítés:** [Telepítőkészlet](#)
 - **Adatbázis EDR Diagram:** [EDR](#)
 - **Források:** [Forrás](#)
 - **Tesztek forráskódja:** [TesztKódok](#)
 - **FlexiStore dizájn képei:** [Melléklet](#)
- **Címek (Address)**
 - **Frontend:** [NewPublicArea](#), [PublicAreaCard](#), [PublicAreaList](#)
 - **React Controller:** [NewPublicArea.jsx](#), [PublicAreaCard.jsx](#),
[PublicAreaList.jsx](#)
 - **Context:** [CrudContext](#), [InitialContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [address](#), [street](#), [user](#)
 - **Model:** [address](#), [street](#), [user](#)
 - **Controller:** [Address](#)
- **Auth és User**
 - **Frontend:** [Login2](#), [Profile](#), [Register2](#)
 - **React Controller:** [Login2.jsx](#), [Profile.jsx](#), [Register2.jsx](#)
 - **Context:** [AuthContext](#), [CartCheckout](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [role](#), [user](#), [address](#), [profile](#)
 - **Model:** [role](#), [user](#), [address](#), [profile](#)
 - **Controller:** [Login](#), [Profile](#), [Register](#)



- **Role**
 - **Frontend:** [NewRole](#), [RoleCard](#), [RoleList](#)
 - **React Controller:** [NewRole.jsx](#), [RoleCard.jsx](#), [RoleList.jsx](#)
 - **Context:** [AuthContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [role](#)
 - **Model:** [role](#)
 - **Controller:** [Register](#)
- **Megrendelések (Cart)**
 - **Frontend:** [CartCheckout](#), [CartView](#), [OrderCheckout](#), [PaymentMethod](#),
[TermofUseInfo](#), [UserOrder](#), [UserDashboard](#), [UserOrdersCard](#)
 - **React Controller:** [CartCheckout.jsx](#), [CartView.jsx](#), [OrderCheckout.jsx](#),
[PaymentMethod.jsx](#), [TermofUseInfo.jsx](#), [UserOrder.jsx](#), [UserDashboard.jsx](#),
[UserOrdersCard.jsx](#)
 - **Context:** [AddressContext](#), [OrderContext](#), [ServiceContext](#), [CartContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [order](#), [orderItems](#), [address](#), [street](#)
 - **Model:** [order](#), [orderItems](#), [address](#), [street](#)
 - **Controller:** [Order](#), [Payment](#), [Address](#)
- **AdminDashboard**
 - **Frontend:** [AdminDashboard](#), [AdminDashboardCard](#), [AdminOrders](#),
[AdminOrdersCard](#), [RegistrationDataEdit](#)
 - **React Controller:** [AdminDashboard.jsx](#), [AdminDashboardCard.jsx](#),
[AdminOrders.jsx](#), [AdminOrdersCard.jsx](#), [RegistrationDataEdit.jsx](#)



- **Context:** [AdminContext](#)
- **Utils:** [SecureStorage](#)
- **Migration:** [user](#)
- **Model:** [user](#)
- **Controller:** [Admin](#), [Order](#), [Register](#), [CheckAdmin](#)
- **Csomagautomaták (Lockers)**
 - **Frontend:** [LockerInfo](#), [LockersCard](#), [LockersList](#), [NewLocker](#)
 - **React Controller:** [LockerInfo.jsx](#), [LockersCard.jsx](#), [LockersList.jsx](#), [NewLocker.jsx](#)
 - **Context:** [ServiceContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [Lockers](#), [LockerProduct](#)
 - **Model:** [Lockers](#), [LockerProduct](#)
 - **Controller:** [Locker](#)
- **FizetésiMód (Payment)**
 - **Frontend:** [NewPaymentMethod](#), [PaymentMethodCard](#), [PaymentMethodList](#)
 - **React Controller:** [NewPaymentMethod.jsx](#), [PaymentMethodCard.jsx](#), [PaymentMethodList.jsx](#)
 - **Context:** [PaymentContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [payment](#)
 - **Model:** [payment](#)
 - **Controller:** [Payment](#)



- **Termék (Products)**
 - **Frontend:** [NewProduct](#), [ProductCard](#), [ProductInfo](#), [ProductList](#)
 - **React Controller:** [NewProduct.jsx](#), [ProductCard.jsx](#), [ProductInfo.jsx](#), [ProductList.jsx](#)
 - **Context:** [CrudContext](#), [InitialContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [Product](#), [category](#)
 - **Model:** [Product](#), [category](#)
 - **Controller:** [Service](#)
- **Kategóriák (Categories)**
 - **Frontend:** [CategoriesCard](#), [CategoriesList](#), [NewCategory](#)
 - **React Controller:** [CategoriesCard.jsx](#), [CategoriesList.jsx](#), [NewCategory.jsx](#)
 - **Context:** [CrudContext](#), [InitialContext](#)
 - **Utils:** [SecureStorage](#)
 - **Migration:** [Categories](#)
 - **Model:** [Categories](#)
 - **Controller:** [Service](#)
- **Weblap Architecture**
 - **Frontend:** [FooterInfoUse](#), [FooterPrivacyInfo](#), [PrivacyPolicy](#), [TermofUse](#), [Main](#), [Menu](#), [App](#), [.env](#)
 - **React Controller:** [FooterInfoUse.jsx](#), [FooterPrivacyInfo.jsx](#), [PrivacyPolicy.jsx](#), [TermofUse.jsx](#), [Main.jsx](#), [Menu.jsx](#)
 - **Context:** [AuthContext](#), [CartContext](#)
 - **Utils:** [SecureStorage](#)



Tartalomjegyzék

Választott téma indoklása – Flexistore	30
Bevezetés	31
FlexiStore – A jövő automatizált bérlesi rendszere.....	31
Menu.jsx – Az alkalmazás menu komponense.....	32
Főbb funkciók.....	32
Reszponzív kialakítás	33
Felhasználói élmény	33
Összegzés	34
Footer.jsx – Az alkalmazás Footer komponense	35
Főbb funkciók.....	35
Reszponzív kialakítás	36
Felhasználói élmény	37
Összegzés	37
App.jsx – Az alkalmazás fő komponense	38
Főbb funkciók.....	38
Felépítés	40
Felhasználói élmény	40
Összegzés	40
FooterInfoUse.jsx – Felhasználási feltételek modális ablak	41
Főbb funkciók.....	41
Felépítés	42
Felhasználói élmény	42
Összegzés	42
TermOfUse.jsx – Felhasználási feltételek.....	43
Főbb funkciók.....	43
Felépítés	44
Felhasználói élmény	44
Tartalom szekciók	44
Összegzés	45
FooterPrivacyInfo.jsx – Adatvédelmi irányelvek modális ablak	46
Főbb funkciók.....	46
Felépítés	47
Felhasználói élmény	47



Összegzés	47
PrivacyPolicy.jsx – Adatvédelmi irányelvek	48
Főbb funkciók.....	48
Tartalom szekciók.....	49
Felhasználói élmény	50
Összegzés	50
MobileTableInfo.jsx – Mobil nézet jogi információk és márkaelemek	51
Főbb funkciók.....	51
Felépítés	52
Felhasználói élmény	52
Összegzés	53
secureStorage.jsx – Biztonságos adattárolás.....	54
Főbb funkciók.....	54
Felépítés	54
Használati példák.....	55
Felhasználói élmény és biztonság	55
Összegzés	55
.env – Környezeti változók.....	56
Főbb funkciók.....	56
Felépítés	56
Használat.....	57
Biztonsági megjegyzések.....	57
Összegzés	57
NewRole.jsx – Új jogosultság létrehozása és módosítása.....	58
Főbb funkciók.....	58
Felépítés	59
Felhasználói élmény	59
Összegzés	59
Főbb funkciók.....	60
Felépítés	60
Felhasználói élmény	61
Összegzés	61
RoleCard.jsx – Jogosultság kártya	62
Főbb funkciók.....	62



Felépítés	62
Felhasználói élmény	63
Összegzés	63
NewCategory.jsx – Új kategória létrehozása és módosítása.....	64
Főbb funkciók.....	64
Felépítés	65
Felhasználói élmény	65
Összegzés	65
CategoriesList.jsx – Kategóriák listázása	66
Főbb funkciók.....	66
Felépítés	66
Felhasználói élmény	67
Összegzés	67
CategoriesCard.jsx – Kategória kártya	68
Főbb funkciók.....	68
Felépítés	68
Felhasználói élmény	69
Összegzés	69
NewLocker.jsx – Új csomagautomata létrehozása és módosítása	70
Főbb funkciók.....	70
Felépítés	71
Felhasználói élmény	71
Összegzés	72
LockersList.jsx – Csomagautomaták listázása	73
Főbb funkciók.....	73
Felépítés	74
Felhasználói élmény	74
Összegzés	75
LockersCard.jsx – Csomagautomata kártya	76
Főbb funkciók.....	76
Felépítés	77
Felhasználói élmény	77
Összegzés	78
LockerInfo.jsx – Csomagautomata információs panel	79



Főbb funkciók	79
Felépítés	79
Felhasználói élmény	80
Összegzés	80
NewProduct.jsx – Új termék létrehozása és módosítása	81
Főbb funkciók	81
Felépítés	82
Felhasználói élmény	83
Összegzés	83
ProductsList.jsx – Termékek listázása	84
Főbb funkciók	84
Felépítés	85
Felhasználói élmény	85
Összegzés	86
ProductsCard.jsx – Termékkártya	87
Főbb funkciók	87
Felépítés	88
Felhasználói élmény	89
Összegzés	89
ProductsInfo.jsx – Termékinformációs panel	90
Főbb funkciók	90
Felépítés	90
Felhasználói élmény	91
Összegzés	91
NewPaymentMethod.jsx – Új fizetési mód létrehozása és módosítása	92
Főbb funkciók	92
Felépítés	93
Felhasználói élmény	93
Összegzés	93
PaymentMethodList.jsx – Fizetési módok listázása	94
Főbb funkciók	94
Felépítés	94
Felhasználói élmény	95
Összegzés	95



PaymentMethodCard.jsx – Fizetési mód kártya	96
Főbb funkciók.....	96
Felépítés	96
Felhasználói élmény	97
Összegzés	97
PublicAreaCard.jsx – Közterület kártyaNewPublicArea.jsx – Új közterület létrehozása és módosítása	98
Főbb funkciók.....	98
Felépítés	99
Felhasználói élmény	99
Összegzés	99
PublicAreaList.jsx – Közterületek listázása	100
Főbb funkciók.....	100
Felépítés	100
Felhasználói élmény	101
Összegzés	101
PublicAreaCard.jsx – Közterület kártya	102
Főbb funkciók.....	102
Felépítés	102
Felhasználói élmény	103
Összegzés	103
Register2.jsx – Felhasználói regisztráció.....	104
Főbb funkciók.....	104
Felépítés	105
Felhasználói élmény	105
Összegzés	106
Login2.jsx – Felhasználói bejelentkezés	107
Főbb funkciók.....	107
Felépítés	108
Felhasználói élmény	108
Összegzés	108
Profile.jsx – Felhasználói profil kezelése	109
Főbb funkciók.....	109
Felépítés	110
Felhasználói élmény	110



Összegzés	111
CartCheckout.jsx – Számlázási cím kezelése és rendelés leadása.....	111
Főbb funkciók.....	111
Felépítés	112
Felhasználói élmény	113
Összegzés	113
CartView.jsx – Kosár megtekintése és kezelése	114
Főbb funkciók.....	114
Felépítés	115
Felhasználói élmény	115
Összegzés	115
OrderCheckout.jsx – Rendelés véglegesítése	116
Főbb funkciók.....	116
Felépítés	117
Felhasználói élmény	117
Összegzés	117
PaymentMethod.jsx – Fizetési mód kiválasztása	118
Főbb funkciók.....	118
Felépítés	118
Felhasználói élmény	119
Összegzés	119
TermofUseInfo.jsx – Felhasználási feltételek elfogadása.....	120
Főbb funkciók.....	120
Felépítés	120
Felhasználói élmény	121
Összegzés	121
UserOrder.jsx – Felhasználói rendelések megtekintése	122
Főbb funkciók.....	122
Felépítés	123
Felhasználói élmény	123
Összegzés	123
UserCard.jsx – Felhasználói kártya kezelése.....	124
Főbb funkciók.....	124
Felépítés	125



Felhasználói élmény	125
Összegzés	125
UserDashboard.jsx – Felhasználói megrendelések kezelése.....	126
Főbb funkciók.....	126
Felépítés	127
Felhasználói élmény	127
Összegzés	127
UserOrdersCard.jsx – Megrendelési kártya.....	128
Főbb funkciók.....	128
Felépítés	128
Felhasználói élmény	129
Összegzés	129
AdminDashboard.jsx – Adminisztrátori felület a felhasználók kezelésére.....	130
Főbb funkciók.....	130
Felépítés	131
Felhasználói élmény	131
Összegzés	132
AdminDashboardCard.jsx – Adminisztrátori felhasználói kártya.....	133
Főbb funkciók.....	133
Felépítés	133
Felhasználói élmény	134
Összegzés	134
AdminOrders.jsx – Adminisztrátori megrendelések kezelése.....	135
Főbb funkciók.....	135
Felépítés	136
Felhasználói élmény	136
Összegzés	137
AdminOrdersCard.jsx – Adminisztrátori megrendelési kártya	138
Főbb funkciók.....	138
Felépítés	138
Felhasználói élmény	139
Összegzés	139
RegistrationDataEdit.jsx – Felhasználói adatok szerkesztése.....	140
Főbb funkciók.....	140



Felépítés	141
Felhasználói élmény	141
Összegzés	141
CrudContext.jsx – CRUD műveletek kezelése kontextusban	143
Főbb funkciók.....	143
Felépítés	144
Felhasználói élmény	144
Összegzés	145
InitialContext.jsx – Alapértelmezett adatok kezelése kontextusban	145
Főbb funkciók.....	145
Felépítés	146
Felhasználói élmény	147
Összegzés	147
AddressContext.jsx – Címek és közterületek kezelése kontextusban.....	148
Főbb funkciók.....	148
Felépítés	149
Felhasználói élmény	149
Összegzés	149
AdminContext.jsx – Adminisztrációs műveletek kezelése kontextusban.....	150
Főbb funkciók.....	150
Felépítés	151
Felhasználói élmény	151
Összegzés	151
AuthContext.jsx – Felhasználói hitelesítés és jogosultságok kezelése kontextusban	152
Főbb funkciók.....	152
Felépítés	153
Felhasználói élmény	153
Összegzés	154
CartContext.jsx – Kosárkezelés kontextusban.....	155
Főbb funkciók.....	155
Felépítés	156
Felhasználói élmény	156
Összegzés	156
OrderContext.jsx – Rendelések kezelése kontextusban.....	157



Főbb funkciók	157
Felépítés	158
Felhasználói élmény	158
Összegzés	159
PaymentContext.jsx – Fizetési módok kezelése kontextusban	159
Főbb funkciók	159
Felépítés	160
Felhasználói élmény	160
Összegzés	160
ServiceContext.jsx – Szolgáltatások és adatok kezelése kontextusban	161
Főbb funkciók	161
Felépítés	162
Felhasználói élmény	163
Összegzés	163
Register.cy.js – Regisztrációs tesztek Cypress-szel	164
Főbb funkciók	164
Felépítés	165
Felhasználói élmény	165
Összegzés	165
Login_test.cy.js – Bejelentkezási tesztek Cypress-szel	166
Főbb funkciók	166
Felépítés	166
Felhasználói élmény	167
Javaslatok a teszt bővítésére	167
Összegzés	167
Full_User_Process.cy.js – Teljes felhasználói folyamat tesztelése Cypress-szel	168
Főbb funkciók	168
Felépítés	169
Felhasználói élmény	169
Javaslatok a teszt bővítésére	170
Összegzés	170
0001_01_01_000000_create_roles_table.php – Laravel migráció a szerepkörök táblájához	171
Főbb funkciók	171
Felépítés	171



Felhasználási példa	172
Felhasználási esetek.....	172
Összegzés	172
0001_01_01_000001_create_users_table.php – Laravel migráció a felhasználók táblájához	173
Főbb funkciók.....	173
Felépítés	174
Felhasználási példa	174
Felhasználási esetek.....	175
Összegzés	175
2025_01_31_233753_create_profiles_table.php – Laravel migráció a profilok táblájához	176
Főbb funkciók.....	176
Felépítés	177
Felhasználási példa	177
Felhasználási esetek.....	177
Felhasználói élmény	178
Összegzés	178
2025_02_01_200058_create_categories_table.php – Laravel migráció a kategóriák táblájához	179
Főbb funkciók.....	179
Felépítés	179
Felhasználási példa	180
Felhasználási esetek.....	180
Felhasználói élmény	180
Összegzés	181
2025_02_01_200955_create_lockers_table.php – Laravel migráció a csomagautomaták táblájához	182
Főbb funkciók.....	182
Felépítés	182
Felhasználási példa	183
Felhasználási esetek.....	183
Felhasználói élmény	183
Összegzés	183



2025_02_01_200956_create_products_table.php – Laravel migráció a termékek táblájához	184
Főbb funkciók	184
Felépítés	185
Felhasználási példa	185
Felhasználási esetek	185
Felhasználói élmény	186
Összegzés	186
2025_02_17_000023_create_street_types_table.php – Laravel migráció az utcanevek típusainak táblájához	187
Főbb funkciók	187
Felépítés	187
Felhasználási példa	188
Felhasználási esetek	188
Felhasználói élmény	188
Összegzés	188
2025_02_17_000030_create_address_table.php – Laravel migráció a címek táblájához	189
Főbb funkciók	189
Felépítés	190
Felhasználási példa	190
Felhasználási esetek	190
Felhasználói élmény	191
Összegzés	191
2025_02_17_000039_create_payment_methods_table.php – Laravel migráció a fizetési módok táblájához	192
Főbb funkciók	192
Felépítés	192
Felhasználási példa	193
Felhasználási esetek	193
Felhasználói élmény	193
Összegzés	193
2025_02_17_000041_create_orders_table.php – Laravel migráció a rendelések táblájához	194
Főbb funkciók	194



Felépítés	195
Felhasználási példa	195
Felhasználási esetek.....	195
Felhasználói élmény	196
Összegzés	196
2025_02_17_000044_create_order_items_table.php – Laravel migráció a rendelési tételek táblájához.....	197
Főbb funkciók.....	197
Felépítés	198
Felhasználási példa	198
Felhasználási esetek.....	199
Felhasználói élmény	199
Összegzés	199
2025_04_12_172737_create_locker_product_table.php – Laravel migráció a csomagautomaták és termékek kapcsolótáblájához	200
Főbb funkciók.....	200
Felépítés	201
Felhasználási példa	201
Felhasználási esetek.....	201
Felhasználói élmény	202
Összegzés	202
RoleSeeder.php – Laravel adatfeltöltő a szerepkörökhez	202
Főbb funkciók.....	202
Felépítés	203
Felhasználási példa	203
Felhasználási esetek.....	203
Felhasználói élmény	204
Összegzés	204
UserSeeder.php – Laravel adatfeltöltő a felhasználókhöz.....	205
Főbb funkciók.....	205
Felépítés	205
Felhasználási példa	206
Felhasználási esetek.....	206
Felhasználói élmény	206
Összegzés	207



Address.php – Laravel modell a címekhez	208
Főbb funkciók.....	208
Felhasználási esetek.....	210
Felhasználói élmény	210
Összegzés	211
Category.php – Laravel modell a kategóriákhoz	212
Főbb funkciók.....	212
Felhasználási esetek.....	213
Felhasználói élmény	213
Összegzés	213
Customer.php – Laravel modell az ügyfelekhez.....	214
Főbb funkciók.....	214
Felhasználási esetek.....	215
Felhasználói élmény	216
Összegzés	216
Locker.php – Laravel modell a csomagautomatákhoz	217
Főbb funkciók.....	217
Felhasználási esetek.....	218
Felhasználói élmény	218
Összegzés	219
Order.php – Laravel modell a rendelésekhez	220
Főbb funkciók.....	220
Felhasználási esetek.....	222
Felhasználói élmény	222
Összegzés	222
OrderItem.php – Laravel modell a rendelési tételekhez	223
Főbb funkciók.....	223
Felhasználási esetek.....	225
Felhasználói élmény	225
Összegzés	225
PaymentMethod.php – Laravel modell a fizetési módokhoz	226
Főbb funkciók.....	226
Felhasználási esetek.....	227
Felhasználói élmény	227



Összegzés	227
Product.php – Laravel modell a termékekhez.....	228
Főbb funkciók.....	228
Felhasználási esetek.....	229
Felhasználói élmény	230
Összegzés	230
Profile.php – Laravel modell a profilokhoz	231
Főbb funkciók.....	231
Felhasználási esetek.....	232
Felhasználói élmény	232
Összegzés	232
Role.php – Laravel modell a szerepkörökhöz.....	233
Főbb funkciók.....	233
Felhasználási esetek.....	234
Felhasználói élmény	234
Összegzés	234
StreetType.php – Laravel modell az utca típusokhoz	235
Főbb funkciók.....	235
Felhasználási esetek.....	236
Felhasználói élmény	236
Összegzés	236
User.php – Laravel modell a felhasználókhoz	237
Főbb funkciók.....	237
Felhasználási esetek.....	239
Felhasználói élmény	240
Összegzés	240
FlexiBox API Útvonalak Dokumentáció	241
1. Hitelesítés (Auth API-k)	241
2. Profilkezelés (Profile API-k).....	241
3. Felhasználói rendelések (User Orders API-k).....	242
4. Adminisztrációs műveletek (Admin API-k).....	243
5. Termékek és kategóriák (Product API-k)	244
6. Címek és utca típusok (Address API-k)	245
CheckAdmin.php – Middleware az adminisztrátori jogosultságok ellenőrzésére	246



Főbb funkciók	246
Felhasználási esetek	247
Példa használatra	248
Felhasználói élmény	248
Összegzés	248
AddressController.php – Laravel vezérlő a címek és utca típusok kezelésére	249
Főbb metódusok	249
Példa API-hívások	251
Felhasználói élmény	252
Összegzés	252
AdminController.php – Laravel vezérlő az adminisztrációs funkciókhoz	253
Főbb metódusok	253
Példa API-hívás	254
Felhasználási esetek	255
Felhasználói élmény	255
Összegzés	255
LockerController.php – Laravel vezérlő a csomagautomaták kezelésére	256
Főbb metódusok	256
Példa API-hívások	259
Felhasználói élmény	260
Összegzés	261
LoginController.php – Laravel vezérlő a bejelentkezés kezelésére	262
Főbb metódusok	262
Példa API-hívás	264
Felhasználási esetek	265
Felhasználói élmény	265
Összegzés	266
OrderController.php – Laravel vezérlő a rendelések kezelésére	266
Főbb metódusok	266
Példa API-hívások	270
Felhasználói élmény	271
Összegzés	271
PaymentController.php – Laravel vezérlő a fizetési módok kezelésére	272
Főbb metódusok	272



Példa API-hívások.....	275
Felhasználói élmény	276
Összegzés	276
ProfileController.php – Laravel vezérlő a profilok kezelésére	277
Főbb metódusok.....	277
Példa API-hívások.....	280
Felhasználói élmény	281
Összegzés	281
RegisterController.php – Laravel vezérlő a regisztráció és felhasználókezelés kezelésére	282
Főbb metódusok.....	282
Felhasználói élmény	287
Összegzés	287
ServiceController.php – Laravel vezérlő a kategóriák és termékek kezelésére	288
Főbb metódusok.....	288
Felhasználói élmény	293
Összegzés	293
React Controller dokumentáció – NewPublicArea.jsx	294
Összegzés	296
React Controller Dokumentáció – PublicAreaCard.jsx	297
Funkciók	297
API-hívások	297
Context használata	298
Adatok kezelése.....	299
Felhasználói Élmény.....	300
Összegzés	300
React Controller Dokumentáció – PublicAreaList.jsx.....	301
Funkciók	301
API-hívások	301
Context használata	302
Adatok kezelése.....	302
Felhasználói Élmény.....	303
Komponens Felépítése	303
Felhasználói Élmény.....	303
Összegzés	304



React Controller Dokumentáció – Login2.jsx.....	305
Funkciók	305
API-hívások	305
Context használata	306
Adatok kezelése.....	307
Felhasználói Élmény.....	308
Komponens Felépítése	309
Összegzés	310
React Controller Dokumentáció – Profile.jsx	311
Funkciók	311
API-hívások	312
Context használata	313
Adatok kezelése.....	313
Felhasználói Élmény.....	314
Komponens Felépítése	314
Összegzés	316
React Controller Dokumentáció – Register2.jsx	317
Funkciók	317
API-hívások	317
Adatok kezelése.....	319
Felhasználói Élmény.....	320
Komponens Felépítése	321
Összegzés	322
React Controller Dokumentáció – CartCheckout.jsx	323
Funkciók	323
API-hívások	324
Adatok kezelése.....	326
Felhasználói Élmény.....	328
Komponens Felépítése	329
Összegzés	330
React Controller Dokumentáció – CartView.jsx	331
Funkciók	331
1. Kosár tartalmának megjelenítése	331
2. Termék mennyiségének módosítása	331
3. Termék eltávolítása a kosáról.....	331



4. Kosár kiürítése	331
5. Tovább a pénztárhoz	332
Adatok kezelése.....	332
1. Kosár adatok kezelése.....	332
2. Átvevőpontok kezelése	332
3. Adatvédelmi feltételek kezelése	332
Felhasználói Élmény.....	333
Komponens Felépítése	333
1. Kosár tartalma	333
2. Mennyiség módosítása	334
3. Kosár kiürítése	334
4. Tovább a pénztárhoz	334
5. Adatvédelmi feltételek modális ablak.....	334
Összegzés	335
Funkciók	336
API-hívások	337
Context használata	338
Adatok kezelése.....	338
Felhasználói Élmény.....	339
Komponens Felépítése	340
Összegzés	340
React Controller Dokumentáció – PaymentMethod.jsx	341
Funkciók	341
API-hívások	342
Context használata	342
Adatok kezelése.....	343
Felhasználói Élmény.....	344
Komponens Felépítése	344
Összegzés	345
React Controller Dokumentáció – TermofUseInfo.jsx.....	346
Funkciók	346
Adatok kezelése.....	347
Felhasználói Élmény.....	348
Komponens Felépítése	348
Összegzés	349



React Controller Dokumentáció – UserOrder.jsx	350
Funkciók	350
API-hívások	351
Context használata	352
Adatok kezelése.....	352
Felhasználói Élmény.....	353
Komponens Felépítése	353
Összegzés	354
React Controller Dokumentáció – CategoriesCard.jsx	355
Funkciók	355
API-hívások	355
Context használata	356
Adatok kezelése.....	357
Felhasználói Élmény.....	357
Komponens Felépítése	358
Összegzés	359
React Controller Dokumentáció – CategoriesList.jsx.....	360
Funkciók	360
API-hívások	360
Context használata	361
Adatok kezelése.....	361
Felhasználói Élmény.....	362
Komponens Felépítése	362
Felhasználói Élmény.....	363
Összegzés	363
React Controller Dokumentáció – NewCategory.jsx	364
Funkciók	364
API-hívások	365
Context használata	366
Adatok kezelése.....	367
Felhasználói Élmény.....	369
Komponens Felépítése	369
Összegzés	370
React Controller Dokumentáció – AdminDashboard.jsx	371
Funkciók	371



API-hívások	372
Context használata	373
Adatok kezelése	374
Felhasználói Élmény	375
Komponens Felépítése	375
Összegzés	377
React Controller Dokumentáció – AdminDashboardCard.jsx	378
Funkciók	378
API-hívások	379
Context használata	379
Adatok kezelése	380
Felhasználói Élmény	380
Komponens Felépítése	381
Összegzés	382
React Controller Dokumentáció – AdminOrders.jsx	383
Funkciók	383
API-hívások	384
Adatok kezelése	386
Felhasználói Élmény	388
Komponens Felépítése	389
Összegzés	390
React Controller Dokumentáció – AdminOrdersCard.jsx	391
Funkciók	391
Adatok kezelése	391
Felhasználói Élmény	392
Komponens Felépítése	393
Összegzés	394
React Controller Dokumentáció – RegistrationDataEdit.jsx	395
Funkciók	395
API-hívások	396
Context használata	397
Adatok kezelése	397
Felhasználói Élmény	398
Komponens Felépítése	399
Összegzés	400



React Controller Dokumentáció – UserCard.jsx.....	401
Funkciók	401
Adatok kezelése.....	402
Felhasználói Élmény.....	403
Komponens Felépítése	403
Összegzés	404
React Controller Dokumentáció – UserDashboard.jsx.....	405
Funkciók	405
API-hívások	406
Adatok kezelése.....	408
Felhasználói Élmény.....	410
Komponens Felépítése	411
Összegzés	411
React Controller Dokumentáció – UserOrdersCard.jsx	412
Funkciók	412
Adatok kezelése.....	412
Felhasználói Élmény.....	413
Komponens Felépítése	414
Összegzés	415
React Controller Dokumentáció – LockerInfo.jsx	416
Funkciók	416
Adatok kezelése.....	416
Felhasználói Élmény.....	417
Komponens Felépítése	417
Összegzés	418
React Controller Dokumentáció – LockersCard.jsx.....	419
Funkciók	419
API-hívások	420
Adatok kezelése.....	421
Felhasználói Élmény.....	422
Komponens Felépítése	422
Összegzés	424
React Controller Dokumentáció – LockersList.jsx	425
Funkciók	425



Adatok kezelése.....	426
Felhasználói Élmény.....	427
Komponens Felépítése	427
Összegzés	429
React Controller Dokumentáció – NewLocker.jsx	430
Funkciók	430
API-hívások	431
Adatok kezelése.....	432
Felhasználói Élmény.....	433
Komponens Felépítése	433
Összegzés	434
React Controller Dokumentáció – NewPaymentMethod.jsx.....	435
Funkciók	435
API-hívások	436
Adatok kezelése.....	437
Felhasználói Élmény.....	438
Komponens Felépítése	438
Összegzés	439
React Controller Dokumentáció – PaymentMethodCard.jsx	440
Funkciók	440
API-hívások	441
Adatok kezelése.....	441
Felhasználói Élmény.....	442
Komponens Felépítése	443
Összegzés	443
React Controller Dokumentáció – PaymentMethodList.jsx	444
Funkciók	444
Adatok kezelése.....	444
Felhasználói Élmény.....	445
Komponens Felépítése	445
Összegzés	446
React Controller Dokumentáció – NewProduct.jsx	447
Funkciók	447
API-hívások	447
Adatok kezelése.....	449



Felhasználói Élmény	451
Komponens Felépítése	451
Összegzés	453
React Controller Dokumentáció – ProductsCard.jsx	454
Funkciók	454
API-hívások	455
Adatok kezelése	456
Felhasználói Élmény	457
Komponens Felépítése	457
Összegzés	459
React Controller Dokumentáció – ProductsInfo.jsx	460
Funkciók	460
Adatok kezelése	460
Felhasználói Élmény	461
Komponens Felépítése	461
Összegzés	462
React Controller Dokumentáció – ProductsList.jsx	463
Funkciók	463
Adatok kezelése	464
Felhasználói Élmény	465
Komponens Felépítése	466
Összegzés	468
React Controller Dokumentáció – NewRole.jsx	469
Funkciók	469
API-hívások	470
Adatok kezelése	471
Felhasználói Élmény	472
Komponens Felépítése	472
Összegzés	473
React Controller Dokumentáció – RoleCard.jsx	474
Funkciók	474
API-hívások	475
Adatok kezelése	475
Felhasználói Élmény	476
Komponens Felépítése	477



Összegzés	477
React Controller Dokumentáció – RolesList.jsx.....	478
Funkciók	478
Adatok kezelése.....	478
Felhasználói Élmény.....	479
Komponens Felépítése	479
Összegzés	480
React Controller Dokumentáció – FooterInfoUse.jsx.....	481
Funkciók	481
Adatok kezelése.....	481
Felhasználói Élmény.....	482
Komponens Felépítése	482
Összegzés	483
React Controller Dokumentáció – FooterPrivacyInfo.jsx	484
Funkciók	484
Adatok kezelése.....	484
Felhasználói Élmény.....	485
Komponens Felépítése	485
Összegzés	486
React Controller Dokumentáció – PrivacyPolicy.jsx	487
Funkciók	487
Adatok kezelése.....	487
Felhasználói Élmény.....	488
Komponens Felépítése	488
Összegzés	489
React Controller Dokumentáció – TermOfUse.jsx.....	490
Funkciók	490
Adatok kezelése.....	490
Felhasználói Élmény.....	491
Komponens Felépítése	491
Összegzés	492
React Controller Dokumentáció – Footer.jsx.....	493
Funkciók	493
Adatok kezelése.....	494



Felhasználói Élmény.....	495
Komponens Felépítése	495
Összegzés	497
React Controller Dokumentáció – Main.jsx	498
Funkciók	498
Adatok kezelése.....	498
Felhasználói Élmény.....	499
Komponens Felépítése	499
Összegzés	501
React Controller Dokumentáció – Menu.jsx.....	502
Funkciók	502
Adatok kezelése.....	503
Felhasználói Élmény.....	503
Komponens Felépítése	504
Összegzés	505
React Controller Dokumentáció – MobileTableInfo.jsx	506
Funkciók	506
Adatok kezelése.....	507
Felhasználói Élmény.....	507
Komponens Felépítése	508
Összegzés	509
FlexiBox Telepítési Útmutató	510
ERD Diagramok	512
Tesztelés	519
Összegzés -Flexistore.....	526
Melléklet:.....	528
Források:.....	529



Választott téma indoklása – Flexistore

A vizsgaremek témájául a **Flexistore** nevű **modern eszközbérlési rendszer** megvalósítását választottuk, amely lehetővé teszi a felhasználók számára, hogy különböző termékeket béröljenek az ország több pontján elhelyezett csomagautomatákból. A téma egyedi ötletként **Szirony Balázs Gábortól** származik, amelyet megosztott **Gombkötő Gáborral** és közösen dolgoztunk ki a projekt megvalósításának részleteit.

A rendszer lényege, hogy a felhasználók az ország különböző pontjain található csomagautomatákból tudnak különféle termékeket kibérelni – egyszerűen, gyorsan és online. A célunk egy olyan digitális megoldás létrehozása volt, amely valós felhasználói igényt szolgál ki, és a gyakorlatban is alkalmazható.

A technológiai választásaink is ezt a célt szolgálták:

- **React** keretrendszer alkalmaztunk a frontend fejlesztéshez, mivel lehetővé teszi egy dinamikus, jól strukturált, komponens-alapú felhasználói felület kialakítását.
- A backend oldalon a **Laravel** keretrendszerre esett a választásunk, mert biztonságos, rugalmas és skálázható architektúrát kínál, valamint támogatja a REST API-k gyors és hatékony fejlesztését.
- A stílus kialakításában a **Tailwind CSS** és a rá épülő **DaisyUI** játszott kulcsszerepet, amelyek gyors, egységes és modern megjelenést biztosítanak az alkalmazásnak anélkül, hogy a dizájn elemeket nulláról kellett volna megírnunk.

A választott téma és a technológiai háttér lehetővé tette számunkra, hogy a szoftverfejlesztő és tesztelő képzés során megszerzett tudást valós, éles környezetben működő rendszerben kamatoztassuk, miközben a csapatmunka és az egyedi ötlet megvalósítása is fontos szerepet kapott.



Bevezetés

FlexiStore – A jövő automatizált bérlesi rendszere

A FlexiStore egy modern platform, amely az automatizált bérlesi rendszerek új generációját képviseli. Az alkalmazás célja, hogy a felhasználók számára egyszerű és hatékony módot biztosítson csomagautomaták kezelésére, regisztrációra, és egyéb szolgáltatások igénybevételére.

Az alkalmazás főbb funkciói:

- Csomagautomaták kezelése:** A felhasználók könnyedén böngészhetik és használhatják a csomagautomatákat.
- Felhasználói regisztráció:** Új felhasználók gyorsan és egyszerűen regisztrálhatnak a platformra.
- Innováció és technológia:** A legújabb technológiák alkalmazásával a FlexiStore célja, hogy jobbá tegye a felhasználók életét.

Készítette:

Szirony Balázs Gábor és Gombkötő Gábor

© minden jog fenntartva.

The screenshot shows the desktop version of the Flexistore website. At the top, there's a dark header bar with the Flexistore logo and a search bar. Below it is a light-colored main content area. On the left, there's a large teal-colored cube icon. In the center, there's a heading "Üdvözölünk a FlexiStore demóban!" followed by some descriptive text and a "Csomagautomaták" button. At the bottom, there are two columns of links: "SZOLGÁLTATÁSOK" (Csomagautomaták, Összes Termék) and "JOGI INFORMÁCIÓK" (Felhasználi feltételek, Adatvédelmi irányelv). The footer contains the Flexistore logo, copyright information ("Flexistore. All rights reserved © 2025"), and social media icons for Instagram and Facebook.

1. ábra Flexistore.hu desktop



Menu.jsx – Az alkalmazás menu komponense

A Menu.jsx az alkalmazás navigációs menűjét valósítja meg, amely reszponzív kialakításának köszönhetően asztali és mobil nézetben is megfelelően működik. A menü a felhasználók és adminisztrátorok számára egyaránt biztosítja a szükséges funkciókat.

Főbb funkciók

1. Logó és kezdőlap link

A menü bal oldalán található a FlexiStore logója, amely a kezdőlapra navigál. A logó SVG ikonként jelenik meg, és a márka vizuális azonosítását segíti.

2. Navigációs linkek

- **Asztali nézet:** A menü középső részén találhatók a főbb funkciókhoz vezető linkek, például a csomagautomatákhoz és a termékekhez.
- **Mobil nézet:** Egy hamburger menü biztosítja a navigációt, amely a kisebb képernyőkön is könnyen használható.

3. Adminisztrációs menü

Az adminisztrátorok számára elérhető egy speciális menü, amely tartalmazza az adminisztrációs funkciókat, például új termékek, kategóriák, vagy jogosultságok kezelését. Ez a menü csak akkor jelenik meg, ha a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik.



4. Kosár ikon

A menü jobb oldalán található a kosár ikon, amely jelzi a kosárban lévő termékek számát és az összesített árat. A kosár tartalma egy legördülő menüben jelenik meg, amely gyors hozzáférést biztosít a kosár megtekintéséhez.

5. Felhasználói profil és hitelesítés

- **Bejelentkezett felhasználók:** A profilkép és a kijelentkezési lehetőség jelenik meg. A profilkép egy legördülő menüben további opciókat kínál, például a profiladatok megtekintését.
- **Vendégek:** Regisztrációs és bejelentkezési gombok jelennek meg, amelyek a megfelelő oldalakra navigálnak.

Reszponzív kialakítás

A Menu.jsx reszponzív kialakítású, így különböző eszközökön is megfelelően működik:

- **Asztali nézet:** A menü teljes szélességben jelenik meg, a navigációs linkek és funkciók jól láthatóak.
- **Mobil nézet:** A menü egy hamburger ikon mögé rejtett, amely érintéssel nyitható meg. Ez biztosítja a könnyű használatot kisebb képernyőkön is.

Felhasználói élmény

A menü kialakítása a felhasználói élményt helyezi előtérbe:

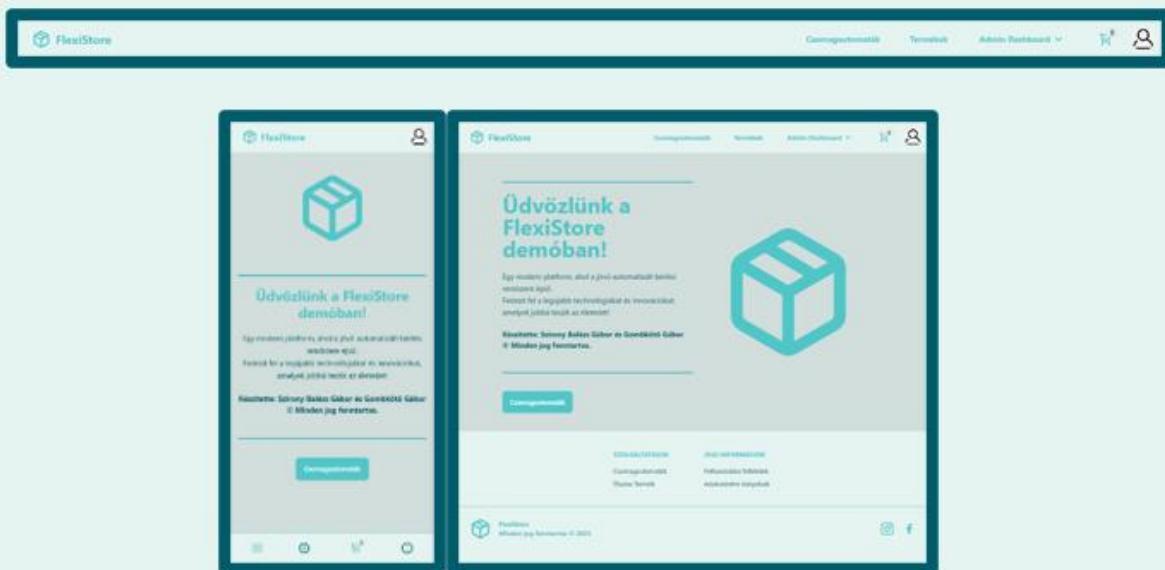
- Az adminisztrációs funkciók csak az arra jogosult felhasználók számára érhetők el, így a menü egyszerűbb és átláthatóbb a hétköznapi felhasználók számára.



- A kosár ikon és a profilkép vizuálisan kiemelkedik, így a felhasználók gyorsan elérhetik a legfontosabb funkciókat.

Összegzés

A Menu.jsx egy jól strukturált, reszponzív navigációs menü, amely az alkalmazás minden felhasználói csoportjának igényeit kielégíti. A menü egyszerre biztosítja a könnyű navigációt, a fontos funkciók gyors elérését, és az adminisztrációs lehetőségeket az arra jogosult felhasználók számára.



2. ábra Menu responsive



Footer.jsx – Az alkalmazás Footer komponense

A [Footer.jsx](#) az alkalmazás láblécét valósítja meg, amely több funkciót is ellát. A lábléc reszponzív kialakítású, így asztali, tablet és mobil nézetben is megfelelően működik. A lábléc tartalmaz navigációs linkekkel, jogi információkat, közösségi média ikonokat, valamint mobil/tablet nézetben egy alsó navigációs menüt.

Főbb funkciók

1. Navigációs linkek (Desktop nézet)

A lábléc felső részén találhatók a navigációs linkek, amelyek a következőket tartalmazzák:

- **Szolgáltatások:** Linkek a csomagautomatákhoz és az összes termékhez.
- **Jogi információk:** Gombok a felhasználási feltételek és az adatvédelmi irányelvezek megnyitásához. Ezek modális ablakban jelennek meg.

2. Alsó lábléc (Desktop nézet)

Az alsó lábléc tartalmazza:

- **Logó és copyright információk:** A FlexiStore logója és a "Minden jog fenntartva © 2025" szöveg.
- **Közösségi média ikonok:** Linkek az Instagram és Facebook oldalakhoz.

3. Mobil/tablet alsó navigáció



Mobil és tablet nézetben az alsó navigációs menü biztosítja a gyors hozzáférést a főbb funkciókhoz:

- **Felhasználói menü:** Linkek a csomagautomatákhoz, termékekhez, és a felhasználói dashboardhoz.
- **Adminisztrációs menü:** Adminisztrátorok számára elérhető funkciók, például új termékek, kategóriák, vagy jogosultságok kezelése.

4. Jogi információk kezelése

A felhasználási feltételek és az adatvédelmi irányelvek megnyitása és bezárása állapotváltozókkal történik:

- A setTermInfo és setPolicyInfo állapotok kezelik a modális ablakok nyitott/zárt állapotát.

Reszponzív kialakítás

A Footer.jsx reszponzív kialakításának köszönhetően különböző eszközökön is megfelelően működik:

- **Asztali nézet:** A lábléc két részből áll: a felső navigációs linkekből és az alsó logóból, copyright információkból, valamint közösségi média ikonokból.
 - **Mobil/tablet nézet:** Az alsó navigációs menü biztosítja a gyors hozzáférést a főbb funkciókhoz, és külön adminisztrációs menü is elérhető az arra jogosult felhasználók számára.
-



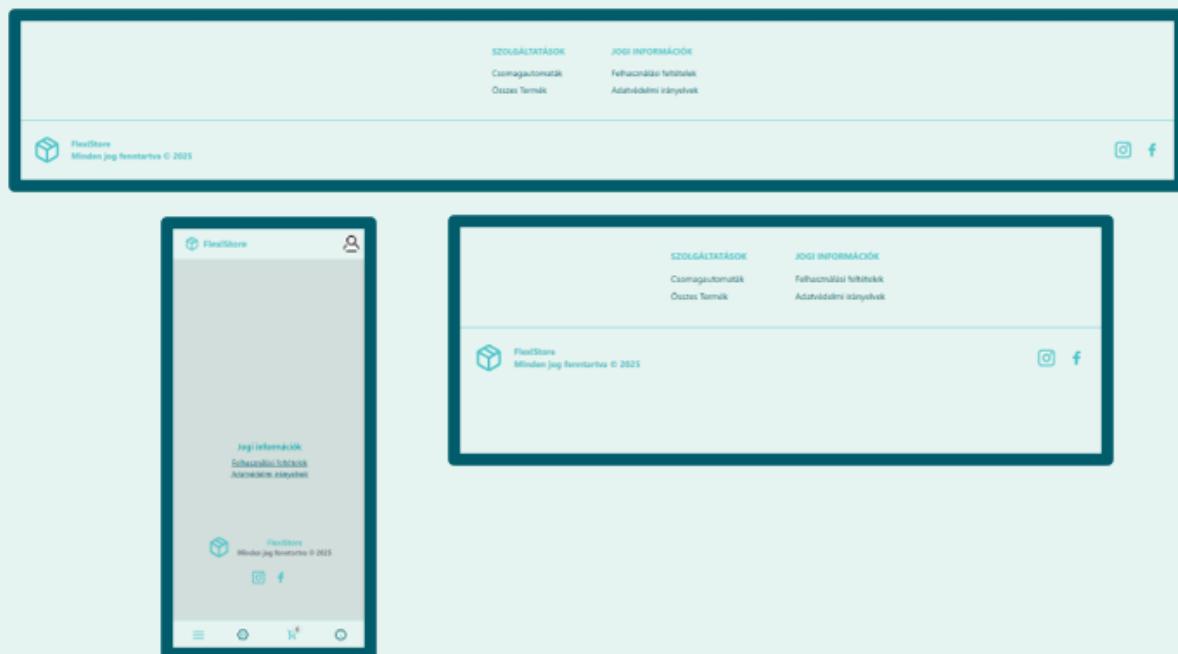
Felhasználi élmény

A lábléc kialakítása a felhasználi élményt helyezi előtérbe:

- A navigációs linkek és jogi információk jól elkülönülnek, így könnyen elérhetők.
- A mobil/tablet nézetben az alsó navigációs menü biztosítja a gyors hozzáférést a legfontosabb funkciókhoz.
- Az adminisztrációs menü csak az arra jogosult felhasználók számára jelenik meg, így a lábléc egyszerűbb és átláthatóbb a hétköznapi felhasználók számára.

Összegzés

A Footer.jsx egy jól strukturált, reszponzív lábléc, amely az alkalmazás minden felhasználi csoportjának igényeit kielégíti. A lábléc egyszerre biztosítja a navigációs linkeket, jogi információkat, és a mobil eszközökre optimalizált alsó navigációs menüt.



3. ábra Footer responsive



App.jsx – Az alkalmazás fő komponense

Az [App.jsx](#) az alkalmazás fő komponense, amely összefogja az alkalmazás különböző részeit, például a routert, a globális állapotkezelést biztosító context-eket, valamint a főbb vizuális elemeket, mint a menü és a lábléc. Ez a fájl határozza meg az alkalmazás alapvető szerkezetét és működését.

Főbb funkciók

1. Globális állapotkezelés (Context Providers)

Az [App.jsx](#) több context providert használ, amelyek az alkalmazás különböző részeinek globális állapotkezelését biztosítják:

- **CrudProvider:** CRUD műveletek kezelése.
- **PaymentProvider:** Fizetési módok kezelése.
- **AdminProvider:** Adminisztrációs funkciók kezelése.
- **CartProvider:** Kosár állapotának kezelése.
- **AuthProvider:** Felhasználói hitelesítés és profilkezelés.
- **OrderProvider:** Rendelések kezelése.
- **ServiceProvider:** Szolgáltatások állapotának kezelése.
- **AddressProvider:** Címek és közterületek kezelése.
- **InitialProvider:** Alapértelmezett beállítások kezelése.

Ezek a context-ek biztosítják, hogy az alkalmazás különböző részei (pl. komponensek, oldalak) könnyen hozzáférjenek a szükséges adatokhoz és funkcióhoz.



2. Router és útvonalak (Routes)

Az [App.jsx](#) a react-router-dom segítségével kezeli az alkalmazás útvonalait. Az útvonalak a következő főbb funkciókat biztosítják:

- a. **Főoldal:** A / útvonal a [Main](#) komponenshez navigál.
- b. **Felhasználói hitelesítés:** Regisztráció (/register2), bejelentkezés (/login2), és profil (/profile).
- c. **Kosár és rendelés:** Kosár megtekintése (/cart), rendelés leadása (/checkout), és rendelési adatok (/userorder).
- d. **Adminisztrációs funkciók:** Admin dashboard (/admashboard), új kategória (/newcategory), új termék (/newproduct), és egyéb adminisztrációs oldalak.
- e. **Egyéb funkciók:** Termékek listája (/products), csomagautomaták (/lockers), és közterületek (/publicareas).

Az útvonalak reszponzívak, és a nem létező útvonalak esetén a felhasználót a főoldalra (/) irányítják.

3. Főbb vizuális elemek

Az [App.jsx](#) tartalmazza az alkalmazás főbb vizuális elemeit:

- a. **Menu:** Az alkalmazás navigációs menüje.
- b. **Footer:** Az alkalmazás lábléce.
- c. **ToastContainer:** Értesítések megjelenítése a react-toastify segítségével.



Felépítés

Az [App.jsx](#) szerkezete a következőképpen épül fel:

- Context Providers:** A context-ek hierarchikusan vannak elhelyezve, hogy biztosítsák az állapotkezelést az alkalmazás minden részén.
- Router:** A [BrowserRouter](#) biztosítja az útvonalak kezelését.
- Komponensek:** A [Menu](#) és a [Footer](#) komponensek az alkalmazás navigációs és lábléc részeit valósítják meg.
- Útvonalak:** A [Routes](#) komponens tartalmazza az összes elérhető útvonalat és azokhoz tartozó komponenseket.

Felhasználói élmény

Az [App.jsx](#) biztosítja, hogy az alkalmazás:

- Könnyen navigálható legyen a különböző oldalak között.
- Reszponzív módon működjön, így minden eszközön megfelelő felhasználói élményt nyújtson.
- A globális állapotkezelés révén hatékonyan kezelje a felhasználói adatokat, a kosár tartalmát, és az adminisztrációs funkciókat.

Összegzés

Az [App.jsx](#) az alkalmazás központi eleme, amely összefogja a különböző context-eket, útvonalakat és vizuális komponenseket. Ez a fájl biztosítja az alkalmazás alapvető működését és szerkezetét, valamint a felhasználói élmény alapját.



FooterInfoUse.jsx – Felhasználási feltételek modális ablak

A [FooterInfoUse.jsx](#) egy egyszerű, de fontos komponens, amely a felhasználási feltételek megjelenítésére szolgál. Ez a komponens egy modális ablakot biztosít, amelyben a felhasználók elolvashatják a felhasználási feltételeket, és egy gomb segítségével bezárhatják az ablakot.

Főbb funkciók

1. Felhasználási feltételek megjelenítése

A [FooterInfoUse.jsx](#) a [TermOfUse](#) nevű alkomponenst használja, amely tartalmazza a felhasználási feltételek szövegét. Ez biztosítja, hogy a jogi információk jól strukturáltan és könnyen olvashatóan jelenjenek meg.

2. Modális ablak

A komponens egy modális ablakot jelenít meg, amely a következőket tartalmazza:

- **Tartalom:** A [TermOfUse](#) komponens által biztosított szöveg.
- **Bezárás gomb:** Egy jól látható "X" gomb, amely lehetővé teszi a felhasználók számára a modális ablak bezárását.

3. Bezárási funkció

A [closeFunction](#) prop segítségével a komponens lehetővé teszi a modális ablak bezárását. Ez a funkció rugalmasan kezelhető, és a szülő komponens biztosítja a megfelelő állapotkezelést.



Felépítés

- **Modális ablak:** A modal és modal-box osztályok biztosítják a modális ablak megjelenését és stílusát.
 - **Tartalom:** A [TermOfUse](#) komponens jeleníti meg a felhasználási feltételek szövegét.
 - **Bezárás gomb:** Egy jól látható gomb, amely a jobb felső sarokban helyezkedik el, és a [closeFunction](#) segítségével zárja be az ablakot.
-

Felhasználói élmény

A [FooterInfoUse.jsx](#) biztosítja, hogy a felhasználók könnyen hozzáférjenek a felhasználási feltételekhez, és egyszerűen bezárhassák a modális ablakot, ha már nincs rá szükségük. A gomb elhelyezése és stílusa intuitív, így a felhasználók számára könnyen kezelhető.

Összegzés

A [FooterInfoUse.jsx](#) egy egyszerű, de hatékony komponens, amely a felhasználási feltételek megjelenítésére szolgál. A modális ablak kialakítása és a bezárási funkció biztosítja a könnyű használatot és a felhasználói élményt.



TermOfUse.jsx – Felhasználási feltételek

A [TermOfUse.jsx](#) egy tartalmi komponens, amely a FlexiStore felhasználási feltételeit jeleníti meg. Ez a komponens a jogi információk strukturált és könnyen olvasható formában történő megjelenítésére szolgál, és általában egy modális ablakban (például a FooterInfoUse.jsx által) kerül megjelenítésre.

Főbb funkciók

1. Felhasználási feltételek bemutatása

A komponens tartalmazza a FlexiStore szolgáltatás használatára vonatkozó legfontosabb jogi és működési információkat. Ezek a következő szekciókra vannak bontva:

- **Általános rendelkezések:** A szolgáltatás használatának alapvető feltételei.
- **A szolgáltatás célja:** A FlexiStore működésének és céljának bemutatása.
- **Használat és jogosultság:** A szolgáltatás igénybevételének feltételei.
- **Felelősség kizárása:** A tesztpalform használatával kapcsolatos felelősségi nyilatkozat.
- **Módosítás:** A feltételek változtatásának lehetősége és értesítési módja.

2. Strukturált tartalom

A komponens jól strukturált, címsorokkal és listákkal tagolt tartalmat jelenít meg, amely megkönnyíti az információk áttekintését.



3. Dátum és hatályosság

A feltételek hatályba lépésének dátuma (2025.04.08.) kiemelten jelenik meg, hogy a felhasználók tisztában legyenek az aktuális verzió érvényességével.

Felépítés

- Címsorok:** A h1 és h3 elemek biztosítják a tartalom logikai tagolását.
 - Szöveges tartalom:** A p elemek tartalmazzák a részletes információkat.
 - Listák:** A li elemek felsorolásként jelenítik meg a legfontosabb pontokat.
-

Felhasználói élmény

A TermOfUse.jsx biztosítja, hogy a felhasználók könnyen hozzáférjenek a jogi információkhöz, és azokat átlátható formában olvashassák. A jól strukturált tartalom és a vizuális kiemelések (pl. címsorok, színek) segítik az információk gyors megértését.

Tartalom szekciók

1. Általános rendelkezések

- A FlexiStore szolgáltatás használatával a felhasználók elfogadják a feltételeket.
- A feltételek ideiglenesek, és a szolgáltatás indulásával frissülhetnek.

2. A szolgáltatás célja

- Automatizált eszközökön keresztül történő termékbérlet biztosítása.
- A végleges működéshez fizikai automaták és digitális platform szükséges.

3. Használat és jogosultság

- A rendszer jelenleg tesztverzióban fut, tényleges bérlet nem történik.
- A jövőben regisztráció és személyes adatok megadása szükséges lehet.



4. Felelősség kizárása

- A tesztpalform információi kizárolag tájékoztató jellegűek.
- A FlexiStore nem vállal felelősséget a tesztpalform használatából eredő károkért.

5. Módosítás

- A feltételek bármikor módosíthatók, és a változásokról értesítést tesznek közzé.
-

Összegzés

A TermOfUse.jsx egy jól strukturált, informatív komponens, amely a FlexiStore felhasználási feltételeit tartalmazza. Ez a komponens biztosítja, hogy a felhasználók tisztában legyenek a szolgáltatás használatának feltételeivel, és könnyen hozzáférjenek a jogi információhoz.



FooterPrivacyInfo.jsx – Adatvédelmi irányelvek modális ablak

1. A [FooterPrivacyInfo.jsx](#) egy egyszerű komponens, amely az adatvédelmi irányelvez megjelenítésére szolgál. Ez a komponens egy modális ablakot biztosít, amelyben a felhasználók elolvashatják az adatvédelmi irányelvezet, és egy gomb segítségével bezárhatják az ablakot.
-

Főbb funkciók

2. **Adatvédelmi irányelvez megjelenítése**

A [FooterPrivacyInfo.jsx](#) a [PrivacyPolicy](#) nevű alkomonenst használja, amely tartalmazza az adatvédelmi irányelvez szövegét. Ez biztosítja, hogy a jogi információk jól strukturáltan és könnyen olvashatóan jelenjenek meg.

3. **Modális ablak**

A komponens egy modális ablakot jelenít meg, amely a következőket tartalmazza:

- **Tartalom:** A [PrivacyPolicy](#) komponens által biztosított szöveg.
- **Bezárás gomb:** Egy jól látható "X" gomb, amely lehetővé teszi a felhasználók számára a modális ablak bezárását.

4. **Bezárási funkció**

A [closeFunction](#) prop segítségével a komponens lehetővé teszi a modális ablak bezárását. Ez a funkció rugalmasan kezelhető, és a szülő komponens biztosítja a megfelelő állapotkezelést.



Felépítés

- Modális ablak:** A modal és modal-box osztályok biztosítják a modális ablak megjelenését és stílusát.
 - Tartalom:** A [PrivacyPolicy](#) komponens jeleníti meg az adatvédelmi irányelvek szövegét.
 - Bezárás gomb:** Egy jól látható gomb, amely a jobb felső sarokban helyezkedik el, és a [closeFunction](#) segítségével zárja be az ablakot.
-

Felhasználói élmény

A [FooterPrivacyInfo.jsx](#) biztosítja, hogy a felhasználók könnyen hozzáférjenek az adatvédelmi irányelvekhez, és azokat átlátható formában olvashassák. A jól elhelyezett bezárás gomb intuitív használatot tesz lehetővé.

Összegzés

A [FooterPrivacyInfo.jsx](#) egy egyszerű, de hatékony komponens, amely az adatvédelmi irányelvek megjelenítésére szolgál. A modális ablak kialakítása és a bezárási funkció biztosítja a könnyű használatot és a felhasználói élményt.



PrivacyPolicy.jsx – Adatvédelmi irányelvek

A [PrivacyPolicy.jsx](#) egy tartalmi komponens, amely a FlexiStore adatvédelmi irányelvezet jeleníti meg. Ez a komponens részletesen bemutatja, hogy a szolgáltatás hogyan kezeli a felhasználói adatokat, milyen adatokat gyűjt, és milyen jogokat biztosít a felhasználók számára. Az adatvédelmi irányelvek jelenleg a tesztelési fázisra vonatkoznak.

Főbb funkciók

1. Adatvédelmi irányelvek bemutatása

A komponens részletesen ismerteti az adatvédelmi irányelvezet, amelyek a következő szekciókra vannak bontva:

- **Milyen adatokat gyűjtünk?**
- **Mire használjuk az adatokat?**
- **Adatok megosztása**
- **Adatbiztonság**
- **Felhasználói jogok**
- **Tesztelési fázis sajátosságai**

2. Strukturált tartalom

A komponens jól strukturált, címsorokkal és listákkal tagolt tartalmat jelenít meg, amely megkönyíti az információk áttekintését.

3. Dátum és hatályosság

Az adatvédelmi irányelvezet hatályba lépésének dátuma (2025.04.08.) kiemelten jelenik meg, hogy a felhasználók tisztában legyenek az aktuális verzió érvényességevel.



Tartalom szekciók

1. Milyen adatokat gyűjtünk?

- Jelenleg a tesztoldalon keresztül nem gyűjtenek személyes adatokat.
- A végleges szolgáltatás során gyűjthető adatok:
 - Név, e-mail cím, telefonszám
 - Helyadatok (pl. automata elhelyezkedése)
 - Bérleti előzmények
 - Fizetési adatok (biztonságos fizetési szolgáltatón keresztül)

2. Mire használjuk az adatokat?

- A bérleti folyamat biztosítására.
- Ügyfélszolgálat és kapcsolattartás céljára.
- A szolgáltatás fejlesztésére.

3. Adatok megosztása

- Az adatokat harmadik félnek nem adják el.
- Megosztás kizárálag az alábbi esetekben történhet:
 - Törvény által előírt esetekben.
 - Technikai partnerekkel, akik segítenek a szolgáltatás működtetésében.

4. Adatbiztonság

- Az adatvédelmet komolyan veszik, és minden ésszerű technikai intézkedést megtesznek az adatok biztonságának érdekében.
- A FlexiStore nem vállal felelősséget a tesztplatform használatából eredő károkért.

5. Felhasználói jogok

- A jövőben a felhasználók számára biztosított jogok:
 - Adatok módosítása vagy törlése.
 - Tájékoztatás kérése az általuk tárolt adatokról.
 - Hozzájárulás visszavonása.

6. Tesztelési fázis sajátosságai

- A jelenlegi verzió tesztelési fázisban van, így:
 - Egyes felhasználói adatok technikai okokból az adatbázisba kerülhetnek, de nem kerülnek feldolgozásra vagy elemzésre.
 - A „kosár” funkció csak teszt célokat szolgál, és a fizetési lehetőségek mögött nincs valós tranzakció.



Felhasználói élmény

A [PrivacyPolicy.jsx](#) biztosítja, hogy a felhasználók könnyen hozzáférjenek az adatvédelmi irányelvekhez, és azokat átlátható formában olvashassák. A jól strukturált tartalom és a vizuális kiemelések (pl. címsorok, színek) segítik az információk gyors megértését.

Összegzés

A [PrivacyPolicy.jsx](#) egy jól strukturált, informatív komponens, amely a FlexiStore adatvédelmi irányelvezetést tartalmazza. Ez a komponens biztosítja, hogy a felhasználók tisztában legyenek azzal, hogyan kezelik az adataikat, és milyen jogokat gyakorolhatnak a szolgáltatás használata során.



MobileTableInfo.jsx – Mobil nézet jogi információk és márkaelemek

A [MobileTableInfo.jsx](#) egy dedikált komponens, amely a mobil nézetben jeleníti meg a jogi információkat, a FlexiStore márkaelemeit, valamint a közösségi média ikonokat. Ez a komponens biztosítja, hogy a mobil felhasználók könnyen hozzáférjenek a jogi információkhoz és a márkalával kapcsolatos tartalmakhoz.

Főbb funkciók

1. Jogi információk megjelenítése

- A komponens tartalmaz linkeket a [Felhasználási feltételek](#) és az [Adatvédelmi irányelvek](#) megnyitásához.
- A jogi információk modális ablakban jelennek meg a [FooterInfoUse](#) és [FooterPrivacyInfo](#) komponensek segítségével.

2. Márkaelemek megjelenítése

- A FlexiStore logója és a "Minden jog fenntartva © 2025" szöveg vizuálisan kiemelve jelenik meg.
- A márkaelemek központi helyet foglalnak el, hogy erősíték a vizuális azonosítást.

3. Közösségi média ikonok

- A komponens tartalmaz linkeket az Instagram és Facebook oldalakhoz, amelyek ikonok formájában jelennek meg.
- Az ikonok modern, minimalista stílusú SVG elemek.



4. Állapotkezelés

- A useState hook segítségével kezeli a jogi információk modális ablakainak nyitott/zárt állapotát:
 - isTermInfo: A felhasználási feltételek modális ablakának állapota.
 - isPolicyInfo: Az adatvédelmi irányelvezek modális ablakának állapota.
-

Felépítés

1. Jogi információk szekció

- A "Jogi információk" címsor alatt két link található:
 - **Felhasználási feltételek:** A FooterInfoUse komponens segítségével jelenik meg.
 - **Adatvédelmi irányelvezek:** A FooterPrivacyInfo komponens segítségével jelenik meg.

2. Márkaelemek szekció

- A FlexiStore logója SVG formátumban jelenik meg.
- A márka neve és a szerzői jogi információk szöveges formában láthatók.

3. Közösségi média ikonok szekció

- Az Instagram és Facebook ikonok SVG formátumban jelennek meg, és linkként funkcionálnak.
-

Felhasználói élmény

A MobileTableInfo.jsx biztosítja, hogy a mobil felhasználók könnyen hozzáférjenek a jogi információkhoz és a márkaival kapcsolatos tartalmakhoz. A jól strukturált elrendezés és a vizuális elemek segítik a gyors navigációt és az információk könnyű elérését.



Összegzés

A [MobileTableInfo.jsx](#) egy reszponzív komponens, amely a mobil nézetben jeleníti meg a FlexiStore jogi információit, márkaelemeket és közösségi média ikonokat. Ez a komponens fontos szerepet játszik a mobil felhasználói élmény javításában, és biztosítja a jogi és márkainformációk könnyű elérhetőségét.



secureStorage.jsx – Biztonságos adattárolás

A `secureStorage.jsx` egy segédfájl, amely a böngésző `sessionStorage` API-ját használja az adatok biztonságos tárolására és visszakeresésére. Az adatok titkosítását a `CryptoJS` könyvtár AES algoritmusa biztosítja, így az érzékeny információk védettek maradnak a kliens oldalon.

Főbb funkciók

1. Adatok titkosított tárolása (`setItem`)

- Az `setItem` metódus lehetővé teszi az adatok titkosított formában történő tárolását a `sessionStorage`-ban.
- Az adatokat JSON formátumra alakítja, majd AES algoritmussal titkosítja a megadott `secretKey` segítségével.

2. Adatok visszafejtése és lekérdezése (`getItem`)

- A `getItem` metódus visszafejtja a `sessionStorage`-ban tárolt titkosított adatokat.
- Ha az adat nem található, vagy a visszafejtés sikertelen, null értéket ad vissza.

3. Adatok törlése (`removeItem`)

- A `removeItem` metódus eltávolítja a megadott kulcsnak tartozó adatot a `sessionStorage`-ból.

Felépítés

1. Titkosítási kules

- A `secretKey` változó tartalmazza az AES titkosításhoz használt kulcsot. Ez a kulcs biztosítja az adatok titkosítását és visszafejtését.

2. Titkosítás és visszafejtés

- A `CryptoJS.AES.encrypt` metódus titkosítja az adatokat.
- A `CryptoJS.AES.decrypt` metódus visszafejtja a titkosított adatokat, majd UTF-8 formátumra alakítja.

3. Hibakezelés

- A `getItem` metódus hibakezelést tartalmaz, amely biztosítja, hogy a visszafejtési hibák ne okozzanak alkalmazásösszeomlást. A hibák a konzolra kerülnek kiírásra.



Használati példák

- **Adatok tárolása:**

```
secureStorage.setItem('user', { id: 1, name: 'John Doe' });
```

Az user kulcshoz tartozó adat titkosított formában kerül tárolásra.

- **Adatok lekérdezése:**

```
const user = secureStorage.getItem('user');
```

```
console.log(user); // { id: 1, name: 'John Doe' }
```

```
secureStorage.removeItem('user');
```

Az user kulcshoz tartozó adat törlésre kerül.

Felhasználói élmény és biztonság

- **Biztonság:** Az AES titkosítás biztosítja, hogy az érzékeny adatok (pl. felhasználói információk, profiladatok) védettek legyenek a kliens oldalon.
- **Egyszerű használat:** Az API egyszerű metódusokat biztosít az adatok tárolására, lekérdezésére és törlésére.
- **Hibakezelés:** A [getItem](#) metódus hibakezelése biztosítja, hogy a titkosítási vagy visszafejtési hibák ne akadályozzák az alkalmazás működését.

Összegzés

A [secureStorage.jsx](#) egy hatékony és biztonságos megoldás az érzékeny adatok kliens oldali tárolására. Az AES titkosítás és a könnyen használható API biztosítja, hogy az adatok védettek legyenek, miközben az alkalmazás fejlesztői számára egyszerű és intuitív megoldást kínál.



.env – Környezeti változók

A `.env` fájl a FlexiStore alkalmazás környezeti változóit tartalmazza, amelyek a fejlesztési és éles környezetek konfigurációját határozzák meg. Ez a fájl a Vite által támogatott formátumban van megadva, és biztosítja, hogy az alkalmazás különböző környezetekben megfelelően működjön.

Főbb funkciók

1. Fejlesztési környezet változói

- **VITE_BASE_URL:** Az API végpont alap URL-je a fejlesztési környezetben.
Példa: <http://localhost:8000/api>
- **VITE_LARAVEL_IMAGE_URL:** A Laravel által kiszolgált képek alap URL-je a fejlesztési környezetben.
Példa: <http://localhost:8000>

2. Éles környezet változói (kommentelve)

- **VITE_BASE_URL:** Az API végpont alap URL-je az éles környezetben.
Példa: <https://flexistore.hu/api>
 - **VITE_LARAVEL_IMAGE_URL:** A Laravel által kiszolgált képek alap URL-je az éles környezetben.
Példa: <https://flexistore.hu>
-

Felépítés

- **Fejlesztési környezet:** Az alapértelmezett változók a fejlesztési környezethez vannak beállítva, így az alkalmazás helyben futtatható.
 - **Éles környezet:** Az éles környezethez tartozó változók kommentelve vannak, és szükség esetén aktiválhatók.
-



Használat

1. Fejlesztési környezetben

A Vite automatikusan betölti a `.env` fájlban megadott változókat, és elérhetővé teszi azokat a `import.meta.env` objektumon keresztül.

Példa:

```
const apiUrl = import.meta.env.VITE_BASE_URL;  
  
console.log(apiUrl); // http://localhost:8000/api
```

2. Éles környezetben

Az éles környezethez tartozó változókat a `.env` fájlban kell aktiválni (a kommentek eltávolításával), vagy külön `.env.production` fájlban kell megadni.

Biztonsági megjegyzések

- Érzékeny adatok:** A `.env` fájl nem tartalmazhat érzékeny adatokat (pl. API kulcsokat, jelszavakat), mivel ez a fájl véletlenül bekerülhet a verziókezelő rendszerbe.
 - Git ignorálás:** A `.env` fájlt általában hozzáadják a `.gitignore` fájlhoz, hogy ne kerüljön be a verziókezelésbe.
-

Összegzés

A `.env` fájl kulcsszerepet játszik az alkalmazás konfigurációjában, lehetővé téve a fejlesztési és éles környezetek közötti egyszerű váltást. A változók használata biztosítja, hogy az alkalmazás rugalmasan alkalmazkodjon a különböző környezeti követelményekhez.



NewRole.jsx – Új jogosultság létrehozása és módosítása

A NewRole.jsx egy olyan komponens, amely lehetővé teszi új jogosultságok létrehozását, illetve meglévő jogosultságok módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új jogosultság hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új jogosultság létrehozása

- Ha a komponens nem kap adatokat (state értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új jogosultságot.

2. Meglévő jogosultság módosítása

- Ha a komponens kap adatokat (state tartalmazza a jogosultság adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa PATCH, és a módosítások mentése után a felhasználó visszairányításra kerül a jogosultságok listájára.

3. Backend kommunikáció

- A backendMuveletRole függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (VITE_BASE_URL) alapján.

4. Űrlapkezelés

- Az űrlapmezők értékei a formData állapotban tárolódnak.
- A writeData függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (cím) dinamikusan változik a művelet típusától függően:
 - Új jogosultság esetén: "Új Jogosultság felvitele".
 - Módosítás esetén: "[Jogosultság neve] módosítása".

2. Űrlapmezők

- Érték (**power**): Szám típusú mező, amely a jogosultság szintjét határozza meg.
- **Jogosultság neve (warrant_name)**: Szöveg típusú mező, amely a jogosultság nevét tartalmazza.

3. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuveletRole függvényt az adatok mentésére.

4. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a jogosultságok listájára (/roles).

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- **Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új jogosultság vagy módosítás).
- **Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a jogosultságok listájára, ahol ellenőrizheti a változásokat.

Összegzés

A NewRole.jsx egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új jogosultságok létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



RolesList.jsx – Jogosultságok listázása

A [RolesList.jsx](#) egy olyan komponens, amely az adminisztrációs felületen belül a jogosultságok listázására szolgál. Ez a komponens a [RoleCard](#) alkomponens segítségével jeleníti meg az egyes jogosultságokat, és a [AuthContext](#)-ból nyeri ki a szükséges adatokat.

Főbb funkciók

1. Jogosultságok listázása

- A [RolesList.jsx](#) a `roles` adatokat a [AuthContext](#)-ból nyeri ki, amely tartalmazza az összes jogosultságot.
- Az adatok megjelenítése dinamikusan történik, a `roles.map()` metódus segítségével.

2. Jogosultságok megjelenítése kártyák formájában

- Az egyes jogosultságokat a [RoleCard](#) komponens jeleníti meg, amely a `role` objektumot kapja propként.
- A kártyák reszponzív elrendezésben jelennek meg, így különböző képernyőméretekben is jól használhatók.

3. Felhasználói élmény

- A komponens egyértelmű címet jelenít meg: "Jogosultságok listája".
- A kártyák vizuálisan elkülönülnek, így a felhasználók könnyen áttekinthetik a jogosultságokat.

Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Jogosultságok listája".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.



3. Jogosultságok megjelenítése

- A roles tömb elemei a RoleCard komponens segítségével kerülnek megjelenésre.
- A key attribútum az egyedi role.id értéket használja, hogy biztosítsa a React hatékony renderelését.

4. Elrendezés

- A kártyák egy rugalmas (flex) elrendezésben jelennek meg, amely automatikusan alkalmazkodik a képernyőmérethez.
- A flex-wrap osztály biztosítja, hogy a kártyák több sorban is megjelenhessenek, ha szükséges.

Felhasználói élmény

- **Átláthatóság:** A jogosultságok jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
- **Reszponzivitás:** A kártyák elrendezése rugalmas, így a komponens különböző eszközökön is megfelelően működik.
- **Egyszerű használat:** A RolesList.jsx egyértelműen és vizuálisan vonzó módon jeleníti meg a jogosultságokat.

Összegzés

A RolesList.jsx egy hatékony komponens, amely az adminisztrátorok számára biztosítja a jogosultságok listázását. A RoleCard alkomponens használata és a reszponzív elrendezés biztosítja a könnyű használatot és a vizuális érthetőséget.



RoleCard.jsx – Jogosultság kártya

A `RoleCard.jsx` egy olyan komponens, amely egyetlen jogosultságot jelenít meg kártya formájában. Ez a komponens az adminisztrátorok számára biztosítja a jogosultságok megtekintését, módosítását és törlését. A kártya vizuálisan elkülönül, és rezponszív kialakítású, így különböző eszközökön is jól használható.

Főbb funkciók

1. Jogosultság megjelenítése

- A kártya tartalmazza a jogosultság nevét (`role.warrant_name`), amely a kártya címeként jelenik meg.

2. Jogosultság módosítása

- A `modosit` függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a `NewRole` komponenshez navigál, ahol a kiválasztott jogosultság adatai előre kitöltve jelennek meg.

3. Jogosultság törlése

- A `torles` függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a `backendMuveletRole` függvényt hívja meg, amely egy DELETE kérést küld a backend API-nak a jogosultság törlésére.

4. Felhasználói jogosultságok ellenőrzése

- A komponens csak akkor jeleníti meg a "Módosítás" és "Törlés" gombokat, ha a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik (`user.isAdmin >= 70`).

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Cím:** A jogosultság neve (`role.warrant_name`).
 - **Gombok:** "Módosítás" és "Törlés" gombok, amelyek az adminisztrátorok számára érhetők el.



2. Backend kommunikáció

- A `torles` függvény a `backendMuveletRole` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a jogosultság azonosítóját (`role.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

3. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a NewRole komponenshez, és átadja a kiválasztott jogosultság adatait.
-

Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a jogosultságok könnyen áttekinthetők.
 - **Egyszerű használat:** A "Módosítás" és "Törlés" gombok egyértelműen jelzik a lehetséges műveleteket.
 - **Biztonság:** A gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a jogosultságok kezelése biztonságos.
-

Összegzés

A `RoleCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi az adminisztrátorok számára a jogosultságok kezelését. A vizuális megjelenés és a backend kommunikáció integrációja biztosítja a könnyű használatot és a hatékony adminisztrációt.



NewCategory.jsx – Új kategória létrehozása és módosítása

A `NewCategory.jsx` egy olyan komponens, amely lehetővé teszi új kategóriák létrehozását, illetve meglévő kategóriák módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új kategória hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új kategória létrehozása

- Ha a komponens nem kap adatokat (`state` értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új kategóriát.

2. Meglévő kategória módosítása

- Ha a komponens kap adatokat (`state` tartalmazza a kategória adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa PATCH, és a módosítások mentése után a felhasználó visszairányításra kerül a kategóriák listájára.

3. Backend kommunikáció

- A `backendMuvelet` függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.

4. Űrlapkezelés

- Az űrlapmezők értékei a `formData` állapotban tárolódnak.
- A `writeData` függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (cím) dinamikusan változik a művelet típusától függően:
 - Új kategória esetén: "Új Kategória Felvitele".
 - Módosítás esetén: "[Kategória neve] módosítása".

2. Űrlapmezők

- **Kategória neve (name):** Szöveg típusú mező, amely a kategória nevét tartalmazza.

3. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuvelet függvényt az adatok mentésére.

4. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a kategóriák listájára (/categories).

Felhasználói élmény

- **Egyeszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- **Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új kategória vagy módosítás).
- **Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a kategóriák listájára, ahol ellenőrizheti a változásokat.

Összegzés

A NewCategory.jsx egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új kategóriák létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



CategoriesList.jsx – Kategóriák listázása

A [CategoriesList.jsx](#) egy olyan komponens, amely az adminisztrációs felületen belül a kategóriák listázására szolgál. Ez a komponens a [CategoriesCard](#) alkomponens segítségével jeleníti meg az egyes kategóriákat, és az [InitialContext](#)-ből nyeri ki a szükséges adatokat.

Főbb funkciók

1. Kategóriák listázása

- A [CategoriesList.jsx](#) a [categories](#) adatokat az [InitialContext](#)-ből nyeri ki, amely tartalmazza az összes kategóriát.
- Az adatok megjelenítése dinamikusan történik a [categories.map\(\)](#) metódus segítségével.

2. Kategóriák megjelenítése kártyák formájában

- Az egyes kategóriákat a [CategoriesCard](#) komponens jeleníti meg, amely a [category](#) objektumot kapja propként.
- A kártyák reszponzív elrendezésben jelennek meg, így különböző képernyőméretekben is jól használhatók.

3. Felhasználói élmény

- A komponens egyértelmű címet jelenít meg: "Kategória Lista".
 - A kártyák vizuálisan elkülönülnek, így a felhasználók könnyen áttekinthetik a kategóriákat.
-

Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Kategória Lista".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.



2. Kategóriák megjelenítése

- A `categories` tömb elemei a `CategoriesCard` komponens segítségével kerülnek megjelenítésre.
- A `key` attribútum az egyedi `category.id` értéket használja, hogy biztosítsa a React hatékony renderelését.

3. Elrendezés

- A kártyák egy rugalmas (flex) elrendezésben jelennek meg, amely automatikusan alkalmazkodik a képernyőmérethez.
- A flex-wrap osztály biztosítja, hogy a kártyák több sorban is megjelenhessenek, ha szükséges.

Felhasználói élmény

- **Átláthatóság:** A kategóriák jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
- **Reszponzivitás:** A kártyák elrendezése rugalmas, így a komponens különböző eszközökön is megfelelően működik.
- **Egyszerű használat:** A `CategoriesList.jsx` egyértelműen és vizuálisan vonzó módon jeleníti meg a kategóriákat.

Összegzés

A `CategoriesList.jsx` egy hatékony komponens, amely az adminisztrátorok számára biztosítja a kategóriák listázását. A `CategoriesCard` alkomponens használata és a reszponzív elrendezés biztosítja a könnyű használatot és a vizuális érthetőséget.



CategoriesCard.jsx – Kategória kártya

A `CategoriesCard.jsx` egy olyan komponens, amely egyetlen kategóriát jelenít meg kártya formájában. Ez a komponens az adminisztrátorok számára biztosítja a kategóriák megtekintését, módosítását és törlését. A kártya vizuálisan elkülönül, és reszponzív kialakítású, így különböző eszközökön is jól használható.

Főbb funkciók

1. Kategória megjelenítése

- A kártya tartalmazza a kategória nevét (`category.name`), amely a kártya címeként jelenik meg.

2. Kategória módosítása

- A `modosit` függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a `NewCategory` komponenshez navigál, ahol a kiválasztott kategória adatai előre kitöltve jelennek meg.

3. Kategória törlése

- A `torles` függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a `backendMuvelet` függvényt hívja meg, amely egy DELETE kérést küld a backend API-nak a kategória törlésére.

4. Felhasználói jogosultságok ellenőrzése

- A komponens csak akkor jeleníti meg a "Módosítás" és "Törlés" gombokat, ha a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik (`user.isAdmin >= 70`).

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Cím:** A kategória neve (`category.name`).
 - **Gombok:** "Módosítás" és "Törlés" gombok, amelyek az adminisztrátorok számára érhetők el.



2. Backend kommunikáció

- A `torles` függvény a `backendMuvelet` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a kategória azonosítóját (`category.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

3. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a `NewCategory` komponenshez, és átadja a kiválasztott kategória adatait.

Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a kategóriák könnyen áttekinthetők.
- **Egyszerű használat:** A "Módosítás" és "Törlés" gombok egyértelműen jelzik a lehetséges műveleteket.
- **Biztonság:** A gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a kategóriák kezelése biztonságos.

Összegzés

A `CategoriesCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi az adminisztrátorok számára a kategóriák kezelését. A vizuális megjelenés és a backend kommunikáció integrációja biztosítja a könnyű használatot és a hatékony adminisztrációt.



NewLocker.jsx – Új csomagautomata létrehozása és módosítása

A `NewLocker.jsx` egy olyan komponens, amely lehetővé teszi új csomagautomaták létrehozását, illetve meglévő csomagautomaták módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új csomagautomata hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új csomagautomata létrehozása

- Ha a komponens nem kap adatokat (`state` értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új csomagautomatát.

2. Meglévő csomagautomata módosítása

- Ha a komponens kap adatokat (`state` tartalmazza a csomagautomata adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa PATCH, és a módosítások mentése után a felhasználó visszairányításra kerül a csomagautomaták listájára.

3. Backend kommunikáció

- A `backendMuvelet` függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.

4. Űrlapkezelés

- Az űrlapmezők értékei a `formData` állapotban tárolódnak.
 - A `writeData` függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.
-



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (**cim**) dinamikusan változik a művelet típusától függően:
 - Új csomagautomata esetén: "Új Csomagautomata Felvitele".
 - Módosítás esetén: "[Csomagautomata neve] módosítása".

2. Űrlapmezők

- **Csomagautomata neve (locker_name)**: Szöveg típusú mező, amely a csomagautomata nevét tartalmazza.
- **Cím (address)**: Szöveg típusú mező, amely a csomagautomata címét tartalmazza.
- **Leírás (description)**: Szövegmező, amely a csomagautomata részletes leírását tartalmazza.

3. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuvelet függvényt az adatok mentésére.

4. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a csomagautomaták listájára (/lockers).

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
 - **Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új csomagautomata vagy módosítás).
 - **Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a csomagautomaták listájára, ahol ellenőrizheti a változásokat.
-



Összegzés

A `NewLocker.jsx` egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új csomagautomaták létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



4. ábra New Locker Desktop



LockersList.jsx – Csomagautomaták listázása

A `LockersList.jsx` egy olyan komponens, amely a csomagautomaták listázását, keresését, szűrését és lapozását valósítja meg. Ez a komponens reszponzív kialakítású, így különböző eszközökön is jól használható, és biztosítja a felhasználók számára a csomagautomaták könnyű áttekintését.

Főbb funkciók

1. Csomagautomaták listázása

- A `LockersList.jsx` a `lockers` adatokat a `ServiceContext`-ből nyeri ki, amely tartalmazza az összes csomagautomatát.
- Az adatok megjelenítése dinamikusan történik a `paginatedLockers` tömb segítségével, amely a lapozás alapján szűrt adatokat tartalmazza.

2. Keresés és szűrés

- A felhasználók kereshetnek a csomagautomaták neve (`locker_name`) vagy címe (`address`) alapján.
- A keresési eredmények automatikusan frissülnek a `searchQuery` állapot változásakor.

3. Lapozás

- A komponens támogatja a lapozást, amely lehetővé teszi a felhasználók számára, hogy egyszerre csak egy adott számú csomagautomatát lássanak.
- A lapozás gombjai (`handlePrevPage`, `handleNextPage`) biztosítják az oldalak közötti navigációt.

4. Reszponzív szűrőpanel

- Mobil nézetben egy beúszó szűrőpanel érhető el, amely a "Hamburger Menü" ikonra kattintva nyitható meg.
 - Asztali nézetben a szűrőpanel a bal oldalon jelenik meg.
-



Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Összes Csomagautomata".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.

2. Keresés és szűrők

- A keresési mező lehetővé teszi a felhasználók számára, hogy gyorsan megtalálják a kívánt csomagautomatát.
- A szűrőpanel mobilon beúszó, asztali nézetben pedig fixen a bal oldalon helyezkedik el.

3. Csomagautomata kártyák

- A csomagautomaták a LockersCard komponens segítségével jelennek meg, amely a locker objektumot kapja propként.
- A kártyák reszponzív rácsos elrendezésben jelennek meg.

4. Lapozás

- A lapozás gombjai a lista alján találhatók, és a felhasználók számára vizuális visszajelzést adnak az aktuális oldalról és az összes oldal számáról.

Felhasználói élmény

- **Átláthatóság:** A csomagautomaták jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
 - **Keresés és szűrés:** A keresési funkció és a szűrőpanel biztosítja, hogy a felhasználók gyorsan megtalálják a kívánt csomagautomatát.
 - **Reszponzivitás:** A komponens mobilon és asztali nézetben is jól használható, a szűrőpanel és a kártyák elrendezése automatikusan alkalmazkodik a képernyőmérethez.
 - **Lapozás:** A lapozási funkció biztosítja, hogy a lista kezelhető maradjon nagy mennyiségű adat esetén is.
-



Összegzés

A `LockersList.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi a csomagautomaták listázását, keresését és szűrését. A vizuális megjelenés, a keresési funkció és a lapozás biztosítja a könnyű használatot és a pozitív felhasználói élményt.



LockersCard.jsx – Csomagautomata kártya

A LockersCard.jsx egy olyan komponens, amely egyetlen csomagautomatát jelenít meg kártya formájában. Ez a komponens az adminisztrátorok számára biztosítja a csomagautomaták megtekintését, módosítását és törlését, valamint a felhasználók számára az elérhető termékek és további információk megtekintését.

Főbb funkciók

1. Csomagautomata megjelenítése

- A kártya tartalmazza a csomagautomata nevét (locker.locker_name) és címét (locker.address).
- Egy alapértelmezett kép (LockerKep) jelenik meg a kártya tetején.

2. Csomagautomata módosítása

- A modosit függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a NewLocker komponenshez navigál, ahol a kiválasztott csomagautomata adatai előre kitöltve jelennek meg.

3. Csomagautomata törlése

- A torles függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a backendMuvelet függvényt hívja meg, amely egy DELETE kérést küld a backend API-nak a csomagautomata törlésére.

4. További információk megjelenítése

- Az "Info" gomb megnyitja a LockerInfo alkókomponenst, amely részletes információkat jelenít meg a csomagautomatról.
- Az információs panel a isInfo állapot alapján jelenik meg vagy záródik be.

5. Elérhető termékek megtekintése

- Az "Elérhető Termékek" gomb lehetőséget biztosít a felhasználóknak a csomagautomatában elérhető termékek megtekintésére (a funkció implementációja nem látható a kódban).

6. Felhasználói jogosultságok ellenőrzése

- A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el (user?.isAdmin >= 70).



Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Kép:** Egy alapértelmezett kép (`LockerKep`) a csomagautomatról.
 - **Cím:** A csomagautomata neve és címe.
 - **Gombok:** "Módosítás", "Törlés", "Elérhető Termékek", és "Info" gombok.

2. Backend kommunikáció

- A `torles` függvény a `backendMuvelet` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a csomagautomata azonosítóját (`locker.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

3. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a NewLocker komponenshez, és átadja a kiválasztott csomagautomata adatait.

4. Információs panel

- A `LockerInfo` alkomponens jeleníti meg a csomagautomata részletes információit, amely a `isInfo` állapot alapján jelenik meg vagy záródik be.

Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a csomagautomaták könnyen áttekinthetők.
- **Egyszerű használat:** A gombok egyértelműen jelzik a lehetséges műveleteket, például módosítás, törlés vagy információk megtekintése.
- **Biztonság:** A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a csomagautomaták kezelése biztonságos.



Összegzés

A `LockersCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi a csomagautomaták kezelését és megtekintését. A vizuális megjelenés, a backend kommunikáció és a felhasználói jogosultságok ellenőrzése biztosítja a könnyű használatot és a hatékony adminisztrációt.



LockerInfo.jsx – Csomagautomata információs panel

A `LockerInfo.jsx` egy egyszerű komponens, amely egy modális ablakban jeleníti meg a csomagautomata részletes leírását. Ez a komponens a felhasználók számára biztosítja a csomagautomatával kapcsolatos további információk megtekintését.

Főbb funkciók

1. Csomagautomata leírásának megjelenítése

- A komponens a `locker.description` mezőt jeleníti meg, amely tartalmazza a csomagautomata részletes leírását.

2. Modális ablak

- A leírás egy modális ablakban jelenik meg, amely vizuálisan elkülönül a fő tartalomtól.
- A modális ablak tartalmaz egy címet, egy elválasztót (divider), és a leírást.

3. Bezárási funkció

- A modális ablak jobb felső sarkában található egy "X" gomb, amely a `closeFunction` prop segítségével zárja be az ablakot.
-

Felépítés

1. Modális ablak

- A modal és modal-box osztályok biztosítják a modális ablak megjelenését és stílusát.
- A cím (`h2`) kiemelt, középre igazított, és a csomagautomata leírására utal.

2. Leírás

- A `p` elem tartalmazza a csomagautomata részletes leírását, amely a `locker.description` mezőből származik.

3. Bezárás gomb

- A jobb felső sarokban található "X" gomb a `closeFunction` segítségével zárja be a modális ablakot.
-



Felhasználói élmény

- **Átláthatóság:** A modális ablak vizuálisan elkülönül, így a felhasználók könnyen azonosíthatják a csomagautomata leírását.
- **Egyszerű használat:** A "X" gomb intuitív módon biztosítja a modális ablak bezárását.
- **Reszponzivitás:** A modális ablak kialakítása biztosítja, hogy különböző eszközökön is megfelelően jelenjen meg.

Összegzés

A [LockerInfo.jsx](#) egy egyszerű, de hatékony komponens, amely lehetővé teszi a csomagautomaták részletes leírásának megjelenítését. A modális ablak kialakítása és a bezárási funkció biztosítja a könnyű használatot és a pozitív felhasználói élményt.



NewProduct.jsx – Új termék létrehozása és módosítása

A `NewProduct.jsx` egy olyan komponens, amely lehetővé teszi új termékek létrehozását, illetve meglévő termékek módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új termék hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új termék létrehozása

- Ha a komponens nem kap adatokat (`state` értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új terméket.

2. Meglévő termék módosítása

- Ha a komponens kap adatokat (`state` tartalmazza a termék adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa POST, és a módosítások mentése után a felhasználó visszairányításra kerül a termékek listájára.

3. Képkezelés

- A felhasználók képet tölhetnek fel a termékhez, amely a backend API-n keresztül kerül mentésre.
- Ha a termékhez már tartozik kép, akkor az elönézet megjelenik az űrlapon.

4. Backend kommunikáció

- A `backendMuveletFile` függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.

5. Űrlapkezelés

- Az űrlapmezők értékei a `formData` állapotban tárolódnak.
- A `writeData` függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (cím) dinamikusan változik a művelet típusától függően:
 - Új termék esetén: "Új termék felvitele".
 - Módosítás esetén: "[Termék neve] módosítása".

2. Űrlapmezők

- **Termék neve (name)**: Szöveg típusú mező, amely a termék nevét tartalmazza.
- **Leírás (description)**: Szövegmező, amely a termék részletes leírását tartalmazza.
- **Bérleti ár (price_per_day)**: Szám típusú mező, amely a termék napi bérleti árát tartalmazza.
- **Kategória (category_id)**: Legördülő lista, amely a termék kategóriáját határozza meg.
- **Csomagautomata (locker_ids)**: Többszörös választási lehetőséget biztosító lista, amely a termékhez kapcsolódó csomagautomatákat tartalmazza.
- **Elérhetőség (available)**: Legördülő lista, amely a termék elérhetőségét határozza meg.

3. Kép feltöltése

- A felhasználók képet tölthetnek fel a termékhez, amely a backend API-n keresztül kerül mentésre.

4. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuveletFile függvényt az adatok mentésére.

5. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a termékek listájára (/products).



Felhasználói élmény

- Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új termék vagy módosítás).
- Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a termékek listájára, ahol ellenőrizheti a változásokat.

Összegzés

A `NewProduct.jsx` egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új termékek létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés, a képkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



ProductsList.jsx – Termékek listázása

A `ProductsList.jsx` egy olyan komponens, amely a termékek listázását, keresését, szűrését és lapozását valósítja meg. Ez a komponens reszponzív kialakítású, így különböző eszközökön is jól használható, és biztosítja a felhasználók számára a termékek könnyű áttekintését.

Főbb funkciók

1. Termékek listázása

- A `ProductsList.jsx` a `products` adatokat az `InitialContext`-ből nyeri ki, amely tartalmazza az összes terméket.
- Az adatok megjelenítése dinamikusan történik a `paginatedProducts` tömb segítségével, amely a lapozás alapján szűrt adatokat tartalmazza.

2. Keresés és szűrés

- A felhasználók kereshetnek a termékek neve (`name`), napi bérleti ára (`price_per_day`), vagy elérhetősége (`available`) alapján.
- Szűrési lehetőségek:
 - **Csomagautomata:** A termékek szűrése a hozzájuk kapcsolt csomagautomaták alapján.
 - **Kategória:** A termékek szűrése kategóriák szerint.
 - **Ár (Ft / nap):** Minimum és maximum ár megadása.

3. Lapozás

- A komponens támogatja a lapozást, amely lehetővé teszi a felhasználók számára, hogy egyszerre csak egy adott számú terméket lássanak.
- A lapozás gombjai (`handlePrevPage`, `handleNextPage`) biztosítják az oldalak közötti navigációt.

4. Reszponzív szűrőpanel

- Mobil nézetben egy beúszó szűrőpanel érhető el, amely a "Hamburger Menü" ikonra kattintva nyitható meg.
- Asztali nézetben a szűrőpanel a bal oldalon jelenik meg.



Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Elérhető termékek".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.

2. Keresés és szűrők

- A keresési mező lehetővé teszi a felhasználók számára, hogy gyorsan megtalálják a kívánt terméket.
- A szűrőpanel mobilon beúszó, asztali nézetben pedig fixen a bal oldalon helyezkedik el.
- Szűrési lehetőségek:
 - **Csomagautomata:** A termékek szűrése a hozzájuk kapcsolt csomagautomaták alapján.
 - **Kategória:** A termékek szűrése kategóriák szerint.
 - **Ár (Ft / nap):** Minimum és maximum ár megadása csúszkák segítségével.

3. Termékkártyák

- A termékek a ProductsCard komponens segítségével jelennek meg, amely a product objektumot kapja propként.
- A kártyák reszponzív rácsos elrendezésben jelennek meg.

4. Lapozás

- A lapozás gombjai a lista alján találhatók, és a felhasználók számára vizuális visszajelzést adnak az aktuális oldalról és az összes oldal számáról.

Felhasználói élmény

- **Átláthatóság:** A termékek jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
- **Keresés és szűrés:** A keresési funkció és a szűrőpanel biztosítja, hogy a felhasználók gyorsan megtalálják a kívánt terméket.



- **Reszponzivitás:** A komponens mobilon és asztali nézetben is jól használható, a szűrőpanel és a kártyák elrendezése automatikusan alkalmazkodik a képernyőmérethez.
- **Lapozás:** A lapozási funkció biztosítja, hogy a lista kezelhető maradjon nagy mennyiségű adat esetén is.

Összegzés

A [ProductsList.jsx](#) egy hatékony és reszponzív komponens, amely lehetővé teszi a termékek listázását, keresését és szűrését. A vizuális megjelenés, a keresési funkció és a lapozás biztosítja a könnyű használatot és a pozitív felhasználói élményt.



ProductsCard.jsx – Termékkártya

A `ProductsCard.jsx` egy olyan komponens, amely egyetlen terméket jelenít meg kártya formájában. Ez a komponens lehetővé teszi a termékek megtekintését, kosárba helyezését, módosítását és törlését, valamint további információk megjelenítését.

Főbb funkciók

1. Termék megjelenítése

- A kártya tartalmazza a termék nevét (`product.name`), napi bérleti árat (`product.price_per_day`), és egy képet (`product.file_path`).
- A termékhez kapcsolódó csomagautomaták listája is megjelenik, amelyből a felhasználó kiválaszthatja a kívánt automatát.

2. Termék módosítása

- A `modosit` függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a `NewProduct` komponenshez navigál, ahol a kiválasztott termék adatai előre kitöltve jelennek meg.

3. Termék törlése

- A `törles` függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a `backendMuvelet` függvényt hívja meg, amely egy `DELETE` kérést küld a backend API-nak a termék törlésére.

4. Kosárba helyezés

- A "Kosárba" gomb lehetővé teszi a termék hozzáadását a kosárhoz a kiválasztott csomagautomatával együtt.
- A `addToCart` függvény a `CartContext`-ből származik, és kezeli a kosárba helyezést.

5. További információk megjelenítése

- Az "Info" gomb megnyitja a `ProductsInfo` alkkomponenst, amely részletes információkat jelenít meg a termékről.
- Az információs panel a `isInfo` állapot alapján jelenik meg vagy záródik be.



7. Felhasználói jogosultságok ellenőrzése

- A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el (`user?.isadmin >= 70`).
 - A "Kosárba" gomb csak bizonyos jogosultsági szinttel rendelkező felhasználók számára érhető el (`user?.isadmin >= 11`).
-

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Kép:** A termékhez tartozó kép (`product.file_path`).
 - **Cím:** A termék neve és napi bérírási ára.
 - **Csomagautomata választó:** Egy legördülő lista, amely lehetővé teszi a felhasználók számára a csomagautomata kiválasztását.
 - **Gombok:** "Módosítás", "Törlés", "Kosárba", és "Info" gombok.

2. Backend kommunikáció

- A `torles` függvény a `backendMuvelet` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a termék azonosítóját (`product.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

3. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a `NewProduct` komponenshez, és átadja a kiválasztott termék adatait.

4. Információs panel

- A `ProductsInfo` alkotmányos komponens jeleníti meg a termék részletes információit, amely a `isInfo` állapot alapján jelenik meg vagy záródik be.
-



Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a termékek könnyen áttekinthetők.
- **Egyszerű használat:** A gombok egyértelműen jelzik a lehetséges műveleteket, például módosítás, törlés, kosárba helyezés vagy információk megtekintése.
- **Biztonság:** A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a termékek kezelése biztonságos.

Összegzés

A `ProductsCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi a termékek kezelését és megtekintését. A vizuális megjelenés, a backend kommunikáció és a felhasználói jogosultságok ellenőrzése biztosítja a könnyű használatot és a hatékony adminisztrációt.



ProductsInfo.jsx – Termékinformációs panel

A `ProductsInfo.jsx` egy egyszerű komponens, amely egy modális ablakban jeleníti meg a termék részletes leírását. Ez a komponens a felhasználók számára biztosítja a termékkel kapcsolatos további információk megtekintését.

Főbb funkciók

1. Termék leírásának megjelenítése

- A komponens a `product.description` mezőt jeleníti meg, amely tartalmazza a termék részletes leírását.

2. Modális ablak

- A leírás egy modális ablakban jelenik meg, amely vizuálisan elkülönül a fő tartalomtól.
- A modális ablak tartalmaz egy címet, egy elválasztót (divider), és a leírást.

3. Bezárási funkció

- A modális ablak jobb felső sarkában található egy "X" gomb, amely a `closeFunction` prop segítségével zárja be az ablakot.
-

Felépítés

1. Modális ablak

- A modal és modal-box osztályok biztosítják a modális ablak megjelenését és stílusát.
- A cím (`h2`) kiemelt, középre igazított, és a termék leírására utal.

2. Leírás

- A `p` elem tartalmazza a termék részletes leírását, amely a `product.description` mezőből származik.

3. Bezárás gomb

- A jobb felső sarokban található "X" gomb a `closeFunction` segítségével zárja be a modális ablakot.
-



Felhasználói élmény

- **Átláthatóság:** A modális ablak vizuálisan elkülönül, így a felhasználók könnyen azonosíthatják a termék leírását.
- **Egyszerű használat:** A "X" gomb intuitív módon biztosítja a modális ablak bezárását.
- **Rezoncivitás:** A modális ablak kialakítása biztosítja, hogy különböző eszközökön is megfelelően jelenjen meg.

Összegzés

A `ProductsInfo.jsx` egy egyszerű, de hatékony komponens, amely lehetővé teszi a termékek részletes leírásának megjelenítését. A modális ablak kialakítása és a bezárási funkció biztosítja a könnyű használatot és a pozitív felhasználói élményt.



NewPaymentMethod.jsx – Új fizetési mód létrehozása és módosítása

A `NewPaymentMethod.jsx` egy olyan komponens, amely lehetővé teszi új fizetési módok létrehozását, illetve meglévő fizetési módok módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új fizetési mód hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új fizetési mód létrehozása

- Ha a komponens nem kap adatokat (`state` értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új fizetési módot.

2. Meglévő fizetési mód módosítása

- Ha a komponens kap adatokat (`state` tartalmazza a fizetési mód adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa PATCH, és a módosítások mentése után a felhasználó visszairányításra kerül a fizetési módok listájára.

3. Backend kommunikáció

- A `backendMuvelet` függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.

4. Űrlapkezelés

- Az űrlapmezők értékei a `formData` állapotban tárolódnak.
 - A `writeData` függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.
-



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (cim) dinamikusan változik a művelet típusától függően:
 - Új fizetési mód esetén: "Új Fizetési mód felvitele".
 - Módosítás esetén: "[Fizetési mód neve] módosítása".

2. Űrlapmezők

- **Fizetési mód neve (card_type):** Szöveg típusú mező, amely a fizetési mód nevét tartalmazza.

3. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuvelet függvényt az adatok mentésére.

4. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a fizetési módok listájára (/paymentmethods).

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- **Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új fizetési mód vagy módosítás).
- **Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a fizetési módok listájára, ahol ellenőrizheti a változásokat.

Összegzés

A NewPaymentMethod.jsx egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új fizetési módok létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



PaymentMethodList.jsx – Fizetési módok listázása

A [PaymentMethodList.jsx](#) egy olyan komponens, amely az adminisztrációs felületen belül a fizetési módok listázására szolgál. Ez a komponens a [PaymentMethodCard](#) alkomponens segítségével jeleníti meg az egyes fizetési módokat, és a [PaymentContext](#)-ből nyeri ki a szükséges adatokat.

Főbb funkciók

1. Fizetési módok listázása

- A [PaymentMethodList.jsx](#) a [payments](#) adatokat a [PaymentContext](#)-ből nyeri ki, amely tartalmazza az összes fizetési módot.
- Az adatok megjelenítése dinamikusan történik a [payments.map\(\)](#) metódus segítségével.

2. Fizetési módok megjelenítése kártyák formájában

- Az egyes fizetési módokat a [PaymentMethodCard](#) komponens jeleníti meg, amely a [payment](#) objektumot kapja propként.
- A kártyák reszponzív elrendezésben jelennek meg, így különböző képernyőméretekben is jól használhatók.

3. Felhasználói élmény

- A komponens egyértelmű címet jelenít meg: "Fizetési módok listája".
 - A kártyák vizuálisan elkülönülnek, így a felhasználók könnyen áttekinthetik a fizetési módokat.
-

Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Fizetési módok listája".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.



3. Fizetési módok megjelenítése

- A [payments](#) tömb elemei a [PaymentMethodCard](#) komponens segítségével kerülnek megjelenítésre.
- A [key](#) attribútum az egyedi [payment.id](#) értéket használja, hogy biztosítsa a React hatékony renderelését.

4. Elrendezés

- A kártyák egy rugalmas (flex) elrendezésben jelennek meg, amely automatikusan alkalmazkodik a képernyőmérethez.
- A flex-wrap osztály biztosítja, hogy a kártyák több sorban is megjelenhessenek, ha szükséges.

Felhasználói élmény

- **Átláthatóság:** A fizetési módok jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
- **Reszponzivitás:** A kártyák elrendezése rugalmas, így a komponens különböző eszközökön is megfelelően működik.
- **Egyszerű használat:** A [PaymentMethodList.jsx](#) egyértelműen és vizuálisan vonzó módon jeleníti meg a fizetési módokat.

Összegzés

A [PaymentMethodList.jsx](#) egy hatékony komponens, amely az adminisztrátorok számára biztosítja a fizetési módok listázását. A [PaymentMethodCard](#) alkotmányos használata és a reszponzív elrendezés biztosítja a könnyű használatot és a vizuális érthetőséget.



PaymentMethodCard.jsx – Fizetési mód kártya

A `PaymentMethodCard.jsx` egy olyan komponens, amely egyetlen fizetési módot jelenít meg kártya formájában. Ez a komponens lehetővé teszi a fizetési módok megtékinthetését, módosítását és törlését az adminisztrátorok számára.

Főbb funkciók

1. Fizetési mód megjelenítése

- A kártya tartalmazza a fizetési mód nevét (`payment.card_type`), amely a kártya címeként jelenik meg.

2. Fizetési mód módosítása

- A `modosit` függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a `NewPaymentMethod` komponenshez navigál, ahol a kiválasztott fizetési mód adatai előre kitöltve jelennek meg.

3. Fizetési mód törlése

- A `torles` függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a `backendMuvelet` függvényt hívja meg, amely egy `DELETE` kérést küld a backend API-nak a fizetési mód törlésére.

4. Felhasználói jogosultságok ellenőrzése

- A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el (`user.isAdmin >= 70`).
-

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Cím:** A fizetési mód neve (`payment.card_type`).
 - **Gombok:** "Módosítás" és "Törlés" gombok, amelyek az adminisztrátorok számára érhetők el.



3. Backend kommunikáció

- A `torles` függvény a `backendMuvelet` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a fizetési mód azonosítóját (`payment.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

4. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a `NewPaymentMethod` komponenshez, és átadja a kiválasztott fizetési mód adatait.

Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a fizetési módok könnyen áttekinthetők.
- **Egyszerű használat:** A gombok egyértelműen jelzik a lehetséges műveleteket, például módosítás vagy törlés.
- **Biztonság:** A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a fizetési módok kezelése biztonságos.

Összegzés

A `PaymentMethodCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi a fizetési módok kezelését és megtekintését. A vizuális megjelenés, a backend kommunikáció és a felhasználói jogosultságok ellenőrzése biztosítja a könnyű használatot és a hatékony adminisztrációt.



PublicAreaCard.jsx – Közterület kártyaNewPublicArea.jsx – Új közterület létrehozása és módosítása

A NewPublicArea.jsx egy olyan komponens, amely lehetővé teszi új közterületek létrehozását, illetve meglévő közterületek módosítását az adminisztrációs felületen. Ez a komponens dinamikusan alkalmazkodik a felhasználói művelethez (új közterület hozzáadása vagy meglévő módosítása), és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Új közterület létrehozása

- Ha a komponens nem kap adatokat (state értéke null), akkor az alapértelmezett űrlapmezők jelennek meg, és a művelet típusa POST lesz.
- Az űrlap kitöltése után a felhasználó a "Felvitel" gombra kattintva hozhat létre új közterületet.

2. Meglévő közterület módosítása

- Ha a komponens kap adatokat (state tartalmazza a közterület adatait), akkor az űrlapmezők előre kitöltődnek a meglévő adatokkal.
- A művelet típusa PATCH, és a módosítások mentése után a felhasználó visszairányításra kerül a közterületek listájára.

3. Backend kommunikáció

- A backendMuvelet függvény segítségével a komponens elküldi az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (VITE_BASE_URL) alapján.

4. Űrlapkezelés

- Az űrlapmezők értékei a formData állapotban tárolódnak.
 - A writeData függvény kezeli az űrlapmezők változásait, és frissíti az állapotot.
-



Felépítés

1. Cím és művelet típusa

- Az űrlap címe (cím) dinamikusan változik a művelet típusától függően:
 - Új közterület esetén: "Új Közterület Felvitele".
 - Módosítás esetén: "[Közterület neve] módosítása".

2. Űrlapmezők

- **Közterület neve (public_area_name)**: Szöveg típusú mező, amely a közterület nevét tartalmazza.

3. Űrlap elküldése

- A onSubmit függvény kezeli az űrlap elküldését, és meghívja a backendMuvelet függvényt az adatok mentésére.

4. Navigáció

- Az űrlap elküldése után a felhasználó automatikusan visszairányításra kerül a közterületek listájára (/publicareas).

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- **Dinamikus működés:** A komponens automatikusan alkalmazkodik a művelet típusához (új közterület vagy módosítás).
- **Visszajelzés:** Az űrlap elküldése után a felhasználó azonnal visszairányításra kerül a közterületek listájára, ahol ellenőrizheti a változásokat.

Összegzés

A NewPublicArea.jsx egy hatékony és rugalmas komponens, amely lehetővé teszi az adminisztrátorok számára új közterületek létrehozását és meglévők módosítását. Az egyszerű űrlapkezelés és a dinamikus működés biztosítja a felhasználói élményt és a hatékony adminisztrációt.



PublicAreaList.jsx – Közterületek listázása

A PublicAreaList.jsx egy olyan komponens, amely az adminisztrációs felületen belül a közterületek listázására szolgál. Ez a komponens a PublicAreaCard alkotmányos segítségével jeleníti meg az egyes közterületeket, és az InitialContext-ből nyeri ki a szükséges adatokat.

Főbb funkciók

1. Közterületek listázása

- A PublicAreaList.jsx az areas adatokat az InitialContext-ből nyeri ki, amely tartalmazza az összes közterületet.
- Az adatok megjelenítése dinamikusan történik az areas.map() metódus segítségével.

2. Közterületek megjelenítése kártyák formájában

- Az egyes közterületeket a PublicAreaCard komponens jeleníti meg, amely a publicarea objektumot kapja propként.
- A kártyák reszponzív elrendezésben jelennek meg, így különböző képernyőméretekben is jól használhatók.

3. Felhasználói élmény

- A komponens egyértelmű címet jelenít meg: "Közterület neveinek Listája".
- A kártyák vizuálisan elkülönülnek, így a felhasználók könnyen áttekinthetik a közterületeket.

Felépítés

1. Cím

- A komponens tetején egy központi cím található: "Közterület neveinek Listája".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.



3. Közterületek megjelenítése

- Az areas tömb elemei a PublicAreaCard komponens segítségével kerülnek megjelenítésre.
- A key attribútum az egyedi area.id értéket használja, hogy biztosítsa a React hatékony renderelését.

4. Elrendezés

- A kártyák egy rugalmas (flex) elrendezésben jelennek meg, amely automatikusan alkalmazkodik a képernyőmérethez.
- A flex-wrap osztály biztosítja, hogy a kártyák több sorban is megjelenhessenek, ha szükséges.

Felhasználói élmény

- **Átláthatóság:** A közterületek jól strukturáltan, kártyák formájában jelennek meg, így könnyen áttekinthetők.
- **Reszponzivitás:** A kártyák elrendezése rugalmas, így a komponens különböző eszközökön is megfelelően működik.
- **Egyszerű használat:** A PublicAreaList.jsx egyértelműen és vizuálisan vonzó módon jeleníti meg a közterületeket.

Összegzés

A PublicAreaList.jsx egy hatékony komponens, amely az adminisztrátorok számára biztosítja a közterületek listázását. A PublicAreaCard alkkomponens használata és a reszponzív elrendezés biztosítja a könnyű használatot és a vizuális érthetőséget.



PublicAreaCard.jsx – Közterület kártya

A `PublicAreaCard.jsx` egy olyan komponens, amely egyetlen közterületet jelenít meg kártya formájában. Ez a komponens lehetővé teszi a közterületek megtekintését, módosítását és törlését az adminisztrátorok számára.

Főbb funkciók

1. Közterület megjelenítése

- A kártya tartalmazza a közterület nevét (`publicarea.public_area_name`), amely a kártya címeként jelenik meg.

2. Közterület módosítása

- A `modosit` függvény a "Módosítás" gombhoz van kötve.
- A gomb kattintásakor a felhasználó a NewPublicArea komponenshez navigál, ahol a kiválasztott közterület adatai előre kitöltve jelennek meg.

3. Közterület törlése

- A `torles` függvény a "Törlés" gombhoz van kötve.
- A gomb kattintásakor a komponens a `backendMuvelet` függvényt hívja meg, amely egy DELETE kérést küld a backend API-nak a közterület törlésére.
- A törlés után az `updatePublicAreaName` függvény frissíti a közterületek listáját.

4. Felhasználói jogosultságok ellenőrzése

- A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el (`user.isAdmin >= 70`).

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a következőket:
 - **Cím:** A közterület neve (`publicarea.public_area_name`).
 - **Gombok:** "Módosítás" és "Törlés" gombok, amelyek az adminisztrátorok számára érhetők el.



3. Backend kommunikáció

- A `torles` függvény a `backendMuvelet` segítségével küldi el a törlési kérést a backend API-nak.
- A kérés tartalmazza a közterület azonosítóját (`publicarea.id`) és a felhasználói token (`usertoken`) azonosítás céljából.

4. Navigáció

- A `modosit` függvény a `useNavigate` hook segítségével navigál a NewPublicArea komponenshez, és átadja a kiválasztott közterület adatait.

Felhasználói élmény

- **Átláthatóság:** A kártyák vizuálisan elkülönülnek, így a közterületek könnyen áttekinthetők.
- **Egyszerű használat:** A gombok egyértelműen jelzik a lehetséges műveleteket, például módosítás vagy törlés.
- **Biztonság:** A "Módosítás" és "Törlés" gombok csak adminisztrátori jogosultsággal rendelkező felhasználók számára érhetők el, így a közterületek kezelése biztonságos.

Összegzés

A `PublicAreaCard.jsx` egy hatékony és reszponzív komponens, amely lehetővé teszi a közterületek kezelését és megtekintését. A vizuális megjelenés, a backend kommunikáció és a felhasználói jogosultságok ellenőrzése biztosítja a könnyű használatot és a hatékony adminisztrációt.



Register2.jsx – Felhasználói regisztráció

A `Register2.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára, hogy regisztrálnak az alkalmazásba. A komponens egy űrlapot jelenít meg, amelyben a felhasználók megadhatják személyes adataikat, és a backend API-val kommunikál az adatok mentése érdekében.

Főbb funkciók

1. Regisztrációs űrlap

- Az űrlap mezői:
 - **Vezetéknév (last_name):** Szöveg típusú mező.
 - **Keresztnév (first_name):** Szöveg típusú mező.
 - **Email:** Email típusú mező.
 - **Jelszó (password):** Jelszó típusú mező.
 - **Jelszó újra (passwordAgain):** Jelszó típusú mező a jelszó megerősítéséhez.

2. Adatok validálása

- Az `onSubmit` függvény ellenőrzi, hogy a két jelszó megegyezik-e. Ha nem, hibaüzenetet jelenít meg.

3. Backend kommunikáció

- A `kuldes` függvény a `fetch` API segítségével küldi el az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.
- Sikeres regisztráció esetén a felhasználó tokenje (`usertoken`) elmentésre kerül a `sessionStorage`-ba.

4. Visszajelzés a felhasználónak

- A `toast` könyvtár segítségével sikeres vagy sikertelen regisztráció esetén visszajelzés jelenik meg.



6. Navigáció

- Sikeres regisztráció után a felhasználó automatikusan átirányításra kerül a főoldalra (/).
 - Az űrlap alján található egy link, amely a bejelentkezési oldalra (/login2) navigál.
-

Felépítés

1. Cím

- Az űrlap tetején egy központi cím található: "Regisztráció".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.

2. Űrlapmezők

- Az űrlapmezők egyértelműen jelzik, hogy milyen adatokat várnak.
- Az ikonok vizuális segítséget nyújtanak a mezők azonosításában.

3. Űrlap elküldése

- A "Regisztráció" gomb elküldi az űrlapot, és meghívja a kuldes függvényt az adatok mentésére.

4. Navigációs lehetőségek

- Az űrlap alján található egy link, amely a bejelentkezési oldalra navigál.
-

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
 - **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a regisztráció sikerességéről vagy sikertelenségéről.
 - **Biztonság:** A jelszavak megerősítése biztosítja, hogy a felhasználók ne véletlenül írjanak be hibás jelszót.
-



Összegzés

A Register2.jsx egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a felhasználók számára a regisztrációt. Az egyszerű űrlapkezelés, a visszajelzések és a backend kommunikáció biztosítja a pozitív felhasználói élményt és a biztonságos adatkezelést.

Regisztráció

👤 Vezetéknév

👤 Keresztnév

✉️ Email

🔒 Jelszó

🔒 Jelszó újra

Regisztráció

 VAGY

Van fiókja?
[Jelentkezzen be](#)

5. ábra Register Desktop



Login2.jsx – Felhasználói bejelentkezés

A `Login2.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára, hogy bejelentkezzenek az alkalmazásba. A komponens egy egyszerű űrlapot jelenít meg, amelyben a felhasználók megadhatják bejelentkezési adataikat, és a backend API-val kommunikál az adatok ellenőrzése érdekében.

Főbb funkciók

1. Bejelentkezési űrlap

- Az űrlap mezői:
 - **Email:** Email típusú mező a felhasználói email cím megadásához.
 - **Jelszó (password):** Jelszó típusú mező a felhasználói jelszó megadásához.

2. Adatok küldése a backendnek

- A `kuldes` függvény a `fetch` API segítségével küldi el az adatokat a backend API-nak.
- Az API végpont URL-je dinamikusan kerül meghatározásra a környezeti változók (`VITE_BASE_URL`) alapján.
- Sikeres bejelentkezés esetén:
 - A felhasználó tokenje (`usertoken`) elmentésre kerül a `sessionStorage`-ba.
 - A felhasználói adatok (`user`) elmentésre kerülnek a `secureStorage`-ba.
 - A felhasználó átirányításra kerül a főoldalra (/).
- Sikertelen bejelentkezés esetén hibaüzenet jelenik meg.

3. Visszajelzés a felhasználónak

- A `toast` könyvtár segítségével sikeres vagy sikertelen bejelentkezés esetén visszajelzés jelenik meg.

4. Navigáció

- Az űrlap alján található egy link, amely a regisztrációs oldalra (/register2) navigál.



Felépítés

1. Cím

- Az űrlap tetején egy központi cím található: "Bejelentkezés".
- A cím kiemelt, nagy méretű és színes (text-primary), hogy könnyen észrevehető legyen.

2. Űrlapmezők

- Az űrlapmezők egyértelműen jelzik, hogy milyen adatokat várnak.
- Az ikonok vizuális segítséget nyújtanak a mezők azonosításában.

3. Űrlap elküldése

- A "Bejelentkezés" gomb elküldi az űrlapot, és meghívja a kuldes függvényt az adatok ellenőrzésére.

4. Navigációs lehetőségek

- Az űrlap alján található egy link, amely a regisztrációs oldalra navigál.

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
- **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a bejelentkezés sikerességről vagy sikertelenségről.
- **Biztonság:** A felhasználói token és adatok biztonságosan tárolódnak a sessionStorage és secureStorage segítségével.

Összegzés

A Login2.jsx egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a felhasználók számára a bejelentkezést. Az egyszerű űrlapkezelés, a visszajelzések és a backend kommunikáció biztosítja a pozitív felhasználói élményt és a biztonságos adatkezelést.



Profile.jsx – Felhasználói profil kezelése

A `Profile.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára a profiljuk kezelését, beleértve a profilkép feltöltését, módosítását és a profil törlését. A komponens a backend API-val kommunikál az adatok frissítése és törlése érdekében.

Főbb funkciók

1. Profilkép feltöltése és módosítása

- A felhasználók feltölthetnek vagy módosíthatnak egy profilképet.
- Az `onFileChange` függvény kezeli a fájl kiválasztását, és frissíti az `image` állapotot.
- Az `onSubmit` függvény a kiválasztott képet elküldi a backend API-nak.

2. Profil törlése

- A `handleDelete` függvény lehetővé teszi a felhasználók számára a profiljuk törlését.
- A törlés során:
 - A backend API egy DELETE kérést kap.
 - A `sessionStorage`-ból eltávolításra kerül a `profile` adat.
 - A `setProfile(null)` hívás törli a profil adatait a helyi állapotból.
 - A felhasználó visszairányításra kerül a főoldalra (/).

3. Navigáció

- A felhasználók a "Back" gomb segítségével visszatérhetnek a főoldalra.
- Sikeres műveletek után a felhasználók automatikusan átirányításra kerülnek a főoldalra.

4. Visszajelzés a felhasználónak

- A `toast` könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletek sikerességéről vagy sikertelenségről.

5. Kosár megjelenítése

- A komponens tartalmazza a `CartCheckout` alkomponenst, amely a felhasználó kosárának tartalmát jeleníti meg.



Felépítés

1. Profilkép kezelése

- A profilkép megjelenik egy img elemben.
- Ha nincs feltöltött kép, egy alapértelmezett kép jelenik meg.
- A kép módosítása egy fájlfeltöltő mező segítségével történik, amely rejte van, és egy gomb aktiválja.

2. Gombok

- **Save:** A profilkép mentésére szolgál.
- **Delete:** A profil törlésére szolgál.
- **Back:** A főoldalra navigál.

3. Backend kommunikáció

- Az axios könyvtár segítségével történik az adatok küldése és törlése a backend API-nak.
- A kérések tartalmazzák a felhasználói azonosítót (user.id) és a hitelesítési tokenet (usertoken).

4. Kosár megjelenítése

- A CartCheckout alkotmányos komponens a profil alján jelenik meg, és a felhasználó kosarának tartalmát mutatja.

Felhasználói élmény

- **Egyszerű használat:** A profilkép feltöltése, módosítása és törlése intuitív és könnyen kezelhető.
 - **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletek sikereségéről vagy hibáiról.
 - **Biztonság:** A felhasználói token és az azonosítók biztonságosan kezelhetők a backend kommunikáció során.
-



Összegzés

A [Profile.jsx](#) egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a felhasználók számára a profiluk kezelését. Az egyszerű képkezelés, a törlési funkció és a visszajelzések biztosítják a pozitív felhasználói élményt és a biztonságos adatkezelést.

CartCheckout.jsx – Számlázási cím kezelése és rendelés leadása

A [CartCheckout.jsx](#) egy olyan komponens, amely lehetővé teszi a felhasználók számára a számlázási cím megadását vagy kiválasztását, valamint a rendelés leadását. A komponens dinamikusan alkalmazkodik a felhasználói igényekhez, például új cím megadásához vagy meglévő cím kiválasztásához.

Főbb funkciók

1. Új cím megadása

- A felhasználók választhatják az "Új címet adok meg" opciót, amely lehetővé teszi számukra, hogy manuálisan megadják a számlázási címet.
- Az űrlap mezői:
 - Város
 - Irányítószám
 - Email cím
 - Közterület jellege (pl. utca, tér)
 - Utca
 - Házszám

2. Meglévő cím kiválasztása

- Ha a felhasználó nem szeretne új címet megadni, választhat egy már meglévő címet a legördülő listából.
- A címek a backend API-ból kerülnek betöltésre, és a felhasználóhoz tartozó címek jelennek meg.



3. Backend kommunikáció

- Az onSubmit függvény kezeli az űrlap elküldését:
 - Új cím esetén a cím adatai egy POST kérésben kerülnek elküldésre a backend API-nak.
 - Meglévő cím esetén a kiválasztott cím azonosítója (address_id) kerül elküldésre.
- A címek betöltése a fetch API segítségével történik.

4. Felhasználói visszajelzés

- A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak:
 - Ha nem választottak közterület jellegeit, hibaüzenet jelenik meg.
 - Ha nem választottak címet, szintén hibaüzenet jelenik meg.

5. Dinamikus állapotkezelés

- Az isNewAddress állapot határozza meg, hogy a felhasználó új címet ad-e meg, vagy meglévőt választ.
- Az addresses állapot tartalmazza a felhasználóhoz tartozó címeket, amelyeket a backend API-ból tölt be.

Felépítés

1. Cím megadása vagy kiválasztása

- Az űrlap dinamikusan változik az isNewAddress állapot alapján:
 - Ha új címet ad meg a felhasználó, akkor az űrlap mezői jelennek meg.
 - Ha meglévő címet választ, akkor egy legördülő lista jelenik meg a címekkel.

2. Űrlapmezők

- Az új cím megadásához szükséges mezők:
 - Város, irányítószám, email cím, közterület jellege, utca, házszám.
- A meglévő cím kiválasztásához egy legördülő lista jelenik meg.

3. Gombok

- **Cím mentése:** Az új cím mentésére szolgál.
- **Rendelés leadása:** A rendelés véglegesítésére szolgál (a funkció implementációja nem látható a kódban).



4. Backend kommunikáció

- Az új címek mentése és a meglévő címek betöltése a backend API-val történik.
-

Felhasználói élmény

- **Egyszerű használat:** Az űrlap intuitív és könnyen kezelhető, a mezők egyértelműen jelzik, hogy milyen adatokat várnak.
 - **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletek sikerességéről vagy hibáiról.
 - **Reszponzivitás:** A komponens kialakítása biztosítja, hogy különböző eszközökön is megfelelően működjön.
-

Összegzés

A CartCheckout.jsx egy hatékony és rugalmas komponens, amely lehetővé teszi a számlázási címek kezelését és a rendelés leadását. Az egyszerű űrlapkezelés, a dinamikus állapotkezelés és a visszajelzések biztosítják a pozitív felhasználói élményt és a biztonságos adatkezelést.



CartView.jsx – Kosár megtekintése és kezelése

A [CartView.jsx](#) egy olyan komponens, amely lehetővé teszi a felhasználók számára a kosár tartalmának megtekintését, a termékek mennyiségének módosítását, a kosár kiürítését, valamint a pénztárhoz való továbblépést. A komponens reszponzív kialakítású, és a kosár tartalmát dinamikusan jeleníti meg. [Kioko, A. \(2023\)](#)

Főbb funkciók

1. Kosár tartalmának megjelenítése

- A kosárban lévő termékek listája jelenik meg, amely tartalmazza:
 - A termék nevét.
 - A napi bérlesi árat.
 - A kiválasztott átvevőpont nevét és címét.
 - A termék mennyiségét.

2. Termékek mennyiségének módosítása

- A felhasználók növelhetik vagy csökkennethetik a termékek mennyiségét a kosárban a + és - gombok segítségével.
- A [addToCart](#) és [removeFromCart](#) függvények kezelik a mennyiség módosítását.

3. Kosár kiürítése

- A "Teljes kosár kiürítése" gomb segítségével a felhasználók egyszerre eltávolíthatják az összes terméket a kosáról.
- A [clearCart](#) függvény végzi el a kosár tartalmának törlését.

4. Pénztárhoz való továbblépés

- A "Tovább a pénztárhoz" gomb megnyitja az adatvédelmi tájékoztatót ([TermofUseInfo](#)), amelyet a felhasználónak el kell fogadnia a pénztárhoz való továbblépés előtt.

5. Kosár összegzése

- A kosár teljes értéke ([getCartTotal](#)) megjelenik a jobb oldali panelen.

6. Navigáció

- A felhasználók a "X" gomb segítségével visszatérhetnek a főoldalra.



Felépítés

1. Termékek listája

- A kosárban lévő termékek egy rugalmas (flex) elrendezésben jelennek meg.
- minden termékhez tartozik egy kép, név, ár, átvevőpont, és a mennyiség módosítására szolgáló gombok.

2. Kosár összegzése

- A jobb oldali panel tartalmazza a kosár teljes értékét, valamint a "Teljes kosár kiürítése" és "Tovább a pénztárhoz" gombokat.

3. Adatvédelmi tájékoztató

- A TermofUseInfo alkomponens jelenik meg, ha a felhasználó a pénztárhoz való továbblépést választja.

4. Üres kosár kezelése

- Ha a kosár üres, egy üzenet jelenik meg: "Kosarad üres jelenleg."

Felhasználói élmény

- **Átláthatóság:** A kosár tartalma jól strukturáltan jelenik meg, így a felhasználók könnyen áttekinthetik a termékeket.
- **Egyszerű használat:** A mennyiség módosítása, a kosár kiürítése és a pénztárhoz való továbblépés intuitív és könnyen kezelhető.
- **Visszajelzés:** Az adatvédelmi tájékoztató biztosítja, hogy a felhasználók tisztában legyenek a vásárlási feltételekkel.

Összegzés

A CartView.jsx egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a kosár tartalmának kezelését és a pénztárhoz való továbblépést. Az egyszerű kezelőfelület, a dinamikus állapotkezelés és a reszponzív kialakítás biztosítja a pozitív felhasználói élményt.



OrderCheckout.jsx – Rendelés véglegesítése

Az [OrderCheckout.jsx](#) egy olyan komponens, amely a rendelési folyamat utolsó lépését valósítja meg. Ez a komponens összefogja a számlázási cím ([CartCheckout](#)) és a fizetési mód ([PaymentMethod](#)) kiválasztását, majd lehetővé teszi a rendelés véglegesítését.

Főbb funkciók

1. Számlázási cím kezelése

- A [CartCheckout](#) alkomponens segítségével a felhasználók megadhatják vagy kiválaszthatják a számlázási címet.

2. Fizetési mód kiválasztása

- A [PaymentMethod](#) alkomponens lehetővé teszi a felhasználók számára, hogy kiválasszák a kívánt fizetési módot.
- A [handlePaymentChange](#) függvény frissíti a [paymentMethod](#) állapotot a kiválasztott fizetési móddal.

3. Rendelés véglegesítése

- A "Rendelés véglegesítése" gomb elküldi a rendelési adatokat a backend API-nak.
- A [finalSubmit](#) függvény:
 - Ellenőrzi, hogy a számlázási cím meg van-e adva.
 - Meghívja a megfelelő rendelésküldő függvényt ([submitOrder](#) vagy [submitOrderisAddress](#)).
 - Sikeres rendelés esetén a felhasználót átirányítja a rendeléseit megjelenítő oldalra (/userorder).
 - Sikertelen rendelés esetén hibaüzenetet jelenít meg a konzolon.

4. Navigáció

- A rendelés véglegesítése után a felhasználó automatikusan átirányításra kerül a rendeléseit megjelenítő oldalra.



Felépítés

1. Űrlap

- Az űrlap tartalmazza a számlázási cím ([CartCheckout](#)) és a fizetési mód ([PaymentMethod](#)) kiválasztására szolgáló alkomponenseket.
- Az űrlap elküldése a [finalSubmit](#) függvényt hívja meg.

2. Gombok

- **Rendelés véglegesítése:** Az űrlap elküldésére szolgál, és elindítja a rendelési folyamatot.

3. Backend kommunikáció

- A rendelési adatok a [submitOrder](#) vagy [submitOrderWithAddress](#) függvényeken keresztül kerülnek elküldésre a backend API-nak.

Felhasználói élmény

- **Egyszerű használat:** A komponens intuitív módon vezeti végig a felhasználót a rendelési folyamat utolsó lépésein.
- **Visszajelzés:** Sikeres rendelés esetén a felhasználó azonnal átirányításra kerül a rendeléseit megjelenítő oldalra.
- **Biztonság:** Az adatok ellenőrzése és a megfelelő függvények meghívása biztosítja a rendelési folyamat helyes működését.

Összegzés

Az [OrderCheckout.jsx](#) egy kulcsfontosságú komponens, amely összefogja a rendelési folyamat utolsó lépéseiit. A számlázási cím és a fizetési mód kezelése, valamint a rendelés véglegesítése biztosítja a felhasználók számára a gördülékeny vásárlási élményt.



PaymentMethod.jsx – Fizetési mód kiválasztása

A `PaymentMethod.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára, hogy kiválasszák a rendelésükhez használni kívánt fizetési módot. A komponens dinamikusan betölti a backend API-ból elérhető fizetési módokat, és a kiválasztott opciót továbbítja a szülő komponensnek.

Főbb funkciók

1. Fizetési módok betöltése

- A `useEffect` hook segítségével a komponens a backend API-ból (`/payment`) lekéri az elérhető fizetési módokat.
- A lekért adatokat a `payments` állapotban tárolja.

2. Fizetési mód kiválasztása

- A felhasználók egy legördülő listából választhatják ki a kívánt fizetési módot.
- A `handleChange` függvény frissíti a `formDataPayment` állapotot a kiválasztott fizetési móddal, és meghívja az `onPaymentChange` propot, hogy a szülő komponens is értesüljön a változásról.

3. Backend kommunikáció

- A fizetési módok lekérése egy GET kérés segítségével történik.
 - A kérés tartalmazza a szükséges fejlécet (Content-Type: application/json).
-

Felépítés

1. Legördülő lista

- A fizetési módok egy `select` elemben jelennek meg.
- Az alapértelmezett opció: "Válasszon fizetési módot".
- A `payments` tömb elemei dinamikusan kerülnek megjelenítésre a `map` metódus segítségével.

2. Úrlapkezelés

- A `handleChange` függvény kezeli a legördülő lista változásait, és frissíti a `formDataPayment` állapotot.



3. Visszajelzés a felhasználónak

- Ha hiba történik a fizetési módok lekérésekor, a hibaüzenet a konzolra kerül ([console.error](#)).
-

Felhasználi élmény

- **Egyszerű használat:** A legördülő lista intuitív módon teszi lehetővé a fizetési mód kiválasztását.
 - **Dinamikus tartalom:** A fizetési módok automatikusan betöltődnek a backend API-ból, így minden naprakészek.
 - **Visszajelzés:** A kiválasztott fizetési mód azonnal frissül az állapotban, és továbbításra kerül a szülő komponensnek.
-

Összegzés

A [PaymentMethod.jsx](#) egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a fizetési módok kezelését a rendelési folyamat során. A dinamikus adatbetöltés, az egyszerű kezelőfelület és a szülő komponenssel való kommunikáció biztosítja a gördülékeny felhasználi élményt.



TermofUseInfo.jsx – Felhasználási feltételek elfogadása

A TermofUseInfo.jsx egy olyan komponens, amely lehetővé teszi a felhasználók számára, hogy megtekintsék és elfogadják a felhasználási feltételeket, mielőtt továbblépnének a rendelési folyamat következő lépéseihez. A komponens egy modális ablakban jelenik meg, és biztosítja, hogy a felhasználók elfogadják a feltételeket.

Főbb funkciók

1. Felhasználási feltételek megjelenítése

- A TermOfUse alkomponens jeleníti meg a felhasználási feltételek szövegét a modális ablakban.

2. Elfogadás ellenőrzése

- A felhasználók egy jelölőnégyzet (checkbox) segítségével fogadhatják el a feltételeket.
- Az isChecked állapot tárolja, hogy a felhasználó elfogadta-e a feltételeket.

3. Tovább a pénztárhoz

- A "Tovább a pénztárhoz" gomb csak akkor működik, ha a felhasználó elfogadta a feltételeket.
- Az openOrder függvény:
 - Ellenőrzi, hogy a jelölőnégyzet be van-e jelölve.
 - Ha igen, sikeres visszajelzést (toast.success) jelenít meg, bezárja a modális ablakot, és a felhasználót a pénztár oldalra (/checkout) navigálja.
 - Ha nem, hibaüzenetet (toast.error) jelenít meg.

4. Bezárás funkció

- A "Bezárás" gomb a closeFunction prop segítségével zárja be a modális ablakot.

Felépítés

1. Modális ablak

- A modal és modal-box osztályok biztosítják a modális ablak megjelenését és stílusát.
- A TermOfUse alkomponens jeleníti meg a felhasználási feltételek szövegét.



2. Jelölőnégyzet

- A jelölőnégyzet (checkbox) lehetővé teszi a felhasználók számára a feltételek elfogadását.
- Az isChecked állapot frissül a handleCheckboxChange függvény segítségével.

3. Gombok

- **Bezárás:** A modális ablak bezárására szolgál.
- **Tovább a pénztárhoz:** Ellenőrzi a feltételek elfogadását, és navigál a pénztár oldalra.

4. Visszajelzés

- A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a feltételek elfogadásáról vagy annak hiányáról.

Felhasználói élmény

- **Átláthatóság:** A modális ablak vizuálisan elkülönül, így a felhasználók könnyen azonosíthatják a felhasználási feltételeket.
- **Egyszerű használat:** A jelölőnégyzet és a gombok intuitív módon teszik lehetővé a feltételek elfogadását és a továbblépést.
- **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveleteikről.

Összegzés

A TermofUseInfo.jsx egy hatékony és felhasználóbarát komponens, amely biztosítja, hogy a felhasználók elfogadják a felhasználási feltételeket, mielőtt továbblépnének a rendelési folyamatban. A modális ablak kialakítása, a visszajelzések és a navigáció biztosítják a gördülékeny felhasználói élményt.



UserOrder.jsx – Felhasználói rendelések megtekintése

A `UserOrder.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára, hogy megtekintsék a korábbi rendeléseiket. A komponens a backend API-val kommunikál a rendelések lekérése érdekében, és részletes információkat jelenít meg a rendelések rölről.

Főbb funkciók

1. Rendelések lekérése

- A `useEffect` hook segítségével a komponens a backend API-ból (`/order`) lekéri a felhasználóhoz tartozó rendeléseket.
- A lekért adatokat a `orders` állapotban tárolja.
- A kérés tartalmazza:
 - A felhasználói azonosítót (`UserId`).
 - A hitelesítési tokent (`Authorization`).

2. Kosár kiürítése

- A rendelési adatok betöltése után a `clearCart` függvény kiüríti a kosár tartalmát.

3. Rendelések megjelenítése

- A komponens részletes információkat jelenít meg minden rendelésről:
 - Rendelés azonosítója.
 - Megrendelő neve és email címe.
 - Számlázási cím.
 - Rendelés teljes összege.
 - Rendelési tételek, beleértve:
 - Csomagautomata neve és címe.
 - Termék azonosítója.
 - Ár, darabszám, és összeg.

4. Navigáció

- A "Bezárás" gomb a `handleClose` függvény segítségével visszairányítja a felhasználót a főoldalra (/).



5. Hiba- és állapotkezelés

- Ha hiba történik a rendelések lekérésekor, a hibaüzenet a konzolra kerül (`console.error`).
- Ha nincs rendelés, egy üzenet jelenik meg: "Nincs rendelés."

Felépítés

1. Rendelések listája

- A rendelések egy kártya (card) formátumban jelennek meg.
- minden rendelés külön szekcióban jelenik meg, amely tartalmazza a rendelés részleteit és a rendelési tételeket.

2. Rendelési tételek

- A rendelési tételek egy külön szekcióban jelennek meg, amely tartalmazza a termékek és a csomagautomaták részleteit.

3. Gombok

- **Bezáras:** A főoldalra navigál.

Felhasználói élmény

- **Átláthatóság:** A rendelések jól strukturáltan jelennek meg, így a felhasználók könnyen áttekinthetik a rendeléseiket.
- **Egyszerű használat:** A "Bezáras" gomb intuitív módon teszi lehetővé a visszatérést a főoldalra.
- **Visszajelzés:** Ha nincs rendelés, egyértelmű üzenet jelenik meg a felhasználónak.

Összegzés

A `UserOrder.jsx` egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a felhasználók számára a rendeléseik részletes megtekintését. A dinamikus adatbetöltés, a részletes megjelenítés és a navigáció biztosítja a pozitív felhasználói élményt.



UserCard.jsx – Felhasználói kártya kezelése

A `UserCard.jsx` egy olyan komponens, amely egyetlen felhasználó adatait jeleníti meg kártya formájában. Ez a komponens lehetővé teszi az adminisztrátorok számára a felhasználók adatainak megtekintését, szerkesztését és törlését.

Főbb funkciók

1. Felhasználói adatok megjelenítése

- A kártya tartalmazza a felhasználó alábbi adatait:
 - Azonosító (`user.id`).
 - Vezetéknév és keresztnév (`user.last_name`, `user.first_name`).
 - Email cím (`user.email`).
 - Jogosultság (`user.role.warrant_name`).
 - Létrehozás dátuma és időpontja (`user.created_at`).

2. Felhasználói adatok szerkesztése

- A "Szerkesztés" gomb megnyitja a `RegistrationDataEdit` alkotmányos komponenst egy modális ablakban.
- A `registrationDataEdit` függvény navigál a /registrationdataedit oldalra, ahol a felhasználó adatai szerkeszthetők.

3. Felhasználó törlése

- A "Törlés" gomb meghívja a `deleteUser` függvényt, amely törli a felhasználót a backend API-n keresztül.
- A törlés a felhasználó azonosítója (`user.id`) alapján történik.

4. Modális ablak kezelése

- Az `isModalOpen` állapot határozza meg, hogy a modális ablak nyitva van-e.
 - Az `openModal` és `closeModal` függvények kezelik a modális ablak megnyitását és bezárását.
-



Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a felhasználó adatait és a műveleti gombokat.
- A kártya vizuálisan elkülönül, és reszponzív kialakítású.

2. Műveleti gombok

- **Szerkesztés:** Megnyitja a `RegistrationDataEdit` alkomponenst modális ablakban.
- **Törlés:** Meghívja a `deleteUser` függvényt a felhasználó törlésére.

3. Modális ablak

- A `RegistrationDataEdit` alkomponens jelenik meg, ha az `isModalOpen` állapot igaz.

Felhasználói élmény

- **Átláthatóság:** A felhasználói adatok jól strukturáltan jelennek meg, így az adminisztrátorok könnyen áttekinthetik az információkat.
- **Egyszerű használat:** A "Szerkesztés" és "Törlés" gombok intuitív módon teszik lehetővé a felhasználói adatok kezelését.
- **Biztonság:** A törlési művelet csak a megfelelő azonosító alapján történik, így minimalizálva a hibalehetőségeket.

Összegzés

A `UserCard.jsx` egy hatékony és felhasználóbarát komponens, amely lehetővé teszi az adminisztrátorok számára a felhasználók adatainak kezelését. A vizuális megjelenés, a modális ablak használata és a műveleti gombok biztosítják a gördülékeny adminisztrációs élményt.



UserDashboard.jsx – Felhasználói megrendelések kezelése

A `UserDashboard.jsx` egy olyan komponens, amely lehetővé teszi a felhasználók számára a megrendeléseik megtekintését, szűrését, lapozását és törlését. Ez a komponens reszponzív kialakítású, és dinamikusan kezeli a megrendeléseket a backend API-val való kommunikáció segítségével.

Főbb funkciók

1. Megrendelések lekérése

- A `useEffect` hook segítségével a komponens a backend API-ból (`/userorderslist`) lekéri a felhasználóhoz tartozó megrendeléseket.
- A lekért adatokat a `orders` és `filteredOrders` állapotokban tárolja.

2. Megrendelések szűrése

- A felhasználók többféle szűrési feltételt alkalmazhatnak:
 - Név szerint (`searchName`).
 - Email cím szerint (`searchEmail`).
 - Csomagautomata neve szerint (`searchLockerName`).
 - Csomagautomata címe szerint (`searchLockerAddress`).
- A `handleSearch` függvény kezeli a szűrési feltételek változását, és frissíti a `filteredOrders` állapotot.

3. Megrendelések lapozása

- A megrendelések lapozása a `currentPage` és `ordersPerPage` állapotok segítségével történik.
- A felhasználók az előző (`handlePrevPage`) és a következő (`handleNextPage`) oldalra navigálhatnak.

4. Megrendelések törlése

- A `handleDelete` függvény lehetővé teszi a megrendelések törlését a backend API-n keresztül.
- A törlés után az `orders` és `filteredOrders` állapotok frissülnek, hogy eltávolítsák a törölt megrendelést.



5. Navigáció

- A "Bezárás" gomb a `handleClose` függvény segítségével visszairányítja a felhasználót a főoldalra (/).
-

Felépítés

1. Szűrők

- A bal oldali panel tartalmazza a szűrési mezőket, amelyek lehetővé teszik a megrendelések gyors keresését.

2. Megrendelések listája

- A megrendelések a `UserOrdersCard` alkomponens segítségével jelennek meg.
- A listázás reszponzív, és a megrendelések kártyák formájában jelennek meg.

3. Lapozás

- Az oldal alján található lapozó gombok lehetővé teszik a megrendelések közötti navigációt.

4. Gombok

- **Lapozás:** Az előző és következő oldalra navigál.
 - **Bezárás:** A főoldalra navigál.
-

Felhasználói élmény

- **Átláthatóság:** A megrendelések jól strukturáltan jelennek meg, így a felhasználók könnyen áttekinthetik az információkat.
 - **Egyszerű használat:** A szűrési mezők és a lapozás intuitív módon teszik lehetővé a megrendelések kezelését.
 - **Visszajelzés:** A `toast` könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a törlési műveletekről.
-

Összegzés

A `UserDashboard.jsx` egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a felhasználók számára a megrendelések kezelését. A szűrési lehetőségek, a lapozás és a törlési funkció biztosítják a gördülékeny felhasználói élményt.



UserOrdersCard.jsx – Megrendelési kártya

A `UserOrdersCard.jsx` egy olyan komponens, amely egyetlen megrendelés részleteit jeleníti meg kártya formájában. Ez a komponens lehetővé teszi az adminisztrátorok vagy felhasználók számára a megrendelések megtekintését és törlését.

Főbb funkciók

1. Megrendelési adatok megjelenítése

- A kártya tartalmazza a megrendelés alábbi részleteit:
 - Azonosító (`order.id`).
 - Megrendelő neve és email címe (`order.user.first_name`, `order.user.last_name`, `order.user.email`).
 - Számlázási cím (`order.address`).
 - Rendelés teljes összege (`order.total`).

2. Rendelési tételek megjelenítése

- Ha a rendeléshez tartoznak tételek (`order.order_item`), azok külön szekcióban jelennek meg:
 - Csomagautomata neve és címe.
 - Termék azonosítója.
 - Ár, darabszám, és összeg.

3. Megrendelés törlése

- A "Törlés" gomb meghívja a `handleDelete` függvényt, amely a rendelés azonosítója (`order.id`) alapján törli a rendelést.

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a megrendelés részleteit és a műveleti gombokat.
- A kártya vizuálisan elkülönül, és reszponzív kialakítású.



2. Rendelési tételek

- A rendelési tételek egy külön szekcióban jelennek meg, amely tartalmazza a termékek és a csomagautomaták részleteit.

3. Műveleti gombok

- **Törlés:** Meghívja a `handleDelete` függvényt a rendelés törlésére.

Felhasználói élmény

- **Átláthatóság:** A megrendelési adatok jól strukturáltan jelennek meg, így a felhasználók könnyen áttekinthetik az információkat.
- **Egyszerű használat:** A "Törlés" gomb intuitív módon teszi lehetővé a megrendelések kezelését.
- **Reszponzivitás:** A kártya kialakítása biztosítja, hogy különböző eszközökön is megfelelően jelenjen meg.

Összegzés

A `UserOrdersCard.jsx` egy hatékony és felhasználóbarát komponens, amely lehetővé teszi a megrendelések részletes megtekintését és törlését. A vizuális megjelenés, a rendelési tételek kezelése és a műveleti gombok biztosítják a gördülékeny felhasználói élményt.



AdminDashboard.jsx – Adminisztrátori felület a felhasználók kezelésére

Az [AdminDashboard.jsx](#) egy olyan komponens, amely lehetővé teszi az adminisztrátorok számára a regisztrált felhasználók adatainak megtekintését, szűrését, lapozását és kezelését. A komponens reszponzív kialakítású, és különböző nézeteket biztosít asztali és mobil eszközökre.

Főbb funkciók

1. Felhasználói adatok lekérése

- A useEffect hook segítségével a komponens a backend API-ból ([users](#)) lekéri az összes regisztrált felhasználót.
- A lekért adatokat a [users](#) állapotban tárolja.
- A kérés tartalmazza a hitelesítési tokent (Authorization).

2. Hozzáférés ellenőrzése

- Ha a felhasználó nem adminisztrátor ([user.isAdmin < 70](#)), a komponens hibaüzenetet jelenít meg, és visszairányítja a főoldalra.

3. Felhasználók szűrése

- A felhasználók szűrése név vagy email cím alapján történik.
- A handleSearch függvény frissíti a [filteredUsers](#) állapotot a szűrési feltételek alapján.

4. Felhasználók lapozása

- A felhasználók lapozása a [currentPage](#) és [itemsPerPage](#) állapotok segítségével történik.
- A felhasználók az előző ([handlePrevPage](#)) és a következő ([handleNextPage](#)) oldalra navigálhatnak.

5. Felhasználók megjelenítése

- **Asztali nézet:** A felhasználók táblázatos formában jelennek meg az [AdminDashboardCard](#) alkotmányos komponens segítségével.
- **Mobil nézet:** A felhasználók kártyák formájában jelennek meg a [UserCard](#) alkotmányos komponens segítségével.



6. Navigáció

- A "Bezárás" gomb a `handleClose` függvény segítségével visszairányítja az adminisztrátort a főoldalra.
-

Felépítés

1. Keresőmező

- A keresőmező lehetővé teszi a felhasználók gyors szűrését név vagy email cím alapján.

2. Felhasználók listája

- **Asztali nézet:** Táblázatos formában jelenik meg, amely tartalmazza a felhasználók adatait és a műveleti oszlopokat.
- **Mobil nézet:** Kártyák formájában jelenik meg, amely tartalmazza a felhasználók adatait.

3. Lapozás

- Az oldal alján található lapozó gombok lehetővé teszik a felhasználók közötti navigációt.

4. Gombok

- **Lapozás:** Az előző és következő oldalra navigál.
 - **Bezárás:** A főoldalra navigál.
-

Felhasználói élmény

- **Átláthatóság:** A felhasználói adatok jól strukturáltan jelennek meg, így az adminisztrátorok könnyen áttekinthetik az információkat.
 - **Egyszerű használat:** A keresőmező, a lapozás és a különböző nézetek intuitív módon teszik lehetővé a felhasználók kezelését.
 - **Rezoncivitás:** A komponens kialakítása biztosítja, hogy különböző eszközökön is megfelelően működjön.
-



Összegzés

Az AdminDashboard.jsx egy hatékony és felhasználóbarát komponens, amely lehetővé teszi az adminisztrátorok számára a regisztrált felhasználók kezelését. A szűrési lehetőségek, a lapozás és a reszponzív kialakítás biztosítják a gördülékeny adminisztrációs élményt.

The screenshot shows a table titled "Regisztrált Felhasználói Adatok" (Registered User Data). The table has columns for Id, Vezetéknév (First Name), Keresztnév (Last Name), Email cím (Email Address), Joga (Role), Létrehozva (Created At), and Műveletek (Actions). There are four rows of data:

Id	Vezetéknév	Keresztnév	Email cím	Joga	Létrehozva	Műveletek
68	Kovács	Anna	anna.kovacs+1.1746301079245@example.com	user	2025-05-03 19:37:48	<button>Szerkesztés</button> <button>Törés</button>
69	Nagy	Béla	bela.nagy+2.1746301081130@example.com	user	2025-05-03 19:37:50	<button>Szerkesztés</button> <button>Törés</button>
70	Tóth	Csilla	csilla.toth+3.1746301083001@example.com	user	2025-05-03 19:37:52	<button>Szerkesztés</button> <button>Törés</button>
71	Szabó	Dániel	daniel.szabo+4.1746301084835@example.com	user	2025-05-03 19:37:54	<button>Szerkesztés</button> <button>Törés</button>

At the bottom, there is a navigation bar with buttons for back, forward, and search, followed by the text "Oldal 3 / 10".

6. ábra Admin User List Desktop

The screenshot shows the same "Regisztrált Felhasználói Adatok" page, but it is displayed in a mobile responsive layout. It features two user cards: "Azonosító #68" and "Azonosító #69". Each card displays the user's first name, last name, email address, and creation date. Below each card are "Szerkesztés" and "Törés" buttons. At the bottom of the screen are standard mobile navigation icons: a menu, a search bar, a shopping cart, and a refresh button.

7. ábra Admin User List Mobile



AdminDashboardCard.jsx – Adminisztrátori felhasználói kártya

Az `AdminDashboardCard.jsx` egy olyan komponens, amely egyetlen felhasználó adatait jeleníti meg táblázatos formában az adminisztrátori felületen. Ez a komponens lehetővé teszi az adminisztrátorok számára a felhasználók adatainak megtekintését, szerkesztését és törlését.

Főbb funkciók

1. Felhasználói adatok megjelenítése

- A táblázat sorai tartalmazzák a felhasználó alábbi adatait:
 - Azonosító (`user.id`).
 - Vezetéknév és keresztnév (`user.last_name`, `user.first_name`).
 - Email cím (`user.email`).
 - Jogosultság (`user.role.warrant_name`).
 - Létrehozás dátuma és időpontja (`user.created_at`).

2. Felhasználói adatok szerkesztése

- A "Szerkesztés" gomb megnyitja a `RegistrationDataEdit` alkókomponenst egy modális ablakban.
- Az `openModal` függvény kezeli a modális ablak megnyitását, míg a `closeModal` függvény annak bezárását.

3. Felhasználó törlése

- A "Törlés" gomb meghívja a `deleteUser` függvényt, amely a felhasználót törli a backend API-n keresztül.
- A törlés a felhasználó azonosítója (`user.id`) alapján történik.

Felépítés

1. Táblázatos megjelenítés

- A felhasználói adatak egy táblázat sorában jelennek meg.
- A táblázat oszlopai tartalmazzák az azonosítót, a neveket, az email címet, a jogosultságot, a létrehozás dátumát, valamint a műveleti gombokat.



2. Műveleti gombok

- **Szerkesztés:** Megnyitja a [RegistrationDataEdit](#) alkomponenst modális ablakban.
- **Törlés:** Meghívja a [deleteUser](#) függvényt a felhasználó törlésére.

3. Modális ablak

- A [RegistrationDataEdit](#) alkomponens jelenik meg, ha az [isModalOpen](#) állapot igaz.

Felhasználói élmény

- **Átláthatóság:** A felhasználói adatok jól strukturáltan jelennek meg a táblázatban, így az adminisztrátorok könnyen áttekinthetik az információkat.
- **Egyszerű használat:** A "Szerkesztés" és "Törlés" gombok intuitív módon teszik lehetővé a felhasználói adatok kezelését.
- **Biztonság:** A törlési művelet csak a megfelelő azonosító alapján történik, így minimalizálva a hibalehetőségeket.

Összegzés

Az [AdminDashboardCard.jsx](#) egy hatékony és felhasználóbarát komponens, amely lehetővé teszi az adminisztrátorok számára a felhasználók adatainak kezelését. A táblázatos megjelenítés, a modális ablak használata és a műveleti gombok biztosítják a gördülékeny adminisztrációs élményt.



AdminOrders.jsx – Adminisztrátori megrendelések kezelése

Az `AdminOrders.jsx` egy olyan komponens, amely lehetővé teszi az adminisztrátorok számára a megrendelések megtekintését, szűrését, lapozását és törlését. A komponens reszponzív kialakítású, és dinamikusan kezeli a megrendelésekkel a backend API-val való kommunikáció segítségével.

Főbb funkciók

1. Megrendelések lekérése

- A `useEffect` hook segítségével a komponens a backend API-ból (`/orderslist`) lekéri az összes megrendelést.
- A lekért adatokat a `orders` és `filteredOrders` állapotokban tárolja.
- A kérés tartalmazza a hitelesítési tokent (Authorization).

2. Megrendelések szűrése

- A felhasználók többféle szűrési feltételt alkalmazhatnak:
 - Név szerint (`searchName`).
 - Email cím szerint (`searchEmail`).
 - Csomagautomata neve szerint (`searchLockerName`).
 - Csomagautomata címe szerint (`searchLockerAddress`).
- A `handleSearch` függvény kezeli a szűrési feltételek változását, és frissíti a `filteredOrders` állapotot.

3. Megrendelések lapozása

- A megrendelések lapozása a `currentPage` és `ordersPerPage` állapotok segítségével történik.
- A felhasználók az előző (`handlePrevPage`) és a következő (`handleNextPage`) oldalra navigálhatnak.



4. Megrendelések törlése

- A `handleDelete` függvény lehetővé teszi a megrendelések törlését a backend API-n keresztül.
- A törlés után a `orders` és `filteredOrders` állapotok frissülnek, hogy eltávolítsák a törölt megrendelést.
- Sikeres törlés esetén visszajelzés (`toast.success`) jelenik meg, míg hiba esetén hibaüzenet (`toast.error`) látható.

5. Navigáció

- A "Bezárás" gomb a `handleClose` függvény segítségével visszairányítja az adminisztrátort a főoldalra (/).

Felépítés

1. Szűrők

- A bal oldali panel tartalmazza a szűrési mezőket, amelyek lehetővé teszik a megrendelések gyors keresését.

2. Megrendelések listája

- A megrendelések az `AdminOrdersCard` alkomponens segítségével jelennek meg.
- A listázás reszponzív, és a megrendelések kártyák formájában jelennek meg.

3. Lapozás

- Az oldal alján található lapozó gombok lehetővé teszik a megrendelések közötti navigációt.

4. Gombok

- **Lapozás:** Az előző és következő oldalra navigál.
- **Bezárás:** A főoldalra navigál.

Felhasználói élmény

- **Átláthatóság:** A megrendelések jól strukturáltan jelennek meg, így az adminisztrátorok könnyen áttekinthetik az információkat.
- **Egyszerű használat:** A szűrési mezők és a lapozás intuitív módon teszik lehetővé a megrendelések kezelését.



- Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a törlési műveletekről.

Összegzés

Az AdminOrders.jsx egy hatékony és felhasználóbarát komponens, amely lehetővé teszi az adminisztrátorok számára a megrendelések kezelését. A szűrési lehetőségek, a lapozás és a törlési funkció biztosítják a gördülékeny adminisztrációs élményt.

The screenshot shows the 'Megrendelések' (Orders) section of the Flexistore Admin Dashboard. On the left, there's a search bar labeled 'Szűrők' (Filters) with four input fields: 'Keresés név szerint', 'Keresés email szerint', 'Keresés csomagautomata név szerint', and 'Keresés csomagautomata cím szerint'. The main area displays two order cards. Order #56 on the left has a recipient named 'Elek Teszt' (testuser@gmail.com), a delivery address '5308 Budapest, Rákóczi út 45A. Út', and a total of 'Összeg: 4500.00 Ft'. It includes a detailed breakdown of the order items. Order #55 on the right has a recipient named 'Elek Teszt' (testuser@gmail.com), a delivery address '4008 Győr, Rákóczi út 7/B. Út', and a total of 'Összeg: 4500.00 Ft'. It also includes a detailed breakdown of the order items. Both orders have a red 'Törles' (Delete) button at the bottom right.

8. ábra Admin All Order List Dekstop



AdminOrdersCard.jsx – Adminisztrátori megrendelési kártya

Az [AdminOrdersCard.jsx](#) egy olyan komponens, amely egyetlen megrendelés részleteit jeleníti meg kártya formájában az adminisztrátori felületen. Ez a komponens lehetővé teszi az adminisztrátorok számára a megrendelések megtekintését és törlését.

Főbb funkciók

1. Megrendelési adatok megjelenítése

- A kártya tartalmazza a megrendelés alábbi részleteit:
 - Azonosító (`order.id`).
 - Megrendelő neve és email címe (`order.user.first_name`, `order.user.last_name`, `order.user.email`).
 - Számlázási cím (`order.address`).
 - Rendelés teljes összege (`order.total`).

2. Rendelési tételek megjelenítése

- Ha a rendeléshez tartoznak tételek (`order.order_item`), azok külön szekcióban jelennek meg:
 - Csomagautomata neve és címe.
 - Termék azonosítója.
 - Ár, darabszám, és összeg.

3. Megrendelés törlése

- A "Törlés" gomb meghívja a `handleDelete` függvényt, amely a rendelés azonosítója (`order.id`) alapján törli a rendelést.

Felépítés

1. Kártya elrendezése

- A kártya egy `div` elemben jelenik meg, amely tartalmazza a megrendelés részleteit és a műveleti gombokat.
- A kártya vizuálisan elkülönül, és reszponzív kialakítású.



2. Rendelési tételek

- A rendelési tételek egy külön szekcióban jelennek meg, amely tartalmazza a termékek és a csomagautomaták részleteit.

3. Műveleti gombok

- **Törlés:** Meghívja a handleDelete függvényt a rendelés törlésére.

Felhasználói élmény

- **Átláthatóság:** A megrendelési adatok jól strukturáltan jelennek meg, így az adminisztrátorok könnyen áttekinthetik az információkat.
- **Egyszerű használat:** A "Törlés" gomb intuitív módon teszi lehetővé a megrendelések kezelését.
- **Rezonans:** A kártya kialakítása biztosítja, hogy különböző eszközökön is megfelelően jelenjen meg.

Összegzés

Az [AdminOrdersCard.jsx](#) egy hatékony és felhasználóbarát komponens, amely lehetővé teszi az adminisztrátorok számára a megrendelések részletes megtekintését és törlését. A vizuális megjelenés, a rendelési tételek kezelése és a műveleti gombok biztosítják a gördülékeny adminisztrációs élményt.



RegistrationDataEdit.jsx – Felhasználói adatok szerkesztése

A `RegistrationDataEdit.jsx` egy olyan komponens, amely lehetővé teszi az adminisztrátorok számára a felhasználói adatok szerkesztését. A komponens egy modális ablakban jelenik meg, és biztosítja, hogy a jogosultsági szintek ellenőrzése után történjenek meg a módosítások.

Főbb funkciók

1. Felhasználói adatok betöltése

- A komponens a `user` prop segítségével kapja meg a szerkesztendő felhasználó adatait.
- Az adatok a `formData` állapotban tárolódnak, amely tartalmazza:
 - Azonosító (`id`).
 - Vezetéknév (`last_name`).
 - Keresztnév (`first_name`).
 - Email cím (`email`).
 - Jogosultság azonosítója (`role_id`).

2. Jogosultsági szintek kezelése

- A `secureStorage` segítségével betöltődnek a jogosultsági szintek (`roles`).
- A módosítás előtt a komponens ellenőri, hogy az adminisztrátor jogosultsági szintje (`adminpower`) magasabb-e, mint a módosítani kívánt jogosultság szintje (`keres`).
- Ha az adminisztrátor jogosultsági szintje nem elegendő, hibaüzenet (`toast.error`) jelenik meg.

3. Adatok mentése

- Az `onSubmit` függvény a `backendMuvelet` segítségével küldi el a módosított adatokat a backend API-nak (`/user/edit`).
- Sikeres mentés esetén:
 - A modális ablak bezáródik (`closeFunction`).
 - A felhasználói lista frissül (`update`).
 - Az adminisztrátor visszairányításra kerül az adminisztrációs felületre (`/admindashboard`).



4. Modális ablak kezelése

- A komponens egy modális ablakban jelenik meg, amely tartalmazza az űrlapot a felhasználói adatok szerkesztéséhez.
-

Felépítés

1. Űrlapmezők

- Az űrlap tartalmazza a következő mezőket:
 - **Vezetéknév:** Szöveg típusú mező.
 - **Keresztnév:** Szöveg típusú mező.
 - **Email cím:** Email típusú mező.
 - **Jogosultság:** Legördülő lista, amely a roles állapotból töltődik be.

2. Műveleti gombok

- **Mentés:** Az űrlap elküldésére szolgál.
- **Bezáras:** A modális ablak bezárására szolgál.

3. Jogosultsági ellenőrzés

- Az onSubmit függvény ellenőrzi, hogy az adminisztrátor jogosultsági szintje elegendő-e a módosításhoz.
-

Felhasználói élmény

- **Átláthatóság:** Az űrlap egyértelműen jelzi, hogy milyen adatokat lehet módosítani.
 - **Biztonság:** A jogosultsági szintek ellenőrzése biztosítja, hogy csak megfelelő jogosultsággal rendelkező adminisztrátorok végezhessenek módosításokat.
 - **Egyszerű használat:** A modális ablak és az intuitív űrlapmezők könnyen kezelhetők.
-

Összegzés

A RegistrationDataEdit.jsx egy hatékony és biztonságos komponens, amely lehetővé teszi az adminisztrátorok számára a felhasználói adatok szerkesztését. A jogosultsági ellenőrzés, a modális ablak használata és az egyszerű űrlapkezelés biztosítja a gördülékeny adminisztrációs élményt.



Vezetéknév

Teszt

Keresztnév

János

Email cím

testuser_1746301077488@example.com

Jogosultság

user

Mentés

Bezárás

9. ábra User Data Edit



CrudContext.jsx – CRUD műveletek kezelése kontextusban

A `CrudContext.jsx` egy globális kontextus, amely a CRUD (Create, Read, Update, Delete) műveletek kezelésére szolgál az alkalmazás különböző részein. Ez a kontextus biztosítja az állapotkezelést, az API-hívásokat és a visszajelzéseket a felhasználók számára.

Főbb funkciók

1. Állapotkezelés

- `refresh`: Egy állapot, amely a komponensek újrarenderelését biztosítja az adatok frissítése után.
- `areas`: A közterületek adatait tárolja, amelyeket a backend API-ból tölt be.

2. Adatok frissítése

- `update`: Egy függvény, amely a `refresh` állapotot változtatja, ezzel biztosítva az adatok újratöltését.

3. API-hívások kezelése

- `backendMuvelet`: Általános függvény az API-hívások kezelésére (pl. POST, PUT, DELETE).
 - Paraméterek: adat, HTTP-módszer, URL, fejléc, sikeres- és hibaüzenetek.
 - Sikeres művelet esetén visszajelzést (`toast.success`) jelenít meg, és frissíti az adatokat.
- `backendMuveletFile`: Fájlok kezelésére szolgáló függvény, amely az `axios` könyvtárat használja.
- `fetchData`: Adatok lekérésére szolgáló függvény, amely a megadott URL-ről tölti be az adatokat, és frissíti az állapotot.
- `deleteItem`: Törlési műveletek kezelésére szolgáló függvény.

4. Közterületek betöltése

- A `useEffect` hook segítségével a komponens a backend API-ból (`/publicareaname`) betölti a közterületek adatait, és azokat a `areas` állapotban tárolja.

5. Visszajelzés a felhasználónak

- A `toast` könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a sikeres vagy sikertelen műveletekről.



Felépítés

1. Kontextus létrehozása

- A [CrudContext](#) a [createContext](#) segítségével jön létre.
- A [CrudProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális függvények

- A kontextusban elérhető függvények:
 - [backendMuvelet](#)
 - [backendMuveletFile](#)
 - [fetchData](#)
 - [deleteItem](#)
 - [update](#)

3. Adatok megosztása

- A [CrudContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a CRUD műveletekhez szükséges logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Hatókonyság:** Az állapotkezelés és az API-hívások központosítása csökkenti a kódisméltést.



Összegzés

A [CrudContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a CRUD műveletek kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a visszajelzések központosítása egyszerűsíti az alkalmazás fejlesztését és karbantartását.

InitialContext.jsx – Alapértelmezett adatok kezelése kontextusban

Az [InitialContext.jsx](#) egy globális kontextus, amely az alkalmazás alapértelmezett adatait kezeli, például közterületek, kategóriák, termékek és csomagautomaták adatait. Ez a kontextus biztosítja az állapotkezelést és az adatok dinamikus frissítését a különböző komponensek számára.

Főbb funkciók

1. Állapotkezelés

- **[areas](#)**: A közterületek adatait tárolja, amelyeket a backend API-ból tölt be.
- **[categories](#)**: A kategóriák adatait tárolja.
- **[products](#)**: A termékek adatait tárolja.
- **[lockers](#)**: A csomagautomaták adatait tárolja.
- **[refresh](#)**: Egy állapot, amely az adatok frissítését biztosítja.
- **[refreshPublicAreaName](#)**: Egy állapot, amely kifejezetten a közterületek adatainak frissítésére szolgál.

2. Adatok frissítése

- **[update](#)**: Egy függvény, amely a [refresh](#) állapotot változtatja, ezzel biztosítva az adatok újrátöltését.
- **[updatePublicAreaName](#)**: Egyfüggvény, amely a [refreshPublicAreaName](#) állapotot változtatja, ezzel frissítve a közterületek adatait.



3. Adatok betöltése

- A `useEffect` hook segítségével a komponens a backend API-ból tölti be az adatokat:
 - **Közterületek:** A `/publicareaname` végpontról.
 - **Kategóriák:** A `/category` végpontról.
 - **Termékek:** A `/product` végpontról.
 - **Csomagautomaták:** A `/locker` végpontról.
- Az adatok betöltése dinamikusan történik a `refresh` vagy `refreshPublicAreaName` állapot változásakor.

4. Hiba- és visszajelzskezelés

- Ha hiba történik az adatok betöltésekor, a hibaüzenet a konzolra kerül (`console.error`), vagy figyelmeztetés jelenik meg (`alert`).

Felépítés

1. Kontextus létrehozása

- Az `InitialContext` a `createContext` segítségével jön létre.
- Az `InitialProvider` biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - `areas`, `categories`, `products`, `lockers`
 - `updatePublicAreaName`, `refreshPublicAreaName`
 - `refresh`, `setRefresh`, `update`

3. Adatok megosztása

- Az `InitialContext.Provider` biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.



Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy az alapértelmezett adatokhoz való hozzáférés és azok frissítése egyszerűen elérhető legyen a gyermekkomponensek számára.
- **Dinamikus frissítés:** Az állapotok változásával az adatok automatikusan frissülnek, így minden naprakészek.
- **Hibakezelés:** A hibaüzenetek biztosítják, hogy az esetleges problémák könnyen azonosíthatók legyenek.

Összegzés

Az [InitialContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja az alapértelmezett adatok kezelését és frissítését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti az alkalmazás fejlesztését és karbantartását.



AddressContext.jsx – Címek és közterületek kezelése kontextusban

Az [AddressContext.jsx](#) egy globális kontextus, amely a felhasználói címek és közterületek kezelésére szolgál. Ez a kontextus biztosítja az állapotkezelést, az API-hívásokat és a visszajelzéseket a címekkel kapcsolatos műveletekhez.

Főbb funkciók

1. Állapotkezelés

- [address](#): A felhasználóhoz tartozó címeket tárolja.
- [areas](#): A közterületek adatait tárolja, amelyeket a backend API-ból tölt be.
- [refresh](#): Egy állapot, amely az adatok frissítését biztosítja.

2. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot változtatja, ezzel biztosítva az adatok újratöltését.

3. Címek betöltése

- A [useEffect](#) hook segítségével a komponens a backend API-ból tölti be a felhasználóhoz tartozó címeket:
 - A /address/{user.id} végpontról.
 - A kérés tartalmazza a felhasználói azonosítót ([user.id](#)) és a hitelesítési tokent (Authorization).
- Hiba esetén hibaüzenet ([toast.error](#)) jelenik meg.

4. Közterületek betöltése

- A közterületek adatai a /publicareaname végpontról kerülnek betöltésre.
- Az adatok a [areas](#) állapotban tárolódnak.

5. API-hívások kezelése

- [backendMuvelet](#): Általános függvény az API-hívások kezelésére (pl. POST, PUT, DELETE).
 - Paraméterek: adat, HTTP-módszer, URL, fejléc.
 - Sikeres művelet esetén visszajelzést ([toast.success](#)) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet ([toast.error](#)) jelenik meg.



Felépítés

1. Kontextus létrehozása

- Az [AddressContext](#) a [createContext](#) segítségével jön létre.
- Az [AddressProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - [areas](#), [address](#), [refresh](#)
 - [backendMuvelet](#), [update](#), [setAddress](#)

3. Adatok megosztása

- Az [AddressContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a címekkel és közterületekkel kapcsolatos adatokhoz való hozzáférés és azok frissítése egyszerűen elérhető legyen a gyermekkomponensek számára.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Hibakezelés:** A hibaüzenetek biztosítják, hogy az esetleges problémák könnyen azonosíthatók legyenek.

Összegzés

Az [AddressContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a címek és közterületek kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti az alkalmazás fejlesztését és karbantartását.



AdminController.jsx – Adminisztrációs műveletek kezelése kontextusban

Az [AdminController.jsx](#) egy globális kontextus, amely az adminisztrációs műveletek kezelésére szolgál, például felhasználók törlésére, módosítására és az adatok frissítésére. Ez a kontextus biztosítja az állapotkezelést, az API-hívásokat és a visszajelzéseket az adminisztrációs feladatokhoz.

Főbb funkciók

1. Állapotkezelés

- [users](#): A regisztrált felhasználók adatait tárolja.
- [refresh](#): Egy állapot, amely az adatok frissítését biztosítja.

2. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot váltogatja, ezzel biztosítva az adatok újratöltését.

3. Felhasználók törlése

- [deleteUser](#): Egy aszinkron függvény, amely a megadott felhasználói azonosító ([id](#)) alapján törli a felhasználót a backend API-n keresztül.
 - Sikeres törlés esetén visszajelzést ([toast.success](#)) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet ([toast.error](#)) jelenik meg.

4. Felhasználók módosítása

- [backendMuvelet](#): Egy általános függvény, amely a felhasználók módosítására szolgál.
 - Paraméterek: adat, HTTP-módszer, URL.
 - Sikeres módosítás esetén visszajelzést ([toast.success](#)) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet ([toast.error](#)) jelenik meg.

5. Hitelesítés kezelése

- A [sessionStorage](#) segítségével a komponens hozzáfér a felhasználói tokenhez ([usertoken](#)), amelyet az API-hívások során az Authorization fejlécben használ.



Felépítés

1. Kontextus létrehozása

- Az [AdminContext](#) a [createContext](#) segítségével jön létre.
- Az [AdminProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - [users](#), [update](#), [setUsers](#)
 - [deleteUser](#), [backendMuvelet](#)

3. Adatok megosztása

- Az [AdminContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy az adminisztrációs műveletekhez szükséges logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Hibakezelés:** A hibaüzenetek biztosítják, hogy az esetleges problémák könnyen azonosíthatók legyenek.

Összegzés

Az [AdminContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja az adminisztrációs műveletek kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti az adminisztrációs feladatok végrehajtását és karbantartását.



AuthContext.jsx – Felhasználói hitelesítés és jogosultságok kezelése kontextusban

Az [AuthContext.jsx](#) egy globális kontextus, amely a felhasználói hitelesítést, a jogosultságok kezelését és a profiladatok kezelését biztosítja az alkalmazás különböző részein. Ez a kontextus központosítja a hitelesítéssel kapcsolatos logikát, és egyszerű hozzáférést biztosít a gyermekkomponensek számára.

Főbb funkciók

1. Állapotkezelés

- [user](#): A bejelentkezett felhasználó adatait tárolja.
- [isLoggedIn](#): Egy állapot, amely jelzi, hogy a felhasználó be van-e jelentkezve.
- [profile](#): A felhasználó profiladatait tárolja.
- [roles](#): A felhasználóhoz tartozó jogosultságokat tárolja.
- [refresh](#): Egy állapot, amely az adatok frissítését biztosítja.

2. Adatok betöltése

- **Profiladatok:** A [useEffect](#) hook segítségével a komponens a /profile/index végpontról tölti be a felhasználó profiladatait.
- **Jogosultságok:** A [useEffect](#) hook a /role végpontról tölti be a felhasználóhoz tartozó jogosultságokat.

3. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot váltogatja, ezzel biztosítva az adatok újratöltését.

4. Kijelentkezés

- [logout](#): Egy függvény, amely törli a felhasználói adatokat a [sessionStorage](#)-ből és a [secureStorage](#)-ből, majd frissíti az állapotokat.
 - Törli a felhasználói tokent, a profiladatokat, a jogosultságokat és a kosár tartalmát.
 - Sikeres kijelentkezés esetén visszajelzést ([toast.success](#)) jelenít meg.



5. API-hívások kezelése

- backendMuveletRole: Egy általános függvény, amely a jogosultságokkal kapcsolatos API-hívásokat kezeli.
 - Paraméterek: adat, HTTP-módszer, URL, fejléc.
 - Sikeres művelet esetén visszajelzést (toast.success) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet (toast.error) jelenik meg.

Felépítés

1. Kontextus létrehozása

- Az AuthContext a createContext segítségével jön létre.
- Az AuthProvider biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - user, isLoggedIn, profile, roles
 - refresh, update, logout, backendMuveletRole

3. Adatok megosztása

- Az AuthContext.Provider biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a hitelesítéssel és jogosultságokkal kapcsolatos logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Biztonság:** A secureStorage használata biztosítja a felhasználói adatok biztonságos tárolását.



Összegzés

Az [AuthContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a felhasználói hitelesítés és jogosultságok kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti a hitelesítéssel kapcsolatos feladatok végrehajtását és karbantartását.



CartContext.jsx – Kosárkezelés kontextusban

A [CartContext.jsx](#) egy globális kontextus, amely a kosár kezelésére szolgál az alkalmazásban. Ez a kontextus biztosítja a kosárhoz kapcsolódó műveletek, például termékek hozzáadása, eltávolítása, a kosár kiürítése és az összeg kiszámítása funkcióit.

Főbb funkciók

1. Állapotkezelés

- [cartItems](#): A kosárban lévő termékeket tárolja. Az adatok a [sessionStorage](#)-ben is mentésre kerülnek, hogy a kosár tartalma megmaradjon az oldal frissítése után.

2. Termékek hozzáadása a kosárhoz

- [addToCart](#): Egy terméket ad a kosárhoz. Ha a termék már létezik a kosárban az adott csomagautomatával ([lockerId](#)), növeli a mennyiséget. Ha nem, új térelként adja hozzá.
- Sikeres hozzáadás esetén visszajelzést ([toast.success](#)) jelenít meg.

3. Termékek eltávolítása a kosárból

- [removeFromCart](#): Egy terméket távolít el a kosárból. Ha a termék mennyisége 1, teljesen eltávolítja a kosárból. Ha több, csökkenti a mennyiséget.

4. Kosár kiürítése

- [clearCart](#): Teljesen kiüríti a kosár tartalmát.

5. Kosár összegének kiszámítása

- [getCartTotal](#): Kiszámítja a kosár teljes összegét a termékek napi ára ([price_per_day](#)) és mennyisége alapján.

6. Adatok mentése és betöltése

- A [useEffect](#) segítségével a kosár tartalma automatikusan mentésre kerül a [sessionStorage](#)-be, és az oldal betöltésekor visszatöltődik.
-



Felépítés

1. Kontextus létrehozása

- A [CartContext](#) a [createContext](#) segítségével jön létre.
- A [CartProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - [cartItems](#), [setCartItems](#)
 - [addToCart](#), [removeFromCart](#), [clearCart](#), [getCartTotal](#)

3. Adatok megosztása

- A [CartContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a kosárkezeléssel kapcsolatos logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a kosár műveleteiről.
- **Tartósság:** A [sessionStorage](#) használata biztosítja, hogy a kosár tartalma megmaradjon az oldal frissítése után.

Összegzés

A [CartContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a kosárkezelés funkcióit az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti a kosárhoz kapcsolódó feladatok végrehajtását és karbantartását.



OrderContext.jsx – Rendelések kezelése kontextusban

Az [OrderContext.jsx](#) egy globális kontextus, amely a rendelési folyamat kezelésére szolgál. Ez a kontextus biztosítja a rendelési adatok, a fizetési módok, a szállítási címek és a kosár tartalmának kezelését, valamint az új rendelések létrehozását.

Főbb funkciók

1. Állapotkezelés

- [formData](#): A rendelési adatok tárolására szolgál, például város, irányítószám, utca, házszám, és felhasználói azonosító.
- [formDataPayment](#): A fizetési mód adatait tárolja, például a kártya típusát.
- [formDataAddress](#): A szállítási cím adatait tárolja.
- [cartItems](#): A kosár tartalmát tárolja, amelyet a [CartContext](#)-ből vesz át.
- [isPrivacyInfo](#): Egy állapot, amely jelzi, hogy az adatvédelmi tájékoztató megvan-e nyitva.

2. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot változtatja, ezzel biztosítva az adatok újratöltését.

3. Rendelési adatok előkészítése

- [formObjPayment](#): A fizetési mód adatait tartalmazó objektum.
- [formObjAddress](#): A szállítási cím adatait tartalmazó objektum.
- [formObjCartItems](#): A kosár tartalmát tartalmazó objektum, amely tartalmazza a termékek mennyiségét, árát, és a csomagautomata azonosítóját.

4. Rendelés létrehozása

- [submitOrder](#): Új rendelést hoz létre, ha a felhasználó nem adott meg külön szállítási címet.
- [submitOrderisAddress](#): Új rendelést hoz létre, ha a felhasználó megadott külön szállítási címet.
- Mindkét függvény a [backendOrder](#) segítségével küldi el az adatokat a backend API-nak.



5. Adatvédelmi tájékoztató kezelése

- [openPrivacyInfo](#): Megnyitja az adatvédelmi tájékoztatót.
- [closePrivacyInfo](#): Bezárja az adatvédelmi tájékoztatót.

6. API-hívások kezelése

- [backendOrder](#): Általános függvény az API-hívások kezelésére. Sikeres művelet esetén visszajelzést ([toast.success](#)) jelenít meg, hiba esetén hibaüzenetet ([toast.error](#)).

Felépítés

1. Kontextus létrehozása

- Az [OrderContext](#) a [createContext](#) segítségével jön létre.
- Az [OrderProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - [formData](#), [formDataPayment](#), [formDataAddress](#), [cartItems](#)
 - [submitOrder](#), [submitOrderIsAddress](#), [openPrivacyInfo](#), [closePrivacyInfo](#)

3. Adatok megosztása

- Az [OrderContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a rendelési folyamathoz szükséges logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a rendelési műveletekről.
- **Biztonság:** Az adatvédelmi tájékoztató kezelése biztosítja, hogy a felhasználók tisztaiban legyenek az adataik kezelésével.



Összegzés

Az [OrderContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a rendelési folyamat kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti a rendelési folyamat végrehajtását és karbantartását.

PaymentContext.jsx – Fizetési módok kezelése kontextusban

A [PaymentContext.jsx](#) egy globális kontextus, amely a fizetési módok kezelésére szolgál. Ez a kontextus biztosítja a fizetési módok betöltését, frissítését és az ezekkel kapcsolatos API-hívások kezelését az alkalmazás különböző részein.

Főbb funkciók

1. Állapotkezelés

- [payments](#): A backend API-ból betöltött fizetési módokat tárolja.
- [refresh](#): Egy állapot, amely az adatok frissítését biztosítja.

2. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot változtatja, ezzel biztosítva az adatok újratöltését.

3. Fizetési módok betöltése

- A [useEffect](#) hook segítségével a komponens a /payment végpontról tölti be a fizetési módokat.
- Az adatok a [payments](#) állapotban tárolódnak.
- Hiba esetén hibaüzenet ([alert](#)) jelenik meg.

4. API-hívások kezelése

- [backendMuvelet](#): Általános függvény az API-hívások kezelésére (pl. POST, PUT, DELETE).
 - Paraméterek: adat, HTTP-módszer, URL, fejléc.
 - Sikeres művelet esetén visszajelzést ([toast.success](#)) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet ([toast.error](#)) jelenik meg.



Felépítés

1. Kontextus létrehozása

- A [PaymentContext](#) a [createContext](#) segítségével jön létre.
- A [PaymentProvider](#) biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - [payments](#), [setPayments](#)
 - [refresh](#), [update](#)
 - [backendMuvelet](#)

3. Adatok megosztása

- A [PaymentContext.Provider](#) biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.

Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a fizetési módokkal kapcsolatos logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A [toast](#) könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Hibakezelés:** A hibaüzenetek biztosítják, hogy az esetleges problémák könnyen azonosíthatók legyenek.

Összegzés

A [PaymentContext.jsx](#) egy hatékony és jól strukturált kontextus, amely biztosítja a fizetési módok kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti a fizetési módokkal kapcsolatos feladatok végrehajtását és karbantartását.



ServiceContext.jsx – Szolgáltatások és adatok kezelése kontextusban

A [ServiceContext.jsx](#) egy globális kontextus, amely az alkalmazás különböző szolgáltatásainak és adatkategóriáinak kezelésére szolgál. Ez a kontextus biztosítja a kategóriák, termékek és csomagautomaták kezelését, valamint az ezekkel kapcsolatos API-hívások kezelését.

Főbb funkciók

1. Állapotkezelés

- [categories](#): A backend API-ból betöltött kategóriákat tárolja.
- [products](#): A termékek adatait tárolja.
- [lockers](#): A csomagautomaták adatait tárolja.
- [refresh](#): Egy állapot, amely az adatok frissítését biztosítja.

2. Adatok frissítése

- [update](#): Egy függvény, amely a [refresh](#) állapotot váltogatja, ezzel biztosítva az adatok újratöltését.

3. Adatok betöltése

- A [useEffect](#) hook segítségével a komponens a backend API-ból tölti be az adatokat:
 - **Kategóriák:** A [/category](#) végpontról.
 - **Termékek:** A [/product](#) végpontról.
 - **Csomagautomaták:** A [/locker](#) végpontról.
- Az adatok a megfelelő állapotokban tárolódnak.
- Hiba esetén hibaüzenet ([alert](#)) jelenik meg.



4. API-hívások kezelése

- backendMuvelet: Általános függvény az API-hívások kezelésére (pl. POST, PUT, DELETE).
 - Paraméterek: adat, HTTP-módszer, URL, fejléc.
 - Sikeres művelet esetén visszajelzést (toast.success) jelenít meg, és frissíti az adatokat.
 - Hiba esetén hibaüzenet (toast.error) jelenik meg.
- backendMuveletFile: Fájlok kezelésére szolgáló függvény, amely az API-hívások során fájlokat küld a backendnek.

Felépítés

1. Kontextus létrehozása

- A ServiceContext a createContext segítségével jön létre.
- A ServiceProvider biztosítja a kontextus értékeit az alkalmazás gyermekkomponensei számára.

2. Globális állapotok és függvények

- A kontextusban elérhető állapotok és függvények:
 - categories, products, lockers
 - setCategories, setProducts, setLockers
 - refresh, update
 - backendMuvelet, backendMuveletFile

3. Adatok megosztása

- A ServiceContext.Provider biztosítja a kontextus értékeit, amelyeket a gyermekkomponensek elérhetnek.



Felhasználói élmény

- **Egyszerű használat:** A kontextus biztosítja, hogy a kategóriák, termékek és csomagautomaták kezeléséhez szükséges logika központosítva legyen, így a gyermekkomponensek egyszerűen használhatják azokat.
- **Visszajelzés:** A toast könyvtár segítségével a felhasználók azonnali visszajelzést kapnak a műveletekről.
- **Hibakezelés:** A hibaüzenetek biztosítják, hogy az esetleges problémák könnyen azonosíthatók legyenek.

Összegzés

A ServiceContext.jsx egy hatékony és jól strukturált kontextus, amely biztosítja a kategóriák, termékek és csomagautomaták kezelését az alkalmazás különböző részein. Az állapotkezelés, az API-hívások és a dinamikus frissítés egyszerűsíti az adatokkal kapcsolatos feladatok végrehajtását és karbantartását.



Register.cy.js – Regisztrációs tesztek Cypress-szel

A [Register.cy.js](#) egy Cypress tesztfájl, amely a regisztrációs folyamatot teszteli az alkalmazásban. A fájl tartalmaz egyedi regisztrációs teszteket, valamint több felhasználóval végzett tömeges regisztrációs teszteket.

Főbb funkciók

1. Egyedi regisztráció tesztelése

- A Sikeres regisztráció végrehajtása teszt egy egyedi email címet generál minden futtatáskor a [Date.now\(\)](#) segítségével.
- A teszt lépései:
 - Megnyitja a regisztrációs oldalt ([cy.visit\(\)](#)).
 - Kitölți a regisztrációs űrlapot (vezetéknév, keresztnév, email, jelszó, jelszó megerősítése).
 - Beküldi az űrlapot ([cy.get\('form'\).submit\(\)](#)).
 - Ellenörzi, hogy a sikeres regisztráció üzenet megjelenik-e ([cy.contains\('Sikeres regisztráció'\).should\('exist'\)](#)).

2. Tömeges regisztráció tesztelése

- A Regisztráció 10 valószerű felhasználóval teszt egy tömbben ([users](#)) tárolja a felhasználói adatokat.
- A [forEach](#) metódus segítségével minden felhasználóra külön tesztet futtat.
- A teszt lépései:
 - Egyedi email címet generál minden felhasználóhoz a [Date.now\(\)](#) segítségével.
 - Megnyitja a regisztrációs oldalt.
 - Kitölți az űrlapot a felhasználó adataival.
 - Beküldi az űrlapot.
 - Ellenörzi, hogy a sikeres regisztráció üzenet megjelenik.



Felépítés

1. **Egyedi regisztráció teszt**
 - Egyetlen felhasználó regisztrációját teszteli dinamikusan generált email címmel.
 2. **Tömeges regisztráció teszt**
 - Egy tömbben tárolt 10 felhasználó adatait használja.
 - minden felhasználóhoz külön tesztet futtat, amely biztosítja, hogy a regisztrációs folyamat minden esetben megfelelően működik.
-

Felhasználói élmény

- **Automatizált tesztelés:** A Cypress biztosítja, hogy a regisztrációs folyamatot gyorsan és megbízhatóan lehessen tesztelni.
 - **Egyedi email címek:** A dinamikusan generált email címek biztosítják, hogy minden tesztfutás egyedi legyen, és ne ütközzön meglévő adatokkal.
 - **Tömeges tesztelés:** A tömeges regisztrációs tesztek biztosítják, hogy a rendszer nagyobb mennyiségű adatot is megfelelően kezeljen.
-

Összegzés

A [Register.cy.js](#) egy hatékony Cypress tesztfájl, amely biztosítja a regisztrációs folyamat helyes működését egyedi és tömeges tesztelési forgatókönyvekben. Az automatizált tesztek segítenek az esetleges hibák gyors azonosításában és javításában.



Login_test.cy.js – Bejelentkezési tesztek Cypress-szel

A `Login_test.cy.js` egy Cypress tesztfájl, amely a bejelentkezési folyamatot teszteli az alkalmazásban. Ez a fájl biztosítja, hogy a bejelentkezési funkció megfelelően működjön, és a felhasználók sikeresen hozzáférjenek a rendszerhez.

Főbb funkciók

1. Oldal betöltése

- A `beforeEach` hook minden teszt előtt betölti a bejelentkezési oldalt (`cy.visit('https://flexistore.hu/login2')`).

2. Bejelentkezési teszt

- A Működik-e a belépés? teszt ellenőrzi, hogy a bejelentkezési folyamat megfelelően működik-e:
 - Kitölte az email mezőt (`cy.get('input#email').type('SuperAdmin@example.com')`).
 - Kitölti a jelszó mezőt (`cy.get('input#password').type('test1234')`).
 - Beküldi az űrlapot (`cy.get('form').submit()`).

Felépítés

1. Bejelentkezési oldal betöltése

- A `beforeEach` hook biztosítja, hogy minden teszt a bejelentkezési oldalról induljon.

2. Bejelentkezési folyamat tesztelése

- A teszt egy előre definiált felhasználói email címet és jelszót használ a bejelentkezéshez.
- Ellenőrzi, hogy a bejelentkezési folyamat hibamentesen lefut-e.



Felhasználói élmény

- **Automatizált tesztelés:** A Cypress segítségével gyorsan és megbízhatóan ellenőrizhető a bejelentkezési funkció működése.
 - **Egyszerűség:** A teszt minimális lépésekkel tartalmaz, így könnyen érthető és karbantartható.
-

Javaslatok a teszt bővítésére

- **Sikeres bejelentkezés ellenőrzése:** Ellenőrizhető, hogy a bejelentkezés után a felhasználó átirányításra kerül-e a megfelelő oldalra (pl. `cy.url().should('include', '/dashboard')`).
 - **Hibás bejelentkezés tesztelése:** Érdemes hozzáadni egy tesztet, amely hibás email vagy jelszó esetén ellenőrzi a hibaüzenet megjelenését.
 - **Több felhasználói szerepkör tesztelése:** Tesztelhető különböző jogosultságú felhasználók (pl. admin, normál felhasználó) bejelentkezése.
-

Összegzés

A `Login_test.cy.js` egy alapvető Cypress tesztfájl, amely a bejelentkezési funkció helyes működését ellenőrzi. A teszt egyszerűsége lehetővé teszi a gyors futtatást és a hibák azonosítását. A javasolt bővítményekkel a teszt még átfogóbbá és robusztusabbá tehető.



Full_User_Process.cy.js – Teljes felhasználói folyamat tesztelése Cypress-szel

A `Full_User_Process.cy.js` egy átfogó Cypress tesztfájl, amely a felhasználói folyamat különböző lépéseinak tesztelését leírja. A fájl beleértve a regisztrációt, bejelentkezést, termékek kosárba helyezését, és a rendelés véglegesítését. Ez a fájl biztosítja, hogy az alkalmazás kulcsfontosságú funkciói megfelelően működjenek.

Főbb funkciók

1. Regisztráció tesztelése

- A Sikeres regisztráció végrehajtása teszt egy új felhasználót regisztrál:
 - Kitölti a regisztrációs űrlapot (vezetéknév, keresztnév, email, jelszó, jelszó megerősítése).
 - Beküldi az űrlapot.
 - Ellenőrzi, hogy a sikeres regisztráció üzenet megjelenik.

2. Bejelentkezés tesztelése

- A Működik-e a belépés? teszt ellenőrzi, hogy a regisztrált felhasználó sikeresen be tud-e jelentkezni:
 - Kitölti az email és jelszó mezőket.
 - Beküldi az űrlapot.
 - Ellenőrzi, hogy a bejelentkezés után az URL nem tartalmazza a /login2 útvonalat.

3. Termék hozzáadása a kosárhoz

- A Bejelentkezés és első termék kosárba tétele teszt:
 - Bejelentkezik a felhasználóval.
 - Navigál a termékek oldalára.
 - Hozzáad egy terméket a kosárhoz.
 - Ellenőrzi, hogy a termék sikeresen hozzáadásra került.



4. Rendelés véglegesítése

- A teszt egy rendelést hoz létre:
 - Kitölti a szállítási cím mezőket (város, irányítószám, utca, házszám).
 - Kiválasztja a közterület jellegét (dropdown).
 - Beküldi a rendelést.
 - Ellenőrzi, hogy a rendelés véglegesítése sikeres volt.
-

Felépítés

1. Regisztráció

- Egyedi email cím generálása minden tesztfutáshoz.
- Az űrlapmezők kitöltése és a regisztrációs folyamat ellenőrzése.

2. Bejelentkezés

- A regisztrált felhasználóval történő bejelentkezés tesztelése.

3. Kosárkezelés

- Termékek hozzáadása a kosárhoz.
- A kosár tartalmának ellenőrzése.

4. Rendelés létrehozása

- Szállítási cím és fizetési adatok megadása.
 - A rendelés véglegesítése.
-

Felhasználói élmény

- **Automatizált tesztelés:** A Cypress segítségével a teljes felhasználói folyamat gyorsan és megbízhatóan tesztelhető.
 - **Egyedi adatok generálása:** A dinamikusan generált email címek és címadatok biztosítják, hogy minden tesztfutás egyedi legyen.
 - **Átfogó tesztelés:** A fájl lefedi az alkalmazás kulcsfontosságú funkcióit, biztosítva azok helyes működését.
-



Javaslatok a teszt bővítésére

- Hibás regisztráció tesztelése:** Ellenőrizhető, hogy hibás adatok esetén megfelelő hibaüzenetek jelennek meg.
- Több termék hozzáadása:** Tesztelhető, hogy több termék hozzáadása a kosárhoz helyesen működik.
- Rendelési visszaigazolás ellenőrzése:** A rendelés véglegesítése után ellenőrizhető, hogy a visszaigazoló oldal vagy üzenet megjelenik.

Összegzés

A `Full_User_Process.cy.js` egy átfogó Cypress tesztfájl, amely a felhasználói folyamat minden fontos lépését lefedi. Az automatizált tesztek segítenek az esetleges hibák gyors azonosításában és javításában, biztosítva az alkalmazás megbízhatóságát.

The screenshot shows a dark-themed Cypress test runner interface. At the top, it displays the file name `Full_User_Process.cy.js` next to a document icon, and a timer showing `00:27`. Below the title, there are two collapsed test case sections. The first section is titled `Regisztráció teszt` and contains one test step: `Sikeres regisztráció végrehajtása`, which is marked as successful with a green checkmark. The second section is titled `Bejelentkezés és rendelés teszteletése!Működik-e?` and contains one test step: `Teljes rendelés végrehajtás!`, also marked as successful with a green checkmark. The overall status of the test run is indicated by a large green checkmark at the bottom left of the interface.

10. ábra Cypress test success



0001_01_01_000000_create_roles_table.php – Laravel migráció a szerepkörök táblájához

Ez a Laravel migrációs fájl a roles tábla létrehozására szolgál az adatbázisban. A tábla a felhasználói szerepköröket és azok jogosultsági szintjeit tárolja.

Főbb funkciók

1. Tábla létrehozása (up metódus)

- A Schema::create metódus segítségével létrejön a roles nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **power:** Egyedi jogosultsági szint, alapértelmezett értéke 11.
 - **warrant_name:** A szerepkör neve, például "Admin" vagy "Felhasználó", szintén egyedi.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a roles táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős a tábla létrehozásáért.
- A power oszlop egyedi, és alapértelmezett értéket kap, amely a jogosultsági szintet jelöli.
- A warrant_name oszlop szintén egyedi, és a szerepkör nevét tárolja.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a roles táblát.



Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a roles táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a roles táblát.

Felhasználási esetek

- **Jogosultságkezelés:**

A roles tábla lehetővé teszi a felhasználók szerepkörök szerinti csoportosítását és jogosultsági szintek kezelését.

- **Adminisztrációs feladatok:**

Az adminisztrátorok különböző szerepköröket hozhatnak létre, például "Admin", "Moderátor", "Felhasználó".

Összegzés

Ez a migrációs fájl egy alapvető, de fontos része a jogosultságkezelésnek egy Laravel-alapú alkalmazásban. A roles tábla lehetővé teszi a felhasználói szerepkörök és jogosultsági szintek kezelését, amely elengedhetetlen egy biztonságos és jól strukturált rendszerhez.



0001_01_01_000001_create_users_table.php – Laravel migráció a felhasználók táblájához

Ez a Laravel migrációs fájl a users tábla és a hozzá kapcsolódó táblák (password_reset_tokens, sessions) létrehozására szolgál az adatbázisban. A users tábla a felhasználók alapvető adatait és jogosultságait tárolja.

Főbb funkciók

1. Felhasználók tábla létrehozása (users)

- A Schema::create metódus segítségével létrejön a users nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **first_name:** A felhasználó keresztneve.
 - **last_name:** A felhasználó vezetékneme.
 - **email:** Egyedi email cím.
 - **email_verified_at:** Az email cím hitelesítésének időbélyege (nullable).
 - **password:** A felhasználó jelszava.
 - **role_id:** Külső kulcs a roles táblára, amely a felhasználó szerepkörét jelöli (alapértelmezett: 1).
 - **remember_token:** A "remember me" funkcióhoz használt token.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Jelszó-visszaállítási tokenek tábla (password_reset_tokens)

- A tábla oszlopai:
 - **email:** Az email cím, amelyhez a token tartozik (elsődleges kulcs).
 - **token:** A jelszó-visszaállítási token.
 - **created_at:** A token létrehozásának időbélyege.



4. Munkamenetek tábla (sessions)

- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, amely a munkamenet azonosítója.
 - **user_id:** Külső kulcs a users táblára, amely a munkamenethez tartozó felhasználót jelöli.
 - **ip_address:** A felhasználó IP-címe.
 - **user_agent:** A felhasználó böngészőjének adatai.
 - **payload:** A munkamenet adatai.
 - **last_activity:** Az utolsó aktivitás időbélyege.

5. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a users, password_reset_tokens és sessions táblákat, ha azok léteznek.

Felépítés

1. up metódus

- Ez a metódus felelős a táblák létrehozásáért.
- A users tábla tartalmazza a felhasználók alapvető adatait és a szerepkörökre vonatkozó külső kulcsot.
- A password_reset_tokens és sessions táblák a jelszó-visszaállítási és munkamenet-kezelési funkciókhoz szükségesek.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a users, password_reset_tokens és sessions táblákat.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a users, password_reset_tokens és sessions táblákat az adatbázisban.



- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a táblákat.

Felhasználási esetek

- **Felhasználók kezelése:**

A users tábla tárolja a felhasználók alapvető adatait, például nevüket, email címüket és jelszavukat.

- **Jelszó-visszaállítás:**

A password_reset_tokens tábla lehetővé teszi a jelszó-visszaállítási funkció implementálását.

- **Munkamenet-kezelés:**

A sessions tábla tárolja a felhasználói munkamenetek adatait, például az IP-címet és a böngésző információit.

Összegzés

Ez a migrációs fájl a felhasználók kezeléséhez szükséges alapvető táblákat hozza létre. A users tábla a felhasználói adatok tárolására szolgál, míg a password_reset_tokens és sessions táblák a jelszó-visszaállítási és munkamenet-kezelési funkciókat támogatják. Ez a migráció elengedhetetlen egy biztonságos és jól strukturált felhasználókezelési rendszerhez.



2025_01_31_233753_create_profiles_table.php – Laravel migráció a profilok táblájához

Ez a Laravel migrációs fájl a profiles tábla létrehozására szolgál az adatbázisban. A tábla a felhasználókhoz kapcsolódó profiladatokat, például fájlneveket és fájlútvonalakat tárolja.

Főbb funkciók

1. Profilok tábla létrehozása (profiles)

- A Schema::create metódus segítségével létrejön a profiles nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **user_id:** Külső kulcs a users táblára, amely a profilhoz tartozó felhasználót jelöli.
 - **onDelete('cascade')**: Ha a felhasználó törlésre kerül, a hozzá tartozó profil is törlődik.
 - **onUpdate('cascade')**: Ha a felhasználó azonosítója frissül, a profil automatikusan frissül.
 - **file_name:** A profilhoz tartozó fájl neve.
 - **file_path:** A fájl elérési útvonala.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a profiles táblát, ha az létezik.



Felépítés

1. up metódus

- Ez a metódus felelős a profiles tábla létrehozásáért.
- A user_id oszlop külön kulcsként kapcsolódik a users táblához, biztosítva az adatok integritását.
- A file_name és file_path oszlopok a profilhoz tartozó fájlok adatait tárolják.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a profiles táblát.

Felhasználási példa

- Migráció futtatása:

```
php artisan migrate
```

Ez a parancs létrehozza a profiles táblát az adatbázisban.

- Migráció visszavonása:

```
php artisan migrate:rollback
```

Ez a parancs törli a profiles táblát.

Felhasználási esetek

- Felhasználói profilk kezelése:

A profiles tábla lehetővé teszi a felhasználókhöz kapcsolódó fájlok, például profilképek vagy dokumentumok tárolását.

- Kapcsolat a felhasználókkal:

A user_id oszlop biztosítja, hogy minden profil egy adott felhasználóhoz legyen kötve.



Felhasználói élmény

- **Adatintegritás:**
Az onDelete('cascade') és onUpdate('cascade') opciók biztosítják, hogy a profilk automatikusan frissüljenek vagy törlődjenek a felhasználói adatok változásakor.
- **Egyszerű fájlkezelés:**
A file_name és file_path oszlopok lehetővé teszik a fájlok könnyű elérését és kezelését.

Összegzés

Ez a migrációs fájl a felhasználói profilk kezeléséhez szükséges alapvető táblát hozza létre. A profiles tábla lehetővé teszi a felhasználókhöz kapcsolódó fájlok tárolását és kezelését, miközben biztosítja az adatok integritását és a könnyű hozzáférést.



2025_02_01_200058_create_categories_table.php – Laravel migráció a kategóriák táblájához

Ez a Laravel migrációs fájl a categories tábla létrehozására szolgál az adatbázisban. A tábla a termékkategóriák adatait tárolja, amelyek segítenek a termékek rendszerezésében és csoportosításában.

Főbb funkciók

1. Kategóriák tábla létrehozása (categories)

- A Schema::create metódus segítségével létrejön a categories nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **name:** A kategória neve, amely egy szöveges mező.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a categories táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős a categories tábla létrehozásáért.
- A name oszlop tárolja a kategóriák nevét, például "Elektronika", "Ruházat", "Könyvek".

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a categories táblát.



Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a categories táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a categories táblát.

Felhasználási esetek

- **Termékkategóriák kezelése:**

A categories tábla lehetővé teszi a termékek kategóriák szerinti csoportosítását, például webshopokban vagy raktárkezelő rendszerekben.

- **Adminisztrációs feladatok:**

Az adminisztrátorok új kategóriákat hozhatnak létre, módosíthatják vagy törölhetik a meglévőket.

Felhasználói élmény

- **Egyszerű rendszerezés:**

A kategóriák segítenek a termékek logikus csoportosításában, ami javítja a felhasználói élményt.

- **Könnyű bővíthetőség:**

Új kategóriák egyszerűen hozzáadhatók az adminisztrációs felületen keresztül.



Összegzés

Ez a migrációs fájl a kategóriák kezeléséhez szükséges alapvető táblát hozza létre. A categories tábla lehetővé teszi a termékek rendszerezését és csoportosítását, ami elengedhetetlen egy jól strukturált adatbázisban.



2025_02_01_200955_create_lockers_table.php – Laravel migráció a csomagautomaták táblájához

Ez a Laravel migrációs fájl a lockers tábla létrehozására szolgál az adatbázisban. A tábla a csomagautomaták adatait tárolja, például nevüket, címüket és leírásukat.

Főbb funkciók

1. Csomagautomaták tábla létrehozása (lockers)

- A Schema::create metódus segítségével létrejön a lockers nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **locker_name:** A csomagautomata neve, például "Locker 1".
 - **address:** A csomagautomata címe.
 - **description:** Egy szöveges mező, amely a csomagautomata leírását tartalmazza.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a lockers táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős a lockers tábla létrehozásáért.
- A locker_name, address és description oszlopok tárolják a csomagautomaták alapvető adatait.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a lockers táblát.



Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a lockers táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a lockers táblát.

Felhasználási esetek

- **Csomagautomaták kezelése:**

A lockers tábla lehetővé teszi a csomagautomaták adatainak tárolását és kezelését, például egy logisztikai rendszerben.

- **Adminisztrációs feladatok:**

Az adminisztrátorok új csomagautomatákat hozhatnak létre, módosíthatják vagy törölhetik a meglévőket.

Felhasználói élmény

- **Egyszerű rendszerezés:**

A csomagautomaták adatai könnyen elérhetők és kezelhetők az adatbázisból.

- **Könnyű bővíthetőség:**

Új csomagautomaták egyszerűen hozzáadhatók az adminisztrációs felületen keresztül.

Összegzés

Ez a migrációs fájl a csomagautomaták kezeléséhez szükséges alapvető táblát hozza létre. A lockers tábla lehetővé teszi a csomagautomaták adatainak tárolását és rendszerezését, ami elengedhetetlen egy logisztikai vagy csomagkezelő rendszerben.



2025_02_01_200956_create_products_table.php – Laravel migráció a termékek táblájához

Ez a Laravel migrációs fájl a products tábla létrehozására szolgál az adatbázisban. A tábla a termékek adatait tárolja, például fájlneveket, leírásokat, árakat és kategóriákat.

Főbb funkciók

1. Termékek tábla létrehozása (products)

- A Schema::create metódus segítségével létrejön a products nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **file_name:** A termékhez tartozó fájl neve (pl. kép).
 - **file_path:** A fájl elérési útvonala.
 - **name:** A termék neve.
 - **description:** A termék részletes leírása.
 - **price_per_day:** A termék napi ára.
 - **category_id:** Külső kulcs a categories táblára, amely a termék kategóriáját jelöli.
 - **onDelete('cascade')**: Ha a kategória törlésre kerül, a hozzá tartozó termékek is törlődnek.
 - **onUpdate('cascade')**: Ha a kategória azonosítója frissül, a termékek automatikusan frissülnek.
 - **available:** Boolean mező, amely jelzi, hogy a termék elérhető-e.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a products táblát, ha az létezik.
-



Felépítés

1. up metódus

- Ez a metódus felelős a products tábla létrehozásáért.
- A category_id oszlop külső kulcsként kapcsolódik a categories táblához, biztosítva az adatok integritását.
- A file_name, file_path, name, description, és price_per_day oszlopok tárolják a termékek alapvető adatait.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a products táblát.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a products táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a products táblát.

Felhasználási esetek

- **Termékek kezelése:**

A products tábla lehetővé teszi a termékek adatainak tárolását és kezelését, például webshopokban vagy kölcsönző rendszerekben.

- **Kategóriákhoz kötött termékek:**

A category_id oszlop biztosítja, hogy minden termék egy adott kategóriához legyen kötve.



Felhasználói élmény

- **Adatintegritás:**
Az onDelete('cascade') és onUpdate('cascade') opciók biztosítják, hogy a termékek automatikusan frissüljenek vagy törlődjenek a kategóriák változásakor.
 - **Könnyű rendszerezés:**
A termékek adatai könnyen elérhetők és kezelhetők az adatbázisból.
-

Összegzés

Ez a migrációs fájl a termékek kezeléséhez szükséges alapvető táblát hozza létre. A products tábla lehetővé teszi a termékek adatainak tárolását, rendszerezését és kategóriához való kötését, ami elengedhetetlen egy jól strukturált adatbázisban.



2025_02_17_000023_create_street_types_table.php – Laravel migráció az utcanevek típusainak táblájához

Ez a Laravel migrációs fájl a street_types tábla létrehozására szolgál az adatbázisban. A tábla az utcanevek típusait tárolja, például "utca", "tér", "körút", stb., amelyek a címek kezeléséhez szükségesek.

Főbb funkciók

1. Utcanevek típusainak tábla létrehozása (street_types)

- A Schema::create metódus segítségével létrejön a street_types nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **public_area_name:** Az utcanevek típusa, például "utca", "tér", "körút". Ez az oszlop egyedi (unique).
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a street_types táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős a street_types tábla létrehozásáért.
- A public_area_name oszlop biztosítja, hogy minden utcanev-típus egyedi legyen.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a street_types táblát.



Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a street_types táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a street_types táblát.

Felhasználási esetek

- **Címek kezelése:**

A street_types tábla lehetővé teszi az utcanevek típusainak tárolását és kezelését, például egy címkezelő rendszerben.

- **Adatok validálása:**

Az utcanevek típusainak előre definiált listája segít az adatok validálásában és egységesítésében.

Felhasználói élmény

- **Egyszerű rendszerezés:**

Az utcanevek típusai könnyen elérhetők és kezelhetők az adatbázisból.

- **Adatintegritás:**

Az egyedi (unique) oszlop biztosítja, hogy ne legyenek duplikált utcanevek típusai.

Összegzés

Ez a migrációs fájl az utcanevek típusainak kezeléséhez szükséges alapvető táblát hozza létre. A street_types tábla lehetővé teszi az utcanevek típusainak rendszerezését és validálását, ami elengedhetetlen egy jól strukturált címkezelő rendszerben.



2025_02_17_000030_create_address_table.php – Laravel migráció a címek táblájához

Ez a Laravel migrációs fájl az addresses tábla létrehozására szolgál az adatbázisban. A tábla a felhasználókhoz kapcsolódó címadatokat tárolja, például irányítószámot, utcanevet, házszámot, várost és email címet.

Főbb funkciók

1. Címek tábla létrehozása (addresses)

- A Schema::create metódus segítségével létrejön az addresses nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **user_id:** Külső kulcs a users táblára, amely a címhez tartozó felhasználót jelöli.
 - **zip:** Az irányítószám (pozitív egész szám, unsigned).
 - **street:** Az utca neve.
 - **house_number:** A házszám.
 - **city:** A város neve.
 - **email:** Az email cím, amely a címhez tartozik.
 - **street_id:** Külső kulcs a street_types táblára, amely az utca típusát jelöli (pl. "utca", "ter").
 - **onDelete('cascade')**: Ha az utca típusa törlésre kerül, a hozzá tartozó címek is törlődnek.
 - **onUpdate('cascade')**: Ha az utca típusa frissül, a címek automatikusan frissülnek.
 - **timestamps**: Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli az addresses táblát, ha az létezik.



Felépítés

1. up metódus

- Ez a metódus felelős az addresses tábla létrehozásáért.
- A user_id és street_id oszlopok különböző kulcsként kapcsolódnak a users és street_types táblákhoz, biztosítva az adatok integritását.
- Az unsigned típusú zip oszlop biztosítja, hogy csak pozitív számokat lehessen tárolni.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli az addresses táblát.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza az addresses táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli az addresses táblát.

Felhasználási esetek

- **Felhasználói címek kezelése:**

Az addresses tábla lehetővé teszi a felhasználókhoz kapcsolódó címek tárolását és kezelését.

- **Kapcsolat az utca típusokkal:**

A street_id oszlop biztosítja, hogy minden cím egy adott utca típushoz legyen kötve.



Felhasználói élmény

- **Adatintegritás:**

Az onDelete('cascade') és onUpdate('cascade') opciók biztosítják, hogy a címek automatikusan frissüljenek vagy törlődjenek a kapcsolódó adatok változásakor.

- **Egyszerű címkezelés:**

Az addresses tábla lehetővé teszi a címadatok könnyű elérését és kezelését az adatbázisból.

Összegzés

Ez a migrációs fájl a címek kezeléséhez szükséges alapvető táblát hozza létre. Az addresses tábla lehetővé teszi a felhasználókhoz kapcsolódó címek tárolását, rendszerezését és az utca típusokkal való kapcsolatát, ami elengedhetetlen egy jól strukturált címkezelő rendszerben.



2025_02_17_000039_create_payment_methods_table.php – Laravel migráció a fizetési módok táblájához

Ez a Laravel migrációs fájl a payment_methods tábla létrehozására szolgál az adatbázisban. A tábla a különböző fizetési módokat tárolja, például bankkártya típusokat vagy más fizetési lehetőségeket.

Főbb funkciók

1. Fizetési módok tábla létrehozása (payment_methods)

- A Schema::create metódus segítségével létrejön a payment_methods nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **card_type:** A fizetési mód típusa (pl. "Visa", "MasterCard"), amely egyedi (unique).
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a payment_methods táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős a payment_methods tábla létrehozásáért.
- A card_type oszlop biztosítja, hogy minden fizetési mód egyedi legyen.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a payment_methods táblát.



Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a payment_methods táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a payment_methods táblát.

Felhasználási esetek

- **Fizetési módok kezelése:**

A payment_methods tábla lehetővé teszi a különböző fizetési lehetőségek tárolását és kezelését.

- **Adatok validálása:**

Az egyedi (unique) card_type oszlop biztosítja, hogy ne legyenek duplikált fizetési módok.

Felhasználói élmény

- **Egyszerű rendszerezés:**

A fizetési módok adatai könnyen elérhetők és kezelhetők az adatbázisból.

- **Adatintegritás:**

Az egyedi oszlop biztosítja, hogy minden fizetési mód egyedi legyen, elkerülve az ismétlődéseket.

Összegzés

Ez a migrációs fájl a fizetési módok kezeléséhez szükséges alapvető táblát hozza létre. A payment_methods tábla lehetővé teszi a különböző fizetési lehetőségek tárolását és rendszerezését, ami elengedhetetlen egy jól strukturált fizetési rendszerben.



2025_02_17_000041_create_orders_table.php – Laravel migráció a rendelések táblájához

Ez a Laravel migrációs fájl az orders tábla létrehozására szolgál az adatbázisban. A tábla a rendelések adatait tárolja, például a felhasználóhoz tartozó címet, a fizetési módot és a rendelés összegét.

Főbb funkciók

1. Rendelések tábla létrehozása (orders)

- A Schema::create metódus segítségével létrejön az orders nevű tábla.
- A tábla oszlopai:
 - **id**: Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **address_id**: Külső kulcs az addresses táblára, amely a rendeléshez tartozó címet jelöli.
 - **onDelete('cascade')**: Ha a cím törlésre kerül, a hozzá tartozó rendelés is törlődik.
 - **user_id**: Külső kulcs a users táblára, amely a rendelést leadó felhasználót jelöli.
 - **total**: A rendelés teljes összege, decimal típusú (10 számjegy, 2 tizedesjegy).
 - **payments_method_id**: Külső kules a payment_methods táblára, amely a rendelés fizetési módját jelöli.
 - **timestamps**: Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli az orders táblát, ha az létezik.



Felépítés

1. up metódus

- Ez a metódus felelős az orders tábla létrehozásáért.
- A address_id, user_id, és payments_method_id oszlopok külső kulesként kapcsolódnak más táblákhoz, biztosítva az adatok integritását.
- A total oszlop tárolja a rendelés összegét, pontosan két tizedesjeggyel.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli az orders táblát.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza az orders táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs töri az orders táblát.

Felhasználási esetek

- **Rendelések kezelése:**

Az orders tábla lehetővé teszi a rendelések adatainak tárolását és kezelését, például webshopokban vagy szolgáltatási rendszerekben.

- **Kapcsolat a címekkel és fizetési módokkal:**

A address_id és payments_method_id oszlopok biztosítják, hogy minden rendelés egy adott címhez és fizetési módhoz legyen kötve.



Felhasználói élmény

- **Adatintegritás:**
Az onDelete('cascade') opció biztosítja, hogy a kapcsolódó adatok törlése esetén a rendelés is automatikusan töröljön.
 - **Egyszerű rendeléskezelés:**
Az orders tábla lehetővé teszi a rendelések adatainak könnyű elérését és kezelését az adatbázisból.
-

Összegzés

Ez a migrációs fájl a rendelések kezeléséhez szükséges alapvető táblát hozza létre. Az orders tábla lehetővé teszi a rendelések adatainak tárolását, rendszerezését és a kapcsolódó címekkel, felhasználókkal és fizetési módokkal való kapcsolatát, ami elengedhetetlen egy jól strukturált rendeléskezelő rendszerben.



2025_02_17_000044_create_order_items_table.php – Laravel migráció a rendelési tételek táblájához

Ez a Laravel migrációs fájl az order_items tábla létrehozására szolgál az adatbázisban. A tábla a rendelésekhez tartozó tételek adatait tárolja, például a termékeket, azok mennyiségett, árat és a csomagautomatákhoz való kapcsolódást.

Főbb funkciók

1. Rendelési tételek tábla létrehozása (order_items)

- A Schema::create metódus segítségével létrejön az order_items nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **quantity:** A rendelési tétel mennyisége.
 - **item_price:** Az adott termék ára.
 - **line_total:** A rendelési tétel teljes ára (mennyiség × egységár).
 - **product_id:** Külső kulcs a products táblára, amely a rendelési tételhez tartozó terméket jelöli.
 - **onDelete('cascade')**: Ha a termék törlésre kerül, a hozzá tartozó rendelési tétel is törlődik.
 - **onUpdate('cascade')**: Ha a termék azonosítója frissül, a rendelési tétel automatikusan frissül.
 - **order_id:** Külső kulcs az orders táblára, amely a rendeléshez tartozó tételeket jelöli.
 - **onDelete('cascade')**: Ha a rendelés törlésre kerül, a hozzá tartozó tételek is törlődnek.
 - **onUpdate('cascade')**: Ha a rendelés azonosítója frissül, a tételek automatikusan frissülnek.
 - **locker_id:** Külső kulcs a lockers táblára, amely a csomagautomatát jelöli, ahol a termék elérhető.
 - **onDelete('cascade')**: Ha a csomagautomata törlésre kerül, a hozzá tartozó rendelési tétel is törlődik.



- **onUpdate('cascade')**: Ha a csomagautomata azonosítója frissül, a rendelési térel automatikusan frissül.
- **timestamps**: Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli az order_items táblát, ha az létezik.

Felépítés

1. up metódus

- Ez a metódus felelős az order_items tábla létrehozásáért.
- A product_id, order_id, és locker_id oszlopok külső kulcsként kapcsolódnak más táblákhoz, biztosítva az adatok integritását.
- A quantity, item_price, és line_total oszlopok tárolják a rendelési térel alapvető adatait.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli az order_items táblát.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza az order_items táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs töri az order_items táblát.



Felhasználási esetek

- Rendelési tételek kezelése:**
Az order_items tábla lehetővé teszi a rendelésekhez tartozó tételek adatainak tárolását és kezelését.
 - Kapcsolat a termékekkel és csomagautomatákkal:**
A product_id és locker_id oszlopok biztosítják, hogy minden rendelési tétel egy adott termékhez és csomagautomatához legyen kötve.
-

Felhasználói élmény

- Adatintegritás:**
Az onDelete('cascade') és onUpdate('cascade') opciók biztosítják, hogy a kapcsolódó adatok törlése vagy frissítése esetén a rendelési tételek is automatikusan frissüljenek vagy törlődjenek.
 - Egyszerű rendeléskezelés:**
Az order_items tábla lehetővé teszi a rendelések részleteinek könnyű elérését és kezelését az adatbázisból.
-

Összegzés

Ez a migrációs fájl a rendelési tételek kezeléséhez szükséges alapvető táblát hozza létre. Az order_items tábla lehetővé teszi a rendelések részleteinek tárolását, rendszerezését és a kapcsolódó termékekkel, rendelésekkel és csomagautomatákkal való kapcsolatát, ami elengedhetetlen egy jól strukturált rendeléskezelő rendszerben.



2025_04_12_172737_create_locker_product_table.php – Laravel migráció a csomagautomaták és termékek kapcsolótáblájához

Ez a Laravel migrációs fájl a locker_product kapcsolótábla létrehozására szolgál az adatbázisban. A tábla a csomagautomaták és a termékek közötti sok-sok kapcsolatot kezeli, például azt, hogy mely termékek érhetők el egy adott csomagautomatában.

Főbb funkciók

1. Kapcsolótábla létrehozása (locker_product)

- A Schema::create metódus segítségével létrejön a locker_product nevű tábla.
- A tábla oszlopai:
 - **id:** Az elsődleges kulcs, automatikusan növekvő egész szám.
 - **product_id:** Külső kulcs a products táblára, amely a kapcsolódó terméket jelöli.
 - **onDelete('cascade')**: Ha a termék törlésre kerül, a hozzá tartozó kapcsolatok is törlődnek.
 - **locker_id:** Külső kulcs a lockers táblára, amely a kapcsolódó csomagautomatát jelöli.
 - **onDelete('cascade')**: Ha a csomagautomata törlésre kerül, a hozzá tartozó kapcsolatok is törlődnek.
 - **timestamps:** Automatikusan kezeli a created_at és updated_at mezőket.

2. Tábla törlése (down metódus)

- A Schema::dropIfExists metódus segítségével törli a locker_product táblát, ha az létezik.



Felépítés

1. up metódus

- Ez a metódus felelős a locker_product tábla létrehozásáért.
- A product_id és locker_id oszlopok külső kulcsként kapcsolódnak a products és lockers táblákhoz, biztosítva az adatok integritását.

2. down metódus

- Ez a metódus visszavonja a migrációt, azaz törli a locker_product táblát.

Felhasználási példa

- **Migráció futtatása:**

```
php artisan migrate
```

Ez a parancs létrehozza a locker_product táblát az adatbázisban.

- **Migráció visszavonása:**

```
php artisan migrate:rollback
```

Ez a parancs törli a locker_product táblát.

Felhasználási esetek

- **Kapcsolatok kezelése:**

A locker_product tábla lehetővé teszi annak nyilvántartását, hogy mely termékek érhetők el egy adott csomagautomatában.

- **Adatok validálása:**

Az onDelete('cascade') opció biztosítja, hogy a kapcsolatok automatikusan törlődjenek, ha egy termék vagy csomagautomata törlésre kerül.



Felhasználói élmény

- **Egyszerű rendszerezés:**
A kapcsolótábla lehetővé teszi a termékek és csomagautomaták közötti kapcsolatok könnyű kezelését.
- **Adatintegritás:**
Az onDelete('cascade') opció biztosítja, hogy a kapcsolódó adatok mindenkorban maradjanak.

Összegzés

Ez a migrációs fájl a csomagautomaták és termékek közötti sok-sok kapcsolat kezeléséhez szükséges alapvető táblát hozza létre. A locker_product tábla lehetővé teszi a kapcsolatok rendszerezését és az adatok integritásának fenntartását, ami elengedhetetlen egy jól strukturált logisztikai rendszerben.

RoleSeeder.php – Laravel adatfeltöltő a szerepkörökhez

A [RoleSeeder.php](#) egy Laravel adatfeltöltő osztály, amely a roles tábla alapértelmezett adatait hozza létre. Ez az osztály biztosítja, hogy a rendszerben mindenkorban elérhetők legyenek az alapvető szerepkörök, például "user", "admin" és "SuperAdmin".

Főbb funkciók

1. Szerepkörök létrehozása vagy frissítése

- A Role::updateOrCreate metódus biztosítja, hogy a szerepkörök létezzenek, és ha már léteznek, frissíti azokat.



- Három alapértelmezett szerepkör kerül létrehozásra:
 - **user:** Alapértelmezett felhasználói szerepkör, amelynek jogosultsági szintje (power) 11.
 - **admin:** Adminisztrátori szerepkör, amelynek jogosultsági szintje 70.
 - **SuperAdmin:** Legmagasabb jogosultsági szinttel rendelkező szerepkör, amelynek jogosultsági szintje 90.

Felépítés

1. run metódus

- Ez a metódus felelős az adatfeltöltésért.
- A Role::updateOrCreate metódus használatával biztosítja, hogy a szerepkörök egyedi módon kerüljenek létrehozásra vagy frissítésre a warrant_name mező alapján.

Felhasználási példa

- Seeder futtatása:

```
php artisan db:seed --class=RoleSeeder
```

Ez a parancs futtatja a RoleSeeder osztályt, és feltölti a roles táblát az alapértelmezett szerepkörökkel.

- Összes seeder futtatása:

```
php artisan db:seed
```

Ez a parancs futtatja az összes regisztrált seeder osztályt, beleérte a RoleSeeder-t is.

Felhasználási esetek

- Alapértelmezett szerepkörök biztosítása:

A RoleSeeder biztosítja, hogy a rendszerben minden elérhetők legyenek az alapvető szerepkörök.



- **Fejlesztési és tesztelési környezetek:**

A seeder gyorsan feltölti a roles táblát, ami hasznos lehet fejlesztési vagy tesztelési környezetekben.

Felhasználói élmény

- **Egyszerű adatkezelés:**

A updateOrCreate metódus biztosítja, hogy a szerepkörök minden naprakészek legyenek, és elkerüli a duplikációkat.

- **Könnyű bővíthetőség:**

Új szerepkörök egyszerűen hozzáadhatók a seederhez, ha szükséges.

Összegzés

A `RoleSeeder.php` egy alapvető adatfeltöltő osztály, amely biztosítja a roles tábla alapértelmezett szerepköreinek létrehozását vagy frissítését. Ez az osztály elengedhetetlen a jogosultságkezeléshez, és biztosítja, hogy a rendszer minden rendelkezzen a szükséges szerepkörökkel.



UserSeeder.php – Laravel adatfeltöltő a felhasználókhoz

A `UserSeeder.php` egy Laravel adatfeltöltő osztály, amely a `users` tábla alapértelmezett adatait hozza létre. Ez az osztály biztosítja, hogy a rendszerben minden elérhető legyen egy alapértelmezett "SuperAdmin" felhasználó.

Főbb funkciók

1. SuperAdmin felhasználó létrehozása vagy frissítése

- A `User::updateOrCreate` metódus biztosítja, hogy a "SuperAdmin" felhasználó létezzen, és ha már létezik, frissíti az adatait.
- A felhasználó adatai:
 - **email:** `SuperAdmin@example.com`
 - **first_name:** `Super`
 - **last_name:** `Admin`
 - **password:** A jelszó titkosítva kerül tárolásra a `Hash::make` metódus segítségével (`test1234`).
 - **role_id:** A "SuperAdmin" szerepkör azonosítója, amelyet a `Role` modell segítségével keres ki.

Felépítés

1. run metódus

- Ez a metódus felelős az adatfeltöltésért.
 - A `Role::where` metódus segítségével lekéri a "SuperAdmin" szerepkört a `roles` táblából.
 - A `User::updateOrCreate` metódus biztosítja, hogy a "SuperAdmin" felhasználó egyedi módon kerüljön létrehozásra vagy frissítésre az email cím alapján.
-



Felhasználási példa

- Seeder futtatása:

```
php artisan db:seed --class=UserSeeder
```

Ez a parancs futtatja a UserSeeder osztályt, és feltölti a users táblát az alapértelmezett "SuperAdmin" felhasználóval.

- Összes seeder futtatása:

```
php artisan db:seed
```

Ez a parancs futtatja az összes regisztrált seeder osztályt, beleértve a UserSeeder-t is.

Felhasználási esetek

- Alapértelmezett adminisztrátor biztosítása:

A UserSeeder biztosítja, hogy a rendszerben minden elérhető legyen egy "SuperAdmin" felhasználó.

- Fejlesztési és tesztelési környezetek:

A seeder gyorsan feltölti a users táblát, ami hasznos lehet fejlesztési vagy tesztelési környezetekben.

Felhasználói élmény

- Egyszerű adatkezelés:

A updateOrCreate metódus biztosítja, hogy a "SuperAdmin" felhasználó minden naprakész legyen, és elkerüli a duplikációkat.

- Biztonságos jelszókezelés:

A Hash::make metódus biztosítja, hogy a jelszó titkosítva kerüljön tárolásra.



Összegzés

A `UserSeeder.php` egy alapvető adatfeltöltő osztály, amely biztosítja a `users` tábla alapértelmezett "SuperAdmin" felhasználójának létrehozását vagy frissítését. Ez az osztály elengedhetetlen a rendszer adminisztrációs funkcióinak biztosításához, és megkönnyíti a fejlesztési és tesztelési folyamatokat.



Address.php – Laravel modell a címekhez

Az Address modell a Laravel Eloquent ORM segítségével az addresses adatbázistáblát reprezentálja. Ez a modell a címek kezelésére szolgál, és kapcsolatokat definiál más modellekkel, például a felhasználókkal, az utcanevek típusaival és a rendelésekhez tartozó címekkel.

Főbb funkciók

1. Tábla megadása

- A modell az addresses táblához kapcsolódik:

```
<?php  
  
protected $table = 'addresses';
```

2. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölthatók ki tömegesen:

```
<?php  
  
protected $fillable = ['zip', 'street', 'house_number', 'city', 'email', 'street_id', 'user_id'];
```

Ezek az oszlopok tartalmazzák az irányítószámot, utcanevet, házszámot, várost, email címet, az utca típusának azonosítóját és a felhasználói azonosítót.



3. Kapcsolatok definiálása

- **Utca típusa (streetType):**

Az Address modell az StreetType modellhez tartozik az street_id oszlop alapján:

```
<?php  
  
public function streetType()  
  
{  
  
    return $this->belongsTo(StreetType::class, 'street_id');  
  
}
```

- **Felhasználó (user):**

Az Address modell az User modellhez tartozik az user_id oszlop alapján:

```
<?php  
  
public function user()  
  
{  
  
    return $this->belongsTo(User::class);  
  
}
```



- **Rendelések (order):**

Az Address modellnek több rendelése lehet az Order modellel való kapcsolat révén:

```
<?php  
  
public function order()  
{  
    return $this->hasMany(Order::class);  
}
```

Felhasználási esetek

- **Címek kezelése:**

Az Address modell lehetővé teszi a címek létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

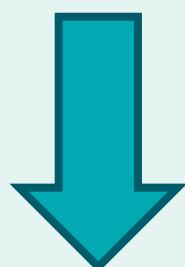
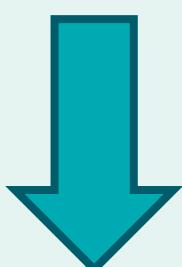
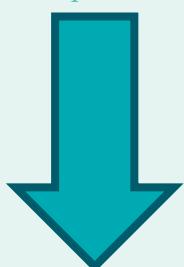
- **Kapcsolatok kezelése:**

Az Address modell segítségével könnyen elérhetők a kapcsolódó utca típusok, felhasználók és rendelések.

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik az adatok egyszerű lekérdezését és kezelését, például:





```
<?php  
  
$address = Address::find(1);  
  
$user = $address->user; // Kapcsolódó felhasználó  
  
$streetType = $address->streetType; // Kapcsolódó utca típusa  
  
$orders = $address->order; // Kapcsolódó rendelések
```

Összegzés

Az Address modell egy jól strukturált Eloquent modell, amely az addresses tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi az adatok egyszerű és hatékony kezelését, miközben biztosítja az adatok integritását és a kapcsolódó táblák közötti összefüggéseket.



Category.php – Laravel modell a kategóriákhoz

A Category modell a Laravel Eloquent ORM segítségével a categories adatbázistáblát reprezentálja. Ez a modell a termékkategóriák kezelésére szolgál, és kapcsolatot definiál a termékekkel.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
  
protected $fillable = ['name'];
```

Ez az oszlop a kategória nevét tárolja.

2. Kapcsolatok definiálása

- Termékek (products):

A Category modell és a Product modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function products()  
{  
    return $this->hasMany(Product::class);  
}
```

Ez azt jelenti, hogy egy kategóriához több termék is tartozhat.



Felhasználási esetek

- **Kategóriák kezelése:**
A Category modell lehetővé teszi a kategóriák létrehozását, frissítését, törlését és lekérdezését az adatbázisból.
- **Kapcsolódó termékek lekérdezése:**
A kategóriához tartozó termékek egyszerűen lekérdezhetők az Eloquent kapcsolat segítségével:

```
<?php  
$category = Category::find(1);  
$products = $category->products; // A kategóriához tartozó termékek
```

Felhasználói élmény

- **Egyszerű adatkezelés:**
Az Eloquent ORM kapcsolatai lehetővé teszik az adatok egyszerű és hatékony kezelését.
- **Tömeges kitöltés támogatása:**
A fillable mezők biztosítják, hogy a kategóriák adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A Category modell egy jól strukturált Eloquent modell, amely a categories tábla kezelésére szolgál. A termékekkel való kapcsolat definiálása lehetővé teszi a kategóriák és a hozzájuk tartozó termékek közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált adatbázisban.



Customer.php – Laravel modell az ügyfelekhez

A Customer modell a Laravel Eloquent ORM segítségével a customers adatbázistáblát reprezentálja. Ez a modell az ügyfelek kezelésére szolgál, és kapcsolatokat definiál a felhasználókkal, címekkel és rendelésekhez.

Főbb funkciók

1. Tábla megadása

- A modell az customers táblához kapcsolódik:

```
<?php  
  
protected $table = 'customers';
```

2. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölthetők ki tömegesen:

```
<?php  
  
protected $fillable = ['user_id', 'address_id', 'order_id'];
```

Ezek az oszlopok tartalmazzák a felhasználói azonosítót, a cím azonosítóját és a rendelés azonosítóját.



4. Kapcsolatok definiálása

- **Felhasználó (user):**

Az Customer modell az User modellhez tartozik az user_id oszlop alapján:

```
<?php  
    public function user()  
    {  
        return $this->belongsTo(User::class);  
    }
```

- **Címek (address):**

Az Customer modellnek több címe is lehet az Address modellel való kapcsolat révén:

```
<?php  
    public function address()  
    {  
        return $this->hasMany(Address::class);  
    }
```

- **Rendelések (order):**

Az Customer modellnek több rendelése is lehet az Order modellel való kapcsolat révén:

```
<?php  
    public function order()  
    {  
        return $this->hasMany(Order::class);  
    }
```

Felhasználási esetek

- **Ügyfelek kezelése:**

A Customer modell lehetővé teszi az ügyfelek adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.



- **Kapcsolatok kezelése:**

Az ügyfelekhez kapcsolódó felhasználók, címek és rendelések egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php  
$customer = Customer::find(1);  
$user = $customer->user; // Kapcsolódó felhasználó  
$addresses = $customer->address; // Kapcsolódó címek  
$orders = $customer->order; // Kapcsolódó rendelések
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik az ügyfelekhez kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy az ügyfelek adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A Customer modell egy jól strukturált Eloquent modell, amely a customers tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi az ügyfelek és a hozzájuk tartozó felhasználók, címek és rendelések közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált adatbázisban.



Locker.php – Laravel modell a csomagautomatákhoz

A Locker modell a Laravel Eloquent ORM segítségével a lockers adatbázistáblát reprezentálja. Ez a modell a csomagautomaták kezelésére szolgál, és kapcsolatokat definiál a termékekkel és rendelési tételekkel.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = [  
    'locker_name',  
    'address',  
    'description',  
];
```

Ezek az oszlopok tartalmazzák a csomagautomata nevét, címét és leírását.

2. Kapcsolatok definiálása

- **Termékek (products):**

A Locker modell és a Product modell között egy sok-sok (Many-to-Many) kapcsolat van definiálva:

```
<?php  
public function products()  
{  
    return $this->belongsToMany(Product::class);  
}
```

Ez azt jelenti, hogy egy csomagautomatában több termék is elérhető, és egy termék több csomagautomatában is megtalálható.



- **Rendelési tételek (orderItems):**

A Locker modell és az OrderItem modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function orderItems()  
{  
    return $this->hasMany(OrderItem::class);  
}
```

Ez azt jelenti, hogy egy csomagautomatához több rendelési térel is tartozhat.

Felhasználási esetek

- **Csomagautomaták kezelése:**

A Locker modell lehetővé teszi a csomagautomaták adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolatok kezelése:**

A csomagautomatákhoz kapcsolódó termékek és rendelési tételek egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php  
$locker = Locker::find(1);  
$products = $locker->products; // Kapcsolódó termékek  
$orderItems = $locker->orderItems; // Kapcsolódó rendelési tételek
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a csomagautomatákhoz kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a csomagautomaták adatai biztonságosan és egyszerűen kezelhetők legyenek.



Összegzés

A Locker modell egy jól strukturált Eloquent modell, amely a lockers tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a csomagautomaták és a hozzájuk tartozó termékek, valamint rendelési tételek közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált logisztikai rendszerben.



Order.php – Laravel modell a rendelésekhez

Az Order modell a Laravel Eloquent ORM segítségével az orders adatbázistáblát reprezentálja. Ez a modell a rendelések kezelésére szolgál, és kapcsolatokat definiál a rendelési tételekkel, felhasználókkal, fizetési módokkal és címekkel.

Főbb funkciók

1. Tábla megadása

- A modell az orders táblához kapcsolódik:

```
<?php  
protected $table = 'orders';
```

2. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = ['total', 'payments_method_id', 'user_id', 'address_id'];
```

Ezek az oszlopok tartalmazzák a rendelés összegét, a fizetési mód azonosítóját, a felhasználói azonosítót és a cím azonosítóját.



3. Kapcsolatok definiálása

- **Rendelési tételek (orderItem):**

Az Order modell és az OrderItem modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function orderItem()  
{  
    return $this->hasMany(OrderItem::class);  
}
```

Ez azt jelenti, hogy egy rendeléshez több rendelési térel is tartozhat.

- **Felhasználó (user):**

Az Order modell az User modellhez tartozik az user_id oszlop alapján:

```
<?php  
public function user()  
{  
    return $this->belongsTo(User::class);  
}
```

- **Fizetési mód (paymentMethod):**

Az Order modell a PaymentMethod modellhez tartozik a payments_method_id oszlop alapján:

```
<?php  
public function paymentMethod()  
{  
    return $this->belongsTo(PaymentMethod::class);  
}
```

- **Cím (address):**

Az Order modell az Address modellhez tartozik az address_id oszlop alapján:

```
<?php  
public function address()  
{  
    return $this->belongsTo(Address::class);  
}
```



Felhasználási esetek

- **Rendelések kezelése:**

Az Order modell lehetővé teszi a rendelések adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolatok kezelése:**

A rendelésekhez kapcsolódó rendelési tételek, felhasználók, fizetési módok és címek egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php
$order = Order::find(1);
$orderItems = $order->orderItem; // Kapcsolódó rendelési tételek
$user = $order->user; // Kapcsolódó felhasználó
$paymentMethod = $order->paymentMethod; // Kapcsolódó fizetési mód
$address = $order->address; // Kapcsolódó cím
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a rendelésekhez kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a rendelések adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

Az Order modell egy jól strukturált Eloquent modell, amely az orders tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a rendelések és a hozzájuk tartozó rendelési tételek, felhasználók, fizetési módok és címek közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált rendeléskezelő rendszerben.



OrderItem.php – Laravel modell a rendelési tételekhez

Az OrderItem modell a Laravel Eloquent ORM segítségével az order_items adatbázistáblát reprezentálja. Ez a modell a rendelési tételek kezelésére szolgál, és kapcsolatokat definiál a termékekkel, rendeléssel és csomagautomatákkal.

Főbb funkciók

1. Tábla megadása

- A modell az order_items táblához kapcsolódik:

```
<?php  
protected $table = 'order_items';
```

2. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = ['quantity', 'item_price', 'line_total', 'product_id', 'order_id',  
'locker_id'];
```

Ezek az oszlopok tartalmazzák a rendelési térel mennyiségét, egységárát, teljes árát, a termék azonosítóját, a rendelés azonosítóját és a csomagautomata azonosítóját.

3. Automatikus line_total számítás

- A boot metódusban definiált creating esemény automatikusan kiszámítja a rendelési térel teljes árát (line_total), amikor egy új rendelési térel jön létre:

```
<?php  
static::creating(function ($orderItem) {  
    $orderItem->line_total = $orderItem->quantity * $orderItem->item_price;  
});
```



4. Kapcsolatok definiálása

- **Termék (product):**

Az OrderItem modell a Product modellhez tartozik a product_id oszlop alapján:

```
<?php
    public function product()
    {
        return $this->belongsTo(Product::class);
    }
```

- **Rendelés (order):**

Az OrderItem modell az Order modellhez tartozik az order_id oszlop alapján:

```
<?php
    public function order()
    {
        return $this->belongsTo(Order::class);
    }
```

- **Csomagautomata (locker):**

Az OrderItem modell a Locker modellhez tartozik a locker_id oszlop alapján:

```
<?php
    public function locker()
    {
        return $this->belongsTo(Locker::class);
    }
```



Felhasználási esetek

- **Rendelési tételek kezelése:**

Az OrderItem modell lehetővé teszi a rendelési tételek adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolatok kezelése:**

A rendelési tételekhez kapcsolódó termékek, rendelések és csomagautomaták egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php
```

```
$orderItem = OrderItem::find(1);  
$product = $orderItem->product; // Kapcsolódó termék  
$order = $orderItem->order; // Kapcsolódó rendelés  
$locker = $orderItem->locker; // Kapcsolódó csomagautomata
```

Felhasználói élmény

- **Automatikus számítások:**

Az line_total automatikus kiszámítása csökkenti a hibalehetőségeket és egyszerűsíti az adatkezelést.

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a rendelési tételekhez kapcsolódó adatok egyszerű és hatékony kezelését.

Összegzés

Az OrderItem modell egy jól strukturált Eloquent modell, amely az order_items tábla kezelésére szolgál. Az automatikus line_total számítás és a kapcsolatok definiálása lehetővé teszi a rendelési tételek és a hozzájuk tartozó termékek, rendelések és csomagautomaták közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált rendeléskezelő rendszerben.



PaymentMethod.php – Laravel modell a fizetési módokhoz

A PaymentMethod modell a Laravel Eloquent ORM segítségével a payment_methods adatbázistáblát reprezentálja. Ez a modell a fizetési módok kezelésére szolgál, és kapcsolatot definiál a rendelésekhez.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölthetők ki tömegesen:

```
<?php  
protected $fillable = ['card_type'];
```

Ez az oszlop a fizetési mód típusát tárolja, például "Visa" vagy "MasterCard".

2. Kapcsolatok definiálása

- Rendelések (orders):

A PaymentMethod modell és az Order modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function orders()  
{  
    return $this->hasMany(Order::class);  
}
```

Ez azt jelenti, hogy egy fizetési módhoz több rendelés is tartozhat.



Felhasználási esetek

- **Fizetési módok kezelése:**

A PaymentMethod modell lehetővé teszi a fizetési módok adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolódó rendelések lekérdezése:**

Egy adott fizetési módhoz tartozó rendelések egyszerűen lekérdezhetők az Eloquent kapcsolat segítségével:

```
<?php  
$paymentMethod = PaymentMethod::find(1);  
$orders = $paymentMethod->orders; // Kapcsolódó rendelések
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a fizetési módokhoz kapcsolódó rendelések egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a fizetési módok adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A PaymentMethod modell egy jól strukturált Eloquent modell, amely a payment_methods tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a fizetési módok és a hozzájuk tartozó rendelések közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált fizetési rendszerben.



Product.php – Laravel modell a termékekhez

A Product modell a Laravel Eloquent ORM segítségével a products adatbázistáblát reprezentálja. Ez a modell a termékek kezelésére szolgál, és kapcsolatokat definiál a kategóriákkal, rendelési tételekkel és csomagautomatákkal.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = [  
    'file_name', 'file_path', 'name', 'description', 'price_per_day', 'category_id', 'available',  
];
```

Ezek az oszlopok tartalmazzák a termék fájlnevét, fájlútvonalát, nevét, leírását, napi árat, kategória azonosítóját és elérhetőségi állapotát.

2. Kapcsolatok definiálása

- **Kategória (category):**

A Product modell és a Category modell között egy sok-egy (Many-to-One) kapcsolat van definiálva:

```
<?php  
public function category()  
{  
    return $this->belongsTo(Category::class, 'category_id');
```

Ez azt jelenti, hogy egy termék egy kategóriához tartozik.



- **Rendelési tételek (orderItems):**

A Product modell és az OrderItem modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function orderItems()  
{  
    return $this->hasMany(OrderItem::class);  
}
```

Ez azt jelenti, hogy egy termék több rendelési térelben is szerepelhet.

- **Csomagautomaták (lockers):**

A Product modell és a Locker modell között egy sok-sok (Many-to-Many) kapcsolat van definiálva:

```
<?php  
public function lockers()  
{  
    return $this->belongsToMany(Locker::class);  
}
```

Ez azt jelenti, hogy egy termék több csomagautomatában is elérhető lehet, és egy csomagautomatában több termék is megtalálható.

Felhasználási esetek

- **Termékek kezelése:**

A Product modell lehetővé teszi a termékek adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolatok kezelése:**

A termékekhez kapcsolódó kategóriák, rendelési tételek és csomagautomaták egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php  
$product = Product::find(1);  
$category = $product->category; // Kapcsolódó kategória  
$orderItems = $product->orderItems; // Kapcsolódó rendelési tételek  
$lockers = $product->lockers; // Kapcsolódó csomagautomaták
```



Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a termékekhez kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a termékek adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A Product modell egy jól strukturált Eloquent modell, amely a products tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a termékek és a hozzájuk tartozó kategóriák, rendelési tételek és csomagautomaták közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált termékkezelő rendszerben.



Profile.php – Laravel modell a profilokhoz

A Profile modell a Laravel Eloquent ORM segítségével a profiles adatbázistáblát reprezentálja. Ez a modell a felhasználói profilok kezelésére szolgál, és kapcsolatot definiál a felhasználókkal.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölthetők ki tömegesen:

```
<?php  
protected $fillable = ['file_name','file_path', 'user_id'];
```

Ezek az oszlopok tartalmazzák a profilhoz tartozó fájl nevét, fájl elérési útvonalát és a felhasználói azonosítót.

2. Kapcsolatok definiálása

- Felhasználó (user):

A Profile modell és a User modell között egy sok-egy (Many-to-One) kapcsolat van definiálva:

```
<?php  
public function user()  
{  
    return $this->belongsTo(User::class);  
}
```

Ez azt jelenti, hogy egy profil egy adott felhasználóhoz tartozik.



Felhasználási esetek

- **Profilok kezelése:**

A Profile modell lehetővé teszi a profilkönyvtár adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolódó felhasználók lekérdezése:**

Egy adott profilhoz tartozó felhasználó egyszerűen lekérdezhető az Eloquent kapcsolat segítségével:

```
<?php  
$profile = Profile::find(1);  
$user = $profile->user; // Kapcsolódó felhasználó
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a profilkönyvtár adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a profilkönyvtár adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A Profile modell egy jól strukturált Eloquent modell, amely a profiles tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a profilkönyvtár adatai és a hozzájuk tartozó felhasználók közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált felhasználókezelő rendszerben.



Role.php – Laravel modell a szerepkörökhez

A Role modell a Laravel Eloquent ORM segítségével a roles adatbázistáblát reprezentálja. Ez a modell a felhasználói szerepkörök kezelésére szolgál, és kapcsolatot definiál a felhasználókkal.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = ['power', 'warrant_name'];
```

Ezek az oszlopok tartalmazzák a szerepkör jogosultsági szintjét (power) és a szerepkör nevét (warrant_name).

2. Kapcsolatok definiálása

- Felhasználók (users):

A Role modell és a User modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function users()  
{  
    return $this->hasMany(User::class);  
}
```

Ez azt jelenti, hogy egy szerepkörhöz több felhasználó is tartozhat.



Felhasználási esetek

- **Szerepkörök kezelése:**

A Role modell lehetővé teszi a szerepkörök adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolódó felhasználók lekérdezése:**

Egy adott szerepkörhöz tartozó felhasználók egyszerűen lekérdezhetők az Eloquent kapcsolat segítségével:

```
<?php  
$role = Role::find(1);  
$users = $role->users; // Kapcsolódó felhasználók
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik a szerepkörökhöz kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy a szerepkörök adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A Role modell egy jól strukturált Eloquent modell, amely a roles tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a szerepkörök és a hozzájuk tartozó felhasználók közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált jogosultságkezelő rendszerben.



StreetType.php – Laravel modell az utca típusokhoz

A StreetType modell a Laravel Eloquent ORM segítségével a street_types adatbázistáblát reprezentálja. Ez a modell az utca típusok kezelésére szolgál, és kapcsolatot definiál a címekkel.

Főbb funkciók

1. Tábla megadása

- A modell a street_types táblához kapcsolódik:

```
<?php  
protected $table = 'street_types';
```

2. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölthetők ki tömegesen:

```
<?php  
protected $fillable = ['public_area_name'];
```

Ez az oszlop az utca típusának nevét tárolja, például "utca", "térr", "körút".

3. Kapcsolatok definiálása

- Címek (address):

A StreetType modell és az Address modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php  
public function address(){  
    return $this->hasMany(Address::class);  
}
```

Ez azt jelenti, hogy egy utca típushoz több cím is tartozhat.



Felhasználási esetek

- **Utca típusok kezelése:**

A StreetType modell lehetővé teszi az utca típusok adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolódó címek lekérdezése:**

Egy adott utca típushoz tartozó címek egyszerűen lekérdezhetők az Eloquent kapcsolat segítségével:

```
<?php  
$streetType = StreetType::find(1);  
$addresses = $streetType->address; // Kapcsolódó címek
```

Felhasználói élmény

- **Egyszerű adatkezelés:**

Az Eloquent ORM kapcsolatai lehetővé teszik az utca típusokhoz kapcsolódó adatok egyszerű és hatékony kezelését.

- **Tömeges kitöltés támogatása:**

A fillable mezők biztosítják, hogy az utca típusok adatai biztonságosan és egyszerűen kezelhetők legyenek.

Összegzés

A StreetType modell egy jól strukturált Eloquent modell, amely a street_types tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi az utca típusok és a hozzájuk tartozó címek közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált címkezelő rendszerben.



User.php – Laravel modell a felhasználókhöz

A User modell a Laravel Eloquent ORM segítségével a users adatbázistáblát reprezentálja. Ez a modell a felhasználók kezelésére szolgál, és kapcsolatokat definiál más modellekkel, például profilokkal, címekkel, rendelésekhez és szerepkörökhez.

Főbb funkciók

1. Tömeges kitöltés engedélyezése

- A fillable mezők határozzák meg, hogy mely oszlopok tölhetők ki tömegesen:

```
<?php  
protected $fillable = ['first_name','last_name','email', 'password','role_id',];
```

Ezek az oszlopok tartalmazzák a felhasználó alapvető adatait, például nevét, email címét, jelszavát és szerepkörének azonosítóját.

2. Kapcsolatok definiálása

- **Profil (profile):**

A User modell és a Profile modell között egy-egy (One-to-One) kapcsolat van definiálva:

```
<?php  
public function profile()  
{  
    return $this->hasOne(Profile::class);  
}
```



- **Címek (address):**

A User modell és az Address modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php
    public function address()
    {
        return $this->hasMany(Address::class);
    }
```

- **Rendelések (order):**

A User modell és az Order modell között egy-egy-több (One-to-Many) kapcsolat van definiálva:

```
<?php
    public function order()
    {
        return $this->hasMany(Order::class);
    }
```

- **Szerepkör (role):**

A User modell és a Role modell között egy sok-egy (Many-to-One) kapcsolat van definiálva:

```
<?php
    public function role()
    {
        return $this->belongsTo(Role::class);
    }
```

3. Rejtett attribútumok

- A hidden mezők határozzák meg, hogy mely attribútumok legyenek elrejtve a JSON serializáció során:

```
<?php
protected $hidden = ['password','remember_token',];
```



4. Attribútumok típuskonverziója

- A casts metódus biztosítja, hogy bizonyos attribútumok automatikusan konvertálódjanak a megfelelő típusra:

```
<?php
protected function casts(): array
{
    return [
        'email_verified_at' => 'datetime',
        'password' => 'hashed',
    ];
}
```

Felhasználási esetek

- **Felhasználók kezelése:**

A User modell lehetővé teszi a felhasználók adatainak létrehozását, frissítését, törlését és lekérdezését az adatbázisból.

- **Kapcsolatok kezelése:**

A felhasználókhöz kapcsolódó profilk, címek, rendelések és szerepkörök egyszerűen lekérdezhetők az Eloquent kapcsolatok segítségével:

```
<?php
$user = User::find(1);
$profile = $user->profile; // Kapcsolódó profil
$addresses = $user->address; // Kapcsolódó címek
$orders = $user->order; // Kapcsolódó rendelések
$role = $user->role; // Kapcsolódó szerepkör
```



Felhasználói élmény

- **Biztonságos jelszókezelés:**
A password mező automatikusan hash-elve kerül tárolásra a hashed típuskonverzió segítségével.
 - **Egyszerű adatkezelés:**
Az Eloquent ORM kapcsolatai lehetővé teszik a felhasználókhoz kapcsolódó adatok egyszerű és hatékony kezelését.
-

Összegzés

A User modell egy jól strukturált Eloquent modell, amely a users tábla kezelésére szolgál. A kapcsolatok definiálása lehetővé teszi a felhasználók és a hozzájuk tartozó profilok, címek, rendelések és szerepkörök közötti összefüggések egyszerű kezelését, ami elengedhetetlen egy jól strukturált felhasználókezelő rendszerben.



FlexiBox API Útvonalak Dokumentáció

Az alábbiakban bemutatjuk a FlexiBox rendszer API útvonalait, amelyek különböző funkciókat biztosítanak, például felhasználói hitelesítést, profilkezelést, rendelések kezelését, termékek és kategóriák adminisztrációját, valamint címek és fizetési módok kezelését. Az API-k RESTful elveket követnek, és a legtöbbük hitelesítést igényel.

1. Hitelesítés (Auth API-k)

Ezek az API-k a felhasználók regisztrációját és bejelentkezését kezelik.

- **Regisztráció**
 - **Módszer:** POST
 - **Útvonal:** /register
 - **Leírás:** Létrehoz egy új felhasználót.
 - **Vezérlő:** RegisterController@register
- **Bejelentkezés**
 - **Módszer:** POST
 - **Útvonal:** /login
 - **Leírás:** Hitelesíti a felhasználót és visszaad egy hozzáférési tokent.
 - **Vezérlő:** LoginController@login

2. Profilkezelés (Profile API-k)

Ezek az API-k a felhasználói profilk kezelésére szolgálnak.

- **Profil létrehozása**
 - **Módszer:** POST
 - **Útvonal:** /profile
 - **Leírás:** Létrehoz egy új profilt.
 - **Vezérlő:** ProfileController@store



- **Profil frissítése**
 - **Módszer:** PATCH
 - **Útvonal:** /profile
 - **Leírás:** Frissíti a felhasználó profilját.
 - **Vezérlő:** RegisterController@update
- **Profil megtekintése**
 - **Módszer:** GET
 - **Útvonal:** /profile/index
 - **Leírás:** Visszaadja a felhasználó profilját.
 - **Vezérlő:** ProfileController@index
 - **Middleware:** auth:sanctum
- **Profil törlése**
 - **Módszer:** DELETE
 - **Útvonal:** /profile/delete
 - **Leírás:** Törli a felhasználó profilját.
 - **Vezérlő:** ProfileController@destroy
 - **Middleware:** auth:sanctum

3. Felhasználói rendelések (User Orders API-k)

Ezek az API-k a felhasználók rendeléseinek kezelésére szolgálnak.

- **Rendelések listázása**
 - **Módszer:** GET
 - **Útvonal:** /userorderslist
 - **Leírás:** Visszaadja a felhasználó rendeléseit.
 - **Vezérlő:** OrderController@userorderindex
 - **Middleware:** auth:sanctum
- **Rendelés törlése**
 - **Módszer:** DELETE
 - **Útvonal:** /userorders
 - **Leírás:** Törli a felhasználó rendelését.
 - **Vezérlő:** OrderController@userdeleteOrder



- **Middleware:** auth:sanctum
-

4. Adminisztrációs műveletek (Admin API-k)

Ezek az API-k az adminisztrátorok számára biztosítanak funkciókat.

- **Felhasználó szerkesztése**
 - **Módszer:** PUT
 - **Útvonal:** /user/edit
 - **Leírás:** Frissíti egy felhasználó adatait.
 - **Vezérlő:** RegisterController@edit
 - **Middleware:** auth:sanctum
- **Felhasználó törlése**
 - **Módszer:** DELETE
 - **Útvonal:** /user/delete
 - **Leírás:** Törli egy felhasználó adatait.
 - **Vezérlő:** RegisterController@destroy
 - **Middleware:** auth:sanctum
- **Rendelések listázása**
 - **Módszer:** GET
 - **Útvonal:** /orderslist
 - **Leírás:** Visszaadja az összes rendelést.
 - **Vezérlő:** OrderController@orderindex
 - **Middleware:** auth:sanctum
- **Rendelés törlése**
 - **Módszer:** DELETE
 - **Útvonal:** /orders
 - **Leírás:** Törli egy rendelést.
 - **Vezérlő:** OrderController@deleteOrder
 - **Middleware:** auth:sanctum
- **Felhasználók listázása**
 - **Módszer:** GET
 - **Útvonal:** users



- **Leírás:** Visszaadja az összes felhasználót.
 - **Vezérlő:** AdminController@index
 - **Middleware:** auth:sanctum
-

5. Termékek és kategóriák (Product API-k)

Ezek az API-k a termékek és kategóriák kezelésére szolgálnak.

- **Új kategória létrehozása**

- **Módszer:** POST
- **Útvonal:** /category
- **Leírás:** Létrehoz egy új kategóriát.
- **Vezérlő:** ServiceController@categoryStore

- **Új termék létrehozása**

- **Módszer:** POST
- **Útvonal:** /product
- **Leírás:** Létrehoz egy új terméket.
- **Vezérlő:** ServiceController@productStore

- **Termékek listázása**

- **Módszer:** GET
- **Útvonal:** /product
- **Leírás:** Visszaadja az összes terméket.
- **Vezérlő:** ServiceController@productIndex

- **Termék frissítése**

- **Módszer:** POST
- **Útvonal:** /product/update
- **Leírás:** Frissíti egy termék adatait.
- **Vezérlő:** ServiceController@productUpdate

- **Termék törlése**

- **Módszer:** DELETE
- **Útvonal:** /product/delete
- **Leírás:** Törli egy terméket.
- **Vezérlő:** ServiceController@productDestroy



- **Middleware:** auth:sanctum
-

6. Címek és utca típusok (Address API-k)

Ezek az API-k a címek és utca típusok kezelésére szolgálnak.

- **Új cím létrehozása**
 - **Módszer:** POST
 - **Útvonal:** /address
 - **Leírás:** Létrehoz egy új címet.
 - **Vezérlő:** AddressController@addressStore
- **Címek listázása**
 - **Módszer:** GET
 - **Útvonal:** /address/{userId}
 - **Leírás:** Visszaadja egy felhasználó címeit.
 - **Vezérlő:** AddressController@addressIndex
- **Utca típusok listázása**
 - **Módszer:** GET
 - **Útvonal:** /publicareaname
 - **Leírás:** Visszaadja az összes utca típust.
 - **Vezérlő:** AddressController@publicAreaIndex



CheckAdmin.php – Middleware az adminisztrátori jogosultságok ellenőrzésére

A CheckAdmin middleware a beérkező kérések adminisztrátori jogosultságainak ellenőrzésére szolgál. Ez biztosítja, hogy csak a megfelelő jogosultsági szinttel rendelkező felhasználók férjenek hozzá bonyos API útvonalakhoz.

Főbb funkciók

1. Jogosultság ellenőrzése

- A middleware ellenőrzi, hogy a felhasználó be van-e jelentkezve (Auth::check()), és hogy az isadmin attribútuma eléri-e vagy meghaladja-e a megadott jogosultsági szintet (\$role).
- Ha a feltételek teljesülnek, a kérés továbbhalad a következő middleware vagy vezérlő felé:

```
<?php
if (Auth::check() && Auth::user()->isadmin >= $role) {
    return $next($request);
}
```

2.

Hozzáférés megtagadása

- Ha a felhasználó nem rendelkezik megfelelő jogosultsággal, a middleware visszaad egy 403 Unauthorized JSON választ.
- A válasz tartalmazza a következő információkat:
 - **auth_check:** A felhasználó be van-e jelentkezve.
 - **user:** A bejelentkezett felhasználó adatai.
 - **required_role:** A szükséges jogosultsági szint.
 - **actual_role:** A felhasználó aktuális jogosultsági szintje.



```
<?php
    return response()->json([
        'error' => 'Unauthorized',
        'auth_check' => Auth::check(),
        'user' => Auth::user(),
        'required_role' => $role,
        'actual_role' => Auth::user()?->isadmin
    ], 403);
```

3. Naplózás (Debugging)

- A middleware naplózza a következő információkat a hibakereséshez:
 - A middleware aktiválódása.
 - A felhasználó bejelentkezési állapota.
 - A bejelentkezett felhasználó adatai.
 - A szükséges és aktuális jogosultsági szintek.

```
<?php
Log::debug('Middleware triggered');
Log::debug('Auth check: ' . Auth::check());
Log::debug('User: ' . json_encode(Auth::user()));
Log::debug('Required Role: ' . $role);
Log::debug('Actual Role: ' . (Auth::user() ? Auth::user()->
isadmin : 'Not Logged In'));
```

Felhasználási esetek

- **Adminisztrációs API-k védelme:**
A middleware biztosítja, hogy csak a megfelelő jogosultsági szinttel rendelkező felhasználók férjenek hozzá az adminisztrációs funkciókhöz.
- **Jogosultsági szintek kezelése:**
A middleware lehetővé teszi különböző jogosultsági szintek (pl. admin, superadmin) kezelését az isadmin attribútum alapján.



Példa használatra

Az api.php útvonalakban a middleware így használható:

```
<?php
Route::group(['middleware' => ['auth:sanctum', 'checkadmin:70']], function () {
    Route::get('/admindashboard', function () {
        return response()->json(['message' => 'Admin access granted']);
    });
});
```

Ebben a példában csak azok a felhasználók férhetnek hozzá az /admindashboard útvonalhoz, akiknek az isadmin értéke legalább 70.

Felhasználói élmény

- Részletes visszajelzés:**
A JSON válasz tartalmazza a jogosultsági ellenőrzés részleteit, ami segíti a fejlesztőket és a felhasználókat a hibák azonosításában.
- Biztonságos hozzáférés:**
A middleware biztosítja, hogy a jogosulatlan felhasználók ne férjenek hozzá az érzékeny adatokhoz vagy funkciókhoz.

Összegzés

A CheckAdmin middleware egy hatékony eszköz az adminisztrátori jogosultságok kezelésére. A naplázási funkciók és a részletes visszajelzések segítik a hibakeresést és a biztonságos hozzáférés biztosítását. Ez elengedhetetlen egy jól strukturált jogosultságkezelő rendszerben.



AddressController.php – Laravel vezérlő a címek és utca típusok kezelésére

Az AddressController a címek (Address) és az utca típusok (StreetType) kezelésére szolgáló vezérlő. Ez a vezérlő biztosítja a címek és utca típusok létrehozását, frissítését, törlését és lekérdezését, valamint a kapcsolódó API-k működését.

Főbb metódusok

1. publicAreaStore(Request \$request)

- **Leírás:** Új utca típus létrehozása.
- **Funkciók:**
 - Ellenőrzi, hogy a public_area_name mező kitöltött és egyedi legyen.
 - Létrehoz egy új StreetType rekordot.
 - Naplózza a kérést és az esetleges hibákat.
- **Válasz:**
 - Sikeres létrehozás esetén: 201 Created
 - Hiba esetén: 500 Internal Server Error

2. publicAreaIndex()

- **Leírás:** Az összes utca típus lekérdezése.
- **Funkciók:**
 - Lekérdezi az összes StreetType rekordot.
- **Válasz:**
 - Sikeres lekérdezés esetén: 200 OK

3. publicAreaDestroy(Request \$request)

- **Leírás:** Egy utca típus törlése.



- **Funkciók:**

- A törlendő utca típus azonosítóját a StreetTypeId fejlécből olvassa ki.
- Törli a megadott StreetType rekordot.

- **Válasz:**

- Sikeres törlés esetén: 200 OK

4. publicAreaUpdate(Request \$request)

- **Leírás:** Egy utca típus frissítése.

- **Funkciók:**

- Ellenőrzi, hogy a public_area_name mező kitöltött legyen.
- Frissíti a megadott StreetType rekordot.

- **Válasz:**

- Sikeres frissítés esetén: 200 OK

5. addressStore(Request \$request)

- **Leírás:** Új cím létrehozása.

- **Funkciók:**

- Ellenőrzi a cím adatait, például az irányítószámot, utcát, várost, email címet stb.
- Létrehoz egy új Address rekordot.

- **Válasz:**

- Sikeres létrehozás esetén: 201 Created

6. addressIndex(\$userId)

- **Leírás:** Egy felhasználóhoz tartozó címek lekérdezése.

- **Funkciók:**

- Ellenőrzi, hogy a megadott felhasználó létezik-e.
- Lekérdezi a felhasználóhoz tartozó címeket, beleértve az utca típusokat is.

- **Válasz:**

- Sikeres lekérdezés esetén: 200 OK
- Ha a felhasználó nem található: 404 Not Found



Példa API-hívások

Új utca típus létrehozása

- **Módszer:** POST
- **Útvonal:**
- **Kérés törzse:**

```
{  
    "public_area_name": "utca"  
}
```

- **Válasz:**

```
{  
    "message": "Public Area created successfully",  
    "publicArea": {  
        "id": 1,  
        "public_area_name": "utca"  
    }  
}
```

Címek lekérdezése felhasználó szerint

- **Módszer:** GET
- **Útvonal:** /address/{userId}
- **Válasz:**

```
{  
    "addresses": [  
        {  
            "id": 1,  
            "zip": 1234,  
            "street": "Fő utca",  
            "city": "Budapest",  
            "email": "example@example.com",  
            "streetType": {  
                "id": 1,  
                "public_area_name": "utca"  
            }  
        }  
    ]  
}
```



Felhasználói élmény

- **Egyszerű cím- és utcatípus-kezelés:**

Az API-k lehetővé teszik a címek és utca típusok gyors létrehozását, frissítését, törlését és lekérdezését.

- **Részletes hibakezelés:**

A naplózás és a részletes válaszok segítik a fejlesztőket a hibák gyors azonosításában.

Összegzés

Az AddressController egy jól strukturált vezérlő, amely hatékonyan kezeli a címek és utca típusok kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiBox rendszerben.



AdminController.php – Laravel vezérlő az adminisztrációs funkciókhoz

Az AdminController a FlexiBox rendszer adminisztrációs funkcióinak kezelésére szolgál. Ez a vezérlő biztosítja az adminisztrátorok számára a felhasználók listázását és a jogosultságok ellenőrzését.

Főbb metódusok

1. index()

- Leírás:**
Ez a metódus visszaadja az összes felhasználót a hozzájuk tartozó szerepkörökkel együtt, de csak akkor, ha a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik.
- Funkciók:**
 - Ellenőri, hogy a bejelentkezett felhasználó szerepkörének jogosultsági szintje (power) legalább 70-e.
 - Ha a jogosultsági szint nem megfelelő, 403 Forbidden válasz küldése:
 - Lekérdezi az összes felhasználót a hozzájuk tartozó szerepkörökkel:

```
<?php
if (Auth::user()->role->power < 70) {
    return response()->json([
        'message' => 'nincs jogosultsága'
    ], 403);
}
```



```
<?php  
$users = User::with('role')->get();  
    o Visszaadja a felhasználók listáját JSON formátumban:  
<?php  
return response()->json($users);
```

- **Válaszok:**
 - o **Sikeress lekérdezés (200 OK):**
Az összes felhasználó és szerepkörük adatai.
 - o **Hozzáférés megtagadva (403 Forbidden):**
Ha a felhasználó nem rendelkezik megfelelő jogosultsággal.

Példa API-hívás

Felhasználók listázása

- **Módszer:** GET
- **Útvonal:** [users](#)
- **Middleware:** auth:sanctum
- **Válasz (sikeress lekérdezés esetén):**

```
[{  
    "id": 1,  
    "name": "John Doe",  
    "email": "john.doe@example.com",  
    "role": {  
        "id": 1,  
        "warrant_name": "admin",  
        "power": 70  
    },  
    {  
        "id": 2,  
        "name": "Jane Smith",  
        "email": "jane.smith@example.com",  
        "role": {  
            "id": 2,  
            "warrant_name": "user",  
            "power": 11  
        }  
    }]
```



- **Válasz (jogosultság hiánya esetén):**

```
{  
    "message": "nincs jogosultsága"  
}
```

Felhasználási esetek

- **Adminisztrátorok számára:**

Az index metódus lehetővé teszi az adminisztrátorok számára, hogy megtekintsék az összes felhasználót és azok szerepköreiit.

- **Jogosultságkezelés:**

A metódus biztosítja, hogy csak a megfelelő jogosultsági szinttel rendelkező felhasználók férjenek hozzá az adminisztrációs funkciókhoz.

Felhasználói élmény

- **Biztonságos hozzáférés:**

A jogosultsági szint ellenőrzése biztosítja, hogy az érzékeny adatokhoz csak adminisztrátorok férjenek hozzá.

- **Egyszerű adatkezelés:**

Az összes felhasználó és szerepkörük egyetlen API-hívással lekérdezhető.

Összegzés

Az AdminController egy egyszerű, de hatékony vezérlő az adminisztrációs funkciók kezelésére. Az index metódus biztosítja a felhasználók listázását és a jogosultságok ellenőrzését, ami elengedhetetlen egy jól strukturált adminisztrációs rendszerben.



LockerController.php – Laravel vezérlő a csomagautomaták kezelésére

A LockerController a csomagautomaták kezelésére szolgáló vezérlő, amely lehetővé teszi a csomagautomaták létrehozását, listázását, frissítését és törlését. Ez a vezérlő biztosítja a szükséges API-k működését a FlexiBox rendszerben.

Főbb metódusok

1. lockerStore(Request \$request)

- **Leírás:**
Új csomagautomata létrehozása.
- **Funkciók:**
 - Ellenőrzi a bemeneti adatokat:
 - locker_name: Kötelező, szöveg, maximum 255 karakter.
 - address: Kötelező, szöveg, maximum 255 karakter.
 - description: Kötelező, szöveg, maximum 1000 karakter.
 - Létrehoz egy új Locker rekordot az adatbázisban.
 - Visszaadja a létrehozott csomagautomata adatait.
- **Válaszok:**
 - **Sikeres létrehozás (200 OK):**

```
{  
  "message": "Locker created successfully",  
  "locker": {  
    "id": 1,  
    "locker_name": "Locker A",  
    "address": "123 Main Street",  
    "description": "A description of Locker A"  
  }  
}
```



2. lockerIndex()

- **Leírás:**
Az összes csomagautomata listázása.
- **Funkciók:**
 - Lekérdezi az összes Locker rekordot az adatbázisból.
 - Visszaadja a csomagautomaták listáját JSON formátumban.
- **Válaszok:**
 - **Sikeres lekérdezés (200 OK):**

```
{  
    "lockers": [  
        {  
            "id": 1,  
            "locker_name": "Locker A",  
            "address": "123 Main Street",  
            "description": "A description of Locker A"  
        },  
        {  
            "id": 2,  
            "locker_name": "Locker B",  
            "address": "456 Elm Street",  
            "description": "A description of Locker B"  
        }  
    ]  
}
```

3. lockerUpdate(Request \$request)

- **Leírás:**
Egy meglévő csomagautomata adatainak frissítése.



- **Funkciók:**

- Ellenőrzi a bemeneti adatokat (ugyanazok a szabályok, mint a lockerStore metódusban).
- Lekérdezi a frissítendő Locker rekordot az id alapján.
- Ha a csomagautomata nem található, 404 Not Found választ küld.
- Frissíti a csomagautomata adatait, majd elmenti az adatbázisba.
- Visszaadja a frissített csomagautomata adatait.

- **Válaszok:**

- **Sikeres frissítés (200 OK):**

```
{  
    "message": "Locker updated successfully",  
    "locker": {  
        "id": 1,  
        "locker_name": "Updated Locker A",  
        "address": "123 Updated Street",  
        "description": "Updated description"  
    }  
}
```

- **Nem található (404 Not Found):**

```
{  
    "message": "Locker not found"  
}
```

4. lockerDestroy(Request \$request)

- **Leírás:**

Egy csomagautomata törlése.

- **Funkciók:**

- Lekérdezi a törlendő Locker rekordot az id alapján.
- Ha a csomagautomata nem található, 404 Not Found választ küld.
- Törli a csomagautomata rekordot az adatbázisból.
- Visszaad egy sikeres törlés üzenetet.



- Válaszok:

- Sikeres törlés (200 OK):

```
{  
    "message": "Locker deleted successfully"  
}
```

- Nem található (404 Not Found):

```
{  
    "message": "Locker not found"  
}
```

Példa API-hívások

Új csomagautomata létrehozása

- Módszer: POST
- Útvonal: /locker
- Kérés törzse:

```
{  
    "locker_name": "Locker A",  
    "address": "123 Main Street",  
    "description": "A description of Locker A"  
}
```

Csomagautomaták listázása

- Módszer: GET
- Útvonal: /locker



Csomagautomata frissítése

- **Módszer:** PATCH
- **Útvonal:** /locker
- **Kérés törzse:**

```
{  
  "id": 1,  
  "locker_name": "Updated Locker A",  
  "address": "123 Updated Street",  
  "description": "Updated description"  
}
```

Csomagautomata törlése

- **Módszer:** DELETE
- **Útvonal:** /locker/delete
- **Kérés törzse:**

```
{  
  "id": 1  
}
```

Felhasználói élmény

- **Egyszerű kezelés:**
Az API-k lehetővé teszik a csomagautomaták gyors létrehozását, frissítését, törlését és lekérdezését.
- **Részletes válaszok:**
Az API-k részletes válaszokat adnak, amelyek segítik a fejlesztőket és a felhasználókat a műveletek eredményének megértésében.



Összegzés

A LockerController egy jól strukturált vezérlő, amely hatékonyan kezeli a csomagautomaták kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiStore rendszerben.



LoginController.php – Laravel vezérlő a bejelentkezés kezelésére

A LoginController a felhasználók bejelentkezését kezeli a FlexiBox rendszerben. Ez a vezérlő ellenőrzi a felhasználói hitelesítési adatokat, és sikeres bejelentkezés esetén hozzáférési tokent generál.

Főbb metódusok

1. login(Request \$request)

- **Leírás:**

A felhasználó bejelentkezését kezeli, és visszaad egy hozzáférési tokent, ha a hitelesítési adatok helyesek.

- **Funkciók:**

1. **Adatellenőrzés:**

- Ellenőrzi, hogy az email és a jelszó mezők helyesen vannak-e kitöltve:

```
<?php  
$validator = Validator::make($request->all(), [  
    'email' => 'required|email',  
    'password' => 'required',  
]);
```

- Ha az ellenőrzés sikertelen, 400 Bad Request válasz küldése:

```
<?php  
if ($validator->fails()) {  
    return response()->json(['error' => $validator->errors()], 400);  
}
```



2. Hitelesítés:

- A Auth::attempt metódus ellenőrzi a felhasználói hitelesítési adatokat:

```
<?php
$credentials = $request->only('email', 'password');
if (Auth::attempt($credentials)) {
    $user = Auth::user();
}
```

3. Token generálása:

- Ha a hitelesítés sikeres, a rendszer létrehoz egy hozzáférési tokent a Sanctum segítségével:

```
<?php
$token = $user->createToken('authToken')->plainTextToken;
```

4. Válasz küldése:

- Sikeres hitelesítés esetén a válasz tartalmazza:
 - A hozzáférési tokent (access_token).
 - A token típusát (Bearer).

A felhasználó adatait, például az azonosítót, keresztnévet és jogosultsági szintet (isadmin):

```
<?php
return response()->json([
    'access_token' => $token,
    'token_type' => 'Bearer',
    'user' => [
        'id' => $user->id,
        'first_name' => $user->first_name,
        'isadmin' => $user->role->power,
    ],
]);
```



5. Hibakezelés:

- Ha a hitelesítés sikertelen, 401 Unauthorized válasz küldése:

```
<?php  
    return response()->json(['error' => 'Unauthorized'], 401);
```

Példa API-hívás

Bejelentkezés

- **Módszer:** POST
- **Útvonal:** /login
- **Kérés törzse:**

```
{  
    "email": "user@example.com",  
    "password": "password123"  
}
```

- **Válasz (sikeres hitelesítés esetén):**

```
{  
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",  
    "token_type": "Bearer",  
    "user": {  
        "id": 1,  
        "first_name": "John",  
        "isadmin": 70  
    }  
}
```



- **Válasz (sikertelen hitelesítés esetén):**

```
{  
    "error": "Unauthorized"  
}
```

- **Válasz (hibás bemenet esetén):**

```
{  
    "error": {  
        "email": ["The email field is required."],  
        "password": ["The password field is required."  
    }  
}
```

Felhasználási esetek

- **Felhasználói hitelesítés:**

A metódus biztosítja, hogy csak a helyes hitelesítési adatokkal rendelkező felhasználók férjenek hozzá a rendszerhez.

- **Hozzáférési token generálása:**

A sikeres bejelentkezés után a felhasználó egy Bearer típusú hozzáférési tokent kap, amelyet az API-k eléréséhez használhat.

Felhasználói élmény

- **Egyszerű bejelentkezés:**

A felhasználók gyorsan és egyszerűen bejelentkezhetnek az email címük és jelszavuk megadásával.

- **Biztonságos hozzáférés:**

A hozzáférési tokenek biztosítják, hogy a felhasználók csak hitelesített módon érhessék el az API-kat.

- **Részletes válaszok:**

A válasz tartalmazza a felhasználó alapvető adatait és jogosultsági szintjét, ami segíti az ügyféloldali alkalmazás működését.



Összegzés

A LoginController egy jól strukturált vezérlő, amely hatékonyan kezeli a felhasználók bejelentkezését és hozzáférési tokenek generálását. A részletes hibakezelés és a biztonságos hitelesítési folyamat biztosítja a rendszer integritását és a felhasználói élmény magas szintjét.

OrderController.php – Laravel vezérlő a rendelések kezelésére

Az OrderController a rendelések kezelésére szolgáló vezérlő, amely lehetővé teszi a rendelések létrehozását, listázását, frissítését és törlését. Ez a vezérlő biztosítja a szükséges API-k működését a FlexiBox rendszerben.

Főbb metódusok

1. store(Request \$request)

- **Leírás:**
Új rendelés létrehozása, beleértve a cím és a rendelési tételek mentését.
- **Funkciók:**
 - Ellenőrzi a bemeneti adatokat, például a várost, irányítószámot, email címet, utcát, házszámot, felhasználói azonosítót, fizetési módöt és a kosár tételeit.
 - Létrehoz egy új Address rekordot.
 - Kiszámítja a rendelés teljes összegét.
 - Létrehoz egy új Order rekordot.
 - Létrehozza a rendelési tételeket (OrderItem).
- **Válasz:**
 - **Sikeres létrehozás (201 Created):**

```
{  
    "message": "Order created successfully"  
}
```



2. index(Request \$request)

- **Leírás:**

Egy adott felhasználó legutóbbi rendelésének lekérdezése.

- **Funkciók:**

- A felhasználói azonosítót a kérés fejlécéből (userId) olvassa ki.
- Lekérdezi a legutóbbi rendelést a kapcsolódó tételekkel, címekkel és felhasználói adatokkal.

- **Válasz:**

- **Sikeres lekérdezés (200 OK):**

A legutóbbi rendelés adatai.

3. userorderindex(Request \$request)

- **Leírás:**

Egy adott felhasználó összes rendelésének lekérdezése.

- **Funkciók:**

- A felhasználói azonosítót a kérés fejlécéből (userId) olvassa ki.
- Lekérdezi az összes rendelést a kapcsolódó tételekkel, címekkel és felhasználói adatokkal.

- **Válasz:**

- **Sikeres lekérdezés (200 OK):**

Az összes rendelés adatai.

4. orderindex(Request \$request)

- **Leírás:**

Az összes rendelés lekérdezése adminisztrátorok számára.



- **Funkciók:**

- Ellenőrzi, hogy a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik-e.
- Lekérdezi az összes rendelést a kapcsolódó tételekkel, címekkel és felhasználói adatokkal.

- **Válasz:**

- **Sikeres lekérdezés (200 OK):**
Az összes rendelés adatai.
 - **Hozzáférés megtagadva (403 Forbidden):**
Ha a felhasználó nem rendelkezik megfelelő jogosultsággal.
-

5. storeIsAddress(Request \$request)

- **Leírás:**

Új rendelés létrehozása meglévő cím használatával.

- **Funkciók:**

- Ellenőrzi a bemeneti adatokat, például a cím azonosítóját, felhasználói azonosítót, fizetési módot és a kosár tételeit.
- Kiszámítja a rendelés teljes összegét.
- Létrehoz egy új Order rekordot.
- Létrehozza a rendelési tételeket (OrderItem).

- **Válasz:**

- **Sikeres létrehozás (201 Created):**

```
{  
    "message": "Order created successfully"  
}
```



6. deleteOrder(Request \$request)

- **Leírás:**

Egy rendelés törlése adminisztrátorok számára.

- **Funkciók:**

- Ellenőrzi, hogy a bejelentkezett felhasználó adminisztrátori jogosultsággal rendelkezik-e.
- Törli a rendelést és a hozzá tartozó rendelési tételeket.

- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Rendelés sikeresen törölve!"  
}
```

- **Hozzáférés megtagadva (403 Forbidden):**

Ha a felhasználó nem rendelkezik megfelelő jogosultsággal.

7. userdeleteOrder(Request \$request)

- **Leírás:**

Egy rendelés törlése a felhasználó által, ha az a saját rendelése.

- **Funkciók:**

- Ellenőrzi, hogy a bejelentkezett felhasználó jogosult-e a rendelés törlésére.
- Törli a rendelést és a hozzá tartozó rendelési tételeket.

- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Rendelés sikeresen törölve!"  
}
```



- **Hozzáférés megtagadva (403 Forbidden):**
Ha a rendelés nem a felhasználóhoz tartozik.
-

Példa API-hívások

Új rendelés létrehozása

- **Módszer:** POST
- **Útvonal:** /order
- **Kérés törzse:**

```
{  
    "city": "Budapest",  
    "zip": "1234",  
    "email": "example@example.com",  
    "street_id": 1,  
    "street": "Fő utca",  
    "house_number": "12",  
    "user_id": 1,  
    "payments_method_id": 2,  
    "cart_items": [  
        {  
            "quantity": 2,  
            "product_id": 1,  
            "item_price": 5000,  
            "lockerId": 3  
        }  
    ]  
}
```

Rendelések listázása adminisztrátorok számára

- **Módszer:** GET
- **Útvonal:** /orderslist

Rendelés törlése adminisztrátor által

- **Módszer:** DELETE
- **Útvonal:** /orders



- Kérés törzse:

```
{  
    "order_id": 1  
}
```

Felhasználói élmény

- Egyszerű rendeléskezelés:

Az API-k lehetővé teszik a rendelések gyors létrehozását, listázását és törlését.

- Biztonságos hozzáférés:

A jogosultsági ellenőrzések biztosítják, hogy csak a megfelelő felhasználók férjenek hozzá az érzékeny adatokhoz.

Összegzés

Az OrderController egy jól strukturált vezérlő, amely hatékonyan kezeli a rendelések kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiBox rendszerben.



PaymentController.php – Laravel vezérlő a fizetési módok kezelésére

A PaymentController a fizetési módok kezelésére szolgáló vezérlő, amely lehetővé teszi a fizetési módok létrehozását, listázását, frissítését és törlését. Ez a vezérlő biztosítja a szükséges API-k működését a FlexiBox rendszerben.

Főbb metódusok

1. store(Request \$request)

- **Leírás:**

Új fizetési mód létrehozása.

- **Funkciók:**

- Ellenőri a bemeneti adatokat:
 - card_type: Kötelező, szöveg.
- Létrehoz egy új PaymentMethod rekordot az adatbázisban.
- Visszaadja a létrehozott fizetési mód adatait.

- **Válasz:**

- **Sikeres létrehozás (201 Created):**

```
{  
    "message": "Payment method created successfully",  
    "paymentMethod": {  
        "id": 1,  
        "card_type": "Visa"  
    }  
}
```



2. index()

- **Leírás:**

Az összes fizetési mód listázása.

- **Funkciók:**

- Lekérdezi az összes PaymentMethod rekordot az adatbázisból.
- Visszaadja a fizetési módok listáját JSON formátumban.

- **Válasz:**

- **Sikeres lekérdezés (200 OK):**

```
{  
    "paymentMethods": [  
        {  
            "id": 1,  
            "card_type": "Visa"  
        },  
        {  
            "id": 2,  
            "card_type": "MasterCard"  
        }  
    ]  
}
```

3. update(Request \$request)

- **Leírás:**

Egy meglévő fizetési mód adatainak frissítése.

- **Funkciók:**

- Ellenörzi a bemeneti adatokat:
 - card_type: Kötelező, szöveg.
- Lekérdezi a frissítendő PaymentMethod rekordot az id alapján.
- Ha a fizetési mód nem található, 404 Not Found választ küld.
- Frissíti a fizetési mód adatait, majd elmenti az adatbázisba.
- Visszaadja a frissített fizetési mód adatait.



- **Válaszok:**

- **Sikeres frissítés (200 OK):**

```
{  
    "message": "Payment method updated successfully",  
    "paymentMethod": {  
        "id": 1,  
        "card_type": "Updated Visa"  
    }  
}
```

- **Nem található (404 Not Found):**

```
{  
    "message": "Payment method not found"  
}
```

4. destroy(Request \$request)

- **Leírás:**

Egy fizetési mód törlése.

- **Funkciók:**

- Lekérdezi a törlendő PaymentMethod rekordot az id alapján.
 - Ha a fizetési mód nem található, 404 Not Found választ küld.
 - Törli a fizetési mód rekordot az adatbázisból.
 - Visszaad egy sikeres törlés üzenetet.

- **Válaszok:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Payment method deleted successfully"  
}
```



- Nem található (404 Not Found):

```
{  
    "message": "Payment method not found"  
}
```

Példa API-hívások

Új fizetési mód létrehozása

- Módszer: POST
- Útvonal: /payment
- Kérés törzse:

```
{  
    "card_type": "Visa"  
}
```

Fizetési módok listázása

- Módszer: GET
- Útvonal: /payment

Fizetési mód frissítése

- Módszer: PATCH
- Útvonal: /payment
- Kérés törzse:

```
{  
    "id": 1,  
    "card_type": "Updated Visa"  
}
```



Fizetési mód törlése

- **Módszer:** DELETE
- **Útvonal:** /payment/delete
- **Kérés fejléc:**

PaymentId: 1

Felhasználói élmény

- **Egyszerű kezelés:**
Az API-k lehetővé teszik a fizetési módok gyors létrehozását, frissítését, törlését és lekérdezését.
 - **Részletes válaszok:**
Az API-k részletes válaszokat adnak, amelyek segítenek a fejlesztőket és a felhasználókat a műveletek eredményének megértésében.
-

Összegzés

A PaymentController egy jól strukturált vezérlő, amely hatékonyan kezeli a fizetési módok kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiStore rendszerben.



ProfileController.php – Laravel vezérlő a profilok kezelésére

A ProfileController a felhasználói profilok kezelésére szolgáló vezérlő, amely lehetővé teszi a profilok létrehozását, megtekintését, frissítését és törlését. Ez a vezérlő biztosítja a szükséges API-k működését a FlexiStore rendszerben.

Főbb metódusok

1. store(Request \$request)

- **Leírás:**

Új profil létrehozása egy kép feltöltésével.

- **Funkciók:**

- Ellenőri a bemeneti adatokat:
 - user_id: Kötelező, létező felhasználói azonosító.
 - image: Kötelező, fájl, amely JPEG, PNG, JPG, GIF vagy SVG formátumú, és legfeljebb 8 MB méretű.
 - Feltölti a képet a public/images mappába.
 - Létrehoz egy új Profile rekordot az adatbázisban.
 - Naplózza a sikeres profil létrehozását.

- **Válasz:**

- **Sikeres létrehozás (201 Created):**

```
{  
    "id": 1,  
    "user_id": 1,  
    "file_name": "example.jpg",  
    "file_path": "/storage/images/example.jpg"  
}
```



2. index(Request \$request)

- **Leírás:**

Egy felhasználó profiljának lekérdezése.

- **Funkciók:**

- A felhasználói azonosítót a kérés fejlécéből (userId) olvassa ki.
- Lekérdezi a felhasználóhoz tartozó Profile rekordot.
- Ha a profil nem található, üres válasz küldése.

- **Válasz:**

- **Sikeres lekérdezés (200 OK):**

```
{  
    "id": 1,  
    "user_id": 1,  
    "file_name": "example.jpg",  
    "file_path": "/storage/images/example.jpg"  
}
```

- **Nem található (200 OK):**

```
{  
    "message": "Profile not found"  
}
```

3. update(Request \$request)

- **Leírás:**

Egy meglévő profil frissítése, vagy új profil létrehozása, ha még nem létezik.

- **Funkciók:**

- Ellenőri a bemeneti adatokat:
 - image: Opcionális, fájl, amely JPEG, PNG, JPG, GIF vagy SVG formátumú, és legfeljebb 8 MB méretű.
- Ha a profil nem létezik, új Profile rekordot hoz létre.



- Ha a profil létezik:
 - Törli a meglévő képet a tárhelyről.
 - Feltölti az új képet.
 - Frissíti a profil adatait.
 - Naplózza a sikeres frissítést.
- Válasz:
- Sikeres frissítés vagy létrehozás (200 OK vagy 201 Created):

```
{  
    "id": 1,  
    "user_id": 1,  
    "file_name": "updated_example.jpg",  
    "file_path": "/storage/images/updated_example.jpg"  
}
```

4. destroy(Request \$request)

- Leírás:
Egy profil törlése.
- Funkciók:
 - A felhasználói azonosítót a kérés fejlécéből (userId) olvassa ki.
 - Lekérdezi a törlendő Profile rekordot.
 - Ha a profil nem található, üres válasz küldése.
 - Törli a profilhoz tartozó képet a tárhelyről.
 - Törli a Profile rekordot az adatbázisból.
 - Naplózza a sikeres törlést.
- Válasz:
 - Sikeres törlés (200 OK):

```
{  
    "message": "Profile deleted successfully"  
}
```



- Nem található (200 OK):

{

"message": "Profile not found"

Példa API-hívások

Új profil létrehozása

- Módszer: POST
- Útvonal: /profile
- Kérés törzse (form-data):
 - user_id: 1
 - image: [Fájl feltöltése]

Profil lekérdezése

- Módszer: GET
- Útvonal: /profile/index
- Fejléc:

userId: 1

Profil frissítése

- Módszer: POST
- Útvonal: /profile/update
- Fejléc:

userId: 1

- Kérés törzse (form-data):
 - image: [Új fájl feltöltése]



Profil törlése

- **Módszer:** DELETE
- **Útvonal:** /profile/delete
- **Fejléc:**

userId: 1

Felhasználói élmény

- **Egyszerű kezelés:**
Az API-k lehetővé teszik a profilkönyvek gyors létrehozását, frissítését, törlését és lekérdezését.
 - **Biztonságos fájlkezelés:**
A képek feltöltése és törlése biztonságosan történik a tárhelyen.
 - **Részletes naplózás:**
A naplózás segíti a hibakeresést és a műveletek nyomon követését.
-

Összegzés

A ProfileController egy jól strukturált vezérlő, amely hatékonyan kezeli a felhasználói profilkönyvek kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiBox rendszerben.



RegisterController.php – Laravel vezérlő a regisztráció és felhasználókezelés kezelésére

A RegisterController a felhasználók regisztrációját, frissítését, törlését és szerepkörök kezelését biztosítja. Ez a vezérlő lehetővé teszi a felhasználók és szerepkörök adminisztrációját a FlexiBox rendszerben.

Főbb metódusok

1. register(Request \$request)

- **Leírás:**
Új felhasználó regisztrációja.
- **Funkciók:**
 - Ellenőrzi a bemeneti adatokat:
 - first_name, last_name, email, password, role_id.
 - Alapértelmezett szerepkörként a "user" szerepkört állítja be, ha nincs megadva.
 - Létrehoz egy új User rekordot, és hash-eli a jelszót.
 - Generál egy hozzáférési tokent.
- **Válasz:**
 - **Sikeres regisztráció (201 Created):**

```
{  
    "message": "Sikeres regisztráció!",  
    "user": {  
        "id": 1,  
        "first_name": "John",  
        "last_name": "Doe",  
        "email": "john.doe@example.com",  
        "role_id": 2  
    },  
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."  
}
```



2. update(Request \$request)

- **Leírás:**

Egy meglévő felhasználó adatainak frissítése.

- **Funkciók:**

- Ellenőrzi a bemeneti adatokat.
- Lekérdezi a frissítendő felhasználót az email alapján.
- Frissíti a felhasználó adatait, beleértve a jelszót is.

- **Válasz:**

- **Sikeres frissítés (201 Created):**

```
{  
    "message": "Sikeres módosítás!",  
    "user": {  
        "id": 1,  
        "first_name": "John",  
        "last_name": "Doe",  
        "email": "john.doe@example.com"  
    }  
}
```

3. edit(Request \$request)

- **Leírás:**

Egy felhasználó adatainak szerkesztése adminisztrátor által.

- **Funkciók:**

- Ellenőrzi a bemeneti adatokat.
- Ellenőrzi, hogy az adminisztrátor jogosult-e a felhasználó adatainak szerkesztésére.
- Frissíti a felhasználó adatait, beleértve a szerepkört is.



- **Válasz:**

- **Sikeres szerkesztés (201 Created):**

```
{  
    "message": "Sikeres módosítás!",  
    "user": {  
        "id": 1,  
        "first_name": "John",  
        "last_name": "Doe",  
        "email": "john.doe@example.com",  
        "role_id": 2  
    }  
}
```

4. destroy(Request \$request)

- **Leírás:**

Egy felhasználó törlése.

- **Funkciók:**

- Ellenőrzi, hogy a bejelentkezett felhasználó jogosult-e a törlésre.
 - Törli a felhasználóhoz tartozó profilképet, címeket és a felhasználót.

- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Sikeres törlés!"  
}
```

5. roleStore(Request \$request)

- **Leírás:**

Új szerepkör létrehozása.

- **Funkciók:**

- Ellenőrzi a bemeneti adatokat.
 - Létrehoz egy új Role rekordot.



- Válasz:

- Sikeres létrehozás (201 Created):

```
{  
    "message": "Role created",  
    "role": {  
        "id": 1,  
        "power": 70,  
        "warrant_name": "admin"  
    }  
}
```

6. roleIndex()

- Leírás:

Az összes szerepkör listázása.

- Funkciók:

- Lekérdezi az összes Role rekordot.

- Válasz:

- Sikeres lekérdezés (200 OK):

```
{  
    "roles": [  
        {  
            "id": 1,  
            "power": 70,  
            "warrant_name": "admin"  
        },  
        {  
            "id": 2,  
            "power": 10,  
            "warrant_name": "user"  
        }  
    ]  
}
```



7. roleUpdate(Request \$request)

- **Leírás:**

Egy szerepkör adatainak frissítése.

- **Funkciók:**

- Ellenőri a bemeneti adatokat.
- Frissíti a megadott Role rekordot.

- **Válasz:**

- **Sikeres frissítés (200 OK):**

```
{  
    "message": "Role updated",  
    "role": {  
        "id": 1,  
        "power": 80,  
        "warrant_name": "superadmin"  
    }  
}
```

8. roleDestroy(Request \$request)

- **Leírás:**

Egy szerepkör törlése.

- **Funkciók:**

- Törli a megadott Role rekordot.

- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Role deleted"  
}
```



Felhasználói élmény

- **Egyszerű regisztráció és felhasználókezelés:**
Az API-k lehetővé teszik a felhasználók gyors regisztrációját, frissítését és törlését.
- **Szerepkörök adminisztrációja:**
Az adminisztrátorok könnyen kezelhetik a szerepköröket, beleértve azok létrehozását, frissítését és törlését.
- **Biztonságos hozzáférés:**
A jogosultsági ellenőrzések biztosítják, hogy csak a megfelelő felhasználók férjenek hozzá az érzékeny adatokhoz.

Összegzés

A RegisterController egy jól strukturált vezérlő, amely hatékonyan kezeli a felhasználók és szerepkörök adminisztrációját. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiBox rendszerben.



ServiceController.php – Laravel vezérlő a kategóriák és termékek kezelésére

A ServiceController a kategóriák és termékek kezelésére szolgáló vezérlő, amely lehetővé teszi a kategóriák és termékek létrehozását, frissítését, törlését és listázását. Ez a vezérlő biztosítja a szükséges API-k működését a FlexiBox rendszerben.

Főbb metódusok

1. categoryStore(Request \$request)

- **Leírás:**
Új kategória létrehozása.
- **Funkciók:**
 - Ellenőrzi a bemeneti adatokat:
 - name: Kötelező, szöveg.
 - Létrehoz egy új Category rekordot az adatbázisban.
- **Válasz:**
 - Sikeres létrehozás (201 Created):

```
{  
    "message": "Category created successfully",  
    "category": {  
        "id": 1,  
        "name": "Electronics"  
    }  
}
```

2. productStore(Request \$request)

- **Leírás:**
Új termék létrehozása képfeltöltéssel.



- **Funkciók:**

- Ellenőrzi a bemeneti adatokat:
 - image, name, description, price_per_day, category_id, available, locker_ids.
- Feltölти a képet a public/product_images mappába.
- Létrehoz egy új Product rekordot az adatbázisban.
- Hozzárendeli a terméket a megadott csomagautomatákhoz.

- **Válasz:**

- **Sikeres létrehozás (201 Created):**

```
{  
    "message": "Product created successfully",  
    "product": {  
        "id": 1,  
        "name": "Laptop",  
        "description": "High-end gaming laptop",  
        "price_per_day": 5000,  
        "category_id": 1,  
        "available": true,  
        "file_path": "/storage/product_images/laptop.jpg",  
        "lockers": [  
            {  
                "id": 1,  
                "locker_name": "Locker A"  
            }  
        ]  
    }  
}
```

3. index()

- **Leírás:**

Az összes kategória listázása.

- **Funkciók:**

- Lekérdezi az összes Category rekordot.



- **Válasz:**

- **Sikeress lekérdezés (200 OK):**

```
{  
  "categories": [  
    {  
      "id": 1,  
      "name": "Electronics"  
    },  
    {  
      "id": 2,  
      "name": "Furniture"  
    }  
  ]  
}
```

4. productIndex()

- **Leírás:**

Az összes termék listázása a kapcsolódó kategóriákkal és csomagautomatákkal.

- **Funkciók:**

- Lekérdezi az összes Product rekordot a kapcsolódó Category és Locker adatokkal.

- **Válasz:**

- **Sikeress lekérdezés (200 OK):**

```
{  
  "products": [{  
    "id": 1,  
    "name": "Laptop",  
    "description": "High-end gaming laptop",  
    "price_per_day": 5000,  
    "category": {  
      "id": 1,  
      "name": "Electronics"  
    },  
    "lockers": [  
      {  
        "id": 1,  
        "locker_name": "Locker A"  
      }]  
  }]
```



5. productUpdate(Request \$request)

- **Leírás:**

Egy meglévő termék adatainak frissítése.

- **Funkciók:**

- Ellenőrzi a bemeneti adatokat.
- Ha új kép van feltöltve, törli a régi képet, és feltölti az újat.
- Frissíti a termék adatait, beleértve a csomagautomaták hozzárendelését is.

- **Válasz:**

- **Sikeres frissítés (200 OK):**

```
{  
    "message": "Product updated successfully",  
    "product": {  
        "id": 1,  
        "name": "Updated Laptop",  
        "description": "Updated description",  
        "price_per_day": 6000,  
        "category_id": 1,  
        "available": true,  
        "file_path": "/storage/product_images/updated_laptop.jpg",  
        "lockers": [  
            {  
                "id": 1,  
                "locker_name": "Locker A"  
            }]  
    }  
}
```

6. productDestroy(Request \$request)

- **Leírás:**

Egy termék törlése.

- **Funkciók:**

- Lekérdezi a törlendő terméket az id alapján.
- Törli a termékhez tartozó képet és az adatbázis rekordot.



- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Product deleted successfully"  
}
```

7. destroy(Request \$request)

- **Leírás:**

Egy kategória törlése.

- **Funkciók:**

- Lekérdezi a törlendő kategóriát az id alapján.
 - Törli az adatbázis rekordot.

- **Válasz:**

- **Sikeres törlés (200 OK):**

```
{  
    "message": "Service deleted successfully"  
}
```

8. update(Request \$request)

- **Leírás:**

Egy kategória adatainak frissítése.

- **Funkciók:**

- Frissíti a kategória nevét.



- **Válasz:**
 - **Sikeres frissítés (200 OK):**

{
}
}

"message": "Service updated successfully"

Felhasználói élmény

- **Egyszerű kezelés:**

Az API-k lehetővé teszik a kategóriák és termékek gyors létrehozását, frissítését, törlését és listázását.
- **Biztonságos fájlkezelés:**

A képek feltöltése és törlése biztonságosan történik a tárhelyen.
- **Részletes válaszok:**

Az API-k részletes válaszokat adnak, amelyek segítik a fejlesztőket és a felhasználókat a műveletek eredményének megértésében.

Összegzés

A ServiceController egy jól strukturált vezérlő, amely hatékonyan kezeli a kategóriák és termékek kezeléséhez szükséges funkciókat. Az API-k könnyen használhatók, és biztosítják a szükséges adatkezelési műveleteket a FlexiBox rendszerben.



React Controller dokumentáció – NewPublicArea.jsx

A [NewPublicArea](#) React komponens a közterületek kezelésére szolgál, lehetővé téve új közterületek felvitelét és meglévők módosítását. A komponens szorosan együttműködik a Laravel backenddel az API-hívásokon keresztül.

API-hívások

1. Új közterület felvitele

- **HTTP-módszer:** POST
- **Végpont:** /publicareaname
- **Küldött adatok:**

```
{  
  "public_area_name": "Fő utca"  
}
```

- **Context:**

A [CrudContext backendMuvelet](#) függvényét használja az API-hívás lebonyolítására.

2. Meglévő közterület módosítása

- **HTTP-módszer:** PATCH
- **Végpont:** /publicareaname
- **Küldött adatok:**

```
{  
  "id": 1,  
  "public_area_name": "Módosított utca"  
}
```

- **Context:**

A [CrudContext backendMuvelet](#) függvényét használja az API-hívás lebonyolítására.



Context használata

A komponens két context-et használ az API-hívásokhoz és az adatok frissítéséhez:

1. CrudContext:

A backendMuvelet függvény segítségével végzi az API-hívásokat. Ez a függvény a következő paramétereket kapja:

- formData: Az űrlap adatai.
- method: Az HTTP-módszer (POST vagy PATCH).
- url: Az API végpont URL-je.
- header: Az API-hívás fejlécadatai, beleértve a hitelesítési tokent.
- successMessage: Sikeres művelet esetén megjelenítendő üzenet.
- errorMessage: Sikertelen művelet esetén megjelenítendő üzenet.

2. InitialContext:

Az updatePublicAreaName függvény segítségével frissíti a közterületek listáját a sikeres művelet után.

Adatok kezelése

• Úrlap adatok kezelése:

A useState hook segítségével kezeli az űrlap adatait. Az alapértelmezett értékek:

```
{  
  public_area_name: ""  
}
```

Módosítás esetén:

```
{  
  id: publicarea.id,  
  public_area_name: publicarea.public_area_name  
}
```



- **Adatok frissítése:**

Az onChange eseménykezelő frissíti az állapotot az űrlap mezőinek változásakor:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

Felhasználói élmény

- **Egyszerű űrlapkezelés:**

Az űrlap intuitív és könnyen használható, a felhasználó gyorsan felvihet vagy módosíthat közterületeket.

- **Valós idejű visszajelzés:**

A sikeres vagy sikertelen műveletekről azonnali visszajelzést kap a felhasználó.

- **Rezonansív dizájn:**

A komponens mobil- és asztali eszközökön egyaránt jól használható.

Összegzés

A NewPublicArea React komponens hatékonyan kezeli a közterületek felvitelét és módosítását, miközben szorosan együttműködik a Laravel backenddel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív.



React Controller Dokumentáció – PublicAreaCard.jsx

A [PublicAreaCard](#) React komponens egy közterület kártyáját jeleníti meg, amely lehetővé teszi a közterület adatainak megtekintését, módosítását és törlését. Ez a komponens szorosan együttműködik a Laravel backenddel az API-hívásokon keresztül.

Funkciók

1. Közterület módosítása

- Leírás:**
A felhasználó a "Módosítás" gombra kattintva navigálhat a közterület módosítására szolgáló oldalra (/newpublicarea), ahol az aktuális közterület adatai előre kitöltésre kerülnek.
- Navigáció:**
A [useNavigate](#) hook segítségével történik.

2. Közterület törlése

- Leírás:**
A felhasználó a "Törlés" gombra kattintva törölheti a közterületet. A komponens egy DELETE API-hívást küld a backendnek, és frissíti a közterületek listáját.
- Backend végpont:**
`DELETE /publicareaname/delete`
- Adatok:**
 - StreetTypeId:** A törlendő közterület azonosítója.

API-hívások

1. Közterület törlése

- HTTP-módszer:** DELETE
- Végpont:** `/publicareaname/delete`



- **Küldött adatok** (fejlécben):

```
{  
    "Content-Type": "application/json",  
    "Authorization": "Bearer <usertoken>",  
    "StreetTypeId": 1  
}
```

- **Context:**

A [CrudContext backendMuvelet](#) függvényét használja az API-hívás lebonyolítására.

- **Sikeres válasz:**

```
{  
    "message": "Közterület sikeresen töröltre!"  
}
```

- **Hibás válasz:**

```
{  
    "error": "Nem sikerült a közterület törlése."  
}
```

Context használata

1. CrudContext

A [backendMuvelet](#) függvény segítségével végzi az API-hívásokat. Ez a függvény a következő paramétereket kapja:

- **formData:** Az API-hívás törzse (jelen esetben null).
- **method:** Az HTTP-módszer (DELETE).
- **url:** Az API végpont URL-je.
- **header:** Az API-hívás fejlécadatai, beleértve a hitelesítési tokenet és a törlendő közterület azonosítóját.
- **successMessage:** Sikeres művelet esetén megjelenítendő üzenet.
- **errorMessage:** Sikertelen művelet esetén megjelenítendő üzenet.



2. InitialContext

Az [updatePublicAreaName](#) függvény segítségével frissíti a közterületek listáját a sikeres törlés után.

Adatok kezelése

- **Módosítás navigációja:**

A [modosit](#) függvény a [useNavigate](#) hook segítségével navigál a közterület módosítására szolgáló oldalra, és az aktuális közterület adatait átadja az állapotban:

```
const modosit = () => {
  navigate("/newpublicarea", { state: { publicarea } });
};
```

- **Törlés API-hívás:**

A [torles](#) függvény a [backendMuvelet](#) függvényt használja az API-hívás lebonyolítására:

```
const torles = (publicarea) => {
  const method = "DELETE";
  const url = `${import.meta.env.VITE_BASE_URL}/publicareaname/delete`;
  const header = {
    "Content-Type": "application/json",
    Authorization: `Bearer ${sessionStorage.getItem("usertoken")}`,
    StreetTypeId: publicarea.id,
  };
  const successMessage = "Közterület sikeresen törölve!";
  const errorMessage = "Nem sikerült a közterület törlése.";
  updatePublicAreaName();
  backendMuvelet(null, method, url, header, successMessage, errorMessage);
};
```



Felhasználói Élmény

- **Adminisztrátori jogosultság ellenőrzése:**

A komponens csak akkor jeleníti meg a "Módosítás" és "Törlés" gombokat, ha a felhasználó adminisztrátori jogosultsággal rendelkezik (`isadmin >= 70`).

- **Valós idejű frissítés:**

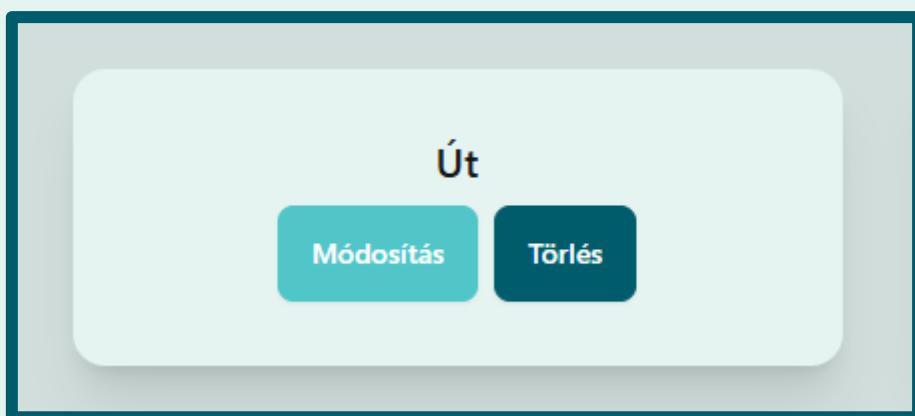
A törlés után a közterületek listája automatikusan frissül az `updatePublicAreaName` függvény segítségével.

- **Rezonansív dizájn:**

A kártya mobil- és asztali eszközökön egyaránt jól használható.

Összegzés

A `PublicAreaCard` React komponens hatékonyan kezeli a közterületek megjelenítését, módosítását és törlését. Az API-hívások és a context-ek használata jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens szorosan együttműködik a Laravel backenddel, biztosítva a RESTful architektúra elveinek betartását.



11. ábra PublicAreaCard



React Controller Dokumentáció – PublicAreaList.jsx

A [PublicAreaList](#) React komponens a közterületek listáját jeleníti meg, és minden közterülethez egy-egy [PublicAreaCard](#) komponenst renderel. Ez a komponens a [InitialContext](#) segítségével szerzi be a közterületek adatait, és felelős azok vizuális megjelenítéséért.

Funkciók

1. Közterületek listázása

- Leírás:**
A komponens a [InitialContext](#)-ből szerzi be a közterületek adatait ([areas](#)), majd minden közterülethez egy [PublicAreaCard](#) komponenst renderel.
- Adatok forrása:**
A közterületek adatai a backend API-ból származnak, amelyeket a [InitialContext](#) tölt be és tárol.

API-hívások

A [PublicAreaList](#) komponens közvetlenül nem végez API-hívásokat, de a [InitialContext](#)-en keresztül hozzáfér a backend által biztosított adatokhoz. A közterületek listáját a következő API végpont biztosítja:

Közterületek lekérdezése

- HTTP-módszer:** GET
- Végpont:** /publicareaname



- Sikeres válasz:

```
{  
  "areas": [  
    {  
      "id": 1,  
      "public_area_name": "Fő utca"  
    },  
    {  
      "id": 2,  
      "public_area_name": "Kossuth tér"  
    }  
  ]  
}
```

Context használata

1. InitialContext

A InitialContext biztosítja a közterületek adatait (areas), amelyeket a komponens a useContext hook segítségével ér el:

```
const { areas } = useContext(InitialContext);
```

Az areas tömb tartalmazza az összes közterület adatait, amelyeket a backend API-ból tölt be a context.

Adatok kezelése

- **Közterületek megjelenítése:**

A komponens a areas tömb minden eleméhez egy PublicAreaCard komponensem renderel:

```
areas.map((area) => (<PublicAreaCard key={area.id} publicarea={area} />))
```

Ez biztosítja, hogy minden közterület külön kártyán jelenjen meg.



Felhasználói Élmény

- **Reszponzív dizájn:**

A közterületek listája rugalmasan igazodik a képernyő méretéhez, és a kártyák egymás mellett vagy alatt jelennek meg a rendelkezésre álló helytől függően.

- **Egyszerű navigáció:**

A PublicAreaCard komponenseken keresztül a felhasználó könnyen módosíthatja vagy törölheti a közterületeket, ha adminisztrátori jogosultsággal rendelkezik.

Komponens Felépítése

1. Context használata

A useContext hook segítségével a komponens hozzáfér a InitialContext által biztosított adatokhoz:

```
const { areas } = useContext(InitialContext);
```

2. Kártyák renderelése

A komponens a areas tömb minden eleméhez egy PublicAreaCard komponenst renderel:

```
<div className="flex flex-row flex-wrap items-center justify-center">
  {
    areas.map((area) => (<PublicAreaCard key={area.id} publicarea={area} />))
  }
</div>
```

Felhasználói Élmény

- **Átlátható megjelenítés:**

A közterületek listája jól strukturált, és minden közterület külön kártyán jelenik meg.

- **Interaktív funkciók:**

A PublicAreaCard komponenseken keresztül a felhasználó módosíthatja vagy törölheti a közterületeket, ha jogosultsággal rendelkezik.



- **Reszponzív kialakítás:**

A lista mobil- és asztali eszközökön egyaránt jól használható.

Összegzés

A PublicAreaList React komponens hatékonyan kezeli a közterületek listázását, miközben a InitialContext segítségével biztosítja az adatok elérését. A komponens egyszerű és intuitív felhasználói élményt nyújt, és szorosan együttműködik a PublicAreaCard komponenssel a közterületek kezelésében.

The screenshot shows a user interface for managing public areas. At the top, there's a navigation bar with links for 'Csomagautomaták', 'Termékek', 'Admin Dashboard', and a user profile icon. Below the navigation, the main title is 'Közterület neveinek Listája'. The interface is organized into a 4x4 grid of cards, each representing a different type of public area. Each card has a small title at the top and two buttons at the bottom labeled 'Módosítás' and 'Törles'.

12. ábra PublicAreas List



React Controller Dokumentáció – Login2.jsx

A Login2 React komponens a felhasználók bejelentkezésére szolgál. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, hogy hitelesítse a felhasználót, és elmentse a hozzáférési tokent a munkamenet során.

Funkciók

1. Bejelentkezés

- **Leírás:**
A felhasználó megadja az email címét és jelszavát, amelyeket a komponens elküld a backendnek egy POST kérésben. Sikeres hitelesítés esetén a rendszer elmenti a hozzáférési tokent és a felhasználói adatokat.
- **Backend végpont:**
POST /login
- **Küldött adatok:**
 - email: A felhasználó email címe.
 - password: A felhasználó jelszava.

API-hívások

1. Bejelentkezés

- **HTTP-módszer:** POST
- **Végpont:** /login
- **Küldött adatok:**

```
{  
  "email": "user@example.com",  
  "password": "password123"  
}
```



- Sikeres válasz:

```
{  
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",  
  "user": {  
    "id": 1,  
    "first_name": "John",  
    "isadmin": 70  
  }  
}
```

- Hibás válasz:

```
{  
  "error": "Unauthorized"  
}
```

Context használata

1. AuthContext

A komponens a AuthContext-et használja a felhasználói adatok és a hitelesítési állapot frissítésére:

- update: Frissíti a hitelesítési állapotot.
- setUser: Beállítja a bejelentkezett felhasználó adatait.

Példa használat:

```
const { update, setUser } = useContext(AuthContext);
```



Adatok kezelése

1. Űrlap adatok kezelése

A useState hook segítségével kezeli az űrlap adatait. Az alapértelmezett értékek:

```
let formObj = {
  email: "",
  password: "",
};
const [formData, setFormData] = useState(formObj);
```

Az onChange eseménykezelő frissíti az állapotot az űrlap mezőinek változásakor:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

2. Űrlap beküldése

Az onSubmit eseménykezelő elküldi az adatokat a backendnek:

```
const onSubmit = (e) => {
  e.preventDefault();
  kuldes(formData, "POST");
};
```



3. API-hívás lebonyolítása

A kuldes függvény végzi az API-hívást:

```
const kuldes = (formData, method) => {
  fetch(`${
    import.meta.env.VITE_BASE_URL
  }/login`, {
    method: method,
    headers: { "Content-type": "application/json" },
    body: JSON.stringify(formData),
  })
  .then((res) => res.json())
  .then((data) => {
    if (!data.error) {
      sessionStorage.setItem("usertoken", data.access_token);
      secureStorage.setItem('user', { ...data.user });
      update();
      setUser(data.user);
      toast.success("Sikeres belépés");
      navigate("/");
    } else {
      toast.error("Kérjük ellenőrizze a bejelentkezési adatait!");
    }
  })
  .catch((err) => toast.error(err));
};
```

Felhasználói Élmény

- **Valós idejű visszajelzés:**

A react-toastify segítségével a felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen bejelentkezésről.

- **Biztonságos adattárolás:**

A hozzáférési token a sessionStorage-ben kerül tárolásra, míg a felhasználói adatok a secureStorage-ben.

- **Egyszerű navigáció:**

Sikeres bejelentkezés után a felhasználó automatikusan a főoldalra (/) kerül átirányításra.



Komponens Felépítése

1. Űrlap

Az űrlap két mezőt tartalmaz:

- Email mező:

```
<input type="email" id="email" placeholder="Felhasználói email"  
required onChange={writeData} value={formData.email}/>
```

- Jelszó mező:

```
<input type="password" id="password" placeholder="Jelszó"  
required onChange={writeData} value={formData.password}/>
```

2. Gombok

- Bejelentkezés gomb:

```
<button type="submit" className="btn btn-primary text-white">  
    Bejelentkezés  
</button>
```

- Regisztráció link:

```
<Link to="/register2" className="link link-primary">  
    Regisztráljon most  
</Link>
```



Összegzés

A Login2 React komponens hatékonyan kezeli a felhasználók bejelentkezését, miközben szorosan együttműködik a Laravel backenddel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendszer többi részével.

The image shows a login form titled "Bejelentkezés". It contains two input fields: "Felhasználói email" (User email) and "Jelszó" (Password). Below the password field is a note: "• Jelszó minimum 8 karakter" (• Password must be at least 8 characters). At the bottom is a large teal button labeled "Bejelentkezés" (Login). Below the button is a horizontal line with the word "VAGY" (OR) in the center. Underneath is the text "Nincs fiókja?" (No account?) and a link "[Regisztráljon most](#)" (Register now).

13. ábra Login Panel



React Controller Dokumentáció – Profile.jsx

A Profile React komponens a felhasználói profil kezelésére szolgál. Ez a komponens lehetővé teszi a felhasználók számára a profilkép feltöltését, módosítását és törlését, miközben API-hívásokkal kommunikál a Laravel backenddel.

Funkciók

1. Profilkép feltöltése vagy módosítása

- **Leírás:**
A felhasználó feltölthet vagy módosíthat egy profilképet. A kép egy POST kérésben kerül elküldésre a backendnek.
- **Backend végpont:**
POST /profile/update
- **Küldött adatok:**
 - user_id: A felhasználó azonosítója.
 - image: A feltöltött kép fájlja.

2. Profil törlése

- **Leírás:**
A felhasználó törölheti a profilját, beleértve a profilképét is. A törlés egy DELETE kérésben történik.
- **Backend végpont:**
DELETE /profile/delete
- **Küldött adatok (fejlécben):**
 - Authorization: A felhasználó hitelesítési tokenje.
 - userId: A felhasználó azonosítója.



API-hívások

1. Profilkép feltöltése vagy módosítása

- **HTTP-módszer:** POST
- **Végpont:** /profile/update
- **Küldött adatok:**

```
{  
    "user_id": 1,  
    "image": "<fájl>"  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Profil sikeresen frissítve!",  
    "profile": {  
        "id": 1,  
        "file_path": "/storage/images/profile.jpg"  
    }  
}
```

- **Hibás válasz:**

```
{  
    "error": "Hiba a profil frissítése során."  
}
```

2. Profil törlése

- **HTTP-módszer:** DELETE
- **Végpont:** /profile/delete
- **Küldött adatok (fejlecben):**

```
{  
    "Authorization": "Bearer <usertoken>",  
    "userId": 1  
}
```



- Sikeres válasz:

```
{  
  "message": "Profil törölve!"  
}
```

- Hibás válasz:

```
{  
  "error": "Hiba a profil törlés során."  
}
```

Context használata

1. AuthContext

A komponens a AuthContext-et használja a felhasználói adatok és a hitelesítési állapot frissítésére:

- update: Frissíti a hitelesítési állapotot.
- setProfile: Beállítja a profil állapotát.

Példa használat:

```
const { user, update, setProfile } = useContext(AuthContext);
```

Adatok kezelése

1. Profilkép feltöltése

A onFileChange eseménykezelő frissíti az állapotot a kiválasztott fájl alapján:

```
const onChange = (e) => {  
  setImage(e.target.files[0]);  
};
```



A onSubmit eseménykezelő elküldi a kiválasztott képet a backendnek:

```
const onSubmit = async (e) => {
  e.preventDefault();
  if (!image) {
    toast.error('Kérlek válassz ki egy képet!');
    return;
  }
  const formData = new FormData();
  formData.append("user_id", user.user_id);
  formData.append("image", image);
  try {
    const response = await axios.post(url, formData, { headers: header });
    console.log(response.data);
    update();
    navigate('/');
  } catch (error) {
    toast.error('Hiba a profil frissítése során.', error);
  }
  toast.success('Sikeresen módosítva/feltöltve!');
  navigate('/');
};
```

2. Profil törlése

A handleDelete függvény végzi a profil törlését:

```
const handleDelete = async () => {
  try {
    const response = await axios.delete(`${import.meta.env.VITE_BASE_URL}/profile/delete`,
    {
      headers: {
        'Authorization': `Bearer ${token}`,
        'userId': user.id
      }
    });
    console.log(response.data);
    sessionStorage.removeItem('profile');
    setProfile(null);
    update();
    toast.success('Profil törölve!');
    navigate('/');
  } catch (error) {
    toast.error('Hiba a profil törlés során.', error);
  }
};
```



1. Profilkép megjelenítése

A komponens megjeleníti a profilképet, vagy egy alapértelmezett képet, ha nincs feltöltve:

```
<img  
src={  
    profile.file_path == null && !image  
    ? "https://images.unsplash.com/photo-1472099645785-5658abf4ff4e?ixlib=rb-  
1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=facearea&facepad=2&w=25  
6&h=256&q=80"  
    : image  
    ? URL.createObjectURL(image)  
    : `${import.meta.env.VITE_LARAVEL_IMAGE_URL}${profile.file_path}`  
}  
alt="Profile"  
className="w-20 h-20 rounded-full object-cover cursor-pointer"/>
```

2. Gombok

- Kép feltöltése/módosítása:

```
<button  
type="button"  
onClick={() => document.getElementById("fileInput").click()}  
className="mt-2 rounded-md bg-white px-4 py-2 text-sm font-semibold text-  
gray-900 shadow-sm ring-1 ring-inset ring-gray-300 hover:bg-gray-50"  
>  
    Kép feltöltése/módosítása  
</button>
```

- Profil mentése:

```
<button type='submit' className="btn btn-primary">Save</button>
```

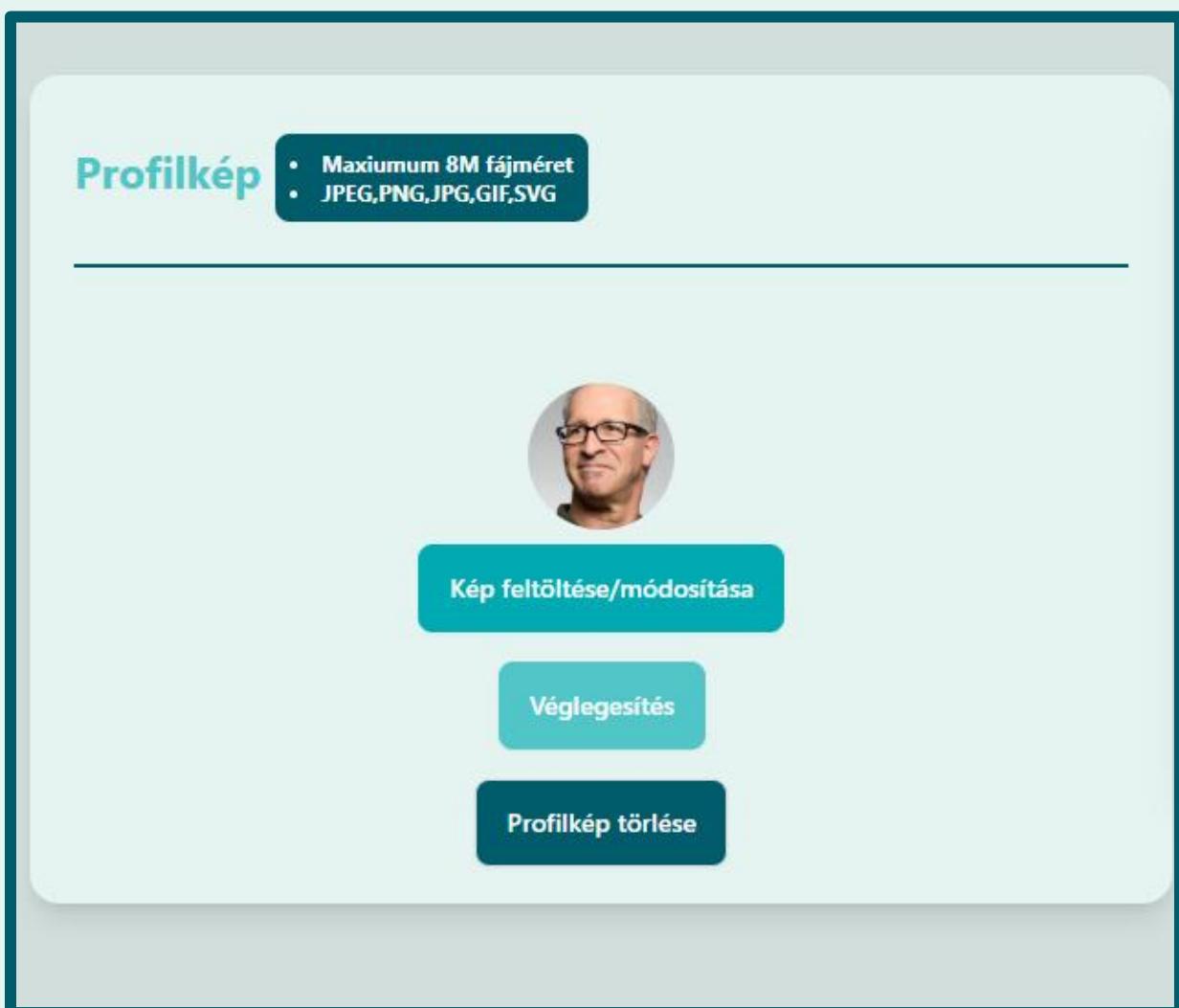
- Profil törlése:

```
<button type="button" onClick={() => handleDelete()} className="btn bg-rose-  
400 rounded-btn">  
    Delete  
</button>
```



Összegzés

A Profile React komponens hatékonyan kezeli a felhasználi profil kezelését, beleértve a profilkép feltöltését, módosítását és törlését. Az API-hívások és az állapotkezelés jól strukturált, a felhasználi élmény pedig egyszerű és intuitív. A komponens szorosan együttműködik a Laravel backenddel, biztosítva a RESTful architektúra elveinek betartását.



14. ábra Profile Picture Upload



React Controller Dokumentáció – Register2.jsx

A Register2 React komponens a felhasználók regisztrációjára szolgál. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, hogy új felhasználói fiókot hozzon létre, és sikeres regisztráció esetén elmentse a hozzáférési tokent.

Funkciók

1. Felhasználói regisztráció

- **Leírás:**
A felhasználó megadja a szükséges adatokat (keresztnév, vezetéknév, email, jelszó), amelyeket a komponens elküld a backendnek egy POST kérésben. Sikeres regisztráció esetén a rendszer elmenti a hozzáférési tokent.
- **Backend végpont:**
POST /register
- **Küldött adatok:**
 - first_name: A felhasználó keresztneve.
 - last_name: A felhasználó vezetékneme.
 - email: A felhasználó email címe.
 - password: A felhasználó jelszava.
 - password_confirmation: A jelszó megerősítése.

API-hívások

1. Regisztráció

- **HTTP-módszer:** POST
- **Végpont:** /register



- Küldött adatok:

```
{  
  "first_name": "John",  
  "last_name": "Doe",  
  "email": "john.doe@example.com",  
  "password": "password123",  
  "password_confirmation": "password123"  
}
```

- Sikeres válasz:

```
{  
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",  
  "user": {  
    "id": 1,  
    "first_name": "John",  
    "last_name": "Doe",  
    "email": "john.doe@example.com"  
  }  
}
```

- Hibás válasz:

```
{  
  "error": "A megadott email cím már létezik."  
}
```



Adatok kezelése

1. Űrlap adatok kezelése

A useState hook segítségével kezeli az űrlap adatait. Az alapértelmezett értékek:

```
let formObj = {
  first_name: "",
  last_name: "",
  email: "",
  password: "",
  passwordAgain: "",
};

const [formData, setFormData] = useState(formObj);
```

Az onChange eseménykezelő frissíti az állapotot az űrlap mezőinek változásakor:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

2. Űrlap beküldése

Az onSubmit eseménykezelő ellenőrzi a jelszavak egyezését, majd elküldi az adatokat a backendnek:

```
const onSubmit = (e) => {
  e.preventDefault();
  if (formData.password !== formData.passwordAgain) {
    toast.error("Jelszavak nem egyeznek!");
    return;
  }
  kuldes(formData, "POST");
};
```



3. API-hívás lebonyolítása

A kuldes függvény végzi az API-hívást:

```
const kuldes = (formData, method) => {
  const dataToSend = {
    first_name: formData.first_name,
    last_name: formData.last_name,
    email: formData.email,
    password: formData.password,
    password_confirmation: formData.passwordAgain,
  };
  fetch(`${
    import.meta.env.VITE_BASE_URL
  }/register`, {
    method: method,
    headers: { "Content-type": "application/json" },
    body: JSON.stringify(dataToSend),
  })
  .then((res) => res.json())
  .then((data) => {
    if (!data.error) {
      sessionStorage.setItem("usertoken", data.token);
      toast.success("Sikeres regisztráció!");
      navigate("/");
    } else {
      toast.error(data.message);
    }
  })
  .catch((error) => {
    toast.error("Hiba történt a regisztráció során!", error);
  });
};
```

Felhasználói Élmény

- Valós idejű visszajelzés:**
A react-toastify segítségével a felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen regisztrációról.
- Biztonságos adattárolás:**
A hozzáférési token a sessionStorage-ben kerül tárolásra.
- Egyszerű navigáció:**
Sikeres regisztráció után a felhasználó automatikusan a főoldalra (/) kerül átirányításra.



Komponens Felépítése

1. Űrlap

Az űrlap mezői:

- **Vezetéknév mező:**

```
<input type="text" id="last_name" className="grow" placeholder="Vezetéknév"  
required onChange={writeData} value={formData.last_name}/>
```

- **Keresztnév mező:**

```
<input type="text" id="first_name" className="grow" placeholder="Keresztnév"  
required onChange={writeData} value={formData.first_name}/>
```

- **Email mező:**

```
<input type="email" id="email" className="grow"  
placeholder="Email" required onChange={writeData} value={formData.email}/>
```

- **Jelszó mező:**

```
<input type="password" id="password" className="grow" placeholder="Jelszó"  
required onChange={writeData} value={formData.password}/>
```

- **Jelszó megerősítése mező:**

```
<input type="password" id="passwordAgain" className="grow"  
placeholder="Jelszó újra" required onChange={writeData}  
value={formData.passwordAgain}/>
```

2. Gombok

- **Regisztráció gomb:**

```
<button type="submit" className="btn btn-primary text-white">  
    Regisztráció  
</button>
```



- Bejelentkezés link:

```
<Link to="/login2" className="link link-primary">  
  Jelentkezzen be  
</Link>
```

Összegzés

A Register2 React komponens hatékonyan kezeli a felhasználók regisztrációját, miközben szorosan együttműködik a Laravel backenddel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendszer többi részével.

The screenshot shows a registration form titled "Regisztráció". It contains five input fields: "Vezetéknév" (First Name), "Keresztnév" (Last Name), "Email", "Jelszó" (Password), and "Jelszó újra" (Password Confirmation). Below these fields is a large teal-colored "Regisztráció" (Registration) button. Underneath the button is a horizontal line with the word "VAGY" (OR) in the center. To the right of "VAGY" is the question "Van fiókja?" (Does he/she have an account?) and a blue underlined link "Jelentkezzen be" (Log in).

15. ábra Register Panel



React Controller Dokumentáció – CartCheckout.jsx

A [CartCheckout](#) React komponens a rendelési folyamat részeként a számlázási cím kezelésére szolgál. Ez a komponens lehetővé teszi a felhasználók számára, hogy új címet adjanak meg, vagy válasszanak egy meglévő címet a rendeléshez. A komponens API-hívásokkal kommunikál a Laravel backenddel az adatok lekérdezése és mentése érdekében.

Funkciók

1. Új cím hozzáadása

- Leírás:**
A felhasználó megadhat egy új címet, amelyet a komponens elküld a backendnek egy POST kérésben.
- Backend végpont:**
POST /address
- Küldött adatok:**
 - city: Város.
 - zip: Irányítószám.
 - email: Email cím.
 - street_id: Közterület jellege.
 - street: Utca neve.
 - house_number: Házszám.
 - user_id: Felhasználó azonosítója.

2. Meglévő cím kiválasztása

- Leírás:**
A felhasználó kiválaszthat egy már meglévő címet a rendeléshez. A cím azonosítója kerül elküldésre a backendnek.
- Backend végpont:**
POST /order
- Küldött adatok:**
 - address_id: A kiválasztott cím azonosítója.



3. Címek lekérdezése

- **Leírás:**
A komponens betölti a felhasználóhoz tartozó címeket a backendből.
- **Backend végpont:**
`GET /address/{user_id}`
- **Kapott adatok:**
 - addresses: A felhasználóhoz tartozó címek listája.

API-hívások

1. Új cím hozzáadása

- **HTTP-módszer:** POST
- **Végpont:** `/address`
- **Küldött adatok:**

```
{  
    "city": "Budapest",  
    "zip": "1234",  
    "email": "user@example.com",  
    "street_id": 1,  
    "street": "Fő utca",  
    "house_number": "12",  
    "user_id": 1  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Cím sikeresen hozzáadva!",  
    "address": {  
        "id": 1,  
        "city": "Budapest",  
        "zip": "1234",  
        "street": "Fő utca",  
        "house_number": "12"  
    }  
}
```



2. Meglévő cím kiválasztása

- **HTTP-módszer:** POST
- **Végpont:** /order
- **Küldött adatok:**

```
{  
    "address_id": 1  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Rendelés sikeresen létrehozva!"  
}
```

3. Címek lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** /address/{user_id}
- **Kapott adatok:**

```
{  
    "addresses": [  
        {  
            "id": 1,  
            "city": "Budapest",  
            "zip": "1234",  
            "street": "Fő utca",  
            "house_number": "12",  
            "street_type": {  
                "public_area_name": "utca"  
            }  
        }  
    ]  
}
```



Adatok kezelése

1. Űrlap adatok kezelése

A useState hook segítségével kezeli az űrlap adatait:

```
const [formData, setFormData] = useState({  
  city: "",  
  zip: "",  
  email: "",  
  street_id: "",  
  street: "",  
  house_number: "",  
  user_id: user ? user.id : null,  
});
```

Az onChange eseménykezelő frissíti az állapotot az űrlap mezőinek változásakor:

```
const writeData = (e) => {  
  setFormData((prevState) => ({  
    ...prevState,  
    [e.target.id]: e.target.value,  
  }));  
};
```



2. Címek betöltése

A komponens a `useEffect` hook segítségével tölti be a felhasználóhoz tartozó címeket a backendből:

```
useEffect(() => {
  fetch(`$import.meta.env.VITE_BASE_URL}/address/${user.id}`, {
    headers: {
      "content-type": "application/json",
    },
  })
  .then((res) => res.json())
  .then((data) => {
    setAddressess(data.addresses);
    if (data.addresses.length > 0) {
      const minId = Math.min(...data.addresses.map((address) => address.id));
      setMinAddressId(minId);
      setFormData2((prevState) => ({
        ...prevState,
        address_id: minId,
      }));
    }
  })
  .catch((err) => {
    console.error(err);
  });
}, [refresh]);
```



3. Új cím hozzáadása

Az onSubmit eseménykezelő elküldi az új címet a backendnek:

```
const onSubmit = (e) => {
  e.preventDefault();
  if (isNewAddress && !formData.street_id) {
    toast.error("Kérjük, válassza ki a közterület jellegét!");
    return;
  }
  const dataToSend = {
    city: formData.city,
    zip: formData.zip,
    email: formData.email,
    street_id: formData.street_id,
    street: formData.street,
    house_number: formData.house_number,
    user_id: formData.user_id,
  };
  backendMuvelet(dataToSend, "POST", `${import.meta.env.VITE_BASE_URL}/address`, {
    "Content-type": "application/json",
  });
};
```

Felhasználói Élmény

- Valós idejű visszajelzés:**
A react-toastify segítségével a felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
 - Reszponzív dizájn:**
Az űrlap mobil- és asztali eszközökön egyaránt jól használható.
 - Egyszerű címkezelés:**
A felhasználó könnyen válthat új cím megadása és meglévő cím kiválasztása között.
-



Komponens Felépítése

1. Új cím űrlap

Az űrlap mezői:

- **Város mező:**

```
<input type="text" id="city" placeholder="Város"  
required onChange={writeData} value={formData.city}/>
```

- **Irányítószám mező:**

```
<input type="number" id="zip" placeholder="Írányítószám"  
required onChange={writeData} value={formData.zip}/>
```

- **Közterület jellege mező:**

```
<select id="street_id" onChange={writeData} value={formData.street_id}>  
    <option value="">Válasszon közterület jellegét</option>  
    {areas.map((area) => (  
        <option key={area.id} value={area.id}>  
            {area.public_area_name}  
        </option>  
    ))}  
</select>
```



2. Meglévő cím kiválasztása

A címek listája:

```
<select id="address_id" onChange={ writeData2 } value={ formData2.address_id }>
  <option value="">Válasszon egy címet</option>
  { addressess.map((address) => (
    <option key={ address.id } value={ address.id }>
      { address.zip } { address.city }, { address.street } { address.street_type.public_area
        _name } { address.house_number }
    </option>
  )))
</select>
```

Összegzés

A CartCheckout React komponens hatékonyan kezeli a számlázási címek kezelését, miközben szorosan együttműködik a Laravel backenddel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendelési folyamatba.

The screenshot shows a user interface element for managing shipping addresses. At the top, there is a label "Számlázási cím" with a small info icon. Below it is a button labeled "Új címet adok meg" with a plus sign icon. A horizontal line follows. Underneath, there is a section labeled "Válassza ki a címet". A dropdown menu is open, showing a single option: "Válasszon egy címet". To the right of the dropdown is a small downward arrow icon.

16. ábra Address Panel



React Controller Dokumentáció – CartView.jsx

A CartView React komponens a kosár tartalmának megjelenítésére és kezelésére szolgál. Ez a komponens lehetővé teszi a felhasználók számára, hogy megtekintsék a kosárban lévő termékeket, módosítsák azok mennyiségét, eltávolítsák azokat, vagy továbblépjenek a rendelési folyamat következő lépéseihez. A komponens több kontextust használ az adatok kezelésére és megjelenítésére.

Funkciók

1. Kosár tartalmának megjelenítése

- **Leírás:** A kosárban lévő termékek listázása, beleértve a termék nevét, árát, mennyiségét és a kiválasztott átvevőpontot.
- **Adatok forrása:** `CartContext` és `ServiceContext`.
- **Megjelenített adatok:**
 - Termék neve.
 - Ár (Ft/nap).
 - Mennyiség.
 - Átvevőpont neve és címe.

2. Termék mennyiségének módosítása

- **Leírás:** A felhasználó növelheti vagy csökkentheti a termék mennyiségét a kosárban.
- **Műveletek:**
 - Hozzáadás: `A addToCart` függvény hívása.
 - Eltávolítás: `A removeFromCart` függvény hívása.

3. Termék eltávolítása a kosárból

- **Leírás:** A felhasználó eltávolíthat egy terméket a kosárból.
- **Művelet:** `A removeFromCart` függvény hívása.

4. Kosár kiürítése

- **Leírás:** Az összes termék eltávolítása a kosárból.
- **Művelet:** `A clearCart` függvény hívása.



5. Tovább a pénztárhoz

- Leírás: A felhasználó tovább léphet a rendelési folyamat következő lépéseihez, ahol elfogadhatja az adatvédelmi feltételeket.
- Művelet: Az openPrivacyInfo függvény hívása.

API-hívások

A CartView komponens nem végez közvetlen API-hívásokat, hanem a kontextusokból (CartContext,ServiceContext) származó adatokat használja.

Adatok kezelése

1. Kosár adatok kezelése

- A CartContext biztosítja a kosárban lévő termékek adatait és a műveletekhez szükséges függvényeket:
 - cartItems: A kosárban lévő termékek listája.
 - addToCart: Termék hozzáadása a kosárhoz.
 - removeFromCart: Termék eltávolítása a kosárból.
 - clearCart: A kosár kiürítése.
 - getCartTotal: A kosár teljes értékének kiszámítása.

2. Átvevőpontok kezelése

- A ServiceContext biztosítja az átvevőpontok adatait:
 - lockers: Az elérhető átvevőpontok listája.

3. Adatvédelmi feltételek kezelése

- Az OrderContext biztosítja az adatvédelmi feltételek kezeléséhez szükséges állapotokat és függvényeket:
 - isPrivacyInfo: Az adatvédelmi feltételek modális ablakának állapota.
 - openPrivacyInfo: Az adatvédelmi feltételek modális ablakának megnyitása.



- closePrivacyInfo: Az adatvédelmi feltételek modális ablakának bezárása.
-

Felhasználói Élmény

- **Valós idejű frissítés:**
 - A kosár tartalma azonnal frissül, amikor a felhasználó módosítja a termékek mennyiségét vagy eltávolítja azokat.
 - **Részponzív dizájn:**
 - A komponens mobil- és asztali eszközökön egyaránt jól használható.
 - **Átvevőpont információk:**
 - A kiválasztott átvevőpont neve és címe megjelenik a termékek mellett.
 - **Üres kosár üzenet:**
 - Ha a kosár üres, a felhasználó egy figyelmeztető üzentet lát.
-

Komponens Felépítése

1. Kosár tartalma

- A kosárban lévő termékek listázása:

```
{cartItems.map((item) => (
  <div className="flex flex-col md:flex-row justify-between items-center rounded-box bg-base-100 m-5" key={item.id}>
    <div className="flex gap-4">
      <img
        src={`${import.meta.env.VITE_LARAVEL_IMAGE_URL}${item.file_path}`}
        alt="termék kép" className="rounded-md h-24 w-24 object-cover" />
      <div className="flex flex-col">
        <h1 className="font-bold text-2xl text-primary">{item.name}</h1>
        <p className="text-info font-medium">{item.price_per_day} Ft/nap</p>
        <p className="font-bold text-secondary">Kiválasztott átvevőpont: {" "}</p>
        {
          lockers.find(locker => locker.id === Number(item.lockerId))
          ? `${lockers.find(locker => locker.id ===
            Number(item.lockerId))?.locker_name} - ${lockers.find(locker => locker.id ===
            Number(item.lockerId))?.address}`
          : "Ismeretlen átvevőpont"
        }
    </div>
  </div>
)}
```



2. Mennyiség módosítása

- Gombok a termék mennyiségek növelésére és csökkentésére:

```
<button className="px-4 py-2 rounded hover:bg-base-200 focus:outline-none" onClick={() => { addToCart(item, item.lockerId) }}>  
  </button>  
  <p className="mr-2 px-4 py-2 text-secondary font-bold">{item.quantity}</p>  
  <button className="mr-2 px-4 py-2 rounded hover:bg-base-200 focus:outline-none" onClick={() => { removeFromCart(item, item.lockerId) }}>  
    </button>
```

3. Kosár kiürítése

- Gomb az összes termék eltávolítására:

```
<button className="px-4 py-2 bg-info text-white text-xs font-bold uppercase rounded" onClick={() => { clearCart() }}>  
  Teljes kosár kiürítése  
</button>
```

4. Tovább a pénztárhoz

- Gomb az adatvédelmi feltételek elfogadásához és a rendelési folyamat folytatásához:

```
<button id="next-to-checkout" onClick={() => openPrivacyInfo()}  
  className="m-2 px-4 py-2 bg-success text-white text-xs font-bold uppercase rounded" >  
  Tovább a pénztárhoz  
</button>
```

5. Adatvédelmi feltételek modális ablak

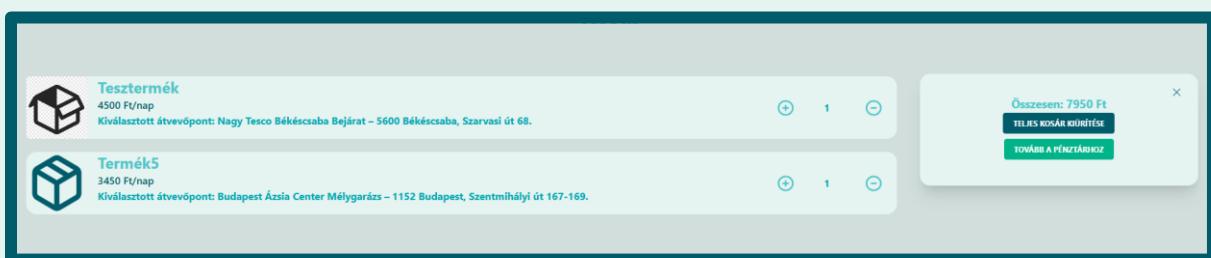
- Az adatvédelmi feltételek elfogadására szolgáló modális ablak:

```
{  
  isPrivacyInfo && (<TermofUseInfo closeFunction={() => closePrivacyInfo()} />)  
}
```



Összegzés

A CartView React komponens hatékonyan kezeli a kosár tartalmának megjelenítését és kezelését. A kontextusok használatával biztosítja az adatok központi kezelését, miközben a felhasználói élmény egyszerű és intuitív. A komponens reszponzív kialakítása és valós idejű frissítése biztosítja, hogy a felhasználók könnyedén kezelhessék a kosarukat a rendelési folyamat során.



17. ábra Cart View Dekstop



18. ábra Cart View Mobile



React Controller Dokumentáció – OrderCheckout.jsx

Az [OrderCheckout](#) React komponens a rendelési folyamat utolsó lépését kezeli, amely magában foglalja a számlázási cím megadását, a fizetési mód kiválasztását, és a rendelés véglegesítését. A komponens szorosan együttműködik a [CartCheckout](#) és [PaymentMethod](#) alkotóelemekkel, valamint az [OrderContext](#)-tel az adatok kezeléséhez és a rendelés leadásához.

Funkciók

1. Számlázási cím kezelése

- **Leírás:**
A [CartCheckout](#) alkotóelem segítségével a felhasználó megadhatja vagy kiválaszthatja a számlázási címet.
- **Alkotóelem:**
[CartCheckout](#)

2. Fizetési mód kiválasztása

- **Leírás:**
A [PaymentMethod](#) alkotóelem segítségével a felhasználó kiválaszthatja a rendeléshez tartozó fizetési módot.
- **Alkotóelem:**
[PaymentMethod](#)

3. Rendelés véglegesítése

- **Leírás:**
A felhasználó a "Rendelés véglegesítése" gombra kattintva elküldi a rendelést a backendnek. A rendelés adatai tartalmazzák a számlázási címet és a kiválasztott fizetési módot.
- **Backend végpont:**
POST /order



API-hívások

1. Rendelés leadása

- **HTTP-módszer:** POST
- **Végpont:** /order
- **Küldött adatok:**
 - Számlázási cím adatai:

```
{  
    "city": "Budapest",  
    "zip": "1234",  
    "street": "Fő utca",  
    "house_number": "12",  
    "street_type": "utca"  
}
```

- Fizetési mód:

```
{  
    "payment_method": "bankkártya"  
}
```

- Sikeres válasz:

```
{  
    "message": "Rendelés sikeresen leadva!",  
    "order_id": 123  
}
```

- Hibás válasz:

```
{  
    "error": "Hiba történt a rendelés leadása során."  
}
```



Context használata

1. OrderContext

Az OrderContext biztosítja a rendelés leadásához szükséges függvényeket és adatokat:

- submitOrder: Rendelés leadása új számlázási címmel.
- submitOrderisAddress: Rendelés leadása meglévő számlázási címmel.
- formDataAddress: A számlázási cím adatai.

Példa használat:

```
const { submitOrder, formDataAddress, submitOrderisAddress } = useContext(OrderContext);
```

Adatok kezelése

1. Fizetési mód kezelése

A PaymentMethod alkomponens a handlePaymentChange függvény segítségével frissíti a kiválasztott fizetési módot:

```
const handlePaymentChange = (paymentType) => {
  setPaymentMethod(paymentType);
};
```

2. Rendelés leadása

A finalSubmit függvény kezeli a rendelés leadását:

- Ha a felhasználó új számlázási címet adott meg, a submitOrder függvény kerül meghívásra.
- Ha a felhasználó meglévő címet választott, a submitOrderisAddress függvény kerül meghívásra.



```
const finalSubmit = async (e) => {
  e.preventDefault();
  let success = false;
  if (formDataAddress.length === 0) {
    success = await submitOrder();
  } else {
    success = await submitOrderisAddress();
  }
  if (success) {
    navigate('/userorder');
  } else {
    console.error('Rendelés mentése nem sikerült');
  }
};
```

Felhasználói Élmény

- Átlátható folyamat:**
A komponens logikusan vezeti végig a felhasználót a rendelési folyamat utolsó lépésein.
- Interaktív funkciók:**
A felhasználó könnyen megadhatja a számlázási címet, kiválaszthatja a fizetési módot, és véglegesítheti a rendelést.
- Valós idejű visszajelzés:**
A rendelés leadása után a felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletről.
- Egyszerű navigáció:**
Sikeres rendelés után a felhasználó automatikusan átirányításra kerül a rendelései oldalra (/userorder).



Komponens Felépítése

1. Számlázási cím

A CartCheckout alkomponens felelős a számlázási cím kezeléséért:

```
<div className="w-full pl-28">
  <CartCheckout />
</div>
```

2. Fizetési mód

A PaymentMethod alkomponens felelős a fizetési mód kiválasztásáért:

```
<div className="w-full pl-28">
  <PaymentMethod />
</div>
```

3. Rendelés véglegesítése

A rendelés leadásához tartozó gomb:

```
<div className="mx-auto mt-10 mb-10 flex justify-center">
  <button
    type="submit"
    className="btn btn-primary text-white text-lg font-semibold rounded-xl">
    Rendelés véglegesítése
  </button>
</div>
```

Összegzés

Az OrderCheckout React komponens hatékonyan kezeli a rendelési folyamat utolsó lépését, miközben szorosan együttműködik a CartCheckout és PaymentMethod alkomponensekkel, valamint az OrderContext-tel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendelési folyamatba.



React Controller Dokumentáció – PaymentMethod.jsx

A [PaymentMethod](#) React komponens a rendelési folyamat részeként a fizetési mód kiválasztására szolgál. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel a fizetési módok lekérdezéséhez, és lehetővé teszi a felhasználó számára, hogy kiválassza a rendeléshez tartozó fizetési módot.

Funkciók

1. Fizetési módok lekérdezése

- **Leírás:**
A komponens betölti a backend által biztosított fizetési módokat egy GET kérés segítségével.
- **Backend végpont:**
`GET /payment`
- **Kapott adatok:**
 - [paymentMethods](#): A fizetési módok listája.

2. Fizetési mód kiválasztása

- **Leírás:**
A felhasználó kiválaszthatja a rendeléshez tartozó fizetési módot egy legördülő menüből. A kiválasztott fizetési módot a komponens továbbítja a [OrderContext](#)-nek.
 - **Függvény:**
[handleChange](#)
-



API-hívások

1. Fizetési módok lekérdezése

- **HTTP-módszer:** [GET](#)
- **Végpont:** [/payment](#)
- **Kapott adatok:**

```
{  
  "paymentMethods": [  
    {  
      "id": 1,  
      "card_type": "Bankkártya"  
    },  
    {  
      "id": 2,  
      "card_type": "Utánvét"  
    }  
  ]  
}
```

- **Hibás válasz:**

```
{  
  "error": "Hiba történt a fizetési módok lekérésekor."  
}
```

Context használata

1. [OrderContext](#)

A [OrderContext](#) biztosítja a fizetési mód kezeléséhez szükséges adatokat és függvényeket:

- [formDataPayment](#): A kiválasztott fizetési mód adatai.
- [setFormDataPayment](#): A fizetési mód adatok frissítésére szolgáló függvény.



Példa használat:

```
const { formDataPayment, setFormDataPayment } = useContext(OrderContext);
```

Adatok kezelése

1. Fizetési módok betöltése

A `useEffect` hook segítségével a komponens betölti a fizetési módokat a backendből:

```
useEffect(() => {
  fetch(`$import.meta.env.VITE_BASE_URL}/payment`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
    },
  })
  .then((response) => response.json())
  .then((data) => {
    setPayments(data.paymentMethods);
  })
  .catch((error) => console.error('Hiba történt a fizetési módok lekérésekor:', error));
}, []);
```

2. Fizetési mód kiválasztása

A `handleChange` függvény frissíti a kiválasztott fizetési módot:

```
const handleChange = (e) => {
  const { value } = e.target;
  setFormDataPayment({ ...formDataPayment, card_type: value });
  onPaymentChange(value);
};
```



Felhasználi Élmény

- **Átlátható megjelenítés:**
A fizetési módok egy legördülő menüben jelennek meg, amely egyszerű és könnyen használható.
- **Valós idejű visszajelzés:**
A kiválasztott fizetési mód azonnal frissül a context-ben, és továbbítható a rendelési folyamat következő lépéseihez.
- **Rezonansív dizájn:**
A komponens mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Fizetési módok megjelenítése

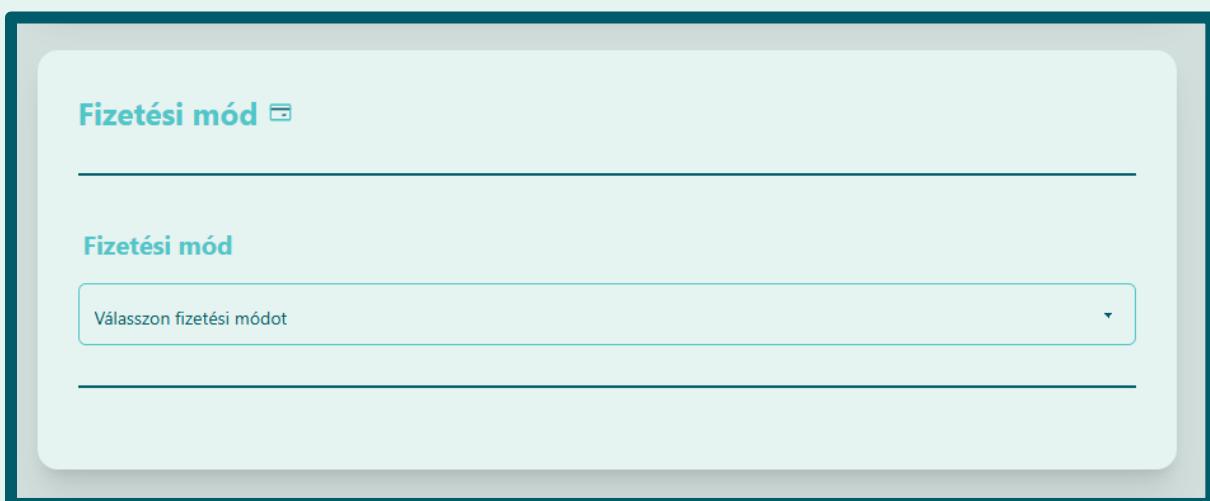
A fizetési módok egy legördülő menüben jelennek meg:

```
<select  
    id="id"  
    className="select select-bordered w-full p-3 mt-2 border border-gray-300 rounded-md focus:outline-none focus:ring-2 focus:ring-blue-500"  
    onChange={handleChange}  
    value={formDataPayment.card_type}>  
    <option value="">Válasszon fizetési módot</option>  
    {payments.map((payment) => (  
        <option key={payment.id} value={payment.id}>  
            {payment.card_type}  
        </option>  
    ))}  
</select>
```

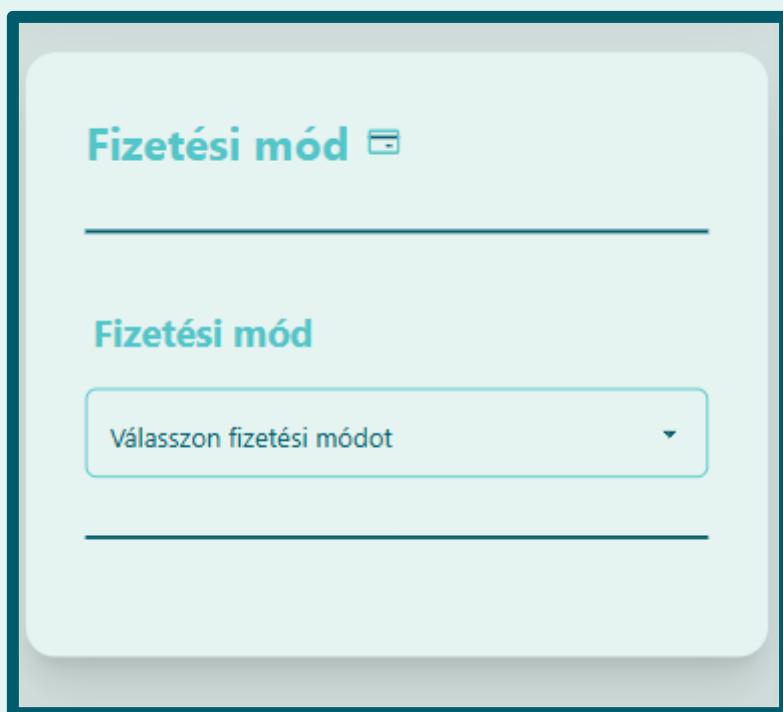


Összegzés

A PaymentMethod React komponens hatékonyan kezeli a fizetési módok lekérdezését és kiválasztását, miközben szorosan együttműködik a Laravel backenddel és az OrderContext-tel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendelési folyamatba.



19. ábra Payment Panel Desktop



20. ábra Payment Panel Mobile



React Controller Dokumentáció – TermofUseInfo.jsx

A TermofUseInfo React komponens a felhasználási feltételek elfogadására szolgál a rendelési folyamat során. Ez a komponens egy modális ablakban jeleníti meg a felhasználási feltételeket, és lehetővé teszi a felhasználó számára, hogy elfogadja azokat, mielőtt továbblépne a pénztárhoz.

Funkciók

1. Felhasználási feltételek megjelenítése

- **Leírás:**

A komponens a TermOfUse alkomponens segítségével jeleníti meg a felhasználási feltételeket egy modális ablakban.

2. Felhasználási feltételek elfogadása

- **Leírás:**

A felhasználó egy jelölőnégyzet segítségével elfogadhatja a felhasználási feltételeket. Az elfogadás után a rendszer továbbnavigálja a felhasználót a pénztár oldalra.

- **Függvény:**

openOrder

3. Modális ablak bezárása

- **Leírás:**

A felhasználó bezárhatja a modális ablakot anélkül, hogy elfogadná a feltételeket.

- **Függvény:**

closeFunction



Adatok kezelése

1. Állapotkezelés

A komponens két állapotot kezel a useState hook segítségével:

- isChecked: A jelölőnégyzet állapota (elfogadta-e a felhasználási feltételeket).
- hasAttemptedSubmit: Annak nyomon követésére, hogy a felhasználó megpróbálta-e már elküldeni az űrlapot.

Példa:

```
const [isChecked, setIsChecked] = useState(false);
const [hasAttemptedSubmit, setHasAttemptedSubmit] = useState(false);
```

2. Jelölőnégyzet kezelése

A handleCheckboxChange függvény frissíti a isChecked állapotot a jelölőnégyzet változásakor:

```
const handleCheckboxChange = (e) => {
  setIsChecked(e.target.checked);
};
```

3. Felhasználási feltételek elfogadása

Az openOrder függvény ellenőrzi, hogy a felhasználó elfogadta-e a feltételeket:

- Ha elfogadta, sikeres visszajelzést kap, és a rendszer továbbnavigálja a pénztár oldalra.
- Ha nem fogadta el, hibaüzenetet kap.



Példa:

```
const openOrder = () => {
  setHasAttemptedSubmit(true);
  if (isChecked) {
    toast.success('A felhasználási feltételeket elfogadtad!');
    closeFunction();
    navigate('/checkout');
  } else {
    toast.error('Kérjük, hogy fogadd el a felhasználási feltételeket!');
  }
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A felhasználási feltételek egy modális ablakban jelennek meg, amely könnyen olvasható és kezelhető.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap, ha nem fogadta el a feltételeket, vagy ha sikeresen elfogadta azokat.
- Egyszerű navigáció:**
Az elfogadás után a felhasználó automatikusan a pénztár oldalra kerül átirányításra.

Komponens Felépítése

1. Felhasználási feltételek megjelenítése

A TermOfUse alkomponens jeleníti meg a felhasználási feltételek tartalmát:

```
<TermOfUse />
```



2. Jelölőnégyzet

A jelölőnégyzet lehetővé teszi a felhasználó számára a feltételek elfogadását:

```
<label className="cursor-pointer">
  <input
    type="checkbox"
    className="checkbox checkbox-primary"
    checked={isChecked}
    onChange={handleCheckboxChange}
  />
  <span className={`ml-2 ${hasAttemptedSubmit && !isChecked ? "text-warning" : ""}`}>
    Elfogadom a felhasználási feltételeket.
  </span>
</label>
```

3. Gombok

- **Bezárás gomb:**

```
<button className="btn text-white bg-info" onClick={closeFunction}>
  Bezárás
</button>
```

- **Elfogadás gomb:**

```
<button id="accepted-term" className="btn text-white bg-primary" onClick={openOrder}>
  Tovább a pénztárhoz
</button>
```

Összegzés

A TermofUseInfo React komponens hatékonyan kezeli a felhasználási feltételek elfogadását a rendelési folyamat során. Az állapotkezelés és a felhasználói visszajelzések jól strukturáltak, a komponens pedig könnyen integrálható a rendelési folyamatba. A modális ablak reszponzív kialakítása biztosítja a felhasználói élményt mobil- és asztali eszközökön egyaránt.



React Controller Dokumentáció – UserOrder.jsx

A [UserOrder](#) React komponens a felhasználó rendeléseinek megjelenítésére szolgál. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel a rendelési adatok lekérdezéséhez, és lehetővé teszi a felhasználó számára, hogy megttekintse a rendelései részleteit.

Funkciók

1. Rendelések lekérdezése

- Leírás:**
A komponens betölti a felhasználó rendeléseit a backendből egy GET kérés segítségével.
- Backend végpont:**
[GET /order](#)
- Kapott adatok:**
 - Rendelések listája, beleértve a rendelési tételeket, számlázási címet, és a rendelés összegét.

2. Kosár kiürítése

- Leírás:**
A rendelési adatok betöltése után a komponens automatikusan kiüríti a kosarat a [CartContext](#) segítségével.
- Függvény:**
[clearCart](#)

3. Bezárás és navigáció

- Leírás:**
A felhasználó a "Bezárás" gombra kattintva visszatérhet a főoldalra.
- Függvény:**
[handleClose](#)



API-hívások

1. Rendelések lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** /order
- **Küldött adatok (fejlécben):**

```
{  
    "Content-Type": "application/json",  
    "Authorization": "Bearer <usertoken>",  
    "UserId": 1  
}
```

Kapott adatok:

```
[{  
    "id": 123,  
    "user": {  
        "first_name": "John",  
        "last_name": "Doe",  
        "email": "john.doe@example.com"  
    },  
    "address": {  
        "zip": "1234",  
        "city": "Budapest",  
        "street": "Fő utca",  
        "house_number": "12",  
        "streettype": {  
            "public_area_name": "utca"  
        }  
    },  
    "total": 15000,  
    "order_item": [  
        {  
            "product_id": 1,  
            "item_price": 5000,  
            "quantity": 3,  
            "line_total": 15000,  
            "locker": {  
                "locker_name": "Csomagautomata 1",  
                "address": "Budapest, Fő utca 12."  
            }  
        }]  
}]
```



Context használata

1. CartContext

A CartContext biztosítja a kosár kiürítéséhez szükséges függvényt:

- clearCart: A kosár teljes tartalmának törlése.

Példa használat:

```
const { clearCart } = useContext(CartContext);
```

Adatok kezelése

1. Rendelések betöltése

A useEffect hook segítségével a komponens betölti a rendeléseket a backendből:

```
useEffect(() => {
  fetch(`${import.meta.env.VITE_BASE_URL}/order`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${sessionStorage.getItem('usertoken')}`,
      'UserId': user.id
    }
  })
  .then(response => response.json())
  .then(data => {
    setOrders(data);
    clearCart();
  })
  .catch(error) => {
    console.error('Error:', error);
  });
}, []);
```



2. Bezárás és navigáció

A `handleClose` függvény a főoldalra navigál:

```
const handleClose = () => {
  navigate('/');
};
```

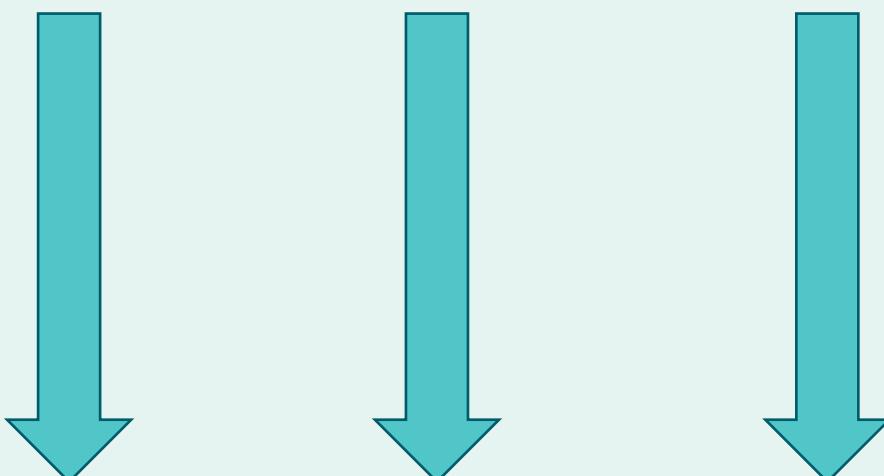
Felhasználói Élmény

- Átlátható megjelenítés:**
A rendelési adatok jól strukturáltan jelennek meg, beleértve a rendelési tételeket, számlázási címet, és a rendelés összegét.
 - Valós idejű visszajelzés:**
A rendelési adatok betöltése után a kosár automatikusan kiürül.
 - Egyszerű navigáció:**
A felhasználó könnyen visszatérhet a főoldalra a "Bezárás" gomb segítségével.
-

Komponens Felépítése

1. Rendelési adatok megjelenítése

A rendelési adatok kártyákban jelennek meg:





```
{orders.map((order, index) => (
  <div key={index} className="pt-4 space-y-3">
    <p className="text-info text-xl font-bold">Azonosító: #{order.id}</p>
    <h2 className="card-title text-xl font-bold text-primary">Megrendelés Adatai</h2>
    <div>
      <p className="font-semibold text-info">Megrendelő:</p>
      <p className="text-sm text-secondary">{order.user.first_name} {order.user.last_name}</p>
      <p className="text-sm text-secondary">{order.user.email}</p>
    </div>
    <div>
      <p className="font-semibold text-info">Számlázási cím:</p>
      <p className="text-sm text-secondary">{order.address.zip} {order.address.city}, {order.address.street} {order.address.house_number}. {order.address.streettype.public_area_name}</p>
    </div>
    <div className="text-lg font-semibold text-secondary">
      <p className="font-semibold text-info">Rendelés összesen:</p>
      Összeg: {order.total} Ft
    </div>
  </div>
))}
```

2. Bezárás gomb

A "Bezárás" gomb a főoldalra navigál:

```
<button className="btn btn-primary text-white" onClick={handleClose}>
  Bezárás
</button>
```

Összegzés

A UserOrder React komponens hatékonyan kezeli a felhasználó rendeléseinek megjelenítését, miközben szorosan együttműködik a Laravel backenddel és a CartContext-tel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a rendelési folyamatba.



React Controller Dokumentáció – CategoriesCard.jsx

A [CategoriesCard](#) React komponens egy kategóriát jelenít meg kártya formájában, és lehetővé teszi a kategória módosítását vagy törlését. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Kategória módosítása

- Leírás:**
A felhasználó a "Módosítás" gombra kattintva navigálhat a kategória módosítására szolgáló oldalra (`/newcategory`), ahol az aktuális kategória adatai előre kitöltésre kerülnek.
- Navigáció:**
A [useNavigate](#) hook segítségével történik.

2. Kategória törlése

- Leírás:**
A felhasználó a "Törlés" gombra kattintva törölheti a kategóriát. A komponens egy `DELETE` API-hívást küld a backendnek, és frissíti a kategóriák listáját.
- Backend végpont:**
`DELETE /category/delete`
- Adatok:**
 - `categoryId`: A törlendő kategória azonosítója.

API-hívások

1. Kategória törlése

- HTTP-módszer:** `DELETE`
- Végpont:** `/category/delete`



- Küldött adatok (fejlécben):

```
{  
  "Content-Type": "application/json",  
  "Authorization": "Bearer <usertoken>",  
  "categoryId": 1  
}
```

- Sikeres válasz:

```
{  
  "message": "Kategória sikeresen törölve!"  
}
```

- Hibás válasz:

```
{  
  "error": "Nem sikerült a kategória törlése."  
}
```

Context használata

1. CrudContext

A CrudContext biztosítja a backendMuvelet függvényt, amely az API-hívások lebonyolítására szolgál:

- backendMuvelet: Az API-hívások általános kezelőfüggvénye.

2. InitialContext

Az InitialContext biztosítja az update függvényt, amely a kategóriák listájának frissítésére szolgál:

- update: A kategóriák listájának frissítése.



Adatok kezelése

1. Kategória módosítása

A modosit függvény a useNavigate hook segítségével navigál a kategória módosítására szolgáló oldalra:

```
const modosit = () => {
  navigate("/newcategory", { state: { category } });
};
```

2. Kategória törlése

A torles függvény a backendMuvelet függvényt használja az API-hívás lebonyolítására:

```
const torles = (category) => {
  const method = "DELETE";
  const url = `${import.meta.env.VITE_BASE_URL}/category/delete`;
  const header = {
    "Content-Type": "application/json",
    "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
    "categoryId": category.id,
  };
  const successMessage = "Kategória sikeresen törölve!";
  const errorMessage = "Nem sikerült a kategória törlése.";
  backendMuvelet(null, method, url, header, successMessage, errorMessage)
    .then(() => {
      update();
    })
    .catch((error) => {
      console.error("Hiba történt a törlés során:", error);
    });
};
```

Felhasználói Élmény

- **Adminisztrátori jogosultság ellenőrzése:**

A komponens csak akkor jeleníti meg a "Módosítás" és "Törlés" gombokat, ha a felhasználó adminisztrátori jogosultsággal rendelkezik (isadmin >= 70).



- **Valós idejű frissítés:**

A törlés után a kategóriák listája automatikusan frissül az `update` függvény segítségével.

- **Reszponzív dizájn:**

A kártya mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Kategória neve

A kategória neve a kártya címeként jelenik meg:

```
<h2 className="card-title justify-center text-primary">{category.name}</h2>
```

2. Műveleti gombok

- **Módosítás gomb:**

```
<button className='btn btn-primary text-white' onClick={() => modosit(category)}>  
    Módosítás  
</button>
```

- **Törlés gomb:**

```
<button className='btn btn-info text-white' onClick={() => torles(category)}>  
    Törlés  
</button>
```

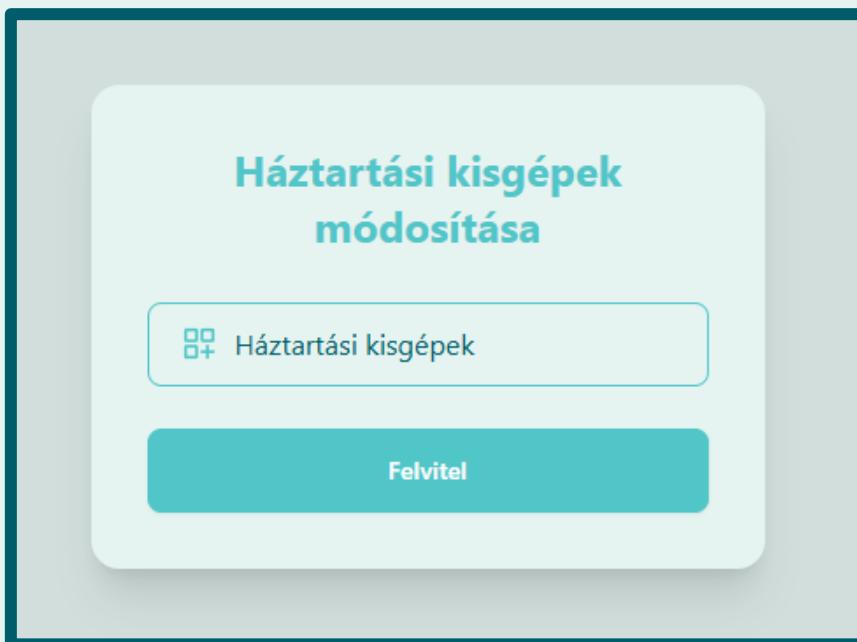


Összegzés

A CategoriesCard React komponens hatékonyan kezeli a kategóriák megjelenítését, módosítását és törlését, miközben szorosan együttműködik a Laravel backenddel és a contextekkel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a kategóriakezelési folyamatba.



21. ábra Category Card Panel



22. ábra Category Edit Panel



React Controller Dokumentáció – CategoriesList.jsx

A [CategoriesList](#) React komponens a kategóriák listájának megjelenítésére szolgál. Ez a komponens a [InitialContext](#) segítségével szerzi be a kategóriák adatait, és minden kategóriához egy-egy [CategoriesCard](#) komponenst renderel.

Funkciók

1. Kategóriák listázása

- **Leírás:**
A komponens a [InitialContext](#)-ből szerzi be a kategóriák adatait ([categories](#)), majd minden kategóriához egy [CategoriesCard](#) komponenst renderel.
- **Adatok forrása:**
A kategóriák adatai a backend API-ból származnak, amelyeket a [InitialContext](#) tölt be és tárol.

API-hívások

A [CategoriesList](#) komponens közvetlenül nem végez API-hívásokat, de a [InitialContext](#)-en keresztül hozzáfér a backend által biztosított adatokhoz. A kategóriák listáját a következő API végpont biztosítja:

Kategóriák lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** </categories>



- Sikeres válasz:

```
{  
  "categories": [  
    {  
      "id": 1,  
      "name": "Elektronika"  
    },  
    {  
      "id": 2,  
      "name": "Bútorok"  
    }  
  ]  
}
```

Context használata

1. InitialContext

A InitialContext biztosítja a kategóriák adatait (categories), amelyeket a komponens a useContext hook segítségével ér el:

```
const { categories } = useContext(InitialContext);
```

A categories tömb tartalmazza az összes kategória adatait, amelyeket a backend API-ból tölt be a context.

Adatok kezelése

- Kategóriák megjelenítése:

A komponens a categories tömb minden eleméhez egy CategoriesCard komponensem renderel:

```
categories.map((category) => (<CategoriesCard key={category.id} category={category} />))
```



Felhasználói Élmény

- **Átlátható megjelenítés:**
A kategóriák listája jól strukturált, és minden kategória külön kártyán jelenik meg.
- **Interaktív funkciók:**
A [CategoriesCard](#) komponenseken keresztül a felhasználó módosíthatja vagy törlheti a kategóriákat, ha adminisztrátori jogosultsággal rendelkezik.
- **Rezonansív dizájn:**
A lista mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Context használata

A [useContext](#) hook segítségével a komponens hozzáfér a [InitialContext](#) által biztosított adatokhoz:

```
const { categories } = useContext(InitialContext);
```

2. Kártyák renderelése

A komponens a [categories](#) tömb minden eleméhez egy [CategoriesCard](#) komponenst renderel:

```
<div className="flex flex-row flex-wrap items-center justify-center">
  {
    categories.map((category) => (<CategoriesCard key={category.id} category={category} />))
  }
</div>
```

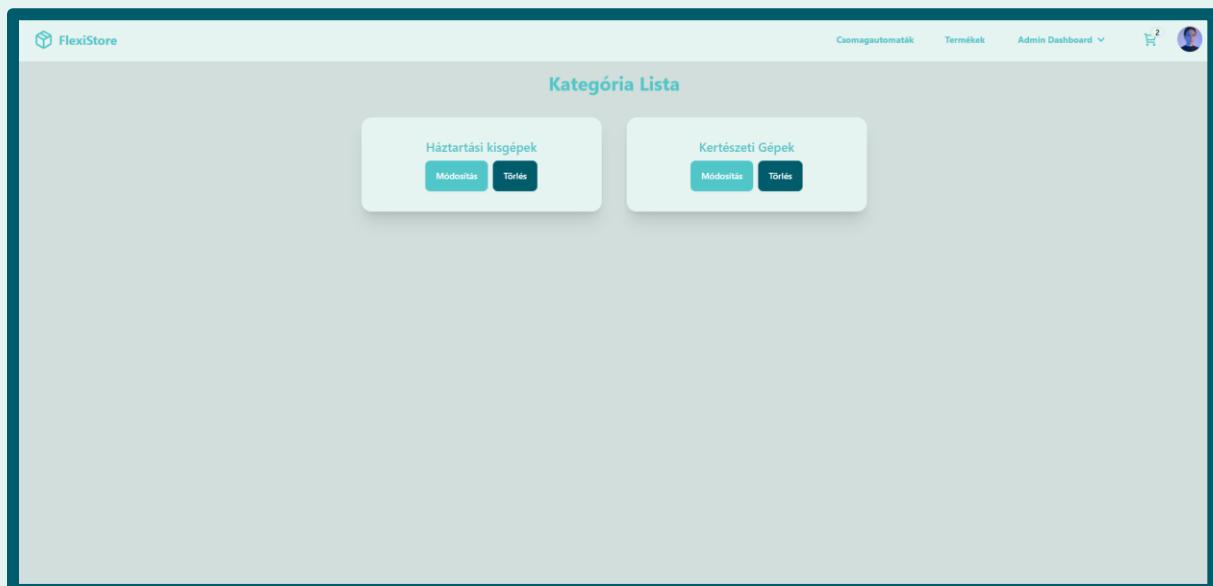


Felhasználói Élmény

- Átlátható megjelenítés:**
A kategóriák listája jól strukturált, és minden kategória külön kártyán jelenik meg.
- Interaktív funkciók:**
A CategoriesCard komponenseken keresztül a felhasználó módosíthatja vagy törlheti a kategóriákat, ha jogosultsággal rendelkezik.
- Rezonansív kialakítás:**
A lista mobil- és asztali eszközökön egyaránt jól használható.

Összegzés

A CategoriesList React komponens hatékonyan kezeli a kategóriák listázását, miközben a InitialContext segítségével biztosítja az adatok elérését. A komponens egyszerű és intuitív felhasználói élményt nyújt, és szorosan együttműködik a CategoriesCard komponenssel a kategóriák kezelésében.



23. ábra Categories List



React Controller Dokumentáció – NewCategory.jsx

A NewCategory React komponens egy új kategória létrehozására vagy meglévő kategória módosítására szolgál. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Új kategória létrehozása

- Leírás:**
A felhasználó megadhatja az új kategória nevét, amelyet a komponens elküld a backendnek egy POST kérésben.
- Backend végpont:**
POST /category
- Küldött adatok:**
 - name: A kategória neve.

2. Meglévő kategória módosítása

- Leírás:**
A felhasználó módosíthatja egy meglévő kategória nevét. A komponens a kategória adatait előre kitölți, és egy PATCH kérésben küldi el a módosított adatokat a backendnek.
- Backend végpont:**
PATCH /category
- Küldött adatok:**
 - id: A kategória azonosítója.
 - name: A kategória új neve.



API-hívások

1. Új kategória létrehozása

- **HTTP-módszer:** POST
- **Végpont:** /category
- **Küldött adatok:**

```
{  
    "name": "Új Kategória"  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Új kategória sikeresen létrehozva!"  
}
```

- **Hibás válasz:**

```
{  
    "error": "Nem sikerült a kategória létrehozása."  
}
```

2. Meglévő kategória módosítása

- **HTTP-módszer:** PATCH
- **Végpont:** /category
- **Küldött adatok:**



- Sikeres válasz:

```
{  
    "id": 1,  
    "name": "Módosított Kategória"  
}
```

```
{  
    "message": "Kategória sikeresen módosítva!"  
}
```

- Hibás válasz:

```
{  
    "error": "Nem sikerült a kategória módosítása."  
}
```

Context használata

1. CrudContext

A CrudContext biztosítja a backendMuvelet függvényt, amely az API-hívások lebonyolítására szolgál:

- backendMuvelet: Az API-hívások általános kezelőfüggvénye.

2. InitialContext

Az InitialContext biztosítja az update függvényt, amely a kategóriák listájának frissítésére szolgál:

- update: A kategóriák listájának frissítése.
-



Adatok kezelése

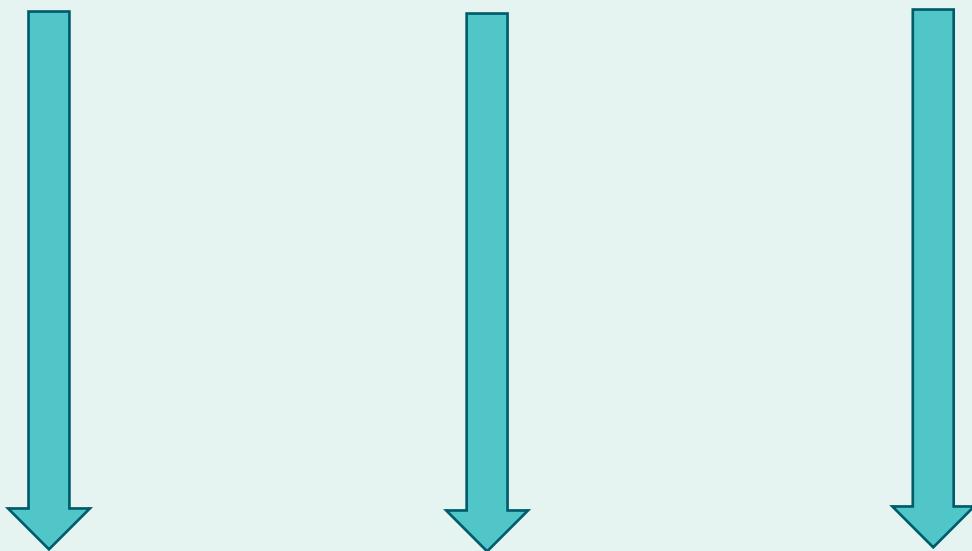
1. Kategória adatok inicializálása

A komponens a `useLocation` hook segítségével ellenőrzi, hogy új kategóriát kell létrehozni, vagy meglévőt kell módosítani:

```
let formObj = {
  id: '',
  name: '',
};
if (state !== null) {
  const { category } = state;
  formObj = {
    id: category.id,
    name: category.name,
  };
  method = "PATCH";
  cim = `${category.name} módosítása`;
}
```

2. Kategória létrehozása vagy módosítása

Az `onSubmit` függvény kezeli az űrlap beküldését:





```
const onSubmit = async (e) => {
  e.preventDefault();
  if (!formData.name) {
    toast.error("A kategória név mező kitöltése kötelező!");
    return;
  }
  const isEdit = !!formData.id;
  const method = isEdit ? "PATCH" : "POST";
  const url = `${import.meta.env.VITE_BASE_URL}/category`;
  const successMessage = isEdit
    ? `${formData.name} sikeresen módosítva!`
    : "Új kategória sikeresen létrehozva!";
  const errorMessage = isEdit
    ? `${formData.name} módosítása sikertelen!`
    : "Nem sikerült a kategória létrehozása!";
  try {
    await backendMuvelet(
      formData,
      method,
      url,
      { "Content-Type": "application/json" },
      successMessage,
      errorMessage
    );
    update();
    navigate("/categories");
  } catch (error) {
    console.error("Hiba történt a kategória felvitelében:", error);
  }
};
```

3. Űrlap mezők kezelése

A writeData függvény frissíti az űrlap adatait:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```



Felhasználói Élmény

- **Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek meglévő kategória módosítása esetén.
- **Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- **Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül a kategóriák listájára.

Komponens Felépítése

1. Űrlap

Az űrlap mezője:

- **Kategória név mező:**

```
<input className="grow placeholder-info" type="text" id="name" placeholder="Kategória neve"  
required onChange={writeData} value={formData.name}/>
```

2. Gombok

- **Felvitel gomb:**

```
<button type="submit" className="btn btn-primary text-white">  
    Felvitel  
</button>
```



Összegzés

A NewCategory React komponens hatékonyan kezeli az új kategóriák létrehozását és a meglévő kategóriák módosítását, miközben szorosan együttműködik a Laravel backenddel és a context-ekkel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a kategóriakezelési folyamatba.



24. ábra New Category Panel



React Controller Dokumentáció – AdminDashboard.jsx

Az [AdminDashboard](#) React komponens az adminisztrátorok számára készült, hogy megtekinthessék és kezelhessék a regisztrált felhasználók adatait. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, és adminisztrátori jogosultságot igényel a hozzáféréshez.

Funkciók

1. Felhasználók listázása

- Leírás:**
A komponens betölti a regisztrált felhasználók adatait a backendből, és megjeleníti azokat táblázatos formában (asztali nézet) vagy kártyákban (mobil nézet).
- Backend végpont:**
`GET /users`
- Kapott adatok:**
 - Felhasználók listája, beleértve az azonosítót, nevet, email címét, jogosultságot, és a létrehozás dátumát.

2. Keresés név vagy email alapján

- Leírás:**
A felhasználók listája szűrhető név vagy email cím alapján.
- Függvény:**
`setSearchQuery`

3. Lapozás

- Leírás:**
A felhasználók listája több oldalra osztható, és a felhasználó navigálhat az oldalak között.



- **Függvények:**
 - handlePrevPage: Előző oldalra lépés.
 - handleNextPage: Következő oldalra lépés.

4. Jogosultság ellenőrzése

- **Leírás:**

A komponens ellenőrzi, hogy a felhasználó adminisztrátori jogosultsággal rendelkezik-e. Ha nem, a hozzáférés megtagadásra kerül, és a felhasználó visszairányításra kerül a főoldalra.
- **Függvény:**

useEffect

5. Bezárás és navigáció

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva visszatérhet a főoldalra.
- **Függvény:**

handleClose

API-hívások

1. Felhasználók lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** users
- **Küldött adatok (fejlécben):**

```
{  
  "Content-Type": "application/json",  
  "Authorization": "Bearer <usertoken>"  
}
```



- Kapott adatok:

```
[  
  {  
    "id": 1,  
    "first_name": "John",  
    "last_name": "Doe",  
    "email": "john.doe@example.com",  
    "isadmin": 70,  
    "created_at": "2025-04-01T12:00:00Z"  
  }  
]
```

Context használata

1. AdminContext

Az AdminContext biztosítja a felhasználók listájának kezeléséhez szükséges adatokat és függvényeket:

- users: A regisztrált felhasználók listája.
- setUsers: A felhasználók listájának frissítése.

Példa használat:

```
const { users, setUsers } = useContext(AdminContext);
```



Adatok kezelése

1. Felhasználók betöltése

A useEffect hook segítségével a komponens betölti a felhasználók adatait a backendből:

```
useEffect(() => {
  if (!user || user.isadmin < 70) {
    toast.error('Hozzáférés megtagadva!');
    navigate('/');
    return;
  }
  fetch(`${import.meta.env.VITE_BASE_URL}/users`, {
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${token}`,
    }
  })
  .then(res => res.json())
  .then(adat => {
    setUsers(adat);
  })
  .catch(err => alert(err));
}, []);
```

2. Keresés név vagy email alapján

A useEffect hook segítségével a komponens szűri a felhasználók listáját a keresési kifejezés alapján:

```
useEffect(() => {
  const filtered = users.filter(u =>
    u.first_name?.toLowerCase().includes(searchQuery.toLowerCase()) ||
    u.last_name?.toLowerCase().includes(searchQuery.toLowerCase()) ||
    u.email?.toLowerCase().includes(searchQuery.toLowerCase())
  );
  setFilteredUsers(filtered);
  setCurrentPage(1);
}, [searchQuery, users]);
```



3. Lapozás

A `paginatedUsers` változó tartalmazza az aktuális oldalon megjelenítendő felhasználókat:

```
const paginatedUsers = filteredUsers.slice(  
  (currentPage - 1) * itemsPerPage,  
  currentPage * itemsPerPage  
)
```

Felhasználi Élmény

- Átlátható megjelenítés:**
A felhasználók adatai táblázatos formában jelennek meg asztali nézetben, és kártyákban mobil nézetben.
- Interaktív keresés:**
A felhasználók listája valós időben szűrhető név vagy email cím alapján.
- Lapozás:**
A felhasználók listája több oldalra osztható, és a felhasználó könnyen navigálhat az oldalak között.
- Jogosultság ellenőrzése:**
A komponens biztosítja, hogy csak adminisztrátori jogosultsággal rendelkező felhasználók férhessenek hozzá az adatokhoz.

Komponens Felépítése

1. Keresőmező

A keresőmező lehetővé teszi a felhasználók szűrését:

```
<input  
  type="text"  
  value={searchQuery}  
  onChange={(e) => setSearchQuery(e.target.value)}  
  placeholder="Keresés név vagy email alapján"  
  className="input input-bordered input-primary w-full max-w-md placeholder-info"  
 />
```



2. Táblázat (asztali nézet)

A felhasználók adatai táblázatos formában jelennek meg:

```
<table className="hidden lg:table w-full bg-base-100 text-info font-bold">
  <thead className="bg-primary text-white">
    <tr>
      <th>Id</th>
      <th>Vezetéknév</th>
      <th>Keresztnév</th>
      <th>Email cím</th>
      <th>Jogosultság</th>
      <th>Létrehozva</th>
      <th colSpan={2} className="text-center">Műveletek</th>
    </tr>
  </thead>
  <tbody>
    {paginatedUsers.map((user) =>
      <AdminDashboardCard key={user.id} user={user} />
    ))}
  </tbody>
</table>
```

3. Kártyák (mobil nézet)

A felhasználók adatai kártyákban jelennek meg:

```
<div className="lg:hidden flex flex-col gap-4">
  {paginatedUsers.map((user) =>
    <UserCard key={user.id} user={user} />
  )}
</div>
```



4. Lapozás

A lapozás gombjai:

```
<div className="join flex justify-center mt-10 mb-24">
  <button
    className="join-item btn btn-secondary"
    onClick={handlePrevPage}
    disabled={currentPage === 1}>
    <<
  </button>
  <button className="join-item btn btn-primary text-white">
    Oldal {currentPage} / {totalPages}
  </button>
  <button
    className="join-item btn btn-secondary"
    onClick={handleNextPage}
    disabled={currentPage === totalPages} >
    >>
  </button>
</div>
```

Összegzés

Az [AdminDashboard](#) React komponens hatékonyan kezeli a regisztrált felhasználók adatainak megjelenítését és kezelését, miközben szorosan együttműködik a Laravel backenddel és a context-ekkel. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható az adminisztrációs folyamatba.



React Controller Dokumentáció – AdminDashboardCard.jsx

Az [AdminDashboardCard](#) React komponens egy felhasználó adatait jeleníti meg egy táblázat sorában az adminisztrációs felületen. Ez a komponens lehetővé teszi a felhasználó adatainak szerkesztését vagy törlését, és szorosan együttműködik az [AdminContext](#)-tel a műveletek végrehajtásához.

Funkciók

1. Felhasználói adatok megjelenítése

- Leírás:**
A komponens megjeleníti a felhasználó azonosítóját, nevét, email címét, jogosultságát, és a regisztráció dátumát egy táblázat sorában.

2. Felhasználói adatok szerkesztése

- Leírás:**
A "Szerkesztés" gombra kattintva a felhasználó adatai szerkeszthetők egy modális ablakban.
- Függvények:**
 - [openModal](#): A modális ablak megnyitása.
 - [closeModal](#): A modális ablak bezárása.

3. Felhasználó törlése

- Leírás:**
A "Törlés" gombra kattintva a felhasználó törölhető a rendszerből. A törlés egy DELETE API-hívással történik.
 - Függvény:**
[deleteUser](#)
-



API-hívások

1. Felhasználó törlése

- **HTTP-módszer:** `DELETE`
- **Végpont:** `/users/{id}`
- **Küldött adatok (fejlécben):**

```
{  
  "Content-Type": "application/json",  
  "Authorization": "Bearer <usertoken>"  
}
```

- **Sikeres válasz:**

```
{  
  "error": "Nem sikerült a felhasználó törlése."  
}
```

- **Hibás válasz:**

```
{  
  "message": "Felhasználó sikeresen törölve!"  
}
```

Context használata

1. AdminContext

Az AdminContext biztosítja a felhasználók kezeléséhez szükséges függvényeket:

- deleteUser: A felhasználó törlésére szolgáló függvény.

Példa használat:

```
const { deleteUser } = useContext(AdminContext);
```



Adatok kezelése

1. Modális ablak kezelése

A useState hook segítségével a komponens kezeli a modális ablak nyitott vagy zárt állapotát:

```
const [isModalOpen, setIsModalOpen] = useState(false);
const openModal = () => {
    setIsModalOpen(true);
};
const closeModal = () => {
    setIsModalOpen(false);
};
```

2. Felhasználó törlése

A deleteUser függvény meghívásával a komponens törli a felhasználót:

```
<button onClick={() => deleteUser(user.id)} className='btn bg-info text-white'>
    Törlés
</button>
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A felhasználók adatai jól strukturáltan jelennek meg egy táblázat sorában.
- Interaktív funkciók:**
A felhasználó adatai szerkeszthetők egy modális ablakban, és törölhetők egy gombnyomással.
- Valós idejű visszajelzés:**
A törlés vagy szerkesztés után a felhasználó azonnali visszajelzést kap a művelet sikereségéről.



Komponens Felépítése

1. Táblázat sor

A felhasználó adatai egy táblázat sorában jelennek meg:

```
<tr>
  <th>{user.id}</th>
  <td>{user.last_name}</td>
  <td>{user.first_name}</td>
  <td>{user.email}</td>
  <td>{user.role.warrant_name}</td>
  <td>{new Date(user.created_at).toISOString().split('T')[0]} {new Date(user.created_a
t).toISOString().split('T')[1].slice(0, 8)}</td>
</tr>
```

2. Szerkesztés gomb

A "Szerkesztés" gomb megnyitja a modális ablakot:

```
<button onClick={() => openModal()} className='btn bg-primary text-white'>
  Szerkesztés
</button>
```

3. Törlés gomb

A "Törlés" gomb meghívja a deleteUser függvényt:

```
<button onClick={() => deleteUser(user.id)} className='btn bg-info text-white'>
  Törlés
</button>
```

4. Modális ablak

A modális ablak a RegistrationDataEdit komponenst jeleníti meg:

```
{isModalOpen && (
  <RegistrationDataEdit user={user} closeFunction={() => closeModal()} />
)}
```



Összegzés

Az [AdminDashboardCard](#) React komponens hatékonyan kezeli a felhasználók adatainak megjelenítését, szerkesztését és törlését az adminisztrációs felületen. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható az adminisztrációs folyamatba.



React Controller Dokumentáció – AdminOrders.jsx

Az [AdminOrders](#) React komponens az adminisztrátorok számára készült, hogy megtekinthessék, szűrhessék és kezelhessék a rendszerben lévő megrendeléseket. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, és adminisztrátori jogosultságot igényel a hozzáféréshez.

Funkciók

1. Megrendelések listázása

- Leírás:**
A komponens betölti a megrendelések adatait a backendből, és megjeleníti azokat kártyák formájában.
- Backend végpont:**
`GET /orderslist`
- Kapott adatok:**
 - Megrendelések listája, beleértve a felhasználói adatokat, csomagautomata adatokat, és a rendelési tételeket.

2. Megrendelések szűrése

- Leírás:**
A megrendelések szűrhetők név, email cím, csomagautomata neve vagy címe alapján.
- Függvény:**
`handleSearch`

3. Lapozás

- Leírás:**
A megrendelések listája több oldalra osztható, és a felhasználó navigálhat az oldalak között.



- **Függvények:**
 - handlePrevPage: Előző oldalra lépés.
 - handleNextPage: Következő oldalra lépés.

4. Megrendelés törlése

- **Leírás:**

A "Törlés" gombra kattintva a megrendelés törölhető a rendszerből. A törlés egy DELETE API-hívással történik.
- **Függvény:**
handleDelete

5. Bezárás és navigáció

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva visszatérhet a főoldalra.
- **Függvény:**
handleClose

API-hívások

1. Megrendelések lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** /orderslist
- **Küldött adatok (fejlécben):**

```
{  
    "Content-Type": "application/json",  
    "Authorization": "Bearer <usertoken>"  
}
```



- Kapott adatok:

```
{  
    "order_id": 1  
}
```

2. Megrendelés törlése

- HTTP-módszer: **DELETE**
- Végpont: **/orders**
- Küldött adatok (törzsben):

```
[  
    {  
        "id": 1,  
        "user": {  
            "first_name": "John",  
            "last_name": "Doe",  
            "email": "john.doe@example.com"  
        },  
        "order_item": [  
            {  
                "locker": {  
                    "locker_name": "Csomagautomata 1",  
                    "address": "Budapest, Fő utca 12."  
                }  
            }  
        ]  
    }  
]
```

- Sikeres válasz:

```
{  
    "message": "Megrendelés sikeresen törölve!"  
}
```



- Hibás válasz:

```
{  
  "error": "Nem sikerült a megrendelés törlése."  
}
```

Adatok kezelése

1. Megrendelések betöltése

A `useEffect` hook segítségével a komponens betölti a megrendeléseket a backendből:

```
useEffect(() => {  
  fetch(`${import.meta.env.VITE_BASE_URL}/orderslist`, {  
    method: 'GET',  
    headers: {  
      'Content-Type': 'application/json',  
      'Authorization': `Bearer ${sessionStorage.getItem('usertoken')}`,  
    },  
  })  
  .then(async (response) => {  
    if (!response.ok) {  
      throw new Error('Hiba történt a megrendelések lekérésekor');  
    }  
    const data = await response.json();  
    setOrders(data);  
    setFilteredOrders(data);  
  })  
  .catch((error) => {  
    console.error('Fetch hiba:', error.message);  
  });  
}, []);
```



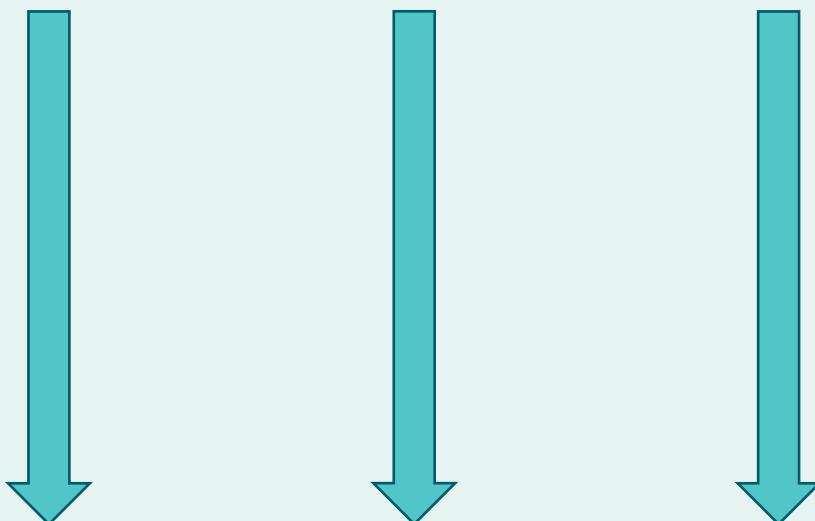
2. Megrendelések szűrése

A `handleSearch` függvény szűri a megrendeléseket a keresési kifejezések alapján:

```
const handleSearch = (event) => {
  const { name, value } = event.target;
  if (name === 'searchName') setSearchName(value);
  if (name === 'searchEmail') setSearchEmail(value);
  if (name === 'searchLockerName') setSearchLockerName(value);
  if (name === 'searchLockerAddress') setSearchLockerAddress(value);
  if (!value) {
    setFilteredOrders(orders);
  } else {
    const filtered = orders.filter((order) => {
      const fullName = `${order.user.first_name} ${order.user.last_name}`.toLowerCase();
      const email = order.user.email.toLowerCase();
      const lockerName = order.order_item[0].locker?.locker_name.toLowerCase() || '';
      const lockerAddress = order.order_item[0].locker?.address.toLowerCase() || '';
      return (
        fullName.includes(searchName.toLowerCase()) &&
        email.includes(searchEmail.toLowerCase()) &&
        lockerName.includes(searchLockerName.toLowerCase()) &&
        lockerAddress.includes(searchLockerAddress.toLowerCase())
      );
    });
    setFilteredOrders(filtered);
  }
};
```

3. Megrendelés törlése

A `handleDelete` függvény meghívásával a komponens törli a megrendelést:





```
const handleDelete = async (orderId) => {
  try {
    const response = await fetch(` ${import.meta.env.VITE_BASE_URL}/orders`, {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${sessionStorage.getItem('usertoken')}`,
      },
      body: JSON.stringify({ order_id: orderId })
    });
    if (!response.ok) {
      toast.error('Hiba történt a rendelés törlésekor');
      return;
    }
    const data = await response.json();
    toast.success(data.message);
    setOrders(orders.filter((order) => order.id !== orderId));
    setFilteredOrders(filteredOrders.filter((order) => order.id !== orderId));
  } catch (error) {
    console.error('Hiba történt a rendelés törlésében:', error);
    toast.error('Hiba történt a rendelés törlésében');
  }
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**

A megrendelések kártyák formájában jelennek meg, és szűrhetők különböző paraméterek alapján.

- Interaktív funkciók:**

A felhasználó törölheti a megrendeléseket, és navigálhat az oldalak között.

- Valós idejű visszajelzés:**

A törlés vagy szűrés után a felhasználó azonnali visszajelzést kap a művelet sikerességéről.



Komponens Felépítése

1. Szűrők

A szűrők lehetővé teszik a megrendelések szűrését:

```
<input  
  type="text"  
  name="searchName"  
  value={searchName}  
  onChange={handleSearch}  
  placeholder="Keresés név szerint"  
  className="input input-bordered w-full input-primary"/>
```

2. Megrendelések kártyái

A megrendelések kártyák formájában jelennek meg:

```
{currentOrders.map((order) =>  
  <AdminOrdersCard key={order.id} order={order} handleDelete={handleDelete} />  
)}
```

3. Lapozás

A lapozás gombjai:

```
<div className="join flex justify-center mt-4">  
  <button  
    className="join-item btn btn-primary"  
    onClick={handlePrevPage}  
    disabled={currentPage === 1}>  
    «  
  </button>  
  <button className="join-item btn btn-info">  
    Oldal {currentPage} / {totalPages}  
  </button>  
  <button  
    className="join-item btn btn-primary"  
    onClick={handleNextPage}  
    disabled={currentPage === totalPages}>  
    »  
  </button>  
</div>
```



Összegzés

Az AdminOrders React komponens hatékonyan kezeli a megrendelések megjelenítését, szűrését és törlését az adminisztrációs felületen. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható az adminisztrációs folyamatba.

The screenshot shows the 'Megrendelések' (Orders) page in the Flexistore Admin Dashboard. On the left, there's a search sidebar with fields for name, email, delivery machine name, and delivery machine address. Two order cards are displayed: Order #40 (Super Admin, Szeged, 6000 Ft) and Order #39 (Super Admin, Janoshalma, 4500 Ft). Each card shows detailed delivery information and a 'Töröl' (Delete) button at the bottom.

25. ábra All Order List Admin Desktop

The screenshot shows the same 'Megrendelések' (Orders) page on a mobile device. It displays Order #40 (Super Admin, Szeged, 6000 Ft). The card includes a 'Tételek megtekintése' (View Details) button and a 'Töröl' (Delete) button.

26. ábra All Order List Admin Mobile



React Controller Dokumentáció – AdminOrdersCard.jsx

Az AdminOrdersCard React komponens egy megrendelés adatait jeleníti meg kártya formájában az adminisztrációs felületen. Ez a komponens lehetővé teszi a megrendelés részleteinek megtekintését, valamint a megrendelés törlését.

Funkciók

1. Megrendelési adatok megjelenítése

- **Leírás:**

A komponens megjeleníti a megrendelés azonosítóját, a megrendelő adatait, a számlázási címet, a rendelés összegét, valamint a rendelési tételeket.

2. Megrendelés törlése

- **Leírás:**

A "Törlés" gombra kattintva a megrendelés törölhető a rendszerből. A törléshez a handleDelete függvényt hívja meg, amelyet a szülő komponens biztosít.

Adatok kezelése

1. Megrendelési adatok megjelenítése

A komponens a order prop segítségével kapja meg a megrendelés adatait, és azokat kártya formájában jeleníti meg:

- **Megrendelő adatai:**

```
<p className="text secondary">{order.user.first_name} {order.user.last_name}</p>
<p className="text-sm text-secondary">{order.user.email}</p>
```



- Számlázási cím:

```
<p className="text-secondary">{order.address.zip} {order.address.city}, {order.address.street} {order.address.house_number}. {order.address.streettype.public_area_name}</p>
```

- Rendelési tételek:

```
{order.order_item.map((item, idx) => (
  <div key={idx} className="mb-2 border-b border-dashed pb-2 last:border-none last:pb-0">
    <p><span className="font-medium">Csomagautómata:</span> {item.locker ? item.locker.locker_name : 'Nincs hozzá rendelt locker'}</p>
    <p><span className="font-medium">Csomagautómata címe:</span> {item.locker ? item.locker.address : 'Nincs hozzá rendelt locker'}</p>
    <p><span className="font-medium">Termék ID:</span> {item.product_id}</p>
    <p><span className="font-medium">Ár:</span> {item.item_price} Ft</p>
    <p><span className="font-medium">Darab:</span> {item.quantity}</p>
    <p><span className="font-medium">Összeg:</span> {item.line_total} Ft</p>
  </div>
))}
```

2. Megrendelés törlése

A "Törlés" gomb meghívja a handleDelete függvényt, amely a szülő komponensben van definiálva:

```
<button className="btn btn-error text-white" onClick={() => handleDelete(order.id)}>
  Törlés
</button>
```

Felhasználói Élmény

- Átlátható megjelenítés:

A megrendelések adatai jól strukturáltan jelennek meg kártya formájában, beleértve a rendelési tételeket is.

- Interaktív funkciók:

A felhasználó egyetlen gombnyomással törölheti a megrendelést.



- **Valós idejű visszajelzés:**

A törlés után a szülő komponens frissíti a megrendelések listáját, így a felhasználó azonnal látja a változást.

Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a megrendelés azonosítója jelenik meg:

```
<p className="text-info text-xl font-bold">Azonosító: #{order.id}</p>
```

2. Megrendelő adatai

```
<p className="text-secondary">{order.user.first_name} {order.user.last_name}</p>
<p className="text-sm text-secondary">{order.user.email}</p>
```

3. Számlázási cím

A számlázási cím adatai:

```
<p className="text-
secondary">{order.address.zip} {order.address.city}, {order.address.street} {order.address.house_-
number}. {order.address.streettype.public_area_name}</p>
```



4. Rendelési tételek

A rendelési tételek listája:

```
{order.order_item.map((item, idx) => (
  <div key={idx} className="mb-2 border-b border-dashed pb-2 last:border-none last:pb-0">
    <p><span className="font-medium">Csomagautomata:</span> {item.locker ? item.locker.locker_name : 'Nincs hozzárendelt locker' }</p>
    <p><span className="font-medium">Csomagautomata címe:</span> {item.locker ? item.locker.address : 'Nincs hozzárendelt locker' }</p>
    <p><span className="font-medium">Termék ID:</span> {item.product_id}</p>
    <p><span className="font-medium">Ár:</span> {item.item_price} Ft</p>
    <p><span className="font-medium">Darab:</span> {item.quantity}</p>
    <p><span className="font-medium">Összeg:</span> {item.line_total} Ft</p>
  </div>
))}
```

5. Törlés gomb

A törlés gomb:

```
<button className="btn btn-error text-white"
  onClick={() => handleDelete(order.id)}>
  Törlés
</button>
```

Összegzés

Az AdminOrdersCard React komponens hatékonyan kezeli a megrendelések megjelenítését és törlését az adminisztrációs felületen. Az adatok jól strukturáltan jelennek meg, és a törlés funkció egyszerűen használható. A komponens könnyen integrálható az adminisztrációs folyamatba, és biztosítja a RESTful architektúra elveinek betartását.



React Controller Dokumentáció – RegistrationDataEdit.jsx

A [RegistrationDataEdit](#) React komponens egy felhasználó adatainak szerkesztésére szolgál az adminisztrációs felületen. Ez a komponens lehetővé teszi a felhasználó nevénk, email címének és jogosultságának módosítását, miközben ellenőrzi az adminisztrátor jogosultsági szintjét.

Funkciók

1. Felhasználói adatok megjelenítése és szerkesztése

- Leírás:**
A komponens előre kitölti a felhasználó adatait egy űrlapon, amely lehetővé teszi azok módosítását.
- Adatok forrása:**
A [user](#) prop tartalmazza a szerkesztendő felhasználó adatait.

2. Jogosultság ellenőrzése

- Leírás:**
A komponens ellenőrzi, hogy az adminisztrátor jogosultsági szintje ([adminpower](#)) elegendő-e a kiválasztott jogosultság módosításához.
- Függvény:**
[onSubmit](#)

3. Felhasználói adatok mentése

- Leírás:**
Az űrlap beküldésekor a komponens elküldi a módosított adatokat a backendnek egy PUT kérésben.

Backend végpont:

PUT /user/edit



4. Bezárás és navigáció

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva bezárhatja a modális ablakot anélkül, hogy mentené a változtatásokat.

- **Függvény:**

[closeFunction](#)

API-hívások

1. Felhasználói adatok módosítása

- **HTTP-módszer:** [PUT](#)
- **Végpont:** [/user/edit](#)
- **Küldött adatok:**

```
{  
  "id": 1,  
  "first_name": "John",  
  "last_name": "Doe",  
  "email": "john.doe@example.com",  
  "role_id": 2  
}
```

- **Sikeres válasz:**

```
{  
  "message": "Felhasználói adatok sikeresen módosítva!"  
}
```

- **Hibás válasz:**

```
{  
  "error": "Nem sikerült a felhasználói adatok módosítása."  
}
```



Context használata

1. AdminContext

Az AdminContext biztosítja a felhasználói adatok módosításához szükséges függvényeket:

- backendMuvelet: Az API-hívások általános kezelőfüggvénye.
- update: A felhasználók listájának frissítésére szolgáló függvény.

Példa használat:

```
const { backendMuvelet, update } = useContext(AdminContext);
```

Adatok kezelése

1. Felhasználói adatok inicializálása

A komponens a user prop segítségével inicializálja az űrlap mezőit:

```
const [formData, setFormData] = useState({  
  id: user.id,  
  first_name: user.first_name,  
  last_name: user.last_name,  
  email: user.email,  
  role_id: user.role_id  
});
```

2. Jogosultságok betöltése

A useEffect hook segítségével a komponens betölti a jogosultságokat a secureStorage-ből:

```
useEffect(() => {  
  const storedRoles = secureStorage.getItem('roles');  
  if (storedRoles) {  
    setRoles(JSON.parse(storedRoles));  
  }  
}, []);
```



3. Adatok módosítása

A `writeData` függvény frissíti az űrlap mezőinek értékeit:

```
const writeData = (e) => {
  setFormData({ ...formData, [e.target.id]: e.target.value });
};
```

4. Adatok mentése

Az `onSubmit` függvény ellenőrzi a jogosultságokat, majd elküldi a módosított adatokat a backendnek:

```
const onSubmit = (e) => {
  e.preventDefault();
  const keres = roles.find((role) => role.id == formData.role_id).power;
  if (adminpower < keres) {
    toast.error('Nincs jogosultságod ehhez a művelethez!');
    return;
  }
  backendMuvelet(formData, "PUT", `${import.meta.env.VITE_BASE_URL}/user/edit`);
  closeFunction();
  update();
  navigate('/admindashboard');
};
```

Felhasználói Élmény

- Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek a meglévő adatokkal.
- Jogosultság ellenőrzése:**
A komponens biztosítja, hogy az adminisztrátor csak a jogosultsági szintjének megfelelő módosításokat végezhesse el.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül az adminisztrációs felületre.



Komponens Felépítése

1. Űrlap mezők

- Vezetéknév mező:

```
<input type="text" id="last_name" className="grow" placeholder="Vezetéknév"  
required onChange={writeData} value={formData.last_name}/>
```

- Keresztnév mező:

```
<input type="text" id="first_name" className="grow" placeholder="Keresztnév"  
required onChange={writeData} value={formData.first_name}/>
```

- Email mező:

```
<input type="email" id="email" className="grow"  
placeholder="Email" required onChange={writeData} value={formData.email}/>
```

- Jogosultság mező:

```
<select  
  className="select select-bordered w-full border-primary"  
  id="role_id"  
  onChange={writeData}  
  value={formData.role_id}>  
>  
  {roles.map((role) => (  
    <option key={role.id} value={role.id}>  
      {role.warrant_name}  
    </option>  
</select>
```



2. Gombok

- **Mentés gomb:**

```
<button className="btn btn-primary text-white">Mentés</button>
```

- **Bezárás gomb:**

```
<button className="btn btn-info text-white" onClick={closeFunction}>  
    Bezárás  
</button>
```

Összegzés

A RegistrationDataEdit React komponens hatékonyan kezeli a felhasználói adatok szerkesztését az adminisztrációs felületen. Az API-hívások és az állapotkezelés jól strukturált, a jogosultságok ellenőrzése pedig biztosítja a rendszer biztonságát. A komponens könnyen integrálható az adminisztrációs folyamatba, és intuitív felhasználói élményt nyújt.



React Controller Dokumentáció – UserCard.jsx

A [UserCard](#) React komponens egy felhasználó adatait jeleníti meg kártya formájában az adminisztrációs felületen. Ez a komponens lehetővé teszi a felhasználó adatainak szerkesztését egy modális ablakban, valamint a felhasználó törlését.

Funkciók

1. Felhasználói adatok megjelenítése

- **Leírás:**

A komponens megjeleníti a felhasználó azonosítóját, nevét, email címét, jogosultságát, és a regisztráció dátumát.

2. Felhasználói adatok szerkesztése

- **Leírás:**

A "Szerkesztés" gombra kattintva a felhasználó adatai szerkeszthetők egy modális ablakban.

- **Függvények:**

- [openModal](#): A modális ablak megnyitása.
- [closeModal](#): A modális ablak bezárása.

3. Felhasználó törlése

- **Leírás:**

A "Törlés" gombra kattintva a felhasználó törölhető a rendszerből. A törléshez a [deleteUser](#) függvényt hívja meg, amelyet a [AdminContext](#) biztosít.



Adatok kezelése

1. Felhasználói adatok megjelenítése

A komponens a user prop segítségével kapja meg a felhasználó adatait, és azokat kártya formájában jeleníti meg:

- Felhasználói adatok:

```
<p className="text-secondary">Vezetéknév: {user.last_name}</p>
<p className="text-secondary">Keresztnév: {user.first_name}</p>
<p className="text-secondary">Email: {user.email}</p>
<p className="text-secondary">Jogosultság: {user.role.warrant_name}</p>
<p className="text-
secondary">Létrehozva: {new Date(user.created_at).toISOString().split('T')[0]} {new Date(user.
created_at).toISOString().split('T')[1].slice(0, 8)}</p>
```

2. Modális ablak kezelése

A useState hook segítségével a komponens kezeli a modális ablak nyitott vagy zárt állapotát:

```
const [isModalOpen, setIsModalOpen] = useState(false);
const openModal = () => {
  setIsModalOpen(true);
};
const closeModal = () => {
  setIsModalOpen(false);
};
```

3. Felhasználó törlése

A "Törlés" gomb meghívja a deleteUser függvényt, amely a AdminContext-ből érhető el:

```
<button onClick={() => deleteUser(user.id)} className="btn bg-info text-white">
  Törlés
</button>
```



4. Felhasználói adatok szerkesztése

A "Szerkesztés" gomb megnyitja a RegistrationDataEdit komponenst egy modális ablakban:

```
{isModalOpen && (  
  <RegistrationDataEdit user={user} closeFunction={() => closeModal()} />  
)}
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A felhasználók adatai jól strukturáltan jelennek meg kártya formájában.
- Interaktív funkciók:**
A felhasználó adatai szerkeszthetők egy modális ablakban, és törölhetők egy gombnyomással.
- Valós idejű visszajelzés:**
A törlés vagy szerkesztés után a szülő komponens frissíti a felhasználók listáját, így a felhasználó azonnal látja a változást.

Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a felhasználó azonosítója jelenik meg:

```
<p className="text-center font-bold text-2xl text-info">Azonosító #{user.id}</p>
```

2. Felhasználói adatok

A felhasználó adatai:

```
<p className="text-secondary">Vezetéknév: {user.last_name}</p>  
<p className="text-secondary">Keresztnév: {user.first_name}</p>  
<p className="text-secondary">Email: {user.email}</p>  
<p className="text-secondary">Jogosultság: {user.role.warrant_name}</p>  
<p className="text-  
secondary">Létrehozva: {new Date(user.created_at).toISOString().split('T')[0]} {new Date(user.  
created_at).toISOString().split('T')[1].slice(0, 8)}</p>
```



3. Szerkesztés gomb

A "Szerkesztés" gomb megnyitja a modális ablakot:

```
<button onClick={() => openModal()} className="btn bg-primary text-white">  
  Szerkesztés  
</button>
```

4. Törlés gomb

A "Törlés" gomb meghívja a deleteUser függvényt:

```
<button onClick={() => deleteUser(user.id)} className="btn bg-info text-white">  
  Törlés  
</button>
```

5. Modális ablak

```
{isModalOpen && (  
  <RegistrationDataEdit user={user} closeFunction={() => closeModal()} />  
)}
```

Összegzés

A UserCard React komponens hatékonyan kezeli a felhasználók adatainak megjelenítését, szerkesztését és törlését az adminisztrációs felületen. Az adatok jól strukturáltan jelennek meg, és a törlés vagy szerkesztés funkció egyszerűen használható. A komponens könnyen integrálható az adminisztrációs folyamatba, és biztosítja a RESTful architektúra elveinek betartását.



React Controller Dokumentáció – UserDashboard.jsx

A UserDashboard React komponens a felhasználók számára készült, hogy megtekinthessék, szűrhessék és kezelhessék saját megrendeléseiket. Ez a komponens API-hívásokkal kommunikál a Laravel backenddel, és biztosítja a megrendelések listázását, szűrését, lapozását és törlését.

Funkciók

1. Megrendelések listázása

- **Leírás:**
A komponens betölti a felhasználó megrendeléseit a backendből, és megjeleníti azokat kártyák formájában.
- **Backend végpont:**
`GET /userorderslist`
- **Kapott adatok:**
 - Megrendelések listája, beleértve a felhasználói adatokat, csomagautomata adatokat, és a rendelési tételeket.

2. Megrendelések szűrése

- **Leírás:**
A megrendelések szűrhetők név, email cím, csomagautomata neve vagy címe alapján.
- **Függvény:**
`handleSearch`

3. Lapozás

- **Leírás:**
A megrendelések listája több oldalra osztható, és a felhasználó navigálhat az oldalak között.



- **Függvények:**
 - handlePrevPage: Előző oldalra lépés.
 - handleNextPage: Következő oldalra lépés.

4. Megrendelés törlése

- **Leírás:**

A "Törlés" gombra kattintva a megrendelés törölhető a rendszerből. A törlés egy DELETE API-hívással történik.
- **Függvény:**

handleDelete

5. Bezárás és navigáció

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva visszatérhet a főoldalra.
- **Függvény:**

handleClose

API-hívások

1. Megrendelések lekérdezése

- **HTTP-módszer:** GET
- **Végpont:** /userorderslist
- **Küldött adatok (fejlécben):**

```
{  
  "Content-Type": "application/json",  
  "Authorization": "Bearer <usertoken>",  
  "userId": 1  
}
```



- Kapott adatok:

```
[  
  {  
    "id": 1,  
    "user": {  
      "first_name": "John",  
      "last_name": "Doe",  
      "email": "john.doe@example.com"  
    },  
    "order_item": [  
      {  
        "locker": {  
          "locker_name": "Csomagautomata 1",  
          "address": "Budapest, Fő utca 12."  
        }  
      }  
    ]  
  }  
]
```

- Küldött adatok (törzsben):

```
{  
  "order_id": 1  
}
```

- Sikeres válasz:

```
{  
  "message": "Megrendelés sikeresen törölve!"  
}
```

- Hibás válasz:

```
{  
  "error": "Nem sikerült a megrendelés törlése."  
}
```



Adatok kezelése

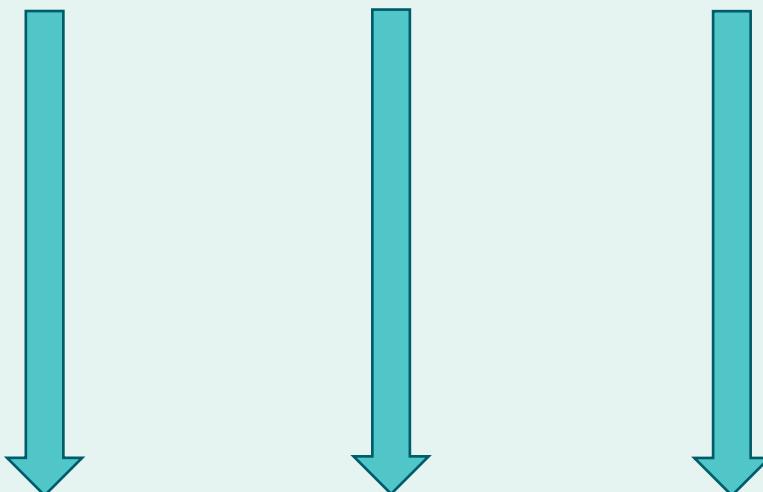
1. Megrendelések betöltése

A useEffect hook segítségével a komponens betölti a megrendeléseket a backendből:

```
useEffect(() => {
  fetch(` ${import.meta.env.VITE_BASE_URL}/userorderslist` , {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${sessionStorage.getItem('usertoken')}` ,
      'userId': user.id,
    },
  })
  .then(async (response) => {
    if (!response.ok) {
      throw new Error('Hiba történt a megrendelések lekérésekor');
    }
    const data = await response.json();
    setOrders(data);
    setFilteredOrders(data);
  })
  .catch((error) => {
    console.error('Fetch hiba:', error.message);
  });
}, []);
```

2. Megrendelések szűrése

A handleSearch függvény szűri a megrendeléseket a keresési kifejezések alapján:

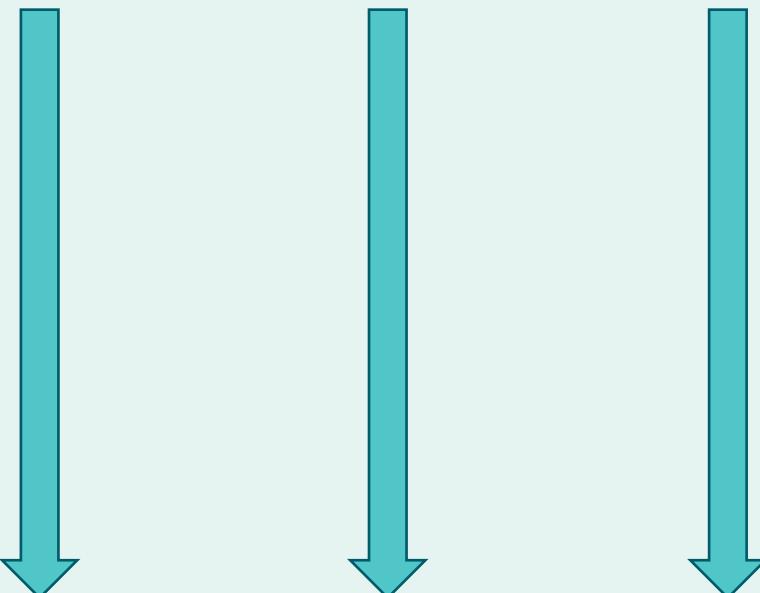




```
const handleSearch = (event) => {
  const { name, value } = event.target;
  if (name === 'searchName') setSearchName(value);
  if (name === 'searchEmail') setSearchEmail(value);
  if (name === 'searchLockerName') setSearchLockerName(value);
  if (name === 'searchLockerAddress') setSearchLockerAddress(value);
  if (!value) {
    setFilteredOrders(orders);
  } else {
    const filtered = orders.filter((order) => {
      const fullName = `${order.user.first_name} ${order.user.last_name}`.toLowerCase();
      const email = order.user.email.toLowerCase();
      const lockerName = order.order_item[0]?.locker?.locker_name.toLowerCase() || '';
      const lockerAddress = order.order_item[0]?.locker?.address.toLowerCase() || '';
      return (
        fullName.includes(searchName.toLowerCase()) &&
        email.includes(searchEmail.toLowerCase()) &&
        lockerName.includes(searchLockerName.toLowerCase()) &&
        lockerAddress.includes(searchLockerAddress.toLowerCase())
      );
    });
    setFilteredOrders(filtered);
  }
};
```

3. Megrendelés törlése

A handleDelete függvény meghívásával a komponens törli a megrendelést:





```
const handleDelete = async (orderId) => {
  try {
    const response = await fetch(`.${import.meta.env.VITE_BASE_URL}/userorders`, {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${sessionStorage.getItem('usertoken')}`,
      },
      body: JSON.stringify({ order_id: orderId })
    });
    if (!response.ok) {
      toast.error('Hiba történt a rendelés törlésekor');
      return;
    }
    const data = await response.json();
    toast.success(data.message);
    setOrders(orders.filter((order) => order.id !== orderId));
    setFilteredOrders(filteredOrders.filter((order) => order.id !== orderId));
  } catch (error) {
    console.error('Hiba történt a rendelés törlésében:', error);
    toast.error('Hiba történt a rendelés törlésében');
  }
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**

A megrendelések kártyák formájában jelennek meg, és szűrhetők különböző paraméterek alapján.

- Interaktív funkciók:**

A felhasználó törölheti a megrendeléseket, és navigálhat az oldalak között.

- Valós idejű visszajelzés:**

A törlés vagy szűrés után a felhasználó azonnali visszajelzést kap a művelet sikereségéről.



Komponens Felépítése

1. Szűrők

A szűrők lehetővé teszik a megrendelések szűrését:

```
<input  
  type="text"  
  name="searchName"  
  value={searchName}  
  onChange={handleSearch}  
  placeholder="Keresés név szerint"  
  className="input input-bordered w-full input-primary placeholder-info"/>
```

2. Megrendelések kártyái

A megrendelések kártyák formájában jelennek meg:

```
{currentOrders.map((order) =>  
  <UserOrdersCard key={order.id} order={order} handleDelete={handleDelete} />  
)}
```

3. Lapozás

A lapozás gombjai:

```
<div className="join flex justify-center mt-4">  
  <button  
    className="join-item btn btnprimary onClick={handlePrevPage} disabled={currentPage === 1}>  
    «  
  </button>  
  <button className="join-item btn btn-info">  
    Oldal {currentPage} / {totalPages}  
  </button>  
  <button  
    className="join-item btn btn-primary"  
    onClick={handleNextPage} disabled={currentPage === totalPages}>  
    »  
  </button>  
</div>
```



React Controller Dokumentáció – UserOrdersCard.jsx

A UserOrdersCard React komponens egy megrendelés adatait jeleníti meg kártya formájában a felhasználói felületen. Ez a komponens lehetővé teszi a megrendelés részleteinek megtekintését, valamint a megrendelés törlését.

Funkciók

1. Megrendelési adatok megjelenítése

- **Leírás:**

A komponens megjeleníti a megrendelés azonosítóját, a megrendelő adatait, a számlázási címét, a rendelés összegét, valamint a rendelési tételeket.

2. Megrendelés törlése

- **Leírás:**

A "Törlés" gombra kattintva a megrendelés törölhető a rendszerből. A törléshez a handleDelete függvényt hívja meg, amelyet a szülő komponens biztosít.

Adatok kezelése

1. Megrendelési adatok megjelenítése

A komponens az order prop segítségével kapja meg a megrendelés adatait, és azokat kártya formájában jeleníti meg:

- **Megrendelő adatai:**

```
<p className="text-secondary">{order.user.first_name} {order.user.last_name}</p>
<p className="text-sm text-secondary">{order.user.email}</p>
```

- **Számlázási cím:**

```
<p className="textsecondary">{order.address.zip} {order.address.city}, {order.address.street}
{order.address.house_number}. {order.address.streettype.public_area_name}</p>
```



- Rendelési tételek:

```
{order.order_item.map((item, idx) => (
  <div key={idx} className="mb-2 border-b border-dashed pb-2 last:border-none last:pb-0">
    <p><span className="font-medium">Csomagautomata:</span> {item.locker ? item.locker.locker_name : 'Nincs hozzárendelt locker'}</p>
    <p><span className="font-medium">Csomagautomata címe:</span> {item.locker ? item.locker.address : 'Nincs hozzárendelt locker'}</p>
    <p><span className="font-medium">Termék ID:</span> {item.product_id}</p>
    <p><span className="font-medium">Ár:</span> {item.item_price} Ft</p>
    <p><span className="font-medium">Darab:</span> {item.quantity}</p>
    <p><span className="font-medium">Összeg:</span> {item.line_total} Ft</p>
  </div>
))}
```

2. Megrendelés törlése

A "Törlés" gomb meghívja a `handleDelete` függvényt, amely a szülő komponensben van definiálva:

```
<button className="btn btn-error text-white" onClick={() => handleDelete(order.id)}>
  Törlés
</button>
```

Felhasználói Élmény

- **Átlátható megjelenítés:**

A megrendelések adatai jól strukturáltan jelennek meg kártya formájában, beleértve a rendelési tételeket is.

- **Interaktív funkciók:**

A felhasználó egyetlen gombnyomással törölheti a megrendelést.

- **Valós idejű visszajelzés:**

A törlés után a szülő komponens frissíti a megrendelések listáját, így a felhasználó azonnal látja a változást.



Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a megrendelés azonosítója jelenik meg:

```
<p className="text-info text-xl font-bold">Azonosító: #{order.id}</p>
```

2. Megrendelő adatai

A megrendelő neve és email címe:

```
<p className="text-secondary">{order.user.first_name} {order.user.last_name}</p>
<p className="text-sm text-secondary">{order.user.email}</p>
```

3. Számlázási cím

A számlázási cím adatai:

```
<p className="text-secondary">{order.address.zip} {order.address.city},
{order.address.street} {order.address.house_number}. {order.address.streettype.public_ar
ea_name}</p>
```

4. Rendelési tételek

A rendelési tételek listája:

```
{order.order_item.map((item, idx) =>
  <div key={idx} className="mb-2 border-b border-dashed pb-2 last:border-none last:pb-0">
    <p><span className="font-
medium">Csomagautomata:</span> {item.locker ? item.locker.locker_name : 'Nincs hozzárendelt lo
cker'}</p>
    <p><span className="font-
medium">Csomagautomata címe:</span> {item.locker ? item.locker.address : 'Nincs hozzárendelt lo
cker'}</p>
    <p><span className="font-medium">Termék ID:</span> {item.product_id}</p>
    <p><span className="font-medium">Ár:</span> {item.item_price} Ft</p>
    <p><span className="font-medium">Darab:</span> {item.quantity}</p>
    <p><span className="font-medium">Összeg:</span> {item.line_total} Ft</p>
  </div>
)}
```



5. Törlés gomb

A törlés gomb:

```
<button className="btn btn-error text-white" onClick={() => handleDelete(order.id)}>  
    Törlés  
</button>
```

Összegzés

A UserOrdersCard React komponens hatékonyan kezeli a megrendelések megjelenítését és törlését a felhasználói felületen. Az adatok jól strukturáltan jelennek meg, és a törlés funkció egyszerűen használható. A komponens könnyen integrálható a felhasználói folyamatokba, és biztosítja a RESTful architektúra elveinek betartását.



React Controller Dokumentáció – LockerInfo.jsx

A LockerInfo React komponens egy modális ablakot jelenít meg, amely egy csomagautomata részletes leírását tartalmazza. Ez a komponens egyszerűen használható a csomagautomaták információinak megtekintésére.

Funkciók

1. Csomagautomata leírásának megjelenítése

- **Leírás:**

A komponens megjeleníti a csomagautomata leírását (locker.description) egy modális ablakban.

2. Modális ablak bezárása

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva bezárhatja a modális ablakot.

- **Függvény:**

closeFunction

Adatok kezelése

1. Csomagautomata adatok

A komponens a locker prop segítségével kapja meg a csomagautomata adatait, és megjeleníti annak leírását:

```
<p className="text-left text-info">{locker.description}</p>
```



2. Modális ablak bezárása

A "Bezárás" gomb meghívja a `closeFunction` függvényt, amelyet a szülő komponens biztosít:

```
<button  
  className="btn btn-ghost btn-circle absolute right-2 top-2 text-info"  
  onClick={closeFunction}>  
  X  
</button>
```

Felhasználói Élmény

- Egyszerű megjelenítés:**
A csomagautomata leírása egy modális ablakban jelenik meg, amely könnyen olvasható és kezelhető.
- Interaktív funkciók:**
A felhasználó egyetlen gombnyomással bezárhatja a modális ablakot.
- Rezonans kialakítás:**
A modális ablak mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Modális fejléc

A modális fejlécében a cím jelenik meg:

```
<h2 className="text-primary font-bold text-center text-lg">Csomagautomata leírás</h2>
```

2. Csomagautomata leírás

A csomagautomata leírása:

```
<p className="text-left text-info">{locker.description}</p>
```



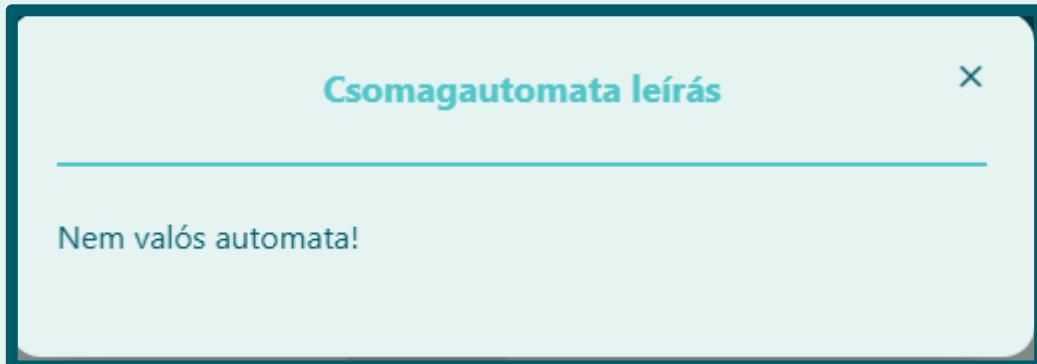
3. Bezárás gomb

A "Bezárás" gomb:

```
<button  
  className="btn btn-ghost btn-circle absolute right-2 top-  
  2 text-info"  
  onClick={closeFunction}>  
  X  
</button>
```

Összegzés

A LockerInfo React komponens egyszerű és hatékony módot biztosít a csomagautomaták leírásának megjelenítésére egy modális ablakban. A komponens könnyen integrálható más felhasználói felületekbe, és intuitív felhasználói élményt nyújt.



27. ábra Locker Info



React Controller Dokumentáció – LockersCard.jsx

A LockersCard React komponens egy csomagautomata adatait jeleníti meg kártya formájában. Ez a komponens lehetővé teszi a csomagautomata részleteinek megtekintését, módosítását, törlését, valamint az elérhető termékek megtekintését.

Funkciók

1. Csomagautomata adatok megjelenítése

- **Leírás:**

A komponens megjeleníti a csomagautomata nevét, címét, és egy alapértelmezett képet.

2. Csomagautomata részleteinek megtekintése

- **Leírás:**

Az "Info" gombra kattintva a csomagautomata részletei egy modális ablakban jelennek meg.

- **Függvények:**

- openInfo: A modális ablak megnyitása.
- closeInfo: A modális ablak bezárása.

3. Csomagautomata módosítása

- **Leírás:**

A "Módosítás" gombra kattintva a felhasználó navigálhat a csomagautomata módosítására szolgáló oldalra.

- **Függvény:**

modosit



4. Csomagautomata törlése

- **Leírás:**
A "Törlés" gombra kattintva a csomagautomata törölhető a rendszerből. A törlés egy DELETE API-hívással történik.
- **Függvény:**
[torles](#)

5. Elérhető termékek megtekintése

- **Leírás:**
Az "Elérhető Termékek" gomb jelenleg statikus, de később navigációra vagy más funkcióra használható.

API-hívások

1. Csomagautomata törlése

- **HTTP-módszer:** [DELETE](#)
- **Végpont:** [/locker/delete](#)
- **Küldött adatok (fejlécben):**

```
{  
    "Content-type": "application/json",  
    "Authorization": "Bearer <usertoken>",  
    "lockerId": 1  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Csomagautomata sikeresen törölve!"  
}
```



- Hibás válasz:

```
{  
  "error": "Nem sikerült a csomagautomata törlése."  
}
```

Adatok kezelése

1. Csomagautomata adatok megjelenítése

A komponens a `locker` prop segítségével kapja meg a csomagautomata adatait, és azokat kártya formájában jeleníti meg:

```
<h2 className="card-title text-primary">{locker.locker_name}</h2>  
<p className="text-info">{locker.address}</p>
```

2. Modális ablak kezelése

A `useState` hook segítségével a komponens kezeli a modális ablak nyitott vagy zárt állapotát:

```
const [isInfo, setInfo] = useState(false);  
const openInfo = () => setInfo(true);  
const closeInfo = () => setInfo(false);
```

3. Csomagautomata módosítása

A `modosit` függvény navigál a csomagautomata módosítására szolgáló oldalra:

```
const modosit = (locker) => {  
  navigate("/newlocker", { state: { locker } });  
};
```



4. Csomagautomata törlése

A torles függvény meghívásával a komponens törli a csomagautomatát:

```
const torles = (locker) => {
  backendMuvelet(
    locker,
    "DELETE",
    `${import.meta.env.VITE_BASE_URL}/locker/delete`,
    {
      "Content-type": "application/json",
      "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
      "lockerId": locker.id
    }
  );
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A csomagautomata adatai jól strukturáltan jelennek meg kártya formájában.
- Interaktív funkciók:**
A felhasználó megtekintheti a csomagautomata részleteit, módosíthatja vagy törölheti azt, ha adminisztrátori jogosultsággal rendelkezik.
- Reszponzív kialakítás:**
A kártya mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a csomagautomata neve és címe jelenik meg:

```
<h2 className="card-title text-primary">{locker.locker_name}</h2>
<p className="text-info">{locker.address}</p>
```



A modális ablak a LockerInfo komponenst jeleníti meg:

```
{isInfo && (  
  <LockerInfo locker={locker} closeFunction={closeInfo}/>  
)}
```

3. Módosítás gomb

A "Módosítás" gomb:

```
<button className="btn btn-primary text-white" onClick={() => modosit(locker)}>  
  Módosítás  
</button>
```

4. Törlés gomb

A "Törlés" gomb:

```
<button className="btn btn-info text-white" onClick={() => torles(locker)}>  
  Törlés  
</button>
```

5. Elérhető Termékek gomb

Az "Elérhető Termékek" gomb:

```
<button className="btn btn-primary text-white">Elérhető Termékek</button>
```

6. Info gomb

Az "Info" gomb:

```
<button className="btn btn-secondary text-white" onClick={openInfo}>  
  Info  
</button>
```



Összegzés

A LockersCard React komponens hatékonyan kezeli a csomagautomaták megjelenítését, részleteinek megtekintését, módosítását és törlését. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens könnyen integrálható a csomagautomata-kezelési folyamatokba.



28. ábra Locker Card



React Controller Dokumentáció – LockersList.jsx

A LockersList React komponens a csomagautomaták listájának megjelenítésére szolgál. Ez a komponens lehetővé teszi a csomagautomaták szűrését, lapozását, és kártyák formájában történő megjelenítését.

Funkciók

1. Csomagautomaták listázása

- **Leírás:**

A komponens betölti a csomagautomaták adatait a ServiceContext-ból, és megjeleníti azokat kártyák formájában.

2. Csomagautomaták szűrése

- **Leírás:**

A felhasználó szűrheti a csomagautomatákat név vagy cím alapján.

- **Függvény:**

handleSearch

3. Lapozás

- **Leírás:**

A csomagautomaták listája több oldalra osztható, és a felhasználó navigálhat az oldalak között.

- **Függvények:**

- handlePrevPage: Előző oldalra lépés.
- handleNextPage: Következő oldalra lépés.

4. Szűrőpanel (mobil nézet)

- **Leírás:**

A mobil nézetben egy szűrőpanel jelenik meg, amely beúszik a képernyő jobb oldaláról.



- **Függvények:**
 - setIsFilterOpen: A szűrőpanel nyitott vagy zárt állapotának kezelése.
-

Adatok kezelése

1. Csomagautomaták betöltése

A ServiceContext biztosítja a csomagautomaták adatait:

```
const { lockers } = useContext(ServiceContext);
```

2. Szűrés

A useEffect hook segítségével a komponens szűri a csomagautomatákat a keresési kifejezés alapján:

```
useEffect(() => {
  const filtered = lockers.filter(locker) =>
    locker.locker_name.toLowerCase().includes(searchQuery.toLowerCase()) ||
    String(locker.address).toLowerCase().includes(searchQuery.toLowerCase());
  setFilteredLockers(filtered);
  setCurrentPage(1);
}, [searchQuery, lockers]);
```

3. Lapozás

A paginatedLockers változó tartalmazza az aktuális oldalon megjelenítendő csomagautomatákat:

```
const paginatedLockers = filteredLockers.slice(
  (currentPage - 1) * itemsPerPage,
  currentPage * itemsPerPage
);
```

4. Szűrőpanel kezelése (mobil nézet)

A szűrőpanel nyitott vagy zárt állapotát a isFilterOpen állapot kezeli:

```
const [isFilterOpen, setIsFilterOpen] = useState(false);
```



Felhasználói Élmény

- **Átlátható megjelenítés:**
A csomagautomaták kártyák formájában jelennek meg, és szűrhetők név vagy cím alapján.
- **Interaktív funkciók:**
A felhasználó navigálhat az oldalak között, és mobil nézetben használhatja a szűrőpanelt.
- **Reszponzív kialakítás:**
A komponens mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Szűrőpanel (mobil nézet)

A szűrőpanel mobil nézetben jelenik meg:

```
<div
  className={`fixed top-0 right-0 w-64 h-full bg-base-100 shadow-lg transform transition-
  transform z-50 ${ isFilterOpen ? "translate-x-0" : "translate-x-full" } lg:hidden`}>
  <div className="p-4">
    <div className="flex justify-end">
      <button
        className="text-primary font-bold text-xl mb-4"
        onClick={() => setIsFilterOpen(false)}
      >
        ×
      </button>
    </div>
    <h2 className="text-xl font-bold mb-4 text-primary text-center">Szűrők</h2>
    <div className="form-control">
      <input
        type="text"
        value={searchQuery}
        onChange={handleSearch}
        placeholder="Keresés..."
        className="input input-bordered w-full border-primary text-info"
      />
    </div>
  </div>
</div>
```



2. Szűrők (asztali nézet)

A szűrők asztali nézetben a bal oldalon jelennek meg:

```
<div className="hidden lg:block w-[20%] p-4">
  <h2 className="text-xl font-bold mb-2 text-center text-primary">Szűrők</h2>
  <div className="form-control">
    <input
      type="text"
      value={searchQuery}
      onChange={handleSearch}
      placeholder="Keresés csomagautomatára"
      className="input input-bordered w-full border-primary placeholder-info"
    />
  </div>
</div>
```

3. Csomagautomata kártyák

A csomagautomaták kártyák formájában jelennek meg:

```
<div className="w-full lg:w-3/4 grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-2 mx-auto pr-6">
  {paginatedLockers.map((locker) => (
    <LockersCard key={locker.id} locker={locker} />
  ))}
</div>
```



4. Lapozás

A lapozás gombjai:

```
<div className="join flex justify-center mt-10 mb-24">
  <button
    className="join-item btn btn-secondary"
    onClick={handlePrevPage} disabled={currentPage === 1}>
    <<
  </button>
  <button className="join-item btn btn-primary text-white">
    Oldal {currentPage} / {totalPages}
  </button>
  <button
    className="join-item btn btn-secondary"
    onClick={handleNextPage} disabled={currentPage === totalPages}>
    >
  </button>
</div>
```

Összegzés

A LockersList React komponens hatékonyan kezeli a csomagautomaták megjelenítését, szűrését és lapozását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens könnyen integrálható a csomagautomata-kezelési folyamatokba, és rezponzív kialakításának köszönhetően minden eszközön jól használható.

The screenshot shows the 'Összes Csomagautomata' (All Lockers) section of the Flexistore Admin Dashboard. It lists three locker locations with their addresses and three action buttons (Edit, Delete, and Info) for each. The interface is clean and modern, using a light blue and white color scheme.

29. ábra Locker List



React Controller Dokumentáció – NewLocker.jsx

A NewLocker React komponens egy új csomagautomata létrehozására vagy meglévő csomagautomata adatainak módosítására szolgál. Ez a komponens API-hívásokkal kommunikál a backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Új csomagautomata létrehozása

- Leírás:**
A felhasználó megadhatja az új csomagautomata nevét, címét és leírását, amelyeket a komponens elküld a backendnek egy POST kérésben.
- Backend végpont:**
POST /locker
- Küldött adatok:**
 - locker_name: A csomagautomata neve.
 - address: A csomagautomata címe.
 - description: A csomagautomata leírása.

2. Meglévő csomagautomata módosítása

- Leírás:**
A felhasználó módosíthatja egy meglévő csomagautomata adatait. A komponens a csomagautomata adatait előre kitölți, és egy PATCH kérésben küldi el a módosított adatokat a backendnek.
- Backend végpont:**
PATCH /locker
- Küldött adatok:**
 - id: A csomagautomata azonosítója.
 - locker_name: A csomagautomata neve.
 - address: A csomagautomata címe.
 - description: A csomagautomata leírása.



API-hívások

1. Új csomagautomata létrehozása

- **HTTP-módszer:** POST
- **Végpont:** /locker
- **Küldött adatok:**

```
{  
    "locker_name": "Új Automata",  
    "address": "Budapest, Fő utca 12.",  
    "description": "Ez egy új csomagautomata."  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Új csomagautomata sikeresen létrehozva!"  
}
```

2. Meglévő csomagautomata módosítása

- **HTTP-módszer:** PATCH
- **Végpont:** /locker
- **Küldött adatok:**

```
{  
    "id": 1,  
    "locker_name": "Módosított Automata",  
    "address": "Budapest, Fő utca 12.",  
    "description": "Ez egy módosított csomagautomata."  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Csomagautomata sikeresen módosítva!"  
}
```



Adatok kezelése

1. Csomagautomata adatok inicializálása

A komponens a useLocation hook segítségével ellenőrzi, hogy új csomagautomatát kell létrehozni, vagy meglévőt kell módosítani:

```
let formObj = {
  id: '',
  locker_name: '',
  address: '',
  description: '',
};

if(state !== null) {
  const { locker } = state;
  formObj = {
    id: locker.id,
    locker_name: locker.locker_name,
    address: locker.address,
    description: locker.description,
  };
  method = "PATCH";
  cim = `${locker.locker_name} módosítása`;
}
```

2. Adatok mentése

Az onSubmit függvény kezeli az űrlap beküldését:

```
const onSubmit = (e) => {
  e.preventDefault();
  backendMuvelet(formData, method, url, header);
  navigate("/lockers");
};
```



3. Űrlap mezők kezelése

A `writeData` függvény frissíti az űrlap mezőinek értékeit:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

Felhasználói Élmény

- Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek meglévő csomagautomata módosítása esetén.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül a csomagautomaták listájára.

Komponens Felépítése

1. Űrlap mezők

- Csomagautomata neve mező:**

```
<input
  className="grow placeholder-info"
  type="text"
  id="locker_name"
  placeholder="Csomagautomata neve"
  required
  onChange={writeData}
  value={formData.locker_name}/>
```



- Cím mező:

```
<input  
    className="grow placeholder-info"  
    type="text"  
    id="address"  
    placeholder="Cím"  
    required  
    onChange={writeData}  
    value={formData.address}/>
```

- Leírás mező:

```
<textarea  
    className="textarea textarea-bordered h-24 textarea-auto w-full border-primary placeholder-info"  
    id="description"  
    placeholder="Csomagautomata leírása"  
    required  
    onChange={writeData}  
    value={formData.description}/>
```

2. Gombok

- Felvitel gomb:

```
<button type="submit" className="btn btn-primary text-white">  
    Felvitel  
</button>
```

Összegzés

A NewLocker React komponens hatékonyan kezeli az új csomagautomaták létrehozását és a meglévő csomagautomaták módosítását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a csomagautomata-kezelési folyamatokba.



React Controller Dokumentáció – NewPaymentMethod.jsx

A NewPaymentMethod React komponens egy új fizetési mód létrehozására vagy meglévő fizetési mód adatainak módosítására szolgál. Ez a komponens API-hívásokkal kommunikál a backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Új fizetési mód létrehozása

- **Leírás:**
A felhasználó megadhatja az új fizetési mód nevét, amelyet a komponens elküld a backendnek egy POST kérésben.
- **Backend végpont:**
POST /payment
- **Küldött adatok:**
 - card_type: A fizetési mód neve.

2. Meglévő fizetési mód módosítása

- **Leírás:**
A felhasználó módosíthatja egy meglévő fizetési mód nevét. A komponens a fizetési mód adatait előre kitölti, és egy PATCH kérésben küldi el a módosított adatokat a backendnek.
 - **Backend végpont:**
PATCH /payment
 - **Küldött adatok:**
 - id: A fizetési mód azonosítója.
 - card_type: A fizetési mód neve.
-



API-hívások

1. Új fizetési mód létrehozása

- **HTTP-módszer:** POST
- **Végpont:** /payment
- **Küldött adatok:**

```
{  
    "card_type": "Bankkártya"  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Új fizetési mód sikeresen létrehozva!"  
}
```

2. Meglévő fizetési mód módosítása

- **HTTP-módszer:** PATCH
- **Végpont:** /payment
- **Küldött adatok:**

```
{  
    "id": 1,  
    "card_type": "Hitelkártya"  
}
```

- **Sikeres válasz:**

```
{  
    "message": "Fizetési mód sikeresen módosítva!"  
}
```



Adatok kezelése

1. Fizetési mód adatok inicializálása

A komponens a `useLocation` hook segítségével ellenőrzi, hogy új fizetési módot kell létrehozni, vagy meglévőt kell módosítani:

```
let formObj = {
  id: '',
  card_type: '',
};

if (state !== null) {
  const { payment } = state;
  formObj = {
    id: payment.id,
    card_type: payment.card_type,
  };
  method = "PATCH";
  cim = `${payment.card_type} módosítása`;
  header = {
    "Content-type": "application/json",
    "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
    "PaymentId": payment.id,
  };
}
```

2. Adatok mentése

Az `onSubmit` függvény kezeli az űrlap beküldését:

```
const onSubmit = (e) => {
  e.preventDefault();
  backendMuvelet(formData, method, url, header);
  navigate("/paymentmethods");
};
```



3. Űrlap mezők kezelése

A `writeData` függvény frissíti az űrlap mezőinek értékeit:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

Felhasználói Élmény

- Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek meglévő fizetési mód módosítása esetén.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül a fizetési módok listájára.

Komponens Felépítése

1. Űrlap mezők

- Fizetési mód neve mező:**

```
<input className="grow placeholder-info" type="text" id="card_type"
placeholder="Fizetési mód neve" required onChange={writeData} value={formData.card_type}>
```



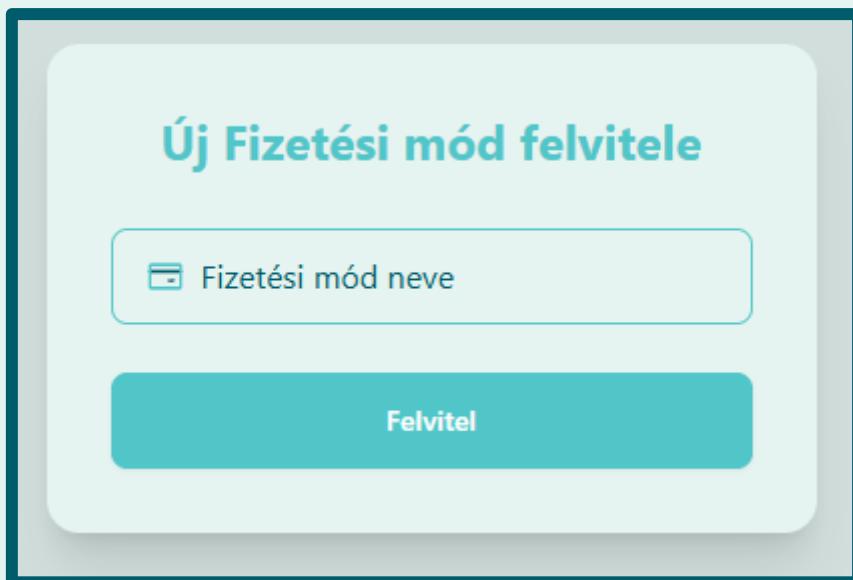
2. Gombok

- **Felvitel gomb:**

```
<button type="submit" className="btn btn-primary text-white">  
    Felvitel  
</button>
```

Összegzés

A [NewPaymentMethod](#) React komponens hatékonyan kezeli az új fizetési módok létrehozását és a meglévő fizetési módok módosítását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a fizetési módok kezelésébe.



30. ábra New Payment Panel



React Controller Dokumentáció – PaymentMethodCard.jsx

A PaymentMethodCard React komponens egy fizetési mód adatait jeleníti meg kártya formájában. Ez a komponens lehetővé teszi a fizetési mód módosítását és törlését, ha a felhasználó adminisztrátori jogosultsággal rendelkezik.

Funkciók

1. Fizetési mód megjelenítése

- **Leírás:**

A komponens megjeleníti a fizetési mód nevét (card_type) egy kártyán.

2. Fizetési mód módosítása

- **Leírás:**

A "Módosítás" gombra kattintva a felhasználó navigálhat a fizetési mód módosítására szolgáló oldalra.

- **Függvény:**

modosit

3. Fizetési mód törlése

- **Leírás:**

A "Törlés" gombra kattintva a fizetési mód törölhető a rendszerből. A törlés egy DELETE API-hívással történik.

- **Függvény:**

torles



API-hívások

1. Fizetési mód törlése

- **HTTP-módszer:** DELETE
- **Végpont:** /payment/delete
- **Küldött adatok (fejlécben):**

```
{  
  "Content-type": "application/json",  
  "Authorization": "Bearer <usertoken>",  
  "PaymentId": 1  
}
```

- **Sikeres válasz:**

```
{  
  "message": "Fizetési mód sikeresen törölve!"  
}
```

- **Hibás válasz:**

```
{  
  "error": "Nem sikerült a fizetési mód törlése."  
}
```

Adatok kezelése

1. Fizetési mód megjelenítése

A komponens a payment prop segítségével kapja meg a fizetési mód adatait, és megjeleníti annak nevét:

```
<h2 className="card-title justify-center text-primary">{payment.card_type}</h2>
```



2. Fizetési mód módosítása

A modosit függvény navigál a fizetési mód módosítására szolgáló oldalra:

```
const modosit = () => {
  navigate("/newpaymentmethod", { state: { payment } });
};
```

3. Fizetési mód törlése

A torles függvény meghívásával a komponens törli a fizetési módot:

```
const torles = (payment) => {
  backendMuvelet(
    payment,
    "DELETE",
    `${import.meta.env.VITE_BASE_URL}/payment/delete`,
    {
      "Content-type": "application/json",
      "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
      "PaymentId": payment.id
    }
  );
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A fizetési mód neve jól láthatóan jelenik meg egy kártyán.
 - Interaktív funkciók:**
Az adminisztrátori jogosultsággal rendelkező felhasználók módosíthatják vagy törölhetik a fizetési módot.
 - Biztonságos hozzáférés:**
A komponens ellenőrzi a felhasználó jogosultságát, mielőtt engedélyezné a módosítást vagy törlést.
-



Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a fizetési mód neve jelenik meg:

```
<h2 className="card-title justify-center text-primary">{payment.card_type}</h2>
```

2. Módosítás gomb

A "Módosítás" gomb:

```
{user.isAdmin >= 70 && (
  <button className="btn btn-primary text-white" onClick={() => modosit(payment)}>
    Módosítás
  </button>
)}
```

3. Törlés gomb

A "Törlés" gomb:

```
{user.isAdmin >= 70 && (
  <button className="btn btn-info text-white" onClick={() => torles(payment)}>
    Törlés
  </button>
)}
```

Összegzés

A PaymentMethodCard React komponens hatékonyan kezeli a fizetési módok megjelenítését, módosítását és törlését. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a fizetési módok kezelésébe.



React Controller Dokumentáció – PaymentMethodList.jsx

A [PaymentMethodList](#) React komponens a fizetési módok listájának megjelenítésére szolgál. Ez a komponens a [PaymentContext](#)-ből kapja a fizetési módok adatait, és azokat kártyák formájában jeleníti meg a [PaymentMethodCard](#) komponens segítségével.

Funkciók

1. Fizetési módok listázása

- **Leírás:**

A komponens a [PaymentContext](#)-ből kapott fizetési módokat jeleníti meg kártyák formájában.

Adatok kezelése

1. Fizetési módok betöltése

A [PaymentContext](#) biztosítja a fizetési módok adatait:

```
const { payments } = useContext(PaymentContext);
```

2. Fizetési módok megjelenítése

A [payments](#) tömb elemei a [PaymentMethodCard](#) komponens segítségével jelennek meg:

```
<div className="flex flex-row flex-wrap items-center justify-center">
  {payments.map((payment) => (
    <PaymentMethodCard key={payment.id} payment={payment} />
  )));
</div>
```



Felhasználói Élmény

- **Átlátható megjelenítés:**
A fizetési módok kártyák formájában jelennek meg, amelyek könnyen áttekinthetők.
- **Interaktív funkciók:**
A kártyákon keresztül a felhasználó módosíthatja vagy törölheti a fizetési módokat (ha adminisztrátori jogosultsággal rendelkezik).

Komponens Felépítése

1. Cím

A lista címe:

```
<h1 className="text-3xl font-bold text-center mb-4 text-primary">  
Fizetési módok listája  
</h1>
```

2. Fizetési módok kártyái

A fizetési módok kártyák formájában jelennek meg:

```
<div className="flex flex-row flex-wrap items-center justify-center">  
{payments.map((payment) => (  
    <PaymentMethodCard key={payment.id} payment={payment} />  
)})  
</div>
```



Összegzés

A [PaymentMethodList](#) React komponens hatékonyan kezeli a fizetési módok megjelenítését. Az adatok jól strukturáltan jelennek meg, és a [PaymentMethodCard](#) komponens segítségével interaktív funkciókat is biztosít. A komponens könnyen integrálható a fizetési módok kezelésébe, és reszponzív kialakításának köszönhetően minden eszközön jól használható.



React Controller Dokumentáció – NewProduct.jsx

A [NewProduct](#) React komponens egy új termék létrehozására vagy meglévő termék adatainak módosítására szolgál. Ez a komponens API-hívásokkal kommunikál a backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Új termék létrehozása

- Leírás:**
A felhasználó megadhatja az új termék nevét, leírását, árát, kategóriáját, elérhetőségét, valamint a hozzá tartozó csomagautomatákat és képet. Az adatokat a komponens elküldi a backendnek egy POST kérésben.
- Backend végpont:**
POST /product

2. Meglévő termék módosítása

- Leírás:**
A felhasználó módosíthatja egy meglévő termék adatait. A komponens a termék adatait előre kitölți, és egy POST kérésben küldi el a módosított adatokat a backendnek.
- Backend végpont:**
POST /product/update

API-hívások

1. Új termék létrehozása

- HTTP-módszer:** POST
- Végpont:** /product



- Küldött adatok:

```
{  
    "name": "Termék neve",  
    "description": "Termék leírása",  
    "price_per_day": 1000,  
    "category_id": 1,  
    "available": 1,  
    "locker_ids": [1, 2],  
    "image": "fájl"  
}
```

- Sikeres válasz:

```
{  
    "message": "Új termék sikeresen létrehozva!"  
}
```

2. Meglévő termék módosítása

- HTTP-módszer: POST
- Végpont: /product/update
- Küldött adatok:

```
{  
    "id": 1,  
    "name": "Módosított termék neve",  
    "description": "Módosított termék leírása",  
    "price_per_day": 1200,  
    "category_id": 2,  
    "available": 0,  
    "locker_ids": [3],  
    "image": "fájl"  
}
```

- Sikeres válasz:

```
{  
    "message": "Termék sikeresen módosítva!"  
}
```



Adatok kezelése

1. Termék adatok inicializálása

A komponens a `useLocation` hook segítségével ellenőrzi, hogy új terméket kell létrehozni, vagy meglévőt kell módosítani:

```
let formObj = {
  id: "",
  name: "",
  description: "",
  price_per_day: "",
  category_id: 1,
  available: 1,
  locker_ids: [],
};

if(state !== null) {
  const { product } = state;
  formObj = {
    id: product.id,
    name: product.name,
    description: product.description,
    price_per_day: product.price_per_day,
    category_id: product.category_id,
    available: product.available,
    locker_ids: product.lockers.map(locker) => locker.id,
  };
}

method = "POST";
url = `${import.meta.env.VITE_BASE_URL}/product/update`;
cím = `${product.name} módosítása`;
```



2. Adatok mentése

Az onSubmit függvény kezeli az űrlap beküldését:

```
const onSubmit = async (e) => {
  e.preventDefault();
  const formDataToSubmit = new FormData();
  formDataToSubmit.append("name", formData.name);
  formDataToSubmit.append("description", formData.description);
  formDataToSubmit.append("price_per_day", formData.price_per_day);
  formDataToSubmit.append("category_id", formData.category_id);
  formDataToSubmit.append("available", formData.available);
  formData.locker_ids.forEach((id) => {
    formDataToSubmit.append("locker_ids[]", id);
  });
  if (image && image.length > 0) {
    formDataToSubmit.append("image", image[0]);
  } else if (!formData.id) {
    toast.error("Nincs kép kiválasztva!");
    return;
  }
  if (formData.id) {
    formDataToSubmit.append("id", formData.id);
  }
  try {
    await backendMuveletFile(
      formDataToSubmit,
      method,
      url,
      { Authorization: `Bearer ${sessionStorage.getItem("usertoken")}` },
      successMessage,
      errorMessage
    );
    update();
    navigate("/products");
  } catch (error) {
    console.error("Hiba történt:", error);
  }
};
```



3. Űrlap mezők kezelése

A `writeData` függvény frissíti az űrlap mezőinek értékeit:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

Felhasználói Élmény

- Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek meglévő termék módosítása esetén.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül a termékek listájára.

Komponens Felépítése

1. Űrlap mezők

- Termék neve mező:**

```
<input type="text" id="name" placeholder="Termék neve" required
      onChange={writeData} value={formData.name}/>
```

- Leírás mező:**

```
<textarea id="description" placeholder="Termék leírása"
      required onChange={writeData} value={formData.description}/>
```



- Ár mező:

```
<input type="number" id="price_per_day" placeholder="Termék Bérlesi ára Ft/nap"  
      required onChange={writeData} value={formData.price_per_day}/>
```

- Kategória mező:

```
<select id="category_id" onChange={writeData} value={formData.category_id}>  
  {categories.map((category) => (  
    <option key={category.id} value={category.id}>  
      {category.name}  
    </option>  
  ))}  
</select>
```

- Csomagautomata mező:

```
<select id="locker_ids" multiple onChange={(e) => {  
  const selected = Array.from(e.target.selectedOptions, (option) => option.value);  
  setFormData((prev) => ({ ...prev, locker_ids: selected }));  
} }value={formData.locker_ids || []}>  
{lockers.map((locker) => (  
  <option key={locker.id} value={locker.id}>  
    {locker.locker_name}  
  </option>  
)})  
</select>
```

2. Gombok

- Felvitel gomb:

```
<button type="submit" className="btn btn-primary text-white">  
  Felvitel  
</button>
```



Összegzés

A NewProduct React komponens hatékonyan kezeli az új termékek létrehozását és a meglévő termékek módosítását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a termékkezelési folyamatokba.

Új termék felvitele

FÁJL KIVÁLASZTÁSA Nincs f...választva

Termék neve

Termék leírása

Termék Bérlési ára Ft/nap

Kategória

Háztartási kiszépek ▾

Csomagautomata

- Nagy Tesco Békéscsaba Bejárat
- Nagy Tesco Szeged Bejárat
- Budapest Ázsia Center Mélygarázs

Elérhető ▾

Felvitel

31. ábra New Product Panel



React Controller Dokumentáció – ProductsCard.jsx

A ProductsCard React komponens egy termék adatait jeleníti meg kártya formájában. Ez a komponens lehetővé teszi a termék részleteinek megtekintését, kosárba helyezését, módosítását és törlését, valamint a csomagautomata kiválasztását.

Funkciók

1. Termék megjelenítése

- **Leírás:**

A komponens megjeleníti a termék nevét, napi bérleti árát, képét, és a hozzá tartozó csomagautomaták listáját.

2. Termék részleteinek megtekintése

- **Leírás:**

Az "Info" gombra kattintva a termék részletei egy modális ablakban jelennek meg.

- **Függvények:**

- openInfo: A modális ablak megnyitása.
- closeInfo: A modális ablak bezárasa.

3. Termék kosárba helyezése

- **Leírás:**

A "Kosárba" gombra kattintva a termék hozzáadható a kosárhoz a kiválasztott csomagautomatával együtt.

- **Függvény:**

addToCart



4. Termék módosítása

- **Leírás:**
A "Módosítás" gombra kattintva a felhasználó navigálhat a termék módosítására szolgáló oldalra.
- **Függvény:**
modosit

5. Termék törlése

- **Leírás:**
A "Törlés" gombra kattintva a termék törölhető a rendszerből. A törlés egy DELETE API-hívással történik.
- **Függvény:**
torles

6. Csomagautomata kiválasztása

- **Leírás:**
A felhasználó kiválaszthatja, hogy melyik csomagautomatából szeretné bérelni a terméket.
- **Függvény:**
handleLockerChange

API-hívások

1. Termék törlése

- **HTTP-módszer:** DELETE
- **Végpont:** /product/delete
- **Küldött adatok (fejlécben):**

```
{  
  "Content-Type": "application/json",  
  "Authorization": "Bearer <usertoken>",  
  "productId": 1  
}
```



- Sikeres válasz:

```
{  
  "message": "Termék sikeresen törölve!"  
}
```

- Hibás válasz:

```
{  
  "error": "Nem sikerült a termék törlése."  
}
```

Adatok kezelése

1. Termék adatok megjelenítése

A komponens a product prop segítségével kapja meg a termék adatait, és azokat kártya formájában jeleníti meg:

```
<h2 className="card-title line-clamp-2 text-center text-primary font-bold">{product.name}</h2>  
<h2 className="card-title line-clamp-2 text-center text-info">{product.price_per_day} Ft/nap</h2>
```

2. Csomagautomata kiválasztása

A handleLockerChange függvény frissíti a kiválasztott csomagautomata értékét:

```
const handleLockerChange = (event) => {  
  setSelectedLocker(event.target.value);  
};
```



3. Termék törlése

A torles függvény meghívásával a komponens törli a terméket:

```
const torles = (product) => {
  const method = "DELETE";
  const url = `${import.meta.env.VITE_BASE_URL}/product/delete`;
  const header = {
    "Content-Type": "application/json",
    Authorization: `Bearer ${sessionStorage.getItem("usertoken")}`,
    productId: product.id,
  };
  const successMessage = `${product.name} sikeresen törölve!`;
  const errorMessage = `${product.name} törlése sikertelen!`;
  backendMuvelet(null, method, url, header, successMessage, errorMessage);
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A termék adatai jól strukturáltan jelennek meg kártya formájában.
 - Interaktív funkciók:**
A felhasználó megtekintheti a termék részleteit, kosárba helyezheti, módosíthatja vagy törölheti azt.
 - Reszponzív kialakítás:**
A kártya mobil- és asztali eszközökön egyaránt jól használható.
-

Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a termék neve és napi bérlesi ára jelenik meg:

```
<h2 className="card-title line-clamp-2 text-center text-primary font-bold">{product.name}</h2>
<h2 className="card-title line-clamp-2 text-center text-info">{product.price_per_day} Ft/nap</h2>
```



2. Csomagautomata kiválasztása

A csomagautomata kiválasztására szolgáló legördülő menü:

```
<select value={selectedLocker} onChange={handleLockerChange}>  
  className="w-full rounded-lg border border-primary bg-base-100 py-2.5 px-3 text-sm text-gray-800 shadow-sm focus:border-secondary focus:ring-2 focus:ring-secondary/50">  
  {product.lockers.map(locker => (  
    <option key={locker.id} value={locker.id}>  
      {locker.address}  
    </option>  
  ))}  
</select>
```

3. Gombok

- **Módosítás gomb:**

```
<button className="btn btn-primary text-white" onClick={() => modosit(product)}>  
  Módosítás  
</button>
```

- **Törlés gomb:**

```
<button className="btn btn-info text-white" onClick={() => torles(product)}>  
  Törlés  
</button>
```

- **Kosárba gomb:**

```
<button className="btn btn-success text-white" id="add-to-cart-button" onClick={() => addToCart(product, selectedLocker)}>  
  Kosárba  
</button>
```

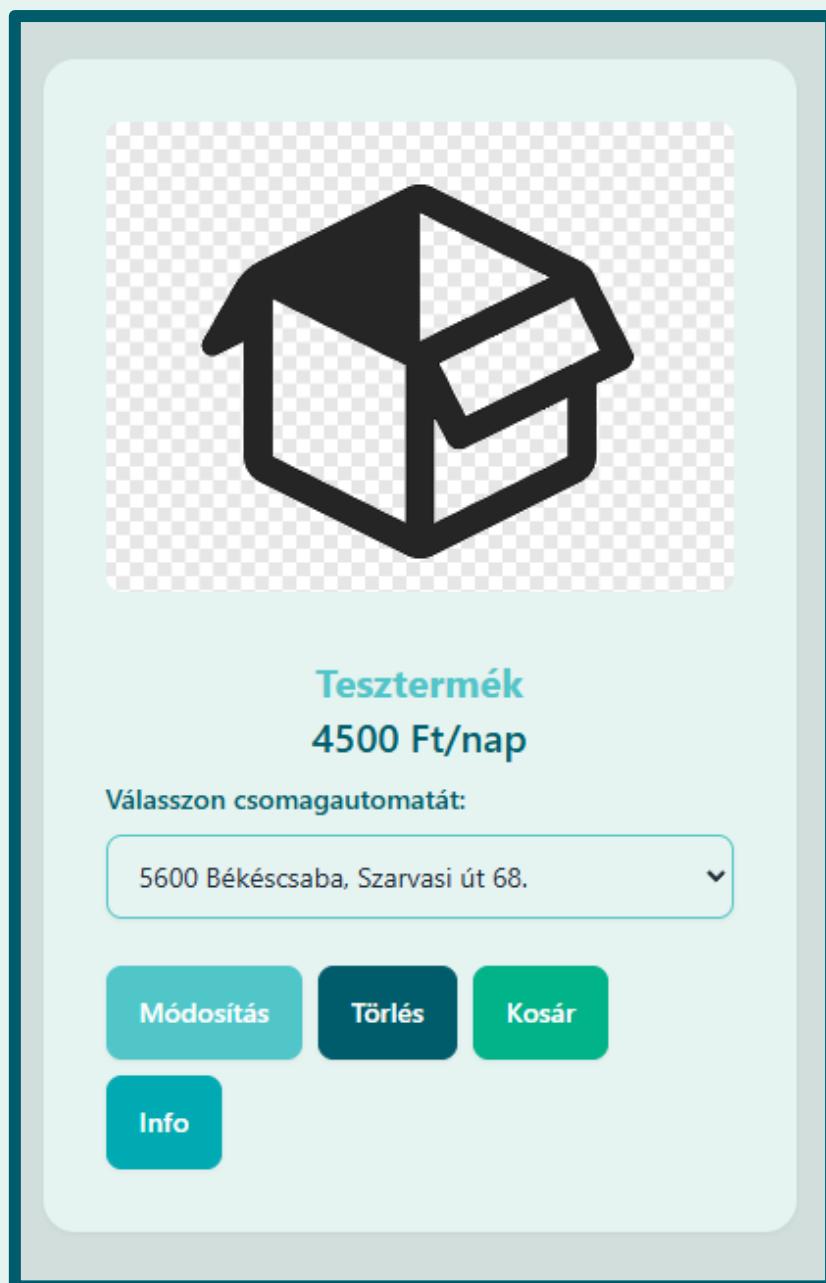
- **Info gomb:**

```
<button className="btn btn-secondary text-white" onClick={() => openInfo()}>  
  Info  
</button>
```



Összegzés

A ProductsCard React komponens hatékonyan kezeli a termékek megjelenítését, részleteinek megtekintését, kosárba helyezését, módosítását és törlését. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens könnyen integrálható a termékkezelési folyamatokba, és reszponzív kialakításának köszönhetően minden eszközön jól használható.



32. ábra Product Card



React Controller Dokumentáció – ProductsInfo.jsx

A ProductsInfo React komponens egy modális ablakot jelenít meg, amely egy termék részletes leírását tartalmazza. Ez a komponens egyszerűen használható a termékek információinak megtekintésére.

Funkciók

1. Termék leírásának megjelenítése

- **Leírás:**

A komponens megjeleníti a termék leírását (product.description) egy modális ablakban.

2. Modális ablak bezárása

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva bezárhatja a modális ablakot.

- **Függvény:**

closeFunction

Adatok kezelése

1. Termék adatok

A komponens a product prop segítségével kapja meg a termék adatait, és megjeleníti annak leírását:

```
<p className="text-info text-left">{product.description}</p>
```



2. Modális ablak bezárása

A "Bezárás" gomb meghívja a `closeFunction` függvényt, amelyet a szülő komponens biztosít:

```
<button
  className="btn btn-ghost btn-circle absolute right-2 top-2 text-info"
  onClick={closeFunction}>
  ×
</button>
```

Felhasználói Élmény

- Egyszerű megjelenítés:**
A termék leírása egy modális ablakban jelenik meg, amely könnyen olvasható és kezelhető.
- Interaktív funkciók:**
A felhasználó egyetlen gombnyomással bezárhatja a modális ablakot.
- Rezonans kialakítás:**
A modális ablak mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Modális fejléc

A modális fejlécében a cím jelenik meg:

```
<h2 className="text-primary font-bold text-center text-lg">Termék leírás</h2>
```

2. Termék leírás

A termék leírása:

```
<p className="text-info text-left">{product.description}</p>
```



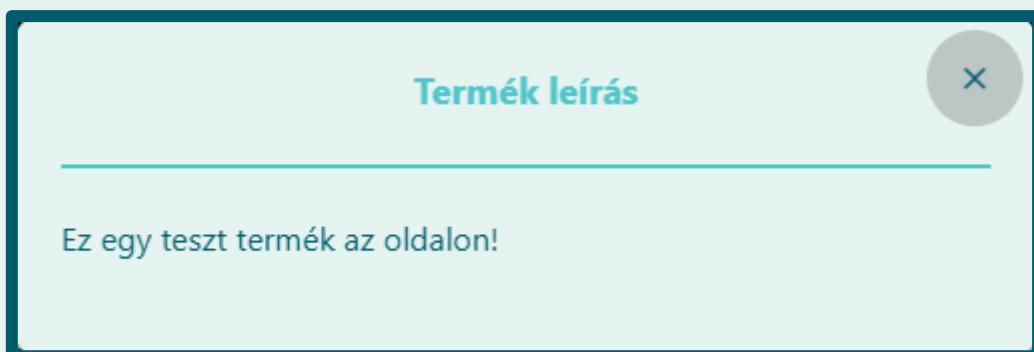
3. Bezárás gomb

A "Bezárás" gomb:

```
<button  
  className="btn btn-ghost btn-circle absolute right-2 top-2 text-info"  
  onClick={closeFunction}>  
  X  
</button>
```

Összegzés

A ProductsInfo React komponens egyszerű és hatékony módot biztosít a termékek leírásának megjelenítésére egy modális ablakban. A komponens könnyen integrálható más felhasználói felületekbe, és intuitív felhasználói élményt nyújt.



33. ábra Product Info



React Controller Dokumentáció – ProductsList.jsx

A ProductsList React komponens a termékek listájának megjelenítésére szolgál. Ez a komponens lehetővé teszi a termékek szűrését, lapozását, és kártyák formájában történő megjelenítését a ProductsCard komponens segítségével.

Funkciók

1. Termékek listázása

- **Leírás:**

A komponens a InitialContext-ből kapott termékeket jeleníti meg kártyák formájában.

2. Termékek szűrése

- **Leírás:**

A felhasználó szűrheti a termékeket név, ár, elérhetőség, csomagautomata vagy kategória alapján.

- **Függvények:**

- handleSearch: Keresési kifejezés alapján szűri a termékeket.
- setFilteredCities: Csomagautomata alapján szűr.
- setFilteredCategories: Kategória alapján szűr.
- setMinPrice és setMaxPrice: Ár alapján szűr.

3. Lapozás

- **Leírás:**

A termékek listája több oldalra osztható, és a felhasználó navigálhat az oldalak között.

- **Függvények:**

- handlePrevPage: Előző oldalra lépés.
- handleNextPage: Következő oldalra lépés.



4. Szűrőpanel (mobil nézet)

- **Leírás:**

A mobil nézetben egy szűrőpanel jelenik meg, amely beúszik a képernyő jobb oldaláról.

- **Függvények:**

- setIsFilterOpen: A szűrőpanel nyitott vagy zárt állapotának kezelése.

Adatok kezelése

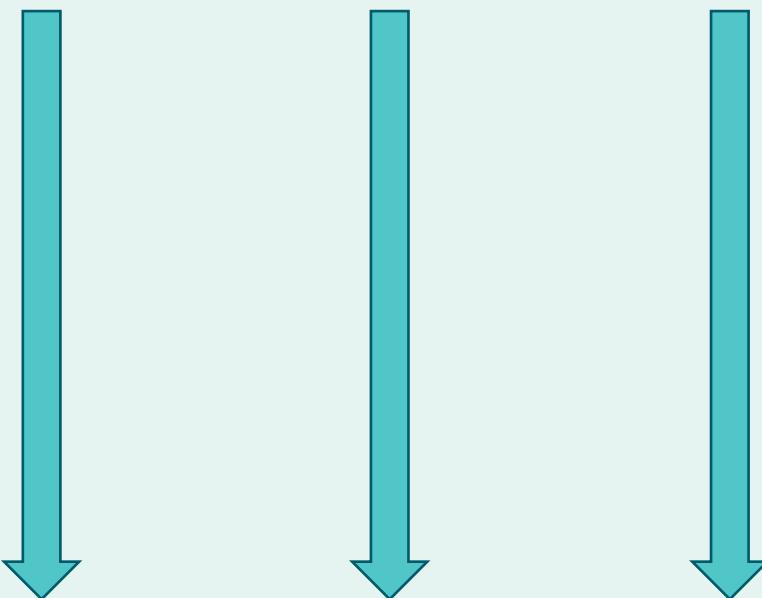
1. Termékek betöltése

A InitialContext biztosítja a termékek, csomagautomaták és kategóriák adatait:

```
const { products, lockers, categories } = useContext(InitialContext);
```

2. Szűrés

A useEffect hook segítségével a komponens szűri a termékeket a keresési kifejezés, csomagautomata, kategória és ár alapján:





```
useEffect(() => {
  let filtered = products.filter((product) =>
    product.name.toLowerCase().includes(searchQuery.toLowerCase()) ||
    String(product.price_per_day).toLowerCase().includes(searchQuery.toLowerCase()) ||
    (product.available ? "Elérhető" : "Nem elérhető").toLowerCase().includes(searchQuery.toLowerCase())
  );
  if (filteredCitys !== "") {
    filtered = filtered.filter((product) =>
      product.lockers.some(locker => locker.id === parseInt(filteredCitys))
    );
  }
  if (filteredCategories !== "") {
    filtered = filtered.filter((product) => product.category.id === parseInt(filteredCategories));
  }
  if (minPrice !== null && maxPrice !== null) {
    filtered = filtered.filter(
      (product) =>
        product.price_per_day >= minPrice && product.price_per_day <= maxPrice
    );
  }
  setFilteredProducts(filtered);
}, [searchQuery, filteredCitys, products, filteredCategories, minPrice, maxPrice]);
```

3. Lapozás

A paginatedProducts változó tartalmazza az aktuális oldalon megjelenítendő termékeket:

```
const paginatedProducts = filteredProducts.slice(
  (currentPage - 1) * itemsPerPage,
  currentPage * itemsPerPage
);
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A termékek kártyák formájában jelennek meg, amelyek könnyen áttekinthetők.
- Interaktív funkciók:**
A felhasználó szűrheti a termékeket, navigálhat az oldalak között, és mobil nézetben használhatja a szűrőpanelt.



- **Reszponzív kialakítás:**

A komponens mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Szűrőpanel (mobil nézet)

A szűrőpanel mobil nézetben jelenik meg:

```
<div
  className={`fixed top-0 right-0 w-64 h-full bg-base-100 shadow-
lg transform transition-transform z-50 ${
    isFilterOpen ? "translate-x-0": "translate-x-full"
  } lg:hidden`}>
  <div className="p-4">
    <div className="flex justify-end">
      <button className="text-xl font-bold text-primary"
        onClick={() => setIsFilterOpen(false)}>
        ×
      </button>
    </div>
    <h2 className="text-xl font-bold mb-4 text-primary text-center">Szűrők</h2>
    <div className="form-control mb-4">
      <input
        type="text"
        value={searchQuery}
        onChange={handleSearch}
        placeholder="Keresés termékek között"
        className="input input-bordered w-full input-primary placeholder-info"
      />
    </div>
    {/* További szűrők */}
  </div>
</div>
```



2. Szűrők (asztali nézet)

A szűrők asztali nézetben a bal oldalon jelennek meg:

```
<div className="hidden lg:block w-[20%] p-4">
  <h2 className="text-xl font-bold mb-2 text-center text-primary">Szűrők</h2>
  <div className="form-control mb-4">
    <input
      type="text"
      value={searchQuery}
      onChange={handleSearch}
      placeholder="Keresés termékek között"
      className="input input-bordered w-full input-primary placeholder-info"
    />
  </div>
  {/* További szűrők */}
</div>
```

3. Termékkártyák

A termékek kártyák formájában jelennek meg:

```
<div className="w-full lg:w-4/5 flex flex-wrap justify-center gap-4">
  {paginatedProducts.map((product) => (
    <ProductsCard key={product.id} product={product} />
  )))
</div>
```

4. Lapozás

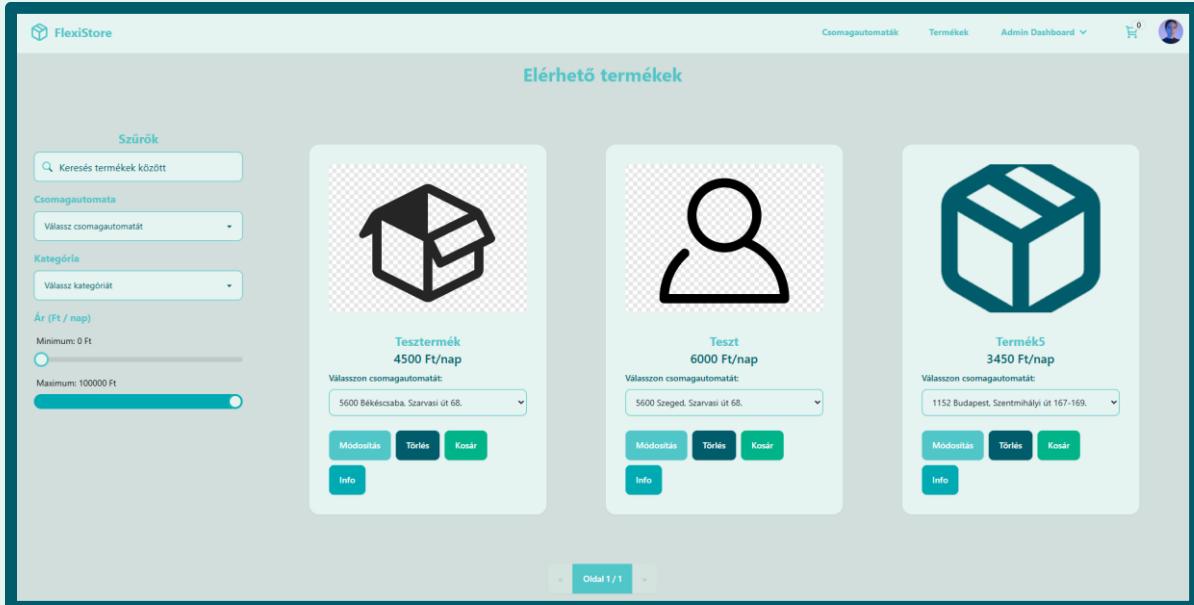
A lapozás gombjai:

```
<div className="join flex justify-center mt-10 mb-24">
  <button className="join-item btn btn-secondary"
    onClick={handlePrevPage} disabled={currentPage === 1}>
    «
  </button>
  <button className="join-item btn btn-primary text-white">
    Oldal {currentPage} / {totalPages}
  </button>
  <button className="join-item btn btn-secondary"
    onClick={handleNextPage} disabled={currentPage === totalPages}>
    »
  </button>
</div>
```

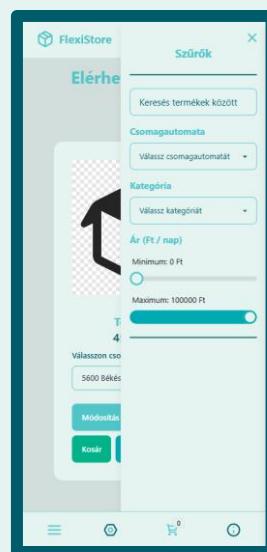


Összegzés

A ProductsList React komponens hatékonyan kezeli a termékek megjelenítését, szűrését és lapozását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és reszponzív kialakításának köszönhetően minden eszközön jól használható.



34. ábra Products List Dekstop



35. ábra Products List Mobile



React Controller Dokumentáció – NewRole.jsx

A NewRole React komponens egy új jogosultság létrehozására vagy meglévő jogosultság adatainak módosítására szolgál. Ez a komponens API-hívásokkal kommunikál a backenddel, és adminisztrátori jogosultságot igényel a műveletek végrehajtásához.

Funkciók

1. Új jogosultság létrehozása

- **Leírás:**
A felhasználó megadhatja az új jogosultság nevét és értékét, amelyeket a komponens elküld a backendnek egy POST kérésben.
- **Backend végpont:**
POST /role
- **Küldött adatok:**
 - power: A jogosultság értéke.
 - warrant_name: A jogosultság neve.

2. Meglévő jogosultság módosítása

- **Leírás:**
A felhasználó módosíthatja egy meglévő jogosultság nevét és értékét. A komponens a jogosultság adatait előre kitölti, és egy PATCH kérésben küldi el a módosított adatokat a backendnek.
- **Backend végpont:** PATCH /role
- **Küldött adatok:**
 - id: A jogosultság azonosítója.
 - power: A jogosultság értéke.
 - warrant_name: A jogosultság neve.



API-hívások

1. Új jogosultság létrehozása

- **HTTP-módszer:** POST
- **Végpont:** /role
- **Küldött adatok:**

```
{  
  "power": 50,  
  "warrant_name": "Adminisztrátor"  
}
```

- **Sikeres válasz:**

```
{  
  "message": "Új jogosultság sikeresen létrehozva!"  
}
```

2. Meglévő jogosultság módosítása

- **HTTP-módszer:** PATCH
- **Végpont:** /role
- **Küldött adatok:**

```
{  
  "id": 1,  
  "power": 70,  
  "warrant_name": "Super Admin"  
}
```

- **Sikeres válasz:**

```
{  
  "message": "Jogosultság sikeresen módosítva!"  
}
```



Adatok kezelése

1. Jogosultság adatok inicializálása

A komponens a `useLocation` hook segítségével ellenőrzi, hogy új jogosultságot kell létrehozni, vagy meglévőt kell módosítani:

```
let formObj = {
  id: '',
  power: '',
  warrant_name: ''
};

if (state !== null) {
  const { role } = state;
  formObj = {
    id: role.id,
    power: role.power,
    warrant_name: role.warrant_name
  };
  method = "PATCH";
  cim = `${role.warrant_name} módosítása`;
  header = {
    "Content-type": "application/json",
    "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
    "RoleId": role.id
  };
}
```

2. Adatok mentése

Az `onSubmit` függvény kezeli az űrlap beküldését:

```
const onSubmit = (e) => {
  e.preventDefault();
  backendMuveletRole(formData, method, url, header);
  navigate("/roles");
};
```



3. Űrlap mezők kezelése

A `writeData` függvény frissíti az űrlap mezőinek értékeit:

```
const writeData = (e) => {
  setFormData((prevState) => ({
    ...prevState,
    [e.target.id]: e.target.value,
  }));
};
```

Felhasználói Élmény

- Átlátható űrlap:**
Az űrlap egyszerű és könnyen használható, a mezők automatikusan kitöltődnek meglévő jogosultság módosítása esetén.
- Valós idejű visszajelzés:**
A felhasználó azonnali visszajelzést kap a sikeres vagy sikertelen műveletekről.
- Egyszerű navigáció:**
A művelet befejezése után a felhasználó automatikusan visszairányításra kerül a jogosultságok listájára.

Komponens Felépítése

1. Űrlap mezők

- Jogosultság értéke mező:**

```
<input className="grow placeholder-info" type="number"
  id="power" placeholder="Érték" required onChange={writeData}
  value={formData.power}/>
```

- Jogosultság neve mező**

```
<input className="grow placeholder-info" type="text" id="warrant_name"
  placeholder="Jogosultság neve" required onChange={writeData}
  value={formData.warrant_name}/>
```



2. Gombok

- **Felvitel gomb:**

```
<button type="submit" className="btn btn-primary text-white">  
  Felvitel  
</button>
```

Összegzés

A NewRole React komponens hatékonyan kezeli az új jogosultságok létrehozását és a meglévő jogosultságok módosítását. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a jogosultságkezelési folyamatokba.



36. ábra New Role Panel



React Controller Dokumentáció – RoleCard.jsx

A RoleCard React komponens egy jogosultság adatait jeleníti meg kártya formájában. Ez a komponens lehetővé teszi a jogosultság módosítását és törlését, ha a felhasználó adminisztrátori jogosultsággal rendelkezik.

Funkciók

1. Jogosultság megjelenítése

- **Leírás:**

A komponens megjeleníti a jogosultság nevét (warrant_name) egy kártyán.

2. Jogosultság módosítása

- **Leírás:**

A "Módosítás" gombra kattintva a felhasználó navigálhat a jogosultság módosítására szolgáló oldalra.

- **Függvény:**

modosit

3. Jogosultság törlése

- **Leírás:**

A "Törlés" gombra kattintva a jogosultság törölhető a rendszerből. A törlés egy DELETE API-hívással történik.

- **Függvény:**

torles



API-hívások

1. Jogosultság törlése

- **HTTP-módszer:** `DELETE`
- **Végpont:** `/role/delete`
- **Küldött adatok (fejlécben):**

```
{  
  "Content-type": "application/json",  
  "Authorization": "Bearer <usertoken>",  
  "RoleId": 1  
}
```

- **Sikeres válasz:**

```
{  
  "message": "Jogosultság sikeresen törölve!"  
}
```

- **Hibás válasz:**

```
{  
  "error": "Nem sikerült a jogosultság törlése."  
}
```

Adatok kezelése

1. Jogosultság megjelenítése

A komponens a `role` prop segítségével kapja meg a jogosultság adatait, és megjeleníti annak nevét:

```
<h2 className="card-title justify-center text-primary">{role.warrant_name}</h2>
```



2. Jogosultság módosítása

A modosit függvény navigál a jogosultság módosítására szolgáló oldalra:

```
const modosit = () => {
  navigate("/newrole", { state: { role } });
};
```

3. Jogosultság törlése

A torles függvény meghívásával a komponens törli a jogosultságot:

```
const torles = (role) => {
  backendMuveletRole(
    role,
    "DELETE",
    `${import.meta.env.VITE_BASE_URL}/role/delete`,
    {
      "Content-type": "application/json",
      "Authorization": `Bearer ${sessionStorage.getItem("usertoken")}`,
      "RoleId": role.id
    }
  );
};
```

Felhasználói Élmény

- Átlátható megjelenítés:**
A jogosultság neve jól láthatóan jelenik meg egy kártyán.
 - Interaktív funkciók:**
Az adminisztrátori jogosultsággal rendelkező felhasználók módosíthatják vagy törölhetik a jogosultságot.
 - Biztonságos hozzáférés:**
A komponens ellenőrzi a felhasználó jogosultságát, mielőtt engedélyezné a módosítást vagy törlést.
-



Komponens Felépítése

1. Kártya fejléc

A kártya fejlécében a jogosultság neve jelenik meg:

```
<h2 className="card-title justify-center text-primary">{role.warrant_name}</h2>
```

2. Módosítás gomb

A "Módosítás" gomb:

```
{user.isAdmin >= 70 && (
  <button className="btn btn-primary text-white" onClick={() => modosit(role)}>
    Módosítás
  </button>
)}
```

3. Törlés gomb

A "Törlés" gomb:

```
{user.isAdmin >= 70 && (
  <button className="btn btn-info text-white" onClick={() => torles(role)}>
    Törlés
  </button>
)}
```

Összegzés

A RoleCard React komponens hatékonyan kezeli a jogosultságok megjelenítését, módosítását és törlését. Az API-hívások és az állapotkezelés jól strukturált, a felhasználói élmény pedig egyszerű és intuitív. A komponens biztosítja a RESTful architektúra elveinek betartását, és könnyen integrálható a jogosultságkezelési folyamatokba.



React Controller Dokumentáció – RolesList.jsx

A RolesList React komponens a jogosultságok listájának megjelenítésére szolgál. Ez a komponens a AuthContext-ből kapja a jogosultságok adatait, és azokat kártyák formájában jeleníti meg a RoleCard komponens segítségével.

Funkciók

1. Jogosultságok listázása

- **Leírás:**

A komponens a AuthContext-ből kapott jogosultságokat jeleníti meg kártyák formájában.

Adatok kezelése

1. Jogosultságok betöltése

A AuthContext biztosítja a jogosultságok adatait:

```
const { roles } = useContext(AuthContext);
```

2. Jogosultságok megjelenítése

A roles tömb elemei a RoleCard komponens segítségével jelennek meg:

```
<div className="flex flex-row flex-wrap items-center justify-center">
  {roles.map((role) => (
    <RoleCard key={role.id} role={role} />
  ))}
</div>
```



Felhasználói Élmény

- **Átlátható megjelenítés:**
A jogosultságok kártyák formájában jelennek meg, amelyek könnyen áttekinthetők.
 - **Interaktív funkciók:**
A kártyákon keresztül a felhasználó módosíthatja vagy törölheti a jogosultságokat (ha adminisztrátori jogosultsággal rendelkezik).
-

Komponens Felépítése

1. Cím

A lista címe:

```
<h1 className="text-3xl font-bold text-center mb-4 text-primary">
  Jogosultságok listája
</h1>
```

2. Jogosultságok kártyái

A jogosultságok kártyák formájában jelennek meg:

```
<div className="flex flex-row flex-wrap items-center justify-center">
  {roles.map((role) => (
    <RoleCard key={role.id} role={role} />
  )))
</div>
```



Összegzés

A RolesList React komponens hatékonyan kezeli a jogosultságok megjelenítését. Az adatok jól strukturáltan jelennek meg, és a RoleCard komponens segítségével interaktív funkciókat is biztosít. A komponens könnyen integrálható a jogosultságkezelési folyamatokba, és reszponzív kialakításának köszönhetően minden eszközön jól használható.



37. ábra Role List



React Controller Dokumentáció – FooterInfoUse.jsx

A [FooterInfoUse](#) React komponens egy modális ablakot jelenít meg, amely a felhasználási feltételeket ([TermOfUse](#) komponens) tartalmazza. Ez a komponens egyszerűen használható a felhasználási feltételek megtekintésére.

Funkciók

1. Felhasználási feltételek megjelenítése

- **Leírás:**

A komponens megjeleníti a [TermOfUse](#) komponenst egy modális ablakban.

2. Modális ablak bezárása

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva bezárhatja a modális ablakot.

- **Függvény:**

[closeFunction](#)

Adatok kezelése

1. Felhasználási feltételek

A [TermOfUse](#) komponens a modális ablak tartalmát biztosítja:

<TermOfUse />



2. Modális ablak bezárása

A "Bezárás" gomb meghívja a `closeFunction` függvényt, amelyet a szülő komponens biztosít:

```
<button  
  className="btn btn-info btn-circle btn-ghost absolute right-2 top-2 font-bold text-primary text-2xl" onClick={closeFunction}>  
  X  
</button>
```

Felhasználói Élmény

- Egyszerű megjelenítés:**
A felhasználási feltételek egy modális ablakban jelennek meg, amely könnyen olvasható és kezelhető.
- Interaktív funkciók:**
A felhasználó egyetlen gombnyomással bezárhatja a modális ablakot.
- Rezonans kialakítás:**
A modális ablak mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Modális tartalom

A modális tartalma a `TermOfUse` komponens:

```
<TermOfUse />
```

2. Bezárás gomb

A "Bezárás" gomb:

```
<button  
  className="btn btn-info btn-circle btn-ghost absolute right-2 top-2 font-bold text-primary text-2xl" onClick={closeFunction}>  
  X  
</button>
```



Összegzés

A [FooterInfoUse](#) React komponens egyszerű és hatékony módot biztosít a felhasználási feltételek megjelenítésére egy modális ablakban. A komponens könnyen integrálható más felhasználói felületekbe, és intuitív felhasználói élményt nyújt.

Felhasználási Feltételek – FlexiStore X

Hatályos: 2025.04.08.

Üdvözölünk a FlexiStore-nál! A FlexiStore egy olyan szolgáltatás, amely automatákból történő termékbérletet tesz lehetővé. Jelenleg a platform fejlesztési szakaszban van, és a szolgáltatás funkciói még változhatnak.

1. Általános rendelkezések

- A FlexiStore szolgáltatás használatával elfogadod az alábbi felhasználási feltételeket.
- A feltételek ideiglenesek, és a szolgáltatás indulásával frissülhetnek.

2. A szolgáltatás célja

- A FlexiStore célja, hogy automatizált eszközökön keresztül elérhetővé tegye különféle termékek rövid távú bérletét.
- A végleges működéshez fizikai automaták és digitális platform (mobilapp vagy weboldal) szükséges, ezek még fejlesztés alatt állnak.

3. Használat és jogosultság

- A rendszer jelenleg tesztverzióban fut, így tényleges bérlet nem történik.
- A jövőben a szolgáltatás igénybevételéhez regisztráció és személyes adatok megadása szükséges lehet.

4. Felelősség kizárása

- A jelen tesztoldal információi kizárolag tájékoztató jellegűek.
- A FlexiStore nem vállal felelősséget semmilyen kárért, amely a tesztplatform használatából ered.

38. ábra Term of use modal



React Controller Dokumentáció – FooterPrivacyInfo.jsx

A FooterPrivacyInfo React komponens egy modális ablakot jelenít meg, amely az adatvédelmi irányelveket ([PrivacyPolicy](#) komponens) tartalmazza. Ez a komponens egyszerűen használható az adatvédelmi irányelvek megtekintésére.

Funkciók

1. Adatvédelmi irányelvek megjelenítése

- **Leírás:**

A komponens megjeleníti a [PrivacyPolicy](#) komponenst egy modális ablakban.

2. Modális ablak bezárása

- **Leírás:**

A felhasználó a "Bezárás" gombra kattintva bezárhatja a modális ablakot.

- **Függvény:**

`closeFunction`

Adatok kezelése

1. Adatvédelmi irányelvek

A [PrivacyPolicy](#) komponens a modális ablak tartalmát biztosítja:

`<PrivacyPolicy />`

2. Modális ablak bezárása

A "Bezárás" gomb meghívja a [closeFunction](#) függvényt, amelyet a szülő komponens biztosít:

```
<button
  className="btn btn-info btn-circle btn-ghost absolute right-2 top-2 font-bold text-primary text-2xl"
  onClick={closeFunction}>
  X
</button>
```



Felhasználói Élmény

- Egyszerű megjelenítés:**

Az adatvédelmi irányelvek egy modális ablakban jelennek meg, amely könnyen olvasható és kezelhető.

- Interaktív funkciók:**

A felhasználó egyetlen gombnyomással bezárhatja a modális ablakot.

- Részponzív kialakítás:**

A modális ablak mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Modális tartalom

A modális tartalma a PrivacyPolicy komponens:

```
<PrivacyPolicy />
```

2. Bezárás gomb

A "Bezárás" gomb:

```
<button  
  className="btn btn-info btn-circle btn-ghost absolute right-2 top-2 font-bold text-primary text-2xl"  
  onClick={closeFunction}>  
  X  
</button>
```



Összegzés

A FooterPrivacyInfo React komponens egyszerű és hatékony módot biztosít az adatvédelmi irányelvek megjelenítésére egy modális ablakban. A komponens könnyen integrálható más felhasználói felületekbe, és intuitív felhasználói élményt nyújt.

Adatvédelmi Irányelvez – FlexiStore X

Hatályos: 2025.04.08.

A FlexiStore számára fontos a felhasználói adatok védelme. Jelen dokumentum ideiglenes irányelveket tartalmaz, melyek a tesztelési fázis során érvényesek.

1. Milyen adatokat gyűjtünk?

Jelenleg a tesztoldalon keresztül nem gyűjtünk személyes adatokat. A végleges szolgáltatás során azonban a következő típusú adatgyűjtés lehetséges:

- Név, e-mail cím, telefonszám
- Helyadatok (pl. automata elhelyezkedése)
- Bérleti előzmények
- Fizetési adatok (pl. kártyaadatok – ezeket kizárolag biztonságos fizetési szolgáltatón keresztül)

2. Mire használjuk az adatokat?

- A bérleti folyamat biztosítására
- Ügyfélszolgálat és kapcsolattartás céljára
- A szolgáltatás fejlesztésére

3. Adatok megosztása

Az adatokat harmadik félnek nem adjuk el. Kizárolag az alábbi esetekben történhet megosztás:

- Törvény által előírt esetek
- Technikai partnerekkel, akik segítenek a szolgáltatás működtetésében

4. Adatbiztonság

39. ábra Privacy Policy modal



React Controller Dokumentáció – PrivacyPolicy.jsx

A [PrivacyPolicy](#) React komponens az adatvédelmi irányelvek megjelenítésére szolgál. Ez a komponens részletes információkat nyújt a felhasználóknak az adatgyűjtésről, adatkezelésről és a tesztelési fázis sajátosságairól.

Funkciók

1. Adatvédelmi irányelvek megjelenítése

- **Leírás:**

A komponens szöveges formában jeleníti meg az adatvédelmi irányelvezet, beleértve az adatgyűjtés típusait, céljait, az adatok megosztását, biztonságát és a felhasználói jogokat.

Adatok kezelése

1. Adatvédelmi irányelvek tartalma

A komponens statikus szöveges tartalmat jelenít meg, amely az adatvédelmi irányelvez különböző szakaszait tartalmazza:

- **Adatgyűjtés típusai:**

- Név, e-mail cím, telefonszám
- Helyadatok (pl. automata elhelyezkedése)
- Bérleti előzmények
- Fizetési adatok (pl. kártyaadatok – ezeket kizárolag biztonságos fizetési szolgáltatón keresztül)

- **Adatok felhasználása:**

- A bérleti folyamat biztosítására
- Ügyfélszolgálat és kapcsolattartás céljára
- A szolgáltatás fejlesztésére



- **Adatbiztonság:**

<p>Az adatvédelmet komolyan vesszük, és minden ésszerű technikai intézkedést megteszünk annak érdekében, hogy az adatok biztonságban legyenek.**</p>**

2. Tesztelési fázis sajátosságai

A komponens kiemeli, hogy a jelenlegi verzió tesztelési fázisban van:

****Egyes felhasználói adatok technikai okokból az adatbázisba kerülhetnek, de ezek nem kerülnek feldolgozásra vagy elemzésre****
****A „kosár” funkció csak teszt célokat szolgál, a benne szereplő fizetési lehetőségek mögött nincs valós tranzakció****

Felhasználói Élmény

- **Átlátható megjelenítés:**

Az adatvédelmi irányelvek jól strukturáltan, szakaszokra bontva jelennek meg, így könnyen olvashatók.

- **Reszponzív kialakítás:**

A komponens mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Fejléc

Az adatvédelmi irányelvek címe és hatálybalépési dátuma:

<h1 className="font-bold text-lg text-primary">
Adatvédelmi Irányelvek – FlexiStore**</h1>**
<p className="py-4 text-info font-bold">Hatályos: 2025.04.08.**</p>**



2. Adatvédelmi szakaszok

A különböző szakaszok címei és tartalma:

```
<h3 className="font-semibold text-primary">1. Milyen adatokat gyűjtünk?</h3>
<p className="py-4 text-info">
  <li>Név, e-mail cím, telefonszám</li>
  <li>Helyadatok (pl. automata elhelyezkedése)</li>
  <li>Bérleti előzmények</li>
</p>
```

3. Tesztelési fázis

A tesztelési fázis sajátosságainak kiemelése:

```
<h3 className="font-semibold text-primary">6. Tesztelési fázis sajátosságai</h3>
<p className="py-4 text-info">
  <li>Egyes felhasználói adatok technikai okokból az adatbázisba kerülhetnek, de ezek nem kerülnek feldolgozásra vagy elemzésre</li>
  <li>A „kosár” funkció csak teszt célokra szolgál, a benne szereplő fizetési lehetőségek mögött nincs valós tranzakció</li>
</p>
```

Összegzés

A [PrivacyPolicy](#) React komponens részletes és jól strukturált módon jeleníti meg az adatvédelmi irányelveket. A komponens könnyen integrálható más felhasználói felületekbe, és biztosítja, hogy a felhasználók tisztában legyenek az adatkezelési gyakorlatokkal. A tesztelési fázis sajátosságainak kiemelése különösen hasznos a jelenlegi fejlesztési szakaszban.



React Controller Dokumentáció – TermOfUse.jsx

A [TermOfUse](#) React komponens a felhasználási feltételek megjelenítésére szolgál. Ez a komponens részletes információkat nyújt a felhasználóknak a szolgáltatás használatának szabályairól, céljáról, felelősségvállalásról és a feltételek módosításáról.

Funkciók

1. Felhasználási feltételek megjelenítése

- **Leírás:**

A komponens szöveges formában jeleníti meg a felhasználási feltételeket, beleértve az általános rendelkezéseket, a szolgáltatás célját, a használati jogosultságokat, a felelősség kizárást és a feltételek módosításának lehetőségét.

Adatok kezelése

1. Felhasználási feltételek tartalma

A komponens statikus szöveges tartalmat jelenít meg, amely a felhasználási feltételek különböző szakaszait tartalmazza:

- **Általános rendelkezések:**

****A FlexiStore szolgáltatás használatával elfogadod az alábbi felhasználási feltételeket.****
****A feltételek ideiglenesek, és a szolgáltatás indulásával frissülhetnek.****

- **A szolgáltatás célja:**

****A FlexiStore célja, hogy automatizált eszközökön keresztül elérhetővé tegye különféle termékek rövid távú bérletét.****
****A végleges működéshez fizikai automaták és digitális platform ([mobilapp](#) vagy [weboldal](#)) szükséges, ezek még fejlesztés alatt állnak.****



- **Használat és jogosultság:**

A rendszer jelenleg tesztverzióban fut, így tényleges bérlet nem történik.
A jövőben a szolgáltatás igénybevételéhez regisztráció és személyes adatok megadása szükséges lehet.

- **Felelősség kizárása:**

A jelen tesztoldal információi kizárolag tájékoztató jellegűek.
A FlexiStore nem vállal felelősséget semmilyen kárért, amely a tesztplatform használatából ered.

- **Módosítás:**

A feltételeket bármikor módosíthatjuk. A változásokról értesítést teszünk közzé ezen az oldalon.

Felhasználói Élmény

- **Átlátható megjelenítés:**

A felhasználási feltételek jól strukturáltan, szakaszokra bontva jelennek meg, így könnyen olvashatók.

- **Rezonansív kialakítás:**

A komponens mobil- és asztali eszközökön egyaránt jól használható.

Komponens Felépítése

1. Fejléc

A felhasználási feltételek címe és hatállyalépési dátuma:

<h1 className="font-bold text-lg text-primary">Felhasználási Feltételek – FlexiStore</h1>
<p className="py-4 text-info font-bold">Hatályos: 2025.04.08.</p>



2. Felhasználási feltételek szakaszai

A különböző szakaszok címei és tartalma

```
<h3 className="font-semibold text-primary">1. Általános rendelkezések</h3>
<p className="py-4 text-info">
  <li>A FlexiStore szolgáltatás használatával elfogadod az alábbi felhasználási
  feltételeket.</li>
  <li>A feltételek ideiglenesek, és a szolgáltatás indulásával frissülhetnek.</li>
</p>
```

3. Felelősség kizárása

A felelősség kizárásnak kiemelése:

```
<h3 className="font-semibold text-primary">4. Felelősség kizárása</h3>
<p className="py-4 text-info">
  <li>A jelen tesztoldal információi kizárolag tájékoztató jellegűek.</li>
  <li>A FlexiStore nem vállal felelősséget semmilyen kárért, amely a tesztplatform
  használatából ered.</li>
</p>
```

Összegzés

A [TermOfUse](#) React komponens részletes és jól strukturált módon jeleníti meg a felhasználási feltételeket. A komponens könnyen integrálható más felhasználói felületekbe, és biztosítja, hogy a felhasználók tisztában legyenek a szolgáltatás használatának szabályaival. A tesztelési fázis sajátosságainak kiemelése különösen hasznos a jelenlegi fejlesztési szakaszban.



React Controller Dokumentáció – Footer.jsx

A Footer React komponens a weboldal láblécét valósítja meg, amely tartalmazza a navigációs linkeket, jogi információkat, felhasználói és adminisztrátori menüket, valamint a kosár és információs ikonokat. A komponens reszponzív kialakítású, és különböző funkciókat biztosít mobil- és asztali nézetekhez.

Funkciók

1. Felhasználási feltételek és adatvédelmi irányelvek megjelenítése

- **Leírás:**
A felhasználó megtekintheti a felhasználási feltételeket és az adatvédelmi irányelveket modális ablakban.
- **Függvények:**
 - openTermInfo: A felhasználási feltételek modális megnyitása.
 - closeInfo: A felhasználási feltételek modális bezárása.
 - openPolicyInfo: Az adatvédelmi irányelvek modális megnyitása.
 - closePolicy: Az adatvédelmi irányelvek modális bezárása.

2. Felhasználói menü

- **Leírás:**
A felhasználó navigálhat a csomagautomatákhoz, termékekhez, vagy a saját felhasználói irányítópultjához.
- **Függvények:**
 - setMenuOpen: A felhasználói menü nyitott vagy zárt állapotának kezelése.

3. Adminisztrátori menü

- **Leírás:**
Az adminisztrátor hozzáférhet az adminisztrációs funkciókhoz, például új termék, kategória vagy jogosultság létrehozásához.



- **Függvények:**
 - setMenuAdminOpen: Az adminisztrátori menü nyitott vagy zárt állapotának kezelése.

4. Kosár megjelenítése

- **Leírás:**

A kosár ikon megjeleníti a kosárban lévő termékek számát és összértékét.

5. Kijelentkezés

- **Leírás:**

A felhasználó kijelentkezhet a rendszerből.
- **Függvény:**
 - handleLogout: A kijelentkezési folyamat kezelése.

Adatok kezelése

1. Felhasználói és adminisztrátori állapot

A komponens a secureStorage és a AuthContext segítségével kezeli a felhasználói adatokat:

```
const user = secureStorage.getItem("user");
const { logout, update, profile } = useContext(AuthContext);
```

2. Kosár adatok

A CartContext biztosítja a kosárban lévő termékek számát és összértékét:

```
const { cartItems, getCartTotal } = useContext(CartContext);
```

3. Modális állapotok

A modális ablakok nyitott vagy zárt állapotát a isTermInfo és isPolicyInfo állapotok kezelik:

```
const [isTermInfo, setTermInfo] = useState(false);
const [isPolicyInfo, setPolicyInfo] = useState(false);
```



Felhasználói Élmény

- Reszponzív kialakítás:**
A lábléc mobil- és asztali nézetekhez igazodik, különböző elrendezésekkel.
- Interaktív funkciók:**
A felhasználók könnyen navigálhatnak az oldalon, megtekinthetik a jogi információkat, és kezelhetik a kosarukat.
- Adminisztrátori hozzáférés:**
Az adminisztrátorok számára külön menü biztosítja a speciális funkciók elérését.

Komponens Felépítése

1. Felhasználási feltételek és adatvédelmi irányelvek modálisok

A modálisok megjelenítése:

```
{isTermInfo && (
  <FooterInfoUse closeFunction={closeInfo} />
)
{isPolicyInfo && (
  <FooterPrivacyInfo closeFunction={closePolicy} />
)}
```

2. Navigációs linkek (asztali nézet)

A navigációs linkek:

```
<nav className="min-w-[150px]">
  <h6 className="footer-title text-secondary">Szolgáltatások</h6>
  <Link to="/lockers" className="link link-hover text-info">Csomagautomaták</Link>
  <Link to="/products" className="link link-hover text-info">Összes Termék</Link>
</nav>
```



3. Kosár ikon

A kosár ikon és tartalma:

```
<Link to="/cart" className="flex flex-col items-center text-primary text-xs">
  <div className="dropdown dropdown-end">
    <div tabIndex={0} role="button" className="btn btn-ghost btn-circle">
      <div className="indicator">
        <svg className="w-6 h-6" fill="none" stroke="#50c6c9" strokeWidth="2" viewBox="0 0 24 24">
          <path d="M6.29977 5H21L19 12H7.37671M20 16H8L6 3H3M9 20C9 20.5523 8.55228 21 8 21C7.44772 21 7 20.5523 7 20C7 19.4477 7.44772 19 8 19C8.55228 19 9 19.4477 9 20ZM20 20C20 20.5523 19.5523 21 19 21C18.4477 21 18 20.5523 18 20C18 19.4477 18.4477 19 19 19C19.5523 19 20 19.4477 20 20Z" />
        </svg>
        <span className="badge badge-sm indicator-item">{user ? cartItems?.length || 0 : 0}</span>
      </div>
    </div>
  </div>
</Link>
```

4. Adminisztrátori menü

Az adminisztrátori menü:

```
{user?.isadmin >= 70 && (
  <button onClick={() => setMenuAdminOpen(true)} className="flex flex-col items-center text-primary text-xs">
    <svg fill="none" width="28" height="28" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg">
      <circle cx="12" cy="12" r="3" className="stroke-[#50c6c9] fill-none" strokeLinecap="round" strokeLinejoin="round" strokeWidth="2" />
      <path d="M7.94,20a1,1,0,0,1-.87-.51l-3.94-7a1,1,0,0,1,0-113.94-7a1,1,0,0,1,7.94,4h8.12a1,1,0,0,1,.87.51l3.94,7a1,1,0,0,1,0,11-3.94,7a1,1,0,0,1,.87.51Z" className="stroke-[#005c6a] fill-none" strokeLinecap="round" strokeLinejoin="round" strokeWidth="2" />
    </svg>
  </button>
)}
```



Összegzés

A Footer React komponens egy komplex, reszponzív láblécet valósít meg, amely számos funkciót biztosít a felhasználók és adminisztrátorok számára. A komponens jól strukturált, könnyen bővíthető, és intuitív felhasználói élményt nyújt.



40. ábra Footer Desktop



41. ábra Footer Mobile



React Controller Dokumentáció – Main.jsx

A Main React komponens a FlexiStore demó kezdőoldalát valósítja meg. Ez a komponens bemutatja a platformot, tartalmazza a navigációs linkekét, és egy illusztrációval egészíti ki a felhasználói élményt.

Funkciók

1. Üdvözlő szöveg megjelenítése

- **Leírás:**

A komponens bemutatja a FlexiStore demót, kiemelve a platform célját és készítőit.

2. Navigációs linkek

- **Leírás:**

A felhasználó navigálhat a csomagautomaták oldalára, vagy regisztrálhat, ha még nem bejelentkezett.

- **Függvények:**

- Link komponensek a react-router-dom-ból.

3. Illusztráció megjelenítése

- **Leírás:**

A komponens egy SVG illusztrációt jelenít meg, amely vizuálisan kiegészíti az oldalt.

Adatok kezelése

1. Felhasználói állapot

A komponens a AuthContext segítségével kezeli a felhasználói adatokat:

```
const { user } = useContext(AuthContext);
```



2. Feltételes megjelenítés

A regisztrációs gomb csak akkor jelenik meg, ha a felhasználó nincs bejelentkezve:

```
{!user && (  
  <Link to="/register2" className="btn btn-info text-white">Regisztráció</Link>  
)}
```

Felhasználói Élmény

- Átlátható megjelenítés:**
Az üdvözlő szöveg és a navigációs linkek jól strukturáltan jelennek meg.
- Interaktív funkciók:**
A felhasználók könnyen navigálhatnak az oldalon, és regisztrálhatnak, ha szükséges.
- Reszponzív kialakítás:**
A komponens mobil- és asztali nézetekhez igazodik, különböző elrendezésekkel.

Komponens Felépítése

1. Üdvözlő szöveg

Az üdvözlő szöveg és a készítők neve:

```
<h1 className="text-3xl sm:text-4xl md:text-5xl font-bold text-primary mb-6">  
Üdvözünk a FlexiStore demóban!  
</h1>  
<p className="text-info font-bold mb-6">  
Készítette: Szirony Balázs Gábor és Gombkötő Gábor <br />  
© minden jog fenntartva.  
</p>
```



2. Navigációs linkek

A navigációs linkek:

```
<div className="flex gap-4 mt-4 flex-wrap justify-center lg:justify-start">
  <Link to="/lockers" className="btn btn-primary text-white">Csomagautomaták</Link>
  { !user && (
    <Link to="/register2" className="btn btn-info text-white">Regisztráció</Link>
  )}
</div>
```

3. Illusztráció

Az SVG illusztráció:

```
<svg
  className="w-[60%] max-w-[155px] sm:w-[75%] sm:max-w-sm md:max-w-md lg:max-w-lg"
  viewBox="0 0 24 24"
  fill="none"
  xmlns="http://www.w3.org/2000/svg"
>
  <path d="M20.3873 7.1575L11.9999 12L3.60913 7.14978" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
  <path d="M12 12V21" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
  <path d="M11 2.57735C11.6188 2.22008 12.3812 2.22008 13 2.57735L19.6603 6.42265C20.2791 6.77992 20.6603 7.44017 20.6603 8.1547V15.8453C20.6603 16.5598 20.2791 17.201 19.6603 17.5774L13 21.4226C12.3812 21.7799 11.6188 21.7799 11 21.4226L4.33975 17.5774C3.72094 17.2201 3.33975 16.5598 3.33975 15.8453V8.1547C3.33975 7.44017 3.72094 6.77992 4.33975 6.42265L11 2.57735Z" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
  <path d="M8.5 4.5L16 9" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
</svg>
```



Összegzés

A Main React komponens egy jól strukturált kezdőoldalt biztosít a FlexiStore demó számára. A komponens reszponzív kialakítású, interaktív funkciókat kínál, és vizuálisan is vonzó. Könnyen bővíthető és testreszabható, hogy megfeleljen a további fejlesztési igényeknek.



42. ábra Main Dekstop



43. ábra Main Mobile



React Controller Dokumentáció – Menu.jsx

A [Menu](#) React komponens a FlexiStore navigációs menüpontját valósítja meg. Ez a komponens tartalmazza a logót, navigációs linkekét, kosár ikont, felhasználói és adminisztrátori menüket, valamint a profilkezelési lehetőségeket. A komponens reszponzív kialakítású, és különböző funkciókat biztosít mobil- és asztali nézetekhez.

Funkciók

1. Navigációs linkek

- **Leírás:**
A felhasználó navigálhat a csomagautomaták, termékek, vagy az adminisztrációs oldalak között.
- **Függvények:**
 - [Link](#) komponensek a react-router-dom-ból.

2. Kosár ikon

- **Leírás:**
A kosár ikon megjeleníti a kosárban lévő termékek számát és összértékét. A felhasználó a kosár megtekintésére navigálhat.
- **Függvények:**
 - [cartItems](#) és [getCartTotal](#) a [CartContext](#)-ből.

3. Felhasználói menü

- **Leírás:**
A felhasználó hozzáférhet a profiljához, kijelentkezhet, vagy regisztrálhat és bejelentkezhet, ha nincs bejelentkezve.
- **Függvények:**
 - [handleLogout](#): A kijelentkezési folyamat kezelése.

4. Adminisztrátori menü



- **Leírás:**
Az adminisztrátor hozzáférhet az adminisztrációs funkciókhoz, például új termék, kategória vagy jogosultság létrehozásához.
- **Függvények:**
 - closeDropdown: Az adminisztrátori menü bezárása.

Adatok kezelése

1. Felhasználói és adminisztrátori állapot

A komponens a secureStorage és a AuthContext segítségével kezeli a felhasználói adatokat:

```
const user = secureStorage.getItem("user");
const { logout, update, profile } = useContext(AuthContext);
```

2. Kosár adatok

A CartContext biztosítja a kosárban lévő termékek számát és összértékét:

```
const { cartItems, getCartTotal } = useContext(CartContext);
```

3. Profil adatok

A profilkép és a felhasználói adatok megjelenítéséhez a profile változót használja:

```
<img
src={
  profile?.file_path
  ? `${import.meta.env.VITE_LARAVEL_IMAGE_URL}${profile.file_path}`
  : "https://img.daisyui.com/images/stock/photo-1534528741775-
53994a69daeb.webp"
alt="Profilkép"/>
```

Felhasználói Élmény

- **Rezsponzív kialakítás:**
A menü mobil- és asztali nézetekhez igazodik, különböző elrendezésekkel.



- **Interaktív funkciók:**

A felhasználók könnyen navigálhatnak az oldalon, kezelhetik a kosarukat, és hozzáférhetnek a profiljukhoz.

- **Adminisztrátori hozzáférés:**

Az adminisztrátorok számára külön menü biztosítja a speciális funkciók elérését.

Komponens Felépítése

1. Logó

A logó és a kezdőlapra navigáló link:

```
<Link to="/" className="btn btn-ghost normal-case text-xl text-primary font-bold flex items-center gap-2">
  <svg className="w-8 h-8" viewBox="0 0 24 24" fill="none">
    <path d="M20.3873 7.1575L11.9999 12L3.60913 7.14978" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
    <path d="M12 12V21" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
    <path d="M11 2.57735C11.6188 2.22008 12.3812 2.22008 13 2.57735L19.6603 6.42265C20.2791 6.77992 20.6603 7.44017 20.6603 8.1547V15.8453C20.6603 16.5598 20.2791 17.2201 19.6603 17.5774L13 21.4226C12.3812 21.7799 11.6188 21.7799 11 21.4226L4.33975 17.5774C3.72094 17.2201 3.33975 16.5598 3.33975 15.8453V8.1547C3.33975 7.44017 3.72094 6.77992 4.33975 6.42265L11 2.57735Z" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
    <path d="M8.5 4.5L16 9" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round" strokeLinejoin="round" />
  </svg>
  FlexiStore
</Link>
```



2. Kosár ikon

A kosár ikon és tartalma

```
<div className="dropdown dropdown-end">
  <div id="cart-button" tabIndex={0} role="button" className="btn btn-ghost btn-circle">
    <div className="indicator">
      <svg className="w-6 h-6" fill="none" stroke="#50c6c9" strokeWidth="2" viewBox="0 0 24 24">
        <path d="M6.29977 5H21L19 12H7.37671M20 16H8L6 3H3M9 20C9 20.5523 8.55228 21 8 21
C7.44772 21 7 20.5523 7 20C7 19.4477 7.44772 19 8 19C8.55228 19 9 19.4477 9 20ZM20 20C20 20
5523 19.5523 21 19 21C18.4477 21 18 20.5523 18 20C18 19.4477 18.4477 19 19 19C19.5523 19 20
19.4477 20 20Z" />
      </svg>
      <span className="badge badge-sm indicator-item">{user ? cartItems?.length || 0 : 0}</span>
    </div>
  </div>
  <div tabIndex={0} className="card card-compact dropdown-content bg-base-100 z-[1] mt-3 w-52 shadow">
    <div className="card-body">
      <span className="text-lg font-bold text-primary">Termék: {cartItems?.length || 0} db</span>
      <span className="text-info">Összesen: {getCartTotal()} Ft</span>
      <div className="card-actions">
        <Link to="/cart" id="cart-view" className="btn btn-primary btn-block text-white">Kosár megtekintése</Link>
      </div>
    </div>
  </div>
</div>
```

3. Adminisztrátori menü

Az adminisztrátori menü:

```
{user?.isadmin >= 70 && (
  <details className="relative">
    <summary className="text-primary font-bold">Admin Dashboard</summary>
    <ul className="bg-base-100 rounded-t-none p-2 z-50 items-start text-left absolute">
      <li><Link to="/newcategory" className="btn btn-ghost text-primary" onClick={closeDropdown}>Új Kategória</Link></li>
      <li><Link to="/newproduct" className="btn btn-ghost text-primary" onClick={closeDropdown}>Új Termék</Link></li>
      <li><Link to="/roles" className="btn btn-ghost text-primary" onClick={closeDropdown}>Jogosultságok</Link></li>
    </ul>
  </details>
)}
```



React Controller Dokumentáció – MobileTableInfo.jsx

A [MobileTableInfo](#) React komponens egy mobilbarát információs szekciót valósít meg, amely tartalmazza a jogi információkat, márkaadatokat, és közösségi média ikonokat. A komponens szisztematikus kialakítású, és különösen mobil eszközökre optimalizált.

Funkciók

1. Jogi információk megjelenítése

- **Leírás:**
A felhasználó megtekintheti a felhasználási feltételeket és az adatvédelmi irányelveket modális ablakban.
- **Függvények:**
 - [openTermInfo](#): A felhasználási feltételek modális megnyitása.
 - [closeInfo](#): A felhasználási feltételek modális bezárása.
 - [openPolicyInfo](#): Az adatvédelmi irányelvek modális megnyitása.
 - [closePolicy](#): Az adatvédelmi irányelvek modális bezárása.

2. Márkaadatok megjelenítése

- **Leírás:**
A FlexiStore logója és márkaadatai jelennek meg, beleértve a szerzői jogi információkat.

3. Közösségi média ikonok

- **Leírás:**
A felhasználó közösségi média ikonokon keresztül navigálhat az Instagram és Facebook oldalakra.



Adatok kezelése

1. Modális állapotok

A modális ablakok nyitott vagy zárt állapotát a `isTermInfo` és `isPolicyInfo` állapotok kezelik:

```
const [isTermInfo, setTermInfo] = useState(false);
const [isPolicyInfo, setPolicyInfo] = useState(false);
```

2. Modális vezérlő függvények

A modális ablakok megnyitásához és bezárásához használt függvények:

```
const openTermInfo = () => setTermInfo(true);
const closeInfo = () => setTermInfo(false);
const openPolicyInfo = () => setPolicyInfo(true);
const closePolicy = () => setPolicyInfo(false);
```

Felhasználói Élmény

- Reszponzív kialakítás:**

A komponens mobil eszközökre optimalizált, de asztali nézetben is jól használható.

- Interaktív funkciók:**

A felhasználók könnyen hozzáférhetnek a jogi információkhoz és a közösségi média linkekhez.

- Egyszerű navigáció:**

A jogi információk és közösségi média ikonok jól elkülönítve jelennek meg.



Komponens Felépítése

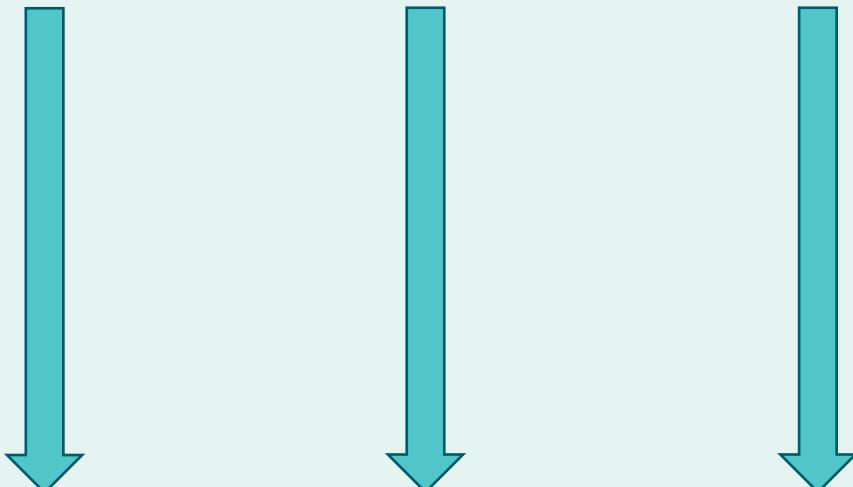
1. Jogi információk

A jogi információk linkjei és modális ablakai:

```
{isTermInfo && <FooterInfoUse closeFunction={closeInfo} />}  
{isPolicyInfo && <FooterPrivacyInfo closeFunction={closePolicy} />}  
<div className="space-y-2">  
  <h3 className="text-secondary text-lg font-semibold">Jogi információk</h3>  
  <ul>  
    <li>  
      <button className="text-info underline hover:text-info-content transition" onClick={openTermInfo}>  
        Felhasználási feltételek  
      </button>  
    </li>  
    <li>  
      <button className="text-info underline hover:text-info-content transition" onClick={openPolicyInfo}>  
        Adatvédelmi irányelvezek  
      </button>  
    </li>  
  </ul>  
</div>
```

2. Márkaadatok

A FlexiStore logója és szerzői jogi információk:





```
<div className="flex items-center gap-4 mt-24">
  <svg width="48" height="48" viewBox="0 0 24 24" fill="none"
    xmlns="http://www.w3.org/2000/svg">
    <path d="M20.3873 7.1575L11.9999 12L3.60913 7.14978" stroke="#50c6c9" strokeWidth="2"
      strokeLinecap="round" strokeLinejoin="round" />
    <path d="M12 12V21" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round"
      strokeLinejoin="round" />
    <path d="M11 2.57735C11.6188 2.22008 12.3812 2.22008 13 2.57735L19.6603 6.42265C20.2
    791 6.77992 20.6603 7.44017 20.6603 8.1547V15.8453C20.6603 16.5598 20.2791 17.2201 19.66
    03 17.5774L13 21.4226C12.3812 21.7799 11.6188 21.7799 11 21.4226L4.33975 17.5774C3.7209
    4 17.2201 3.33975 16.5598 3.33975 15.8453V8.1547C3.33975 7.44017 3.72094 6.77992 4.33975
    6.42265L11 2.57735Z" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round"
      strokeLinejoin="round" />
    <path d="M8.5 4.5L16 9" stroke="#50c6c9" strokeWidth="2" strokeLinecap="round"
      strokeLinejoin="round" />
  </svg>
  <div className="text-primary font-bold">
    <p>FlexiStore</p>
    <p className="text-sm text-gray-500">Minden jog fenntartva © 2025</p>
  </div>
</div>
```

3. Közösségi média ikonok

Az Instagram és Facebook ikonok:

```
<div className="flex gap-4">
  <a href="#" aria-label="Instagram">
    <svg width="32" height="32" fill="#50c6c9" viewBox="0 0 24 24">
      <path d="M2 6C2 3.79086 3.79086 2 6 2H18C20.2091 2 22 3.79086 22 6V18C22 20.2091 20
      .2091 22 18 22H6C3.79086 22 2 20.2091 2 18V6ZM6 4C4.89543 4 4 4.89543 4 6V18C4 19.1046
      4.89543 20 6 20H18C19.1046 20 20 19.1046 20 18V6C20 4.89543 19.1046 4 18 4H6ZM12 9C1
      0.3431 9 9 10.3431 9 12C9 13.6569 10.3431 15 12 15C13.6569 15 15 13.6569 15 12C15 10.3431
      13.6569 9 12 9ZM7 12C7 9.23858 9.23858 7 12 7C14.7614 7 17 9.23858 17 12C17 14.7614 14.
      7614 17 12 17C9.23858 17 7 14.7614 7 12ZM17.5 8C18.3284 8 19 7.32843 19 6.5C19 5.67157 1
      8.3284 5 17.5 5C16.6716 5 16 5.67157 16 6.5C16 7.32843 16.6716 8 17.5 8Z" />
    </svg>
  </a>
  <a href="#" aria-label="Facebook">
    <svg width="32" height="32" fill="#50c6c9" viewBox="0 0 32 32">
      <path d="M21.95 5.0051-3.306-.004c-3.206 0-5.277 2.124-
      5.277 5.415v2.495H10.05v4.515h3.317l-.004 9.575h4.641l.004-9.575h3.806l-.003-4.514h-
      3.803v-2.117c0-1.018.241-1.533 1.566-1.533l2.366-.001.01-4.256z" />
    </svg>
  </a>
</div>
```



FlexiBox Telepítési Útmutató

1. Projekt klónozása

Először klónozd a projektet a GitHub-ról a helyi gépedre:

```
git clone https://github.com/Szyron/FlexiBox.git  
cd FlexiBox
```

2. Függőségek telepítése

Telepítsd a szükséges PHP és JavaScript csomagokat:

```
composer install  
npm install
```

3. Környezeti változók beállítása

Másold a .env.example fájlt .env néven:

```
cp .env.example .env
```

Nyisd meg a .env fájlt egy szövegszerkesztőben, és állítsd be az adatbázis kapcsolatot:

```
DB_DATABASE=flexibox  
DB_USERNAME=your_database_username  
DB_PASSWORD=your_database_password
```

4. Alkalmazás kulcs generálása

Generálj egy új alkalmazáskulcsot:

```
php artisan key:generate
```

5. Adatbázis migrációk és seederek futtatása

Hozd létre az adatbázis táblákat és törlsd fel az alapadatokat:

```
php artisan migrate --seed
```

Ez a parancs lefuttatja a migrációkat és a seedereket, beleértve a RoleSeeder és UserSeeder osztályokat.



6. Storage link létrehozása

Hozz létre egy szimbolikus linket a képek eléréséhez:

```
php artisan storage:link
```

7. Fejlesztői környezet indítása

Indítsd el a Laravel szervert:

```
php artisan serve
```

8. Indítsd el a React fejlesztői környezetet:

```
npm run dev
```

9. Tesztelés

Nyisd meg a böngészőt, és látogasd meg az alábbi URL-eket:

Backend: <http://localhost:8000>

Frontend: <http://localhost:5173>

Fontos Megjegyzések

- Győződj meg róla, hogy a gépeden telepítve van a következő szoftverek: PHP, Composer, Node.js, NPM, MySQL.

10. Az adatbázis létrehozása előtt győződj meg róla, hogy a MySQL szerver fut, és a flexibox nevű adatbázis létezik. Ha nem, hozd létre a következő parancs segítségével:

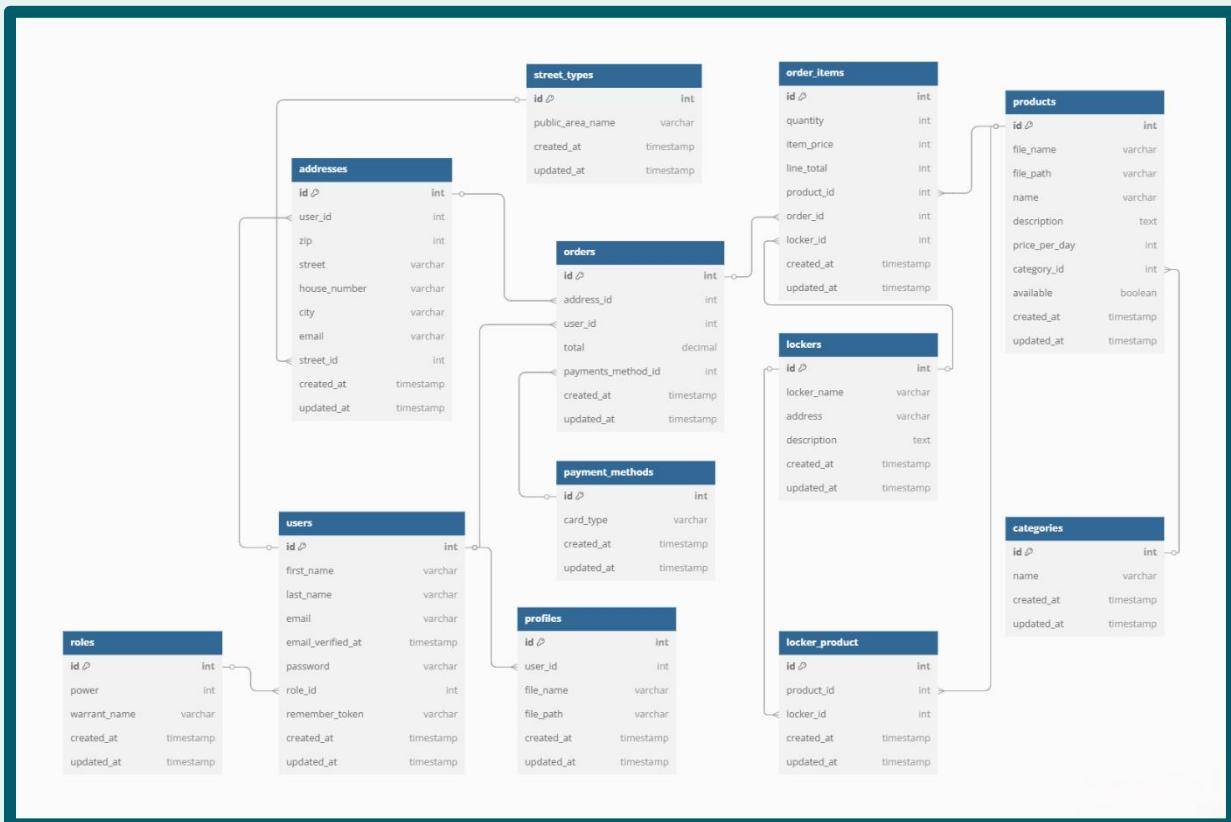
```
CREATE DATABASE flexibox;
```

Ezekkel a lépésekkel bárki könnyedén telepítheti és elindíthatja a FlexiBox projektet.



ERD Diagramok

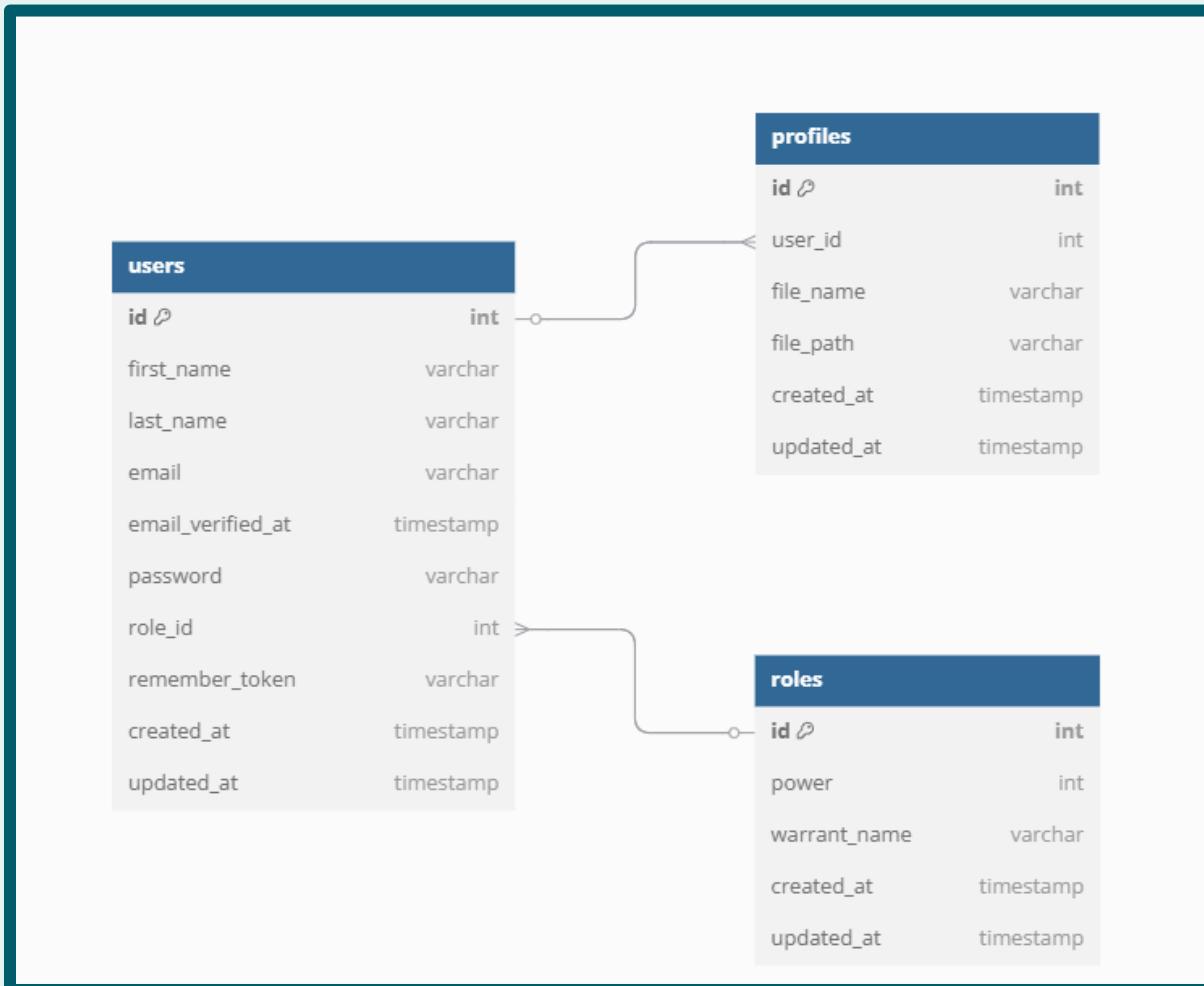
Full ERD



44. ábra Full ERD



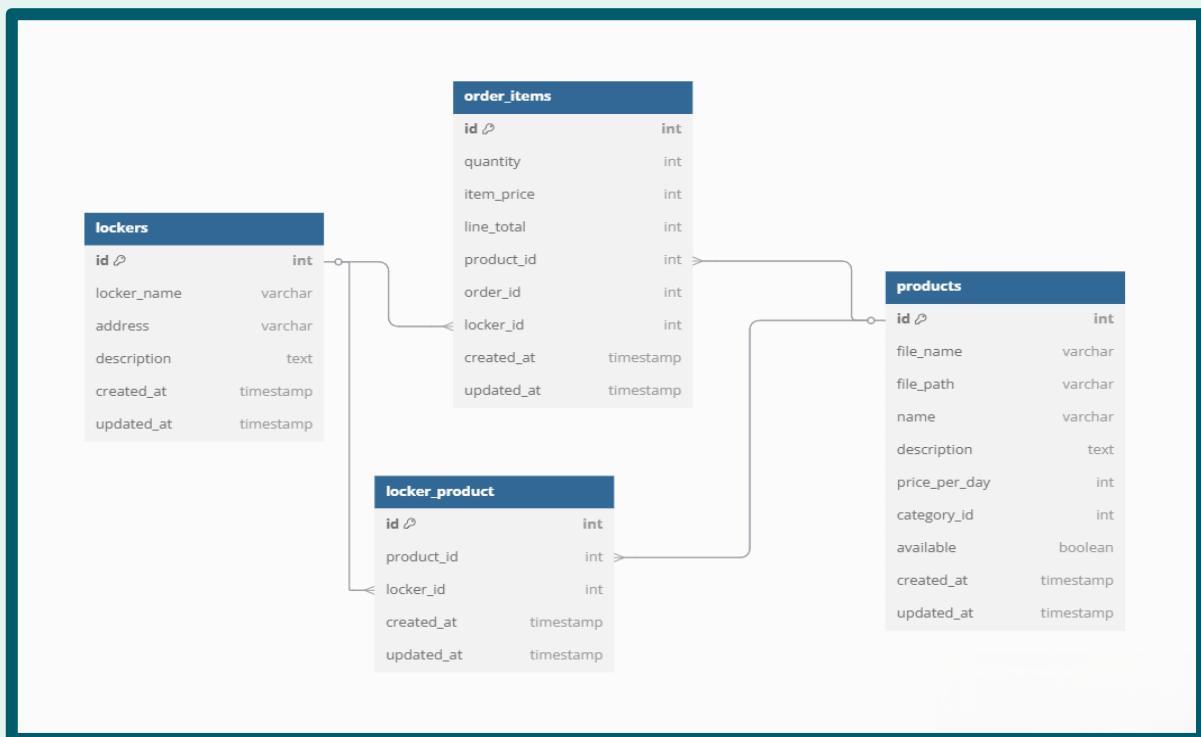
User ERD



45. ábra User ERD

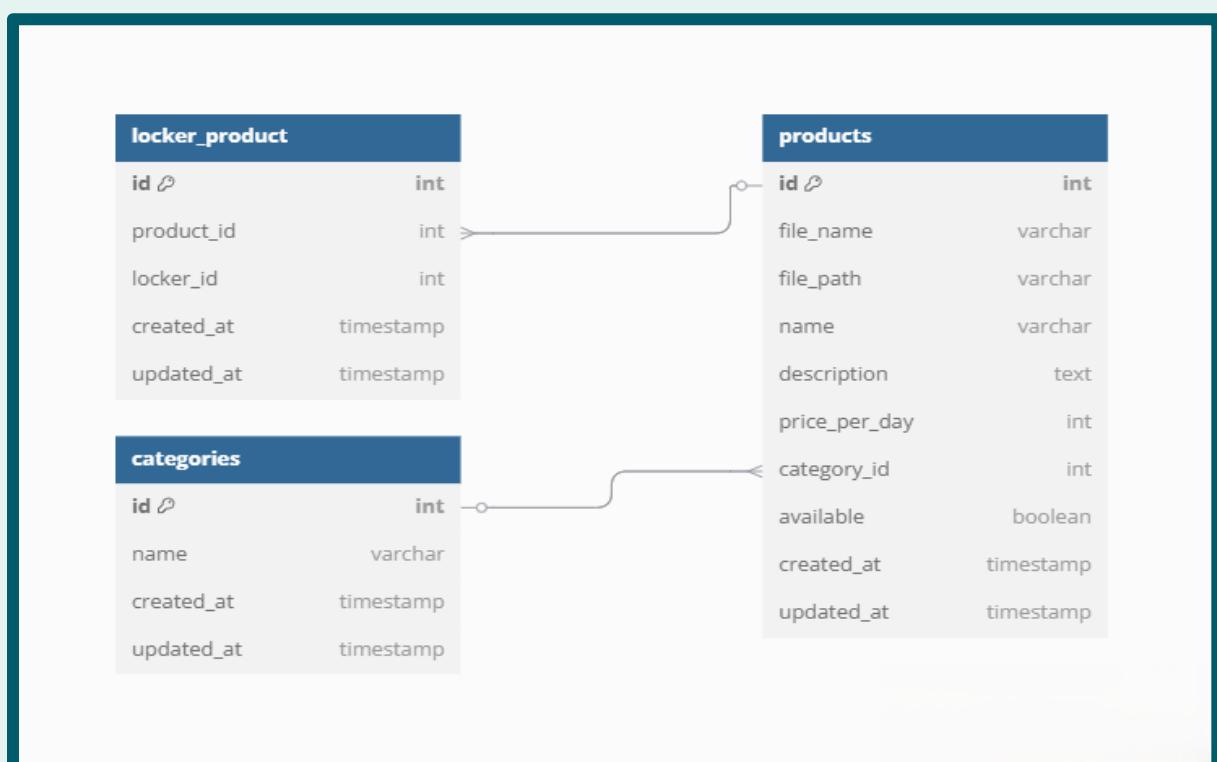


Order ERD



46. ábra Order ERD

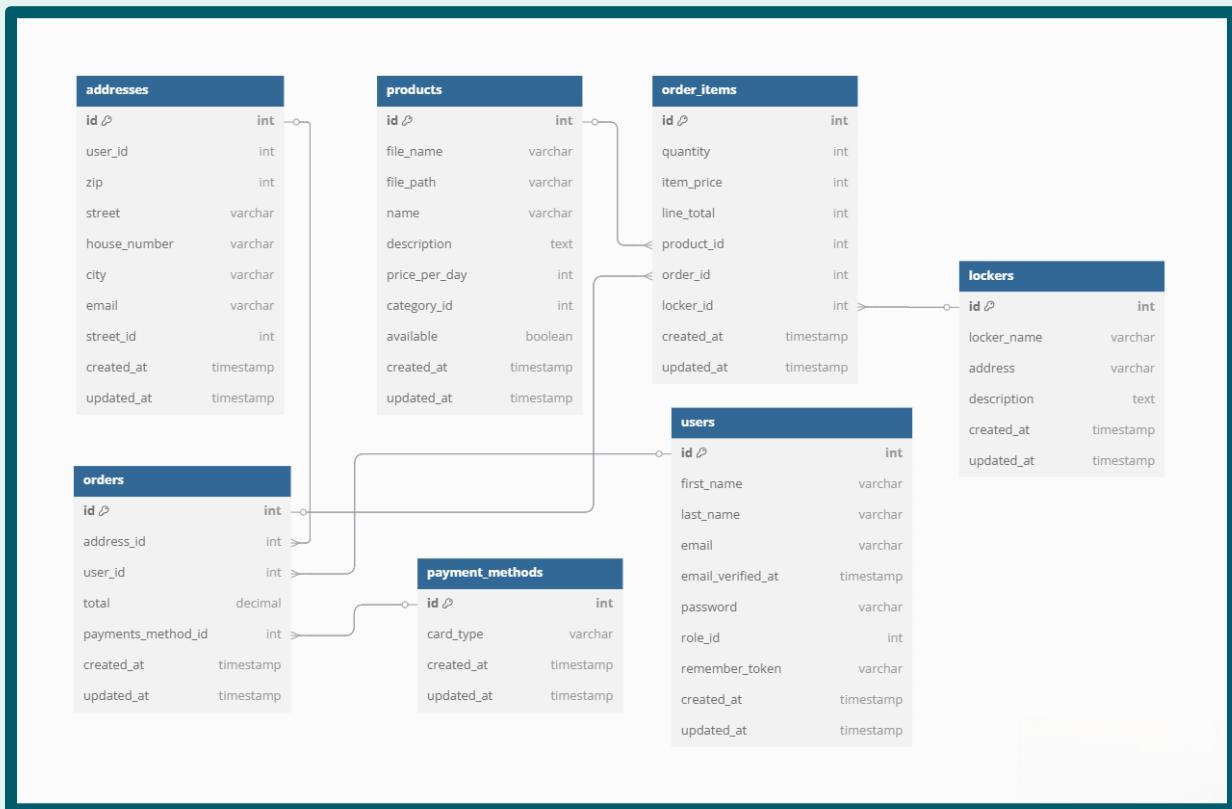
Product ERD



47. ábra Product ERD



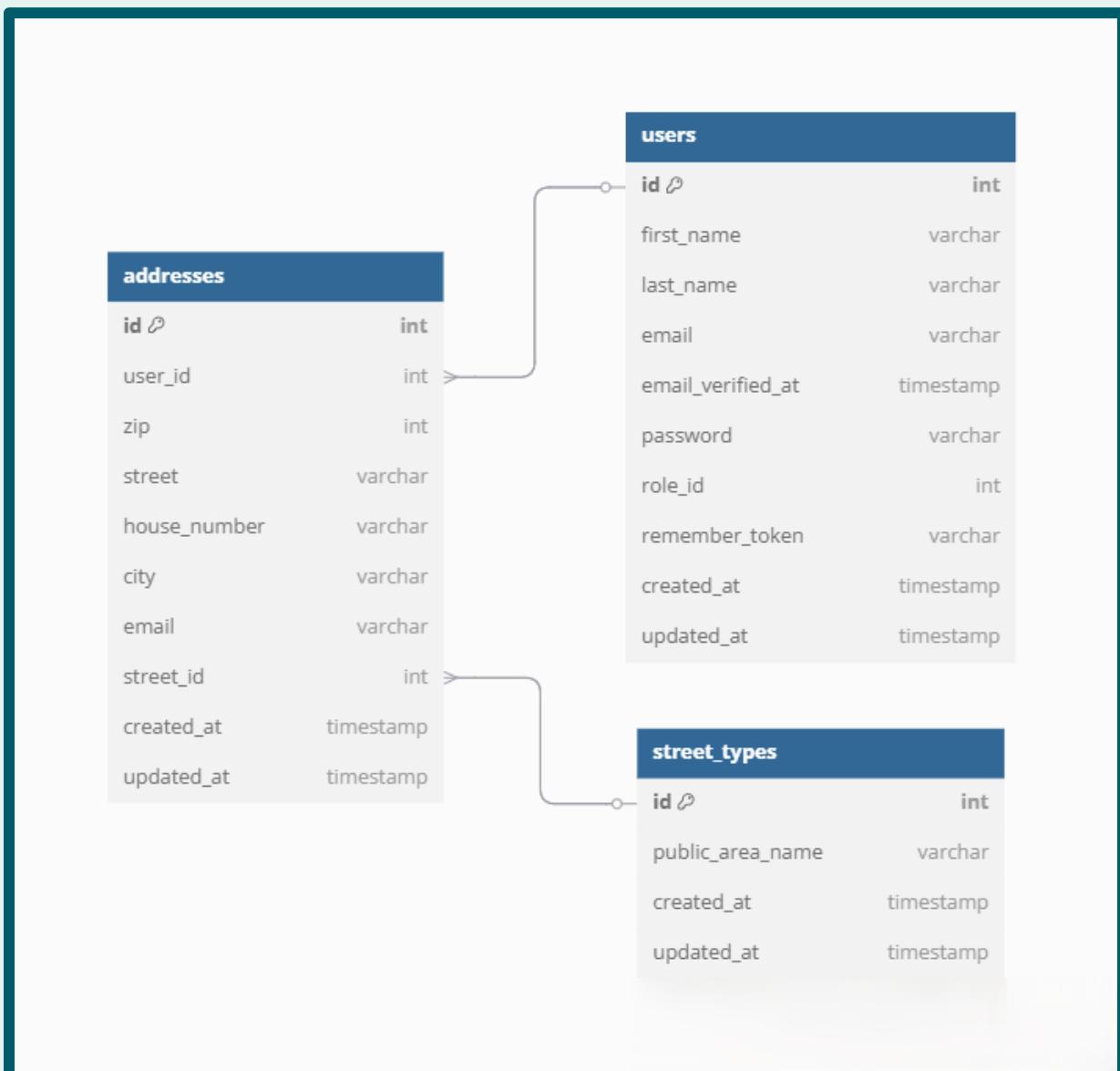
Orders ERD



48. ábra Orders ERD



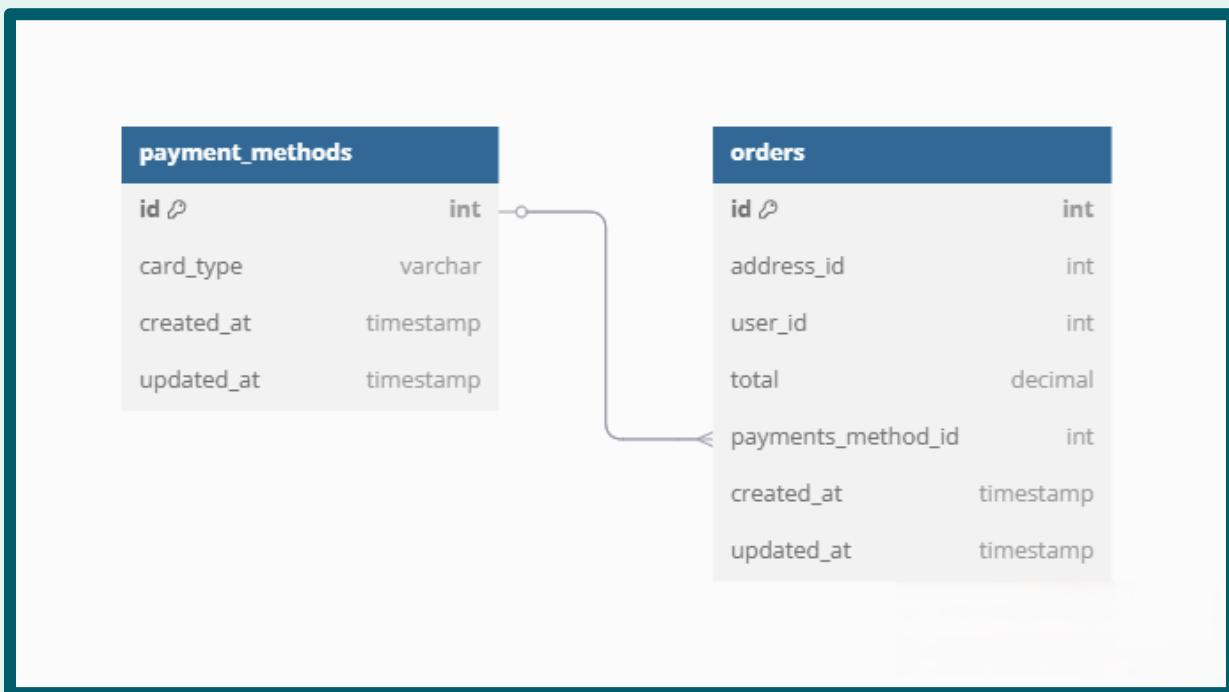
Address ERD



49. ábra Address ERD



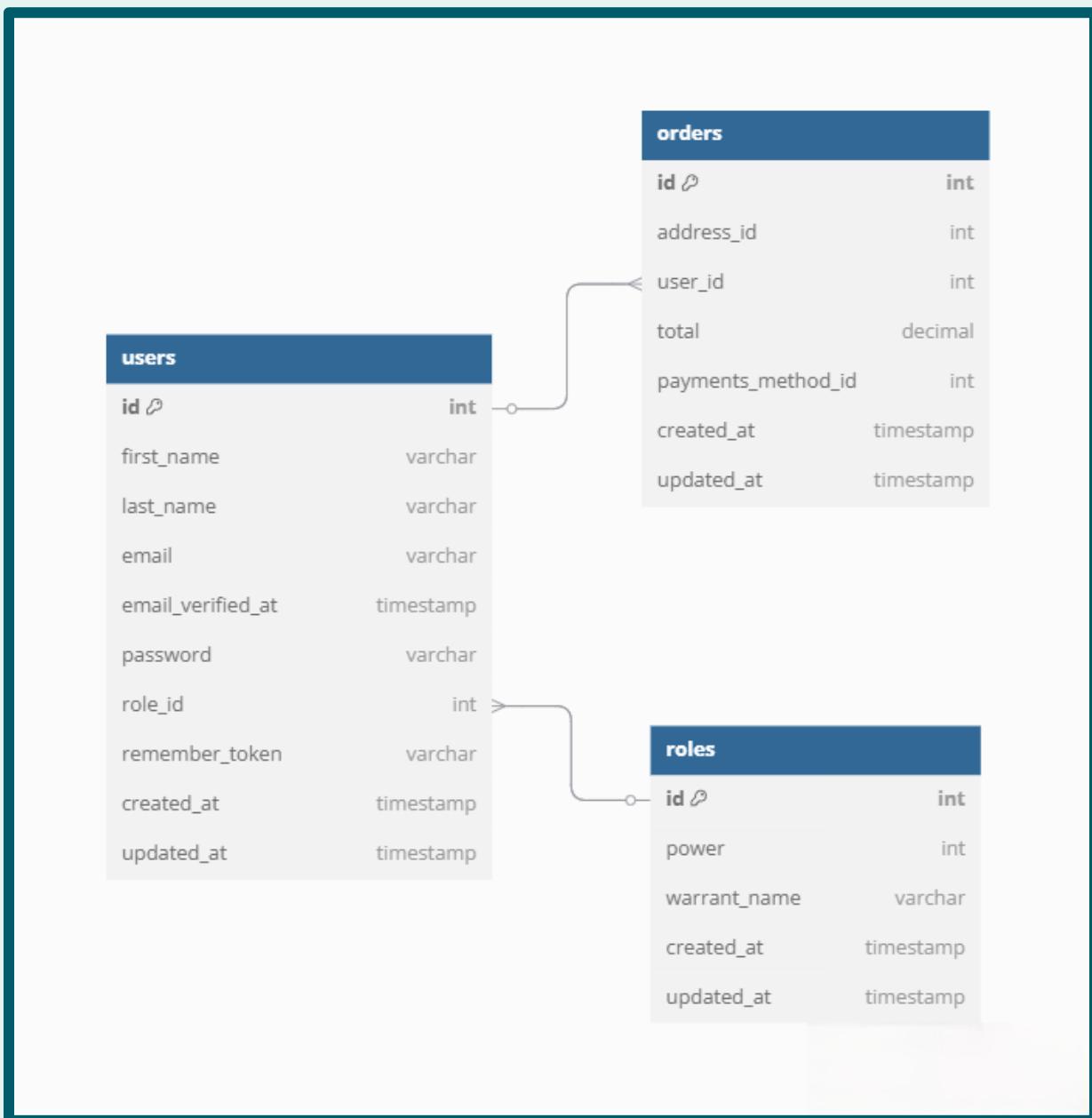
Payment ERD



50. ábra Payment ERD



Role EDR



51. ábra Role EDR



Tesztelés

A projekt megbízhatóságának és stabilitásának biztosítása érdekében automatizált végpont-teszteket készítettünk Cypress tesztkeretrendszerrel. Ezek a tesztek a rendszer legkritikusabb felhasználói folyamatait fedik le: a regisztrációt, bejelentkezést, termékkiválasztást, kosárkezelést, címadatok megadását, valamint a rendelés végrehajtását.

1. Register.cy.js

Regisztrációs teszt egyedi generált email címmel:

```
describe('Regisztráció teszt', () => {
  beforeEach(() => {
    cy.visit('https://flexistore.hu/register2');
  });

  it('Sikeress regisztráció végrehajtása', () => {
    // Generálunk egy egyedi email címet minden tesztfutáshoz
    const uniqueEmail = `testuser_${Date.now()}@example.com`;

    cy.get('input#last_name').type('Teszt');
    cy.get('input#first_name').type('János');
    cy.get('input#email').type(uniqueEmail);
    cy.get('input#password').type('Jelszo123!');
    cy.get('input#passwordAgain').type('Jelszo123!');

    cy.get('form').submit();
    cy.contains('Sikeress regisztráció').should('exist');
  });
});
```



Regisztrációs teszt 10 véletlenszerű felhasználóval:

```
const users = [
  { first_name: "Anna", last_name: "Kovács", email: "anna.kovacs@example.com",
    password: "Jelszo123!" },
  { first_name: "Béla", last_name: "Nagy", email: "bela.nagy@example.com", password:
    "Jelszo123!" },
  { first_name: "Csilla", last_name: "Tóth", email: "csilla.toth@example.com",
    password: "Jelszo123!" },
  { first_name: "Dániel", last_name: "Szabó", email: "daniel.szabo@example.com",
    password: "Jelszo123!" },
  { first_name: "Eszter", last_name: "Kiss", email: "eszter.kiss@example.com",
    password: "Jelszo123!" },
  { first_name: "Ferenc", last_name: "Varga", email: "ferenc.varga@example.com",
    password: "Jelszo123!" },
  { first_name: "Gábor", last_name: "Molnár", email: "gabor.molnar@example.com",
    password: "Jelszo123!" },
  { first_name: "Hanna", last_name: "Németh", email: "hanna.nemeth@example.com",
    password: "Jelszo123!" },
  { first_name: "István", last_name: "Farkas", email: "istvan.farkas@example.com",
    password: "Jelszo123!" },
  { first_name: "Judit", last_name: "Horváth", email: "judit.horvath@example.com",
    password: "Jelszo123!" },
]
]

describe('Regisztráció 10 valószerű felhasználóval', () => {
  beforeEach(() => {
    cy.viewport(1920, 1080);
  });
  users.forEach((user, index) => {
    it(`Regisztráció teszt: ${user.first_name} ${user.last_name}`, () => {
      cy.visit('https://flexistore.hu/register2');

      cy.get('input#last_name').type(user.last_name);
      cy.get('input#first_name').type(user.first_name);
      cy.get('input#email').type(user.email);
      cy.get('input#password').type(user.password);
      cy.get('input#passwordAgain').type(user.password);
      cy.get('form').submit();
      cy.contains('Sikeres regisztráció').should('exist');
    });
  });
});
```



2. Login_test.cy.js

Login teszt már regisztrált felhasználóval:

```
describe('template spec', () => {
  beforeEach('Oldal betöltése', () => {
    cy.visit('https://flexistore.hu/login2');
  });

  it('Működik-e a belépés?', () => {
    cy.get('input#email').type('SuperAdmin@example.com');
    cy.get('input#password').type('Elekes4455!');
    cy.get('form').submit();
  });
});
```



3. Full_User_Process.cy.js

Tesztelt funkciók:

- Sikeres regisztráció valid adatokkal.
- Bejelentkezés valós felhasználói fiókkal.
- Termék kiválasztása és kosárba helyezése.
- Kosár tartalmának ellenőrzése, értesítések kezelése.
- Rendelési folyamat végigjárása, beleértve:
 - Felhasználási feltételek elfogadása.
 - Valós címadatok és elérhetőségek dinamikus megadása.
 - Fizetési mód kiválasztása.
 - Rendelés véglegesítése.

A tesztek során dinamikus adatokat is alkalmaztunk (például véletlenszerű e-mail cím, városnév, irányítószám stb.), hogy a szkriptek ne csak statikus környezetben működjenek, hanem valós felhasználói viselkedést is szimuláljanak.

Ez az automatizált megközelítés nemcsak a manuális tesztelési időt csökkentette jelentősen, hanem elősegítette a hibák gyors azonosítását, és biztosította a funkciók folyamatos működését az éles környezetben is.

További előnyök:

- Könnyen újrafuttatható, CI/CD folyamatba illeszthető.
- Skálázható más modulokra is (pl. admin felület tesztelése).
- Segíti az agilis fejlesztési környezetben a gyors visszajelzést.



Regisztráció, bejelentkezés és teljes rendelési folyamat teszt:

```
describe('Regisztráció teszt', () => {
  beforeEach(() => {
    cy.visit('https://flexistore.hu/register2');
    cy.viewport(1920, 1080);
  });

  it('Sikeres regisztráció végrehajtása', () => {
    const uniqueEmail = `testuser@gmail.com`;

    //Regisztrációs adatok kitöltése
    cy.get('input#last_name').type('Teszt');
    cy.get('input#first_name').type('Elek');
    cy.get('input#email').type(uniqueEmail);
    cy.get('input#password').type('Elekes4455!');
    cy.get('input#passwordAgain').type('Elekes4455!');

    cy.get('form').submit();
    cy.contains('Sikeres regisztráció').should('exist');
  });
});
```

```
describe('Bejelentkezés és rendelés teszteletése!Működik-e?', () => {
  beforeEach('Oldal betöltése', () => {
    cy.viewport(1920, 1080);
  });
  it('Teljes rendelés végrehajtás!', () => {
    // Bejelentkezés
    cy.visit('https://flexistore.hu/login2');
    cy.get('input#email').type('testuser@gmail.com');
    cy.get('input#password').type('Elekes4455!');
    cy.get('form').submit();
    cy.url().should('not.include', '/login2');

    //Termék kiválasztása
    cy.visit('https://flexistore.hu/products');

    cy.get('.card').first().should('exist');

    //Termék Kosárba helyezése
    cy.get('#add-to-cart-button')
      .should('be.visible')
      .click();

    //Termék ellenőrzése ,hogy sikeresen hozzáadva a kosárhoz és toastify eltűnt-e
  });
});
```



```
cy.contains('Termék hozzáadva a kosárhoz!').should('exist');
cy.scrollTo('top');
cy.wait(4000);
cy.get('.Toastify__toast').should('not.exist');

cy.wait(1000)
cy.get('.hidden > .dropdown > .btn-ghost')
  .should('be.visible')
  .scrollIntoView()
  .click();

cy.wait(1000);
cy.get('.dropdown > .card > .card-body > .card-actions > .btn')
  .should('be.visible')
  .click();

cy.scrollTo('top');
cy.wait(2000);

//Tovább a pénztárhoz gomb
cy.get('#next-to-checkout')
  .should('be.visible')
  .click();

//TermOfUse elfogadása,checkbox bejelölése
cy.scrollTo('top');
cy.wait(2000);
cy.contains('Elfogadom a felhasználási feltételeket.').scrollIntoView();
cy.get('input[type="checkbox"]').check({ force: true });
cy.wait(1000);

cy.get('#accepted-term')
  .should('be.visible')
  .click();

//Címadatok megadása
cy.get('input[type="checkbox"]').check({ force: true });
cy.wait(1000);
const randomZip = Math.floor(1000 + Math.random() * 8999);
const cities = ['Budapest', 'Debrecen', 'Szeged', 'Pécs', 'Győr'];
const streets = ['Petőfi Sándor utca', 'Kossuth Lajos tér', 'Ady Endre
út', 'Rákóczi út', 'Jókai utca'];
const houseNumbers = ['12', '45A', '7/B', '102', '18'];

const randomCity = cities[Math.floor(Math.random() * cities.length)];
const randomStreet = streets[Math.floor(Math.random() * streets.length)];
```



```
const randomHouseNumber = houseNumbers[Math.floor(Math.random() * houseNumbers.length)];
const email = `teszt${Date.now()}@gmail.com`;

cy.get('#city').type(randomCity);
cy.get('#zip').type(randomZip.toString());
cy.get('#email').type(email);
cy.get('#street').type(randomStreet);
cy.get('#house_number').type(randomHouseNumber);

cy.get('#street_id').then(($select) => {
  const options = $select.find('option');
  if (options.length > 1) {
    cy.get('#street_id').select(options.eq(1).val());
  }
});

cy.scrollTo('top');
cy.wait(5000);

//Fizetési mód kiválasztása
cy.get('select#id')
.should('be.visible')
.then(($select) => {
  const options = $select.find('option');
  if (options.length > 1) {
    const firstValidOption = options.eq(1);
    const value = firstValidOption.val();
    const text = firstValidOption.text();

    cy.log(`Első valódi opció kiválasztva: "${text}" (value="${value}")`);
    console.log(`Cypress kiválasztja: "${text}" (value="${value}")`);

    cy.wrap($select).select(value);
  } else {
    cy.log('Nincs elérhető valódi opció.');
    console.warn('Nincs elérhető valódi opció a select mezőben.');
  }
});
cy.wait(2000);
cy.contains('button', 'Megrendelés')
.should('be.visible')
.click();
cy.scrollTo('top');
});
```



Összegzés -Flexistore

A **Flexistore** eszközbérlési rendszer a célzott fejlesztésével sikeresen megvalósított egy olyan online platformot, amely lehetővé teszi a felhasználók számára a különböző termékek gyors és kényelmes bérletét az ország különböző pontjain található csomagautomatákból. A projekt során a csapattal az **React** frontend technológiai megoldást alkalmaztuk, amely dinamikus és rövidre válaszoló felhasználói élményt biztosított. A backend fejlesztése a **Laravel** keretrendszerre épült, amely lehetővé tette az adatkezelés és a biztonságos API-k gyors implementálását. Az esztétikai és felhasználói élmény kialakításában a **Tailwind CSS** és **DaisyUI** segített, gyors, letisztult megjelenést biztosítva.

A fejlesztés során kiemelt figyelmet fordítottunk a stabil működésre, a felhasználói élményre és az adminisztrációs felület egyszerű kezelésére, amelyet a rendszer biztonságos és könnyen skálázható módon valósítottunk meg. Az alkalmazás az ország bármely pontján képes fogadni a bérleti igényeket, biztosítva ezzel a termékek gyors elérhetőségét és a felhasználói kényelmet.

A **tanulságok** alapján a projekt fejlesztése során megértettük, hogy egy ilyen komplex rendszerben különösen fontos a **rugalmas adatkezelés**, a **felhasználói élmény optimalizálása**, és a **biztonságos tranzakciók biztosítása**. A tesztelési fázis során számos hibát és fejlesztési lehetőséget fedeztünk fel, amelyek segítettek a rendszer finomhangolásában.

Fejlesztési lehetőségek

A Flexistore rendszer továbbfejlesztésére több lehetőség is kínálkozik, amelyek a felhasználói élmény javítását, a rendszer skálázhatóságát és a jövőbeni bővíthetőséget szolgálják. A következő fejlesztési irányokat javasoljuk:

- Adminisztrációs részleg szétválasztása és külön domain használata**
A rendszer adminisztrációs felületét érdemes lenne egy külön domain néven elérhetővé tenni. Ez nemcsak a biztonságot növeli, hanem a rendszer adminisztrációs és felhasználói részeit is könnyebben szétválaszthatja, biztosítva az egyszerűbb kezelhetőséget és a gyorsabb skálázást.
- Mobil alkalmazás fejlesztése**
A Flexistore weboldalának és a csomagautomaták bérleti rendszerének mobilalkalmazásra való "átállása" egy fontos lépés lehet a felhasználói élmény továbbfejlesztésében. A mobilalkalmazás lehetőséget biztosít arra, hogy a felhasználók gyorsabban és kényelmesebben intézhessék bérleseiket, és a rendszer szinkronban működjön a weboldallal. Ez nemcsak a felhasználók elégedettségét növeli, hanem bővíti a rendszer funkcionálitását.



4. Csomagautomaták vizuális megjelenítése térképen

A felhasználók számára rendkívül hasznos lehet, ha a csomagautomaták helyszínei vizuálisan is megjelennek egy térképen, például **Google Maps** vagy egyéb térkép alapú alkalmazás segítségével. Ez lehetővé teszi, hogy a felhasználók könnyedén megtalálják a legközelebbi automatát, és információkat kapjanak a hozzá tartozó szolgáltatásokról, például az elérhetőségről vagy a bérlet során felmerülő lehetőségekről. Ez a fejlesztés jelentős mértékben növelte a felhasználói élményt és a rendszer funkcionalitását.

Jövőbeli kilátások

A projekt folytatása és további fejlesztése lehetőséget biztosít arra, hogy a Flexistore még inkább piacképes és felhasználóbarát megoldássá váljon. A fenti fejlesztési javaslatok megvalósításával tovább növelhető a rendszer skálázhatósága, a felhasználói élmény és a jövőbeli bővítési lehetőségek is biztosítva lesznek.



Melléklet:

A képeket a domumentum FlexiStoreKépek mellékleteiben találja meg.

1. ábra Flexistore.hu desktop	31
2. ábra Menu responsive	34
3. ábra Footer responsive.....	37
4. ábra New Locker Desktop.....	72
5. ábra Register Desktop	106
6. ábra Admin User List Desktop.....	132
7. ábra <i>Admin User List Mobile</i>	132
8. ábra Admin All Order List Dekstop	137
9. ábra User Data Edit	142
10. ábra Cypress test success	170
11. ábra PublicAreaCard	300
12. ábra PublicAreas List.....	304
13. ábra Login Panel	310
14. ábra Profile Picture Upload	316
15. ábra Register Panel	322
16. ábra Address Panel	330
17. ábra Cart View Dekstop	335
18. ábra Cart View Mobile	335
19. ábra Payment Panel Desktop	345
20. ábra Payment Panel Mobile	345
21. ábra Category Card Panel	359
22. ábra Category Edit Panel	359
23. ábra Categories List	363
24. ábra New Category Panel.....	370
25. ábra All Order List Admin Desktop.....	390
26. ábra All Order List Admin Mobile.....	390
27. ábra Locker Info	418
28. ábra Locker Card	424
29. ábra Locker List	429
30. ábra New Payment Panel.....	439
31. ábra New Product Panel	453
32. ábra Product Card.....	459
33. ábra Product Info	462
34. ábra Products List Dekstop	468
35. ábra Products List Mobile	468
36. ábra New Role Panel	473
37. ábra Role List.....	480
38. ábra Term of use modal	483
39. ábra Privacy Policy modal.....	486
40. ábra Footer Desktop.....	497
41. ábra Footer Mobile	497
42. ábra Main Dekstop	501
43. ábra Main Mobile	501



44. ábra Full ERD	512
45. ábra User ERD.....	513
46. ábra Order ERD	514
47. ábra Product ERD.....	514
48. ábra Orders ERD	515
49. ábra Address ERD	516
50. ábra Payment ERD.....	517
51. ábra Role ERD	518

Források:

Kioko, A. (2023) 'Cart Functionality in React with Context API', Dev.to, 4 June. Available at:
<https://dev.to/anne46/cart-functionality-in-react-with-context-api-2k2f> (Accessed:26 April 2025)

SVGRepo (n.d.) *Free SVG Vectors and Icons*. Available at: <https://www.svgrepo.com/> (Accessed: 28 April 2025).

Tailwind Labs (n.d.) *Tailwind CSS - Rapidly build modern websites without ever leaving your HTML*. Available at: <https://tailwindcss.com/> (Accessed: 28 April 2025).

daisyUI (n.d.) *daisyUI - Tailwind CSS Components*. Available at: <https://daisyui.com/> (Accessed: 28 April 2025).

W3Schools.com. (n.d.). Available at: <https://www.w3schools.com/> (Accessed 6 May 2025).

Stack Overflow. (n.d.). *Programming Q&A platform*. Available at: <https://stackoverflow.com/> (Accessed 6 May 2025).

Laravel. (n.d.). *Laravel 11.x Documentation*. Available at: <https://laravel.com/docs/11.x> (Accessed 6 May 2025).

Laracasts. (n.d.). *30 Days to Learn Laravel 11*. Available at: <https://laracasts.com/series/30-days-to-learn-laravel-11> (Accessed 6 May 2025).

Kovacsrud. (n.d.). *geszi_frontend_24-25*. GitHub. Available at:
https://github.com/kovacsrud/geszi_frontend_24-25 (Accessed 6 May 2025).