МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа № 5**

*по дисциплине «Программирование мобильных устройств»*

Выполнил студен группы ПИбд-32

Емельянов А. С.

Проверил доцент кафедры

«Информационные системы»

Филиппов А.А.

Ульяновск, 2023

**Задание**

Взаимодействие с REST API. Необходимо:

1. Создать сервис для работы с REST API с помощью Retrofit. В качестве сервера можно использовать json-server или самостоятельную реализацию.

2. Создать репозитории для абстрагирования от реализации механизма доступа к данным. Новые репозитории должны реализовывать интерфейсы-репозитории из ЛР 4, а также использовать классы-репозитории для работы с БД.

3. Для работы со страничными данными (paging) необходимо реализовать RemoteMediator в соответствующем новом репозитории.

4. Адаптировать ViewModel для работы с новыми репозиториями. Бизнеслогика должна быть изменена минимально.

5. Отчет и изменения проекта загрузить в репозиторий по адресу <http://student.git.athene.tech>

1. Сервер написан с использованием фреймоврка Java Spring. Создадим сервис с помощью Retrofit для их взаимодействия.
2. interface BackendService {  
     
    //SNEAKER  
    @GET("sneaker/get/{id}")  
    suspend fun getSneaker(  
    @Path("id") id: Int,  
    ): SneakerRemote  
     
    @GET("sneaker/getAll")  
    suspend fun getSneakers(  
    @Query("page") page: Int,  
    @Query("size") size: Int,  
    ): List<SneakerRemote>  
     
    @POST("sneaker/create")  
    suspend fun createSneaker(  
    @Body sneaker: SneakerRemote,  
    ): SneakerRemote  
     
    @PUT("sneaker/update/{id}")  
    suspend fun updateSneaker(  
    @Path("id") id: Int,  
    @Body sneaker: SneakerRemote  
    ): SneakerRemote  
     
    @DELETE("sneaker/delete/{id}")  
    suspend fun deleteSneaker(  
    @Path("id") id: Int  
    )  
     
    //USER  
    @POST("user/signup")  
    suspend fun SignUp(  
    @Body user: UserRemote,  
    ): UserRemote  
     
    @POST("user/signin")  
    suspend fun SignIn(  
    @Body user: UserRemoteSignIn  
    ): UserRemote  
     
    //BASKET  
    @POST("basket/createBasketSneaker")  
    suspend fun createBasketSneaker(  
    @Body basketSneaker: BasketSneakerRemote  
    )  
     
    @GET("basket/getUserBasketSneakers/{id}")  
    suspend fun getUserBasketSneakers(  
    @Path("id") id: Int  
    ): List<SneakerRemote>  
     
    @GET("basket/getUserBasket/{id}")  
    suspend fun getUserBasket(  
    @Path("id") id: Int  
    ): Int  
     
    @GET("basket/getQuantity/{basketId}/{sneakerId}")  
    suspend fun getQuantity(  
    @Path("basketId") basketId: Int,  
    @Path("sneakerId") sneakerId: Int,  
    ): Int  
     
    @PUT("basket/incrementQuantity/{basketId}/{sneakerId}")  
    suspend fun increment(  
    @Path("basketId") basketId: Int,  
    @Path("sneakerId") sneakerId: Int,  
    )  
     
    @PUT("basket/decrementQuantity/{basketId}/{sneakerId}")  
    suspend fun decrement(  
    @Path("basketId") basketId: Int,  
    @Path("sneakerId") sneakerId: Int,  
    )  
     
    @GET("basket/getSneaker/{basketId}/{sneakerId}")  
    suspend fun getSneaker(  
    @Path("basketId") basketId: Int,  
    @Path("sneakerId") sneakerId: Int,  
    ): Boolean  
     
    @GET("basket/removeSneaker/{basketId}/{sneakerId}")  
    suspend fun deleteSneakerFromBasket(  
    @Path("basketId") basketId: Int,  
    @Path("sneakerId") sneakerId: Int,  
    )  
     
    @GET("basket/getUserPrice/{userId}")  
    suspend fun getTotalPriceForUserBasket(  
    @Path("userId") userId: Int  
    ): Double  
     
    @GET("basket/deleteAllSneakerFromBasket/{basketId}")  
    suspend fun deleteAllSneakerFromBasket(  
    @Path("basketId") basketId: Int  
    )  
     
    //ORDER  
    @POST("order/createOrderSneaker")  
    suspend fun createOrderSneaker(  
    @Body orderSneaker: OrderSneakerRemote  
    )  
     
    @POST("order/create")  
    suspend fun createOrder(  
    @Body order: OrderRemote  
    ): Long  
     
    @GET("order/getUserOrders/{userId}")  
    suspend fun getUserOrders(  
    @Path("userId") userId: Int  
    ) : List<OrderRemote>  
     
    @GET("order/getSneakerFromOrder/{orderId}")  
    suspend fun getSneakerFromOrder(  
    @Path("orderId") orderId: Int  
    ) : List<SneakerRemote>  
     
    @GET("order/deleteOrder/{orderId}")  
    suspend fun deleteOrder(  
    @Path("orderId") orderId: Int  
    )  
     
    companion object {  
    private const val BASE\_URL = "https://59k4pfj3-8080.euw.devtunnels.ms/api/"  
     
    @Volatile  
    private var INSTANCE: BackendService? = null  
     
    fun getInstance(): BackendService {  
    return INSTANCE ?: *synchronized*(this) **{** val logger = HttpLoggingInterceptor()  
    logger.level = HttpLoggingInterceptor.Level.*BASIC* val client = OkHttpClient.Builder()  
    .addInterceptor(logger)  
    .build()  
    return Retrofit.Builder()  
    .baseUrl(BASE\_URL)  
    .client(client)  
    .addConverterFactory(Json.*asConverterFactory*("application/json".*toMediaType*()))  
    .build()  
    .create(BackendService::class.*java*)  
    .*also* **{** INSTANCE = **it }  
    }** }  
    }  
   }
3. Реализация интерфейсов для доступа к данным с сервера

class RestBasketRepository(  
 private var service: BackendService  
): BasketRepository {  
 override suspend fun insertBasketSneaker(basketSneaker: BasketSneakers) {  
 service.createBasketSneaker(basketSneaker.*toBasketSneakerRemote*())  
 }  
  
 override suspend fun getBasketWithSneakers(id: Int): Flow<List<Sneaker>> {  
 val sneakersRemoteList = service.getUserBasketSneakers(id)  
 val sneakersList = sneakersRemoteList.*map* **{ it**.*toSneaker*() **}** return *flowOf*(sneakersList.*toList*())  
 }  
  
 override suspend fun getUserBasketId(id: Int): Flow<Int> {  
 return *flowOf*(service.getUserBasket(id))  
 }  
  
 override suspend fun getQuantity(basketId: Int, sneakerId: Int): Int? {  
 return service.getQuantity(basketId, sneakerId)  
 }  
//  
// override fun getAllBasket(): Flow<List<Basket>> {  
// *TODO("Not yet implemented")*// }  
//  
// override suspend fun delete(basket: Basket) {  
// *TODO("Not yet implemented")*// }  
//  
// override suspend fun createBasket(basket: Basket): Long {  
// *TODO("Not yet implemented")*// }  
//  
 override suspend fun removeSneakerFromBasket(basketId: Int, sneakerId: Int) {  
 service.deleteSneakerFromBasket(basketId, sneakerId)  
 }  
//  
// override suspend fun updateSneakerQuantity(basketId: Int, sneakerId: Int, quantity: Int) {  
// *TODO("Not yet implemented")*// }  
//  
 override suspend fun incrementSneakerQuantity(basketId: Int, sneakerId: Int) {  
 service.increment(basketId, sneakerId)  
 }  
  
 override suspend fun decrementSneakerQuantity(basketId: Int, sneakerId: Int) {  
 service.decrement(basketId, sneakerId)  
 }  
//  
// override suspend fun getQuantity(basketId: Int, sneakerId: Int): Int? {  
// *TODO("Not yet implemented")*// }  
//  
 override suspend fun existSneaker(basketId: Int, sneakerId: Int): Boolean {  
 return service.getSneaker(basketId, sneakerId)  
 }  
  
 override suspend fun getTotalPriceForUser(userId: Int): Double? {  
 return service.getTotalPriceForUserBasket(userId)  
 }  
  
 override suspend fun deleteAllSneakerFromBasket(basketId: Int) {  
 service.deleteAllSneakerFromBasket(basketId)  
 }  
}

class RestOrderRepository(private val service: BackendService): OrderRepository {  
 override suspend fun createOrder(order: Order): Long {  
 return service.createOrder(order.*toOrderRemote*())  
 }  
  
 override suspend fun insertOrderSneaker(orderSneaker: OrderSneaker) {  
 service.createOrderSneaker(orderSneaker.*toOrderSneakerRemote*())  
 }  
  
 override suspend fun delete(orderId: Int) {  
 service.deleteOrder(orderId)  
 }  
  
 override suspend fun getSneakerFromOrder(id: Int): Flow<List<Sneaker>> {  
 val sneakersRemoteList = service.getSneakerFromOrder(id)  
 val sneakersList = sneakersRemoteList.*map* **{ it**.*toSneaker*() **}** return *flowOf*(sneakersList.*toList*())  
 }  
  
 override suspend fun getUserOrders(id: Int): Flow<List<Order>> {  
 val ordersRemoteList = service.getUserOrders(id)  
 val ordersList = ordersRemoteList.*map* **{ it**.*toOrder*() **}** return *flowOf*(ordersList.*toList*())  
 }  
}

class RestSneakerRepository(  
 private val service: BackendService,  
 private val dbSneakerRepository: SneakerRepoImpl,  
 private val database: AppDatabase,  
 private val dbRemoteKeyRepository: RemoteKeysRepositoryImpl  
  
) : SneakerRepository {  
  
 override fun getAllSneakers(): Flow<PagingData<Sneaker>> {  
 val pagingSourceFactory = **{** dbSneakerRepository.getAllSneakersPagingSource() **}** @OptIn(ExperimentalPagingApi::class)  
 return Pager(  
 config = PagingConfig(  
 pageSize = AppContainer.LIMIT,  
 enablePlaceholders = false  
 ),  
 remoteMediator = SneakerRemoteMediator(  
 service,  
 dbSneakerRepository,  
 database,  
 dbRemoteKeyRepository,  
 ),  
 pagingSourceFactory = pagingSourceFactory  
 ).flow  
 }  
  
 override suspend fun getSneakerById(id: Int): Sneaker = service.getSneaker(id).*toSneaker*()  
 override suspend fun insertSneaker(sneaker: Sneaker) {  
 service.createSneaker(sneaker.*toSneakerRemote*())  
 }  
  
 override suspend fun updateSneaker(sneaker: Sneaker) {  
 sneaker.sneakerId?.*let* **{** service.updateSneaker(**it**, sneaker.*toSneakerRemote*()) **}** }  
  
 override suspend fun deleteSneaker(sneaker: Sneaker) {  
 sneaker.sneakerId?.*let* **{** service.deleteSneaker(**it**) **}** }  
}

class RestUserRepository(  
 private var service: BackendService  
): UserRepository {  
 override suspend fun createUser(user: User) {  
 service.SignUp(user.*toUserRemote*())  
 }  
  
 override suspend fun updateUser(user: User) {  
 *println*()  
 }  
  
 override suspend fun deleteUser(user: User) {  
 *println*()  
 }  
 override suspend fun authUser(user: UserRemoteSignIn): User {  
 return service.SignIn(user).*toUser*()  
 }  
}

Также создадим модели, для data transfer object

@Serializable  
data class BasketRemote (  
 val id: Int? = 0,  
 val userId: Int = 0,  
 )

@Serializable  
data class BasketSneakerRemote (  
 val basketId: Int = 0,  
 val sneakerId: Int = 0,  
 val quantity: Int = 0  
)  
  
fun BasketSneakerRemote.toBasketSneaker(): BasketSneakers = BasketSneakers(  
 basketId,  
 sneakerId,  
 quantity  
)  
  
fun BasketSneakers.toBasketSneakerRemote():BasketSneakerRemote = BasketSneakerRemote(  
 basketId,  
 sneakerId,  
 quantity  
)

@Serializable  
data class OrderRemote(  
 val id: Int? = 0,  
 val date: Long = 0L,  
 val city: String = "",  
 val street: String = "",  
 val house: String = "",  
 val subtotal: Double = 0.0,  
 val taxes: Double = 0.0,  
 val total: Double = 0.0,  
 val userId: Int = 0  
)  
  
fun OrderRemote.toOrder(): Order = Order(  
 id,  
 date,  
 city,  
 street,  
 house,  
 subtotal,  
 taxes,  
 total,  
 userId  
)  
  
fun Order.toOrderRemote():OrderRemote = OrderRemote(  
 orderId,  
 date,  
 city,  
 street,  
 house,  
 subtotal,  
 taxes,  
 total,  
 creatorUserId  
)

@Serializable  
data class OrderSneakerRemote (  
 val orderId: Int = 0,  
 val sneakerId: Int = 0,  
 val quantity: Int = 0  
)  
  
fun OrderSneakerRemote.toOrderSneaker(): OrderSneaker = OrderSneaker(  
 orderId,  
 sneakerId,  
 quantity  
)  
  
fun OrderSneaker.toOrderSneakerRemote():OrderSneakerRemote = OrderSneakerRemote(  
 orderId,  
 sneakerId,  
 quantity  
)

@Serializable  
data class SneakerRemote (  
 val id: Int? = 0,  
 val brand: String = "",  
 val model: String = "",  
 val description: String = "",  
 val price: Double = 0.0,  
 val photo: Int = 0  
)  
  
fun SneakerRemote.toSneaker():Sneaker = Sneaker(  
 id,  
 brand,  
 model,  
 description,  
 price,  
 photo  
)  
  
fun Sneaker.toSneakerRemote():SneakerRemote = SneakerRemote(  
 sneakerId,  
 brand,  
 model,  
 description,  
 price,  
 photo  
)

@Serializable  
data class UserRemote (  
 val id: Int? = 0,  
 val name: String = "",  
 val surname: String = "",  
 val email: String = "",  
 val password: String = "",  
 val role: String = "",  
 val photo: Int? = 0,  
)  
  
  
fun UserRemote.toUser(): User = User(  
 id,  
 name,  
 surname,  
 email,  
 password,  
 role,  
 photo  
)  
  
fun User.toUserRemote():UserRemote = UserRemote(  
 userId,  
 name,  
 surname,  
 email,  
 password,  
 role,  
 photo  
)

@Serializable  
data class UserRemoteSignIn(  
 val email: String = "",  
 val password: String = "",  
)

1. Реализуем RemoteMediator для работы со списком каталога кроссовок
2. @OptIn(ExperimentalPagingApi::class)  
   class SneakerRemoteMediator(  
    private val service: BackendService,  
    private val sneakerRepository: SneakerRepoImpl,  
    private val database: AppDatabase,  
    private val dbRemoteKeyRepository: RemoteKeysRepositoryImpl  
   ) : RemoteMediator<Int, Sneaker>() {  
    override suspend fun initialize(): InitializeAction {  
    return InitializeAction.*LAUNCH\_INITIAL\_REFRESH* }  
    override suspend fun load(  
    loadType: LoadType,  
    state: PagingState<Int, Sneaker>  
    ): MediatorResult {  
    val page = when (loadType) {  
    LoadType.*REFRESH* -> {  
    val remoteKeys = getRemoteKeyClosestToCurrentPosition(state)  
    remoteKeys?.nextKey?.minus(1) ?: 1  
    }  
     
    LoadType.*PREPEND* -> {  
    val remoteKeys = getRemoteKeyForFirstItem(state)  
    remoteKeys?.prevKey  
    ?: return MediatorResult.Success(endOfPaginationReached = remoteKeys != null)  
    }  
     
    LoadType.*APPEND* -> {  
    val remoteKeys = getRemoteKeyForLastItem(state)  
    remoteKeys?.nextKey  
    ?: return MediatorResult.Success(endOfPaginationReached = remoteKeys != null)  
    }  
    }  
     
    try {  
    val sneakers = service.getSneakers(page, state.config.pageSize).*map* **{ it**.*toSneaker*() **}** val endOfPaginationReached = sneakers.isEmpty()  
    database.withTransaction **{** if (loadType == LoadType.*REFRESH*) {  
    dbRemoteKeyRepository.deleteRemoteKey(RemoteKeyType.*SNEAKER*)  
    sneakerRepository.clearSneakers()  
    }  
    val prevKey = if (page == 1) null else page - 1  
    val nextKey = if (endOfPaginationReached) null else page + 1  
    val keys = sneakers.*map* **{** RemoteKeys(  
    entityId = **it**.sneakerId!!,  
    type = RemoteKeyType.*SNEAKER*,  
    prevKey = prevKey,  
    nextKey = nextKey  
    )  
    **}** dbRemoteKeyRepository.createRemoteKeys(keys)  
    sneakerRepository.insertSneakers(sneakers)  
    **}** return MediatorResult.Success(endOfPaginationReached = endOfPaginationReached)  
    } catch (exception: IOException) {  
    return MediatorResult.Error(exception)  
    } catch (exception: HttpException) {  
    return MediatorResult.Error(exception)  
    }  
    }  
     
    private suspend fun getRemoteKeyForLastItem(state: PagingState<Int, Sneaker>): RemoteKeys? {  
    return state.pages.*lastOrNull* **{ it**.data.*isNotEmpty*() **}**?.data?.*lastOrNull*()  
    ?.*let* **{** sneaker **->** sneaker.sneakerId?.*let* **{** dbRemoteKeyRepository.getAllRemoteKeys(**it**, RemoteKeyType.*SNEAKER*) **}  
    }** }  
     
    private suspend fun getRemoteKeyForFirstItem(state: PagingState<Int, Sneaker>): RemoteKeys? {  
    return state.pages.*firstOrNull* **{ it**.data.*isNotEmpty*() **}**?.data?.*firstOrNull*()  
    ?.*let* **{** sneaker **->** sneaker.sneakerId?.*let* **{** dbRemoteKeyRepository.getAllRemoteKeys(**it**, RemoteKeyType.*SNEAKER*) **}  
    }** }  
     
    private suspend fun getRemoteKeyClosestToCurrentPosition(  
    state: PagingState<Int, Sneaker>  
    ): RemoteKeys? {  
    return state.anchorPosition?.*let* **{** position **->** state.closestItemToPosition(position)?.sneakerId?.*let* **{** sneakerUid **->** dbRemoteKeyRepository.getAllRemoteKeys(sneakerUid, RemoteKeyType.*SNEAKER*)  
    **}  
    }** }  
   }

Также поменяем в di реализацию вместо бд, на апи

class AppDataContainer(private val context: Context) : AppContainer {  
 override val sneakerRepo: SneakerRepository by *lazy* **{** RestSneakerRepository(  
 BackendService.getInstance(),  
 sneakerRepository,  
 AppDatabase.getInstance(context),  
 remoteKeyRepository  
 )  
 **}** override val userRepo: UserRepository by *lazy* **{** RestUserRepository(BackendService.getInstance())  
 **}** override val basketRepo: BasketRepository by *lazy* **{** RestBasketRepository(BackendService.getInstance())  
 **}** override val orderRepo: OrderRepository by *lazy* **{** RestOrderRepository(BackendService.getInstance())  
 **}** private val sneakerRepository: SneakerRepoImpl by *lazy* **{** SneakerRepoImpl(AppDatabase.getInstance(context).sneakerDao())  
 **}** private val remoteKeyRepository: RemoteKeysRepositoryImpl by *lazy* **{** RemoteKeysRepositoryImpl(AppDatabase.getInstance(context).remoteKeysDao())  
 **}**}

1. Адаптируем все view модели под новую логику взаимодействия с данными
2. class BasketViewModel(private val basketRepository: BasketRepository): ViewModel() {  
     
    private val \_quantityStateMap = *mutableMapOf*<Int, MutableStateFlow<Int>>()  
     
    private val \_sneakerList = *MutableStateFlow*<List<Sneaker>>(*emptyList*())  
    val sneakerList: StateFlow<List<Sneaker>> = \_sneakerList  
     
    fun getQuantityState(basketId: Int, sneakerId: Int): StateFlow<Int> {  
    val quantityStateFlow = \_quantityStateMap.*getOrPut*(sneakerId) **{** *MutableStateFlow*(0)  
    **}** *viewModelScope*.*launch* **{** val quantityFromDb = basketRepository.getQuantity(basketId, sneakerId)  
    quantityFromDb?.*let* **{** quantityStateFlow.value = **it }  
    }** return quantityStateFlow  
    }  
     
    suspend fun isSneakerInBasket(basketId: Int, sneakerId: Int): Boolean {  
    return basketRepository.existSneaker(basketId, sneakerId)  
    }  
    fun addToBasket(basketSneakers: BasketSneakers) = *viewModelScope*.*launch* **{** val isSneakerInBasket = isSneakerInBasket(basketSneakers.basketId, basketSneakers.sneakerId)  
     
    if (isSneakerInBasket) {  
    incrementQuantity(basketSneakers.basketId, basketSneakers.sneakerId)  
    } else {  
    basketRepository.insertBasketSneaker(basketSneakers)  
    }  
    **}** fun fetchBasketSneakers(userId: Int) {  
    *viewModelScope*.*launch* **{** basketRepository.getBasketWithSneakers(userId).collect **{** \_sneakerList.value = **it  
    }  
    }** }  
     
    suspend fun getUserBasketId(userId: Int) : Flow<Int>{  
    return basketRepository.getUserBasketId(userId)  
    }  
     
     
    fun deleteSneakerFromBasket(basketId: Int, sneakerId: Int) = *viewModelScope*.*launch* **{** basketRepository.removeSneakerFromBasket(basketId, sneakerId)  
    fetchBasketSneakers(GlobalUser.getInstance().getUser()?.userId!!)  
    **}** fun incrementQuantity(basketId: Int, sneakerId: Int) {  
    val currentQuantity = \_quantityStateMap[sneakerId]?.value ?: 1  
    \_quantityStateMap[sneakerId]?.value = currentQuantity + 1  
     
    *viewModelScope*.*launch* **{** basketRepository.incrementSneakerQuantity(basketId, sneakerId)  
    **}** }  
     
    fun decrementQuantity(basketId: Int, sneakerId: Int) {  
    val currentQuantity = \_quantityStateMap[sneakerId]?.value ?: 1  
    if (currentQuantity > 1) {  
    \_quantityStateMap[sneakerId]?.value = currentQuantity - 1  
     
    *viewModelScope*.*launch* **{** basketRepository.decrementSneakerQuantity(basketId, sneakerId)  
    **}** }  
    }  
    fun deleteAllSneakerFromBasket(basketId: Int) = *viewModelScope*.*launch* **{** basketRepository.deleteAllSneakerFromBasket(basketId)  
    **}**}

class OrderViewModel(private val orderRepository: OrderRepository, private val basketRepository: BasketRepository) : ViewModel() {  
  
 var city = *mutableStateOf*("")  
 val street = *mutableStateOf*("")  
 val house = *mutableStateOf*("")  
 private val \_selectedItems = MutableLiveData<List<Sneaker>>()  
 private val \_subTotal = *mutableStateOf*(0.0)  
 val subTotal: State<Double> get() = \_subTotal  
 val selectedItems: LiveData<List<Sneaker>> get() = \_selectedItems  
  
 fun updateSelectedItems(items: List<Sneaker>) {  
 \_selectedItems.*value* = items  
 }  
  
 fun deleteOrder(orderId: Int) = *viewModelScope*.*launch* **{** orderRepository.delete(orderId)  
 **}** suspend fun getOrderList(id: Int) : Flow<List<Order>> {  
 return orderRepository.getUserOrders(id)  
 }  
  
 suspend fun getOrderWithSneakers(id: Int) : Flow<List<Sneaker>> {  
 return orderRepository.getSneakerFromOrder(id)  
 }  
  
 fun createOrder() = *viewModelScope*.*launch* **{** val userId = GlobalUser.getInstance().getUser()?.userId!!  
 val subTotal = getSubTotal(userId)  
 val order = Order(  
 date = Date().*time*,  
 city = city.value,  
 street = street.value,  
 house = house.value,  
 subtotal = subTotal,  
 taxes = "%.2f".*format*(subTotal \* 0.05).*toDouble*(),  
 total = "%.2f".*format*(subTotal \* 0.05 + subTotal).*toDouble*(),  
 creatorUserId = GlobalUser.getInstance().getUser()?.userId!!  
 )  
  
 val orderId = orderRepository.createOrder(order)  
  
 for (sneaker in selectedItems.*value*.*orEmpty*()) {  
 val userId = GlobalUser.getInstance().getUser()?.userId!!  
 val orderSneaker = OrderSneaker( orderId.toInt(), sneaker.sneakerId!!, 1)  
 if (orderSneaker != null) {  
 orderRepository.insertOrderSneaker(orderSneaker)  
 }  
 }  
 city.value = ""  
 street.value = ""  
 house.value = ""  
 **}** fun updateSubTotal(userId: Int) {  
 *viewModelScope*.*launch* **{** \_subTotal.value = getSubTotal(userId)  
 **}** }  
  
 suspend fun getSubTotal(userId: Int): Double {  
 return basketRepository.getTotalPriceForUser(userId) ?: 0.0  
 }  
}

class SneakerViewModel(private val sneakerRepository: SneakerRepository): ViewModel() {  
 val sneakerList = sneakerRepository.getAllSneakers()  
 var brand = *mutableStateOf*("")  
 val model = *mutableStateOf*("")  
 val description = *mutableStateOf*("")  
 val price = *mutableStateOf*("")  
 val photo = *mutableIntStateOf*(R.drawable.*img*)  
 fun insertSneaker() = *viewModelScope*.*launch* **{** val sneaker = Sneaker(  
 brand = brand.value,  
 model = model.value,  
 description = description.value,  
 price = price.value.*toDouble*(),  
 photo = photo.value  
 )  
 sneakerRepository.insertSneaker(sneaker)  
 **}** fun deleteSneaker(sneaker : Sneaker) = *viewModelScope*.*launch* **{** sneakerRepository.deleteSneaker(sneaker)  
 **}** fun UpdateSneaker(sneaker: Sneaker) = *viewModelScope*.*launch* **{** sneakerRepository.updateSneaker(sneaker)  
 **}**}

class UserViewModel(private val userRepository: UserRepository): ViewModel() {  
  
 var name = *mutableStateOf*("")  
 val surname = *mutableStateOf*("")  
 val email = *mutableStateOf*("")  
 val password = *mutableStateOf*("")  
 fun createUser() = *viewModelScope*.*launch* **{** val user = User(  
 name = name.value,  
 surname = surname.value,  
 email = email.value,  
 password = password.value,  
 role = "USER",  
 photo = R.drawable.*shailushai* )  
 userRepository.createUser(user)  
 **}** fun authUser() = *viewModelScope*.*launch* **{** val user = userRepository.authUser(UserRemoteSignIn(email.value, password.value))  
 GlobalUser.getInstance().setUser(user)  
 **}** fun isValidEmail(email: String): Boolean {  
 return android.util.Patterns.*EMAIL\_ADDRESS*.matcher(email).matches()  
 }  
}