Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Ubiquitous Applications Tecnologies

Documentation of the project

Book API

Project realized by:

Alexandros Mitsis

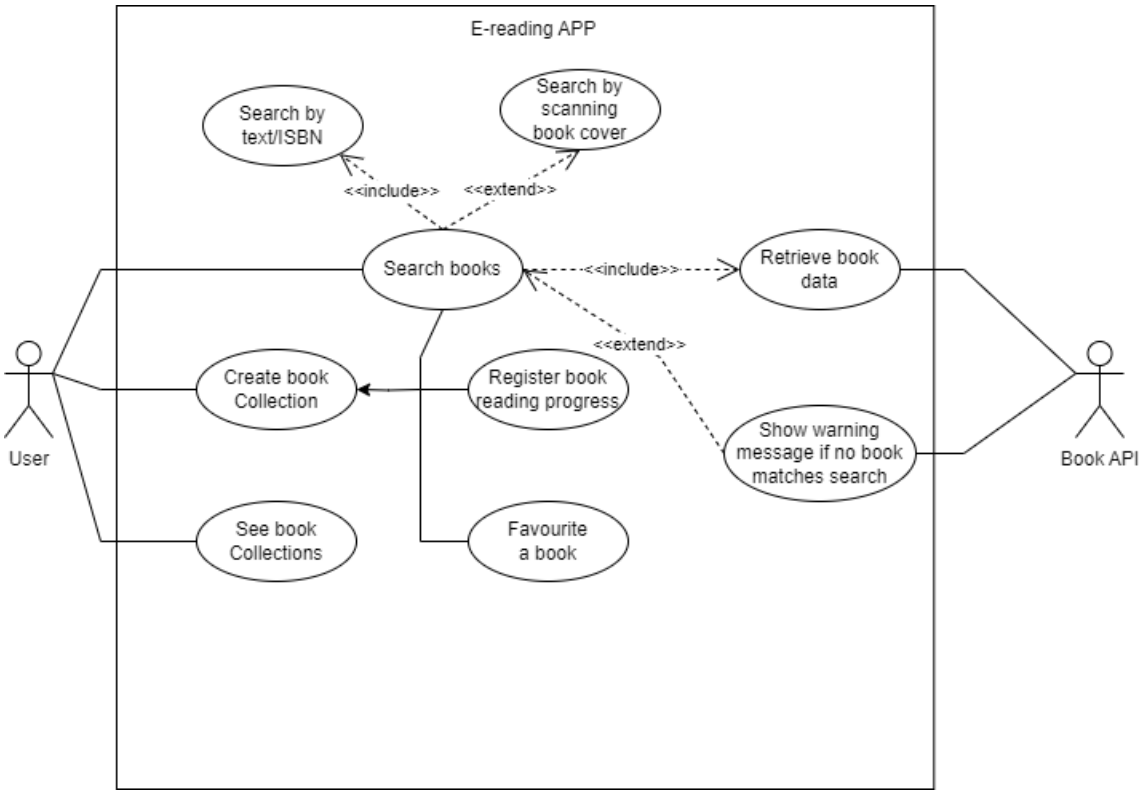
Bruno Silva

Luís Barradas

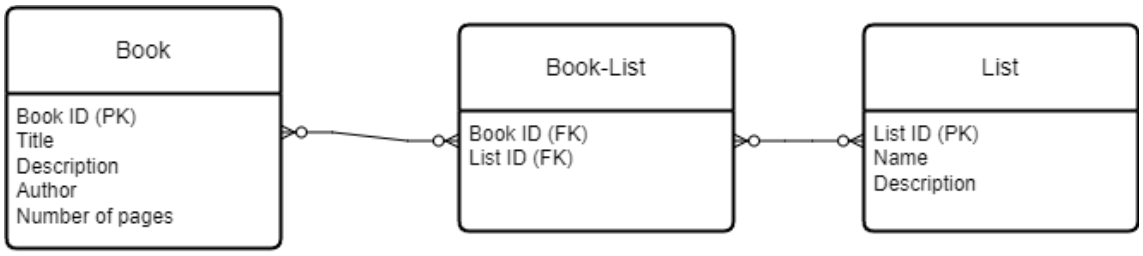
Index

1.Documentation

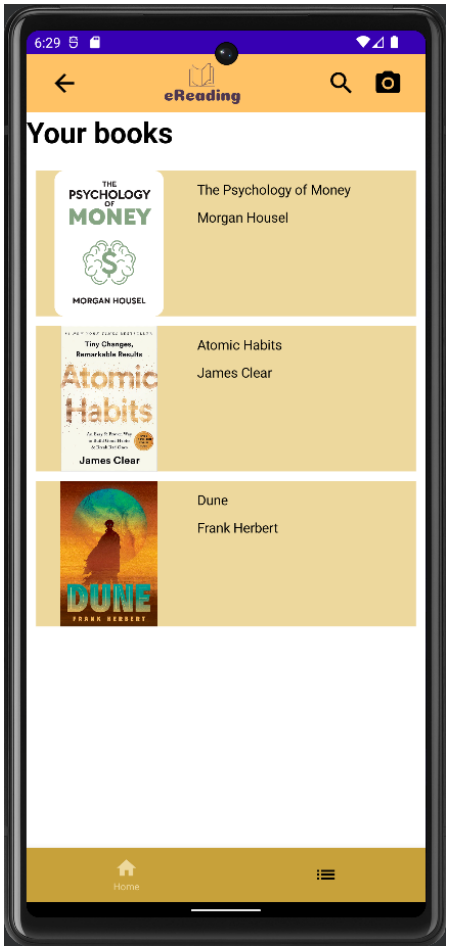
1.1.Use-case diagram:



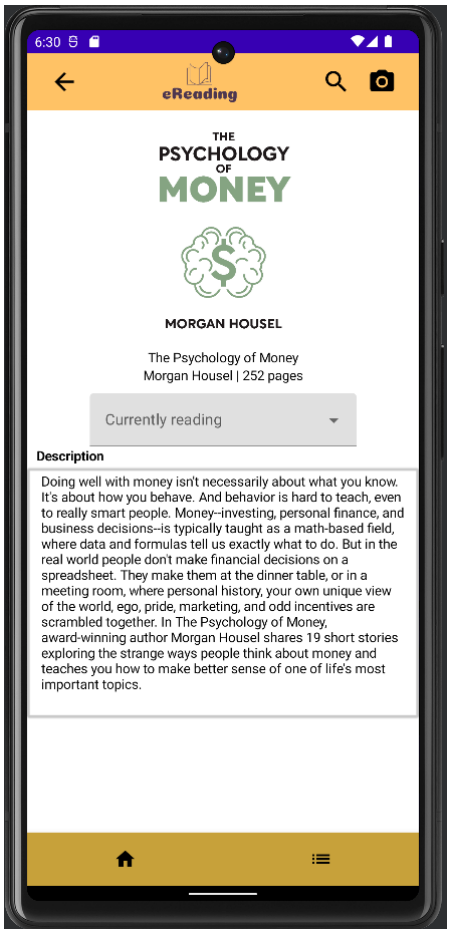
1.2.E-R diagram:



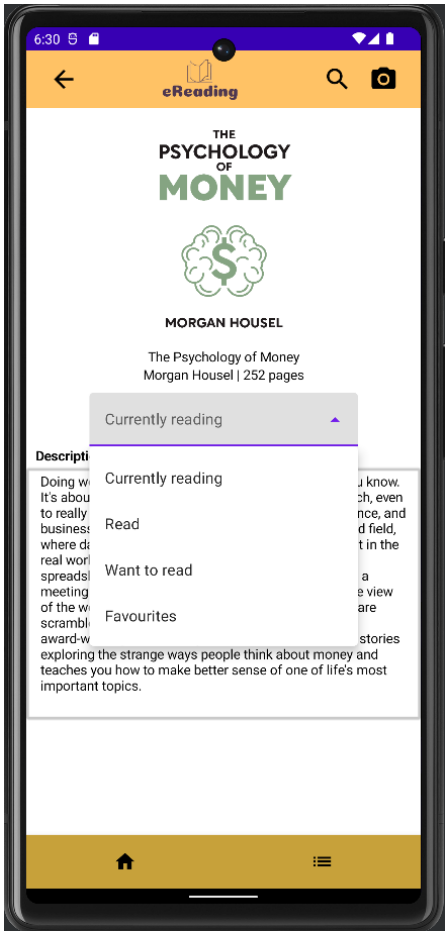
1.3.Graphical interface screenshots:



Here we have the main page where we can see some books the search bar and, in the bottom, we can switch to the list page.

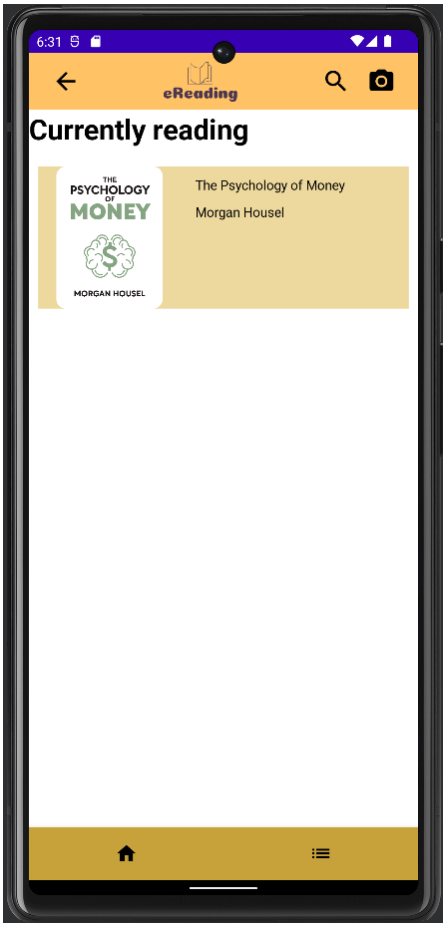


Here we have the page of the book when we press the book in the main page, in this page we have the Title, the number of pages, the author, the content box where we can add the book to some list and the description.

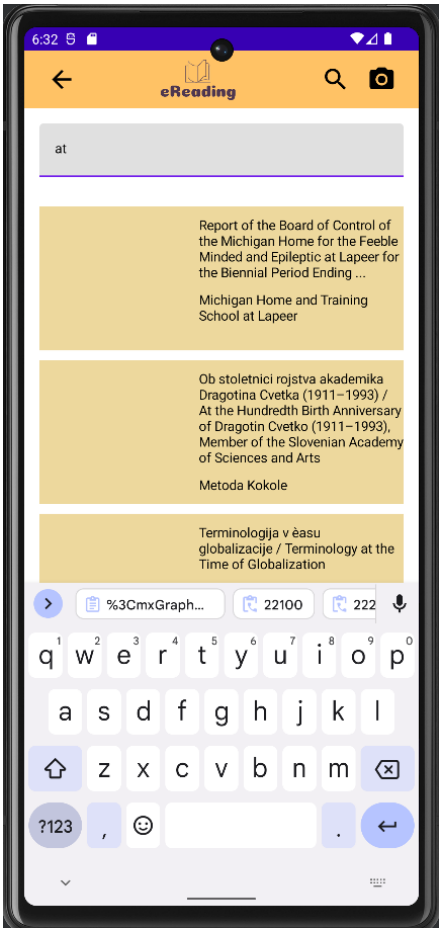


Here we have the book page again but with the content box open and we can see the list that we can add the book.

Here we have the Lists pages where we can see the Lists we saw before.



Here we have the one of the List page open with a book inside.



In this screenshot we have the search bar working with wi-fi.

And for the last screenshot we have the search bar working without wi-fi.

Presenting of Ubiquitous Technology:

Jetpack compose:

For our project we used the Jetpack Compose to our search bar here we gonna explain how it was implemented the offline “part”.

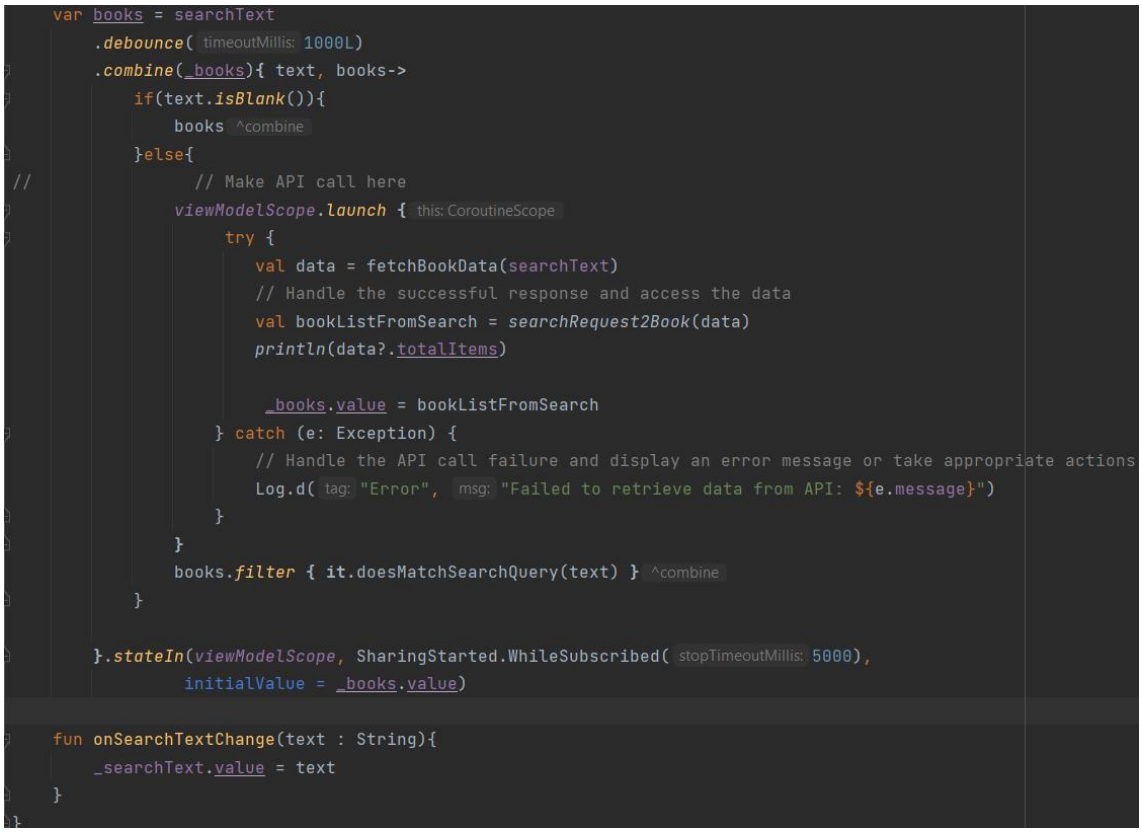
So first we gonna define the SearchViewModel class using the ViewModel class. In this class we gonna define the logic behind the search functionality and data retrieval, and the ViewModel It is designed to store and manage UI-related data in a lifecycle-aware manner.

Now we need to define 3 variables:

```
@OptIn(FlowPreview::class)
class SearchViewModel: ViewModel(){

    private val _searchText = MutableStateFlow( value: "")
    val searchText = _searchText.asStateFlow()

    private val _isSearching = MutableStateFlow( value: false)
    val isSearching = _isSearching.asStateFlow()

    private var _books = MutableStateFlow(book_data_base)
```

So the first variable is responsible for updating the value of the _searchText that initially its empty, and then a public read-only state flow named searchText is created using the asStateFlow() extension function, which provides read-only access to the _searchText flow. The second variable its responsible to tell us if the user its searching or not defining initially as false because no one its searching and we do the same thing as the first variable creating a new variable with the asStateFlow. The third variable its responsible to access the database.

And now we gonna have the search part:



```
    var books = searchText
        .debounce( timeoutMillis: 1000L)
        .combine(_books){ text, books->
            if(text.isBlank()){
                books ^combine
            }else{
//                // Make API call here
                viewModelScope.launch { this: CoroutineScope
                    try {
                        val data = fetchBookData(searchText)
                        // Handle the successful response and access the data
                        val bookListFromSearch = searchRequest2Book(data)
                        println(data?.totalItems)

                         _books.value = bookListFromSearch
                    } catch (e: Exception) {
                        // Handle the API call failure and display an error message or take appropriate actions
                        Log.d( tag: "Error",  msg: "Failed to retrieve data from API: ${e.message}")
                    }
                }
                books.filter { it.doesMatchSearchQuery(text) } ^combine
            }

        }.stateIn(viewModelScope, SharingStarted.WhileSubscribed( stopTimeoutMillis: 5000),
                 initialValue = _books.value)

    fun onSearchTextChange(text : String){
        _searchText.value = text
    }
}
```

First, we have the variable book is gonna receive the value of the search where the searchText it’s the search query, the debounce it’s a delay of 1000 miliseconds so the user can finish typing before the search its trigger, .combine combines the searchText flow and the _books flow allowing to react to changes in both flows, then its gonna verify if the text its blank or not, if its not blank its gonna do a try catch block if there is no error its gonna call the API to retrieve the book data based on the search query if there is error its gonna send a message to the console, then its gonna filter the list and we use the .stateIn basically to convert the flow into stateFlow to refresh automatically to get the latest values, and finally we have the onSearchTextChange function to receive the text the use puts into the search bar.

So now we need to define how its gonna work in the search bar:

```
@Composable
fun SearchScreen(navController: NavController) {

    val viewModel = viewModel<SearchViewModel>()
    val searchText by viewModel.searchText.collectAsState()
    val books by viewModel.books.collectAsState()
    val isSearching by viewModel.isSearching.collectAsState()
```

Here we gonna create these variables to get the values from the SearchViewModel.

```
Column(modifier = Modifier
    .fillMaxSize()
    .padding(16.dp)) { this: ColumnScope

    TextField(
        value = searchText,
        onValueChange = viewModel::onSearchTextChange,
        modifier = Modifier.fillMaxWidth(),
        placeholder = { Text(text = "Search")}
            )
    Spacer(modifier = Modifier.height(20.dp))
```

Here we gonna define the column so the values are styling in a column and then we have the TextField where we gonna receive the value from the searchbar, the onValueChange its to update the value in the viewModel , the modifier its just to fill the width of our screen, then a placeholder and the spacer adds a vertical space with a height of 20.dp between the TextField and the next UI component.

```
if(isSearching){
    Box(modifier = Modifier.fillMaxSize()){ this: BoxScope
        CircularProgressIndicator(modifier = Modifier.align(Alignment.Center))
    }
} else {
    if(searchText.isNotBlank()){
        LazyColumn(modifier = Modifier
            .fillMaxWidth()
            .weight(1f)){ this: LazyListScope

            items(books){ this: LazyItemScope book->
                BookDisplayHome(book = book, navController = navController)
            }
        }
    }
}
```

The first if it just to indicate that the search operation its in progress and if the isSearching its false its gonna verify if the text is blank and if not its gonna use a lazyColumn to show the books that we search.