Lab 08: Networking in Flutter Lab

In this lab, you will expand the shopping list app developed in the handling forms lab by adding networking capabilities. You will use the Provider package for state management and the http package to interact with a server and retrieve, create, update, and delete data.

Prerequisites

Before starting this lab, make sure you have completed the handling forms lab and have the starter code ready. You will also need to install the Provider and http packages, nodejs, json-server, and json-server-auth.

Step 1: Install the Provider and http packages

Start by adding the Provider and http packages to your project's dependencies. Open the terminal tap in Android Studio and run the following command:

```
flutter  pub add provider  http mocktail
```

Step 2: Install nodejs, json-server, and json-server-auth

Next, you will need to install nodejs, json-server, and json-server-auth. Follow the instructions for your operating system to install nodejs. Then open a terminal or command prompt and run the following commands to install json-server and json-server-auth:

```
npm install -g json-server
npm install -g json-server-auth
```

Step 3: Create a db.json file

Now you will create a backend/db.json file that will serve as your mock database. Create a new file named db.json add the following content:

```
{
  "users": [
    {
      "email": "1@example.com",
      "password":
"$2a$10$p3llO4K67K7XzuT4bCJGW.TIGRYLjMQN1RACz.qkcvGSLX3.nUvy2",
      "id": 1
    }
  ],
  "categories": [
    {
      "id": 1,
      "name": "Groceries",
      "icon": 57902
    },
    {
      "id": 2,
      "name": "Frozen and Cooled Foods",
      "icon": 57399
    },
    {
      "id": 3,
      "name": "Cleaning",
      "icon": 57703
    },
    {
      "id": 4,
      "name": "Meats and Fish",
      "icon": 58750
    }
  ],
  "products": [
    {
      "id": 1,
      "name": "Tea Bags",
      "category": {
        "id": 1,
        "name": "Groceries",
        "icon": 57902
      },
      "quantity": 1,
      "isBought": true,
      "userId": 1
    },
    {
      "userId": 1,
```

```
      "id": 2,
      "name": "Orange Juice",
      "category": {
        "id": 2,
        "name": "Frozen and Cooled Foods",
        "icon": 57399
      },
      "quantity": 2,
      "isBought": false
    },
    {
      "userId": 1,
      "id": 3,
      "name": "Detergent",
      "category": {
        "id": 3,
        "name": "Cleaning",
        "icon": 57703
      },
      "quantity": 1,
      "isBought": true
    },
    {
      "userId": 1,
      "id": 1690904048045,
      "name": "Fish",
      "category": {
        "id": 4,
        "name": "Meats and Fish",
        "icon": 58750
      },
      "quantity": 1,
      "isBought": false
    }
  ]
}
```


This file defines three collections: categories, products, and users. The categories collection contains a list of categories with an id and name property. The products collection contains a list of products with an id, name, category, quantity, and isBought property. The users collection contains a list of users with an email and password property.

Step 4: Serve the db.json file

Next, you will use json-server to serve the db.json file. Open the terminal tap then run the following command:

```
json-server-auth --watch backend\db.json --host 0.0.0.0 --port 3000
```

This will start a server on port 3000 that serves the data from your db.json file. You can access the data by opening a web browser and navigating to http://{your_ip}:3000/categories or http://{your_ip}:3000/products.

Step 6: Get the categories from the server

Now that you have set up your server and installed the necessary packages, it's time to update your Flutter app to use the http package to get the data from the server.

Start by removing the hard coded map of categories, list of products, and the enumerator of categories in the shopping_list_provider.dart. Then create a method named getCategories that populates the list of categories from the server.
Here's an example of what this function might look like:

```
import 'dart:convert';
import 'package:http/http.dart' as http;

class ShoppingListProvider extends ChangeNotifier {
final String categoriesUrl;
final String productsUrl;
final String loginUrl;
final String registerUrl;
http.Client? client;


ShoppingListProvider({this.categoriesUrl='http://{your IP}:3000/categories',
this.productsUrl='http://{your IP}:3000/products', this.loginUrl =
'http://{your IP}:3000/login',
this.registerUrl = 'http://{your IP}:3000/register',
this.client,
}) {
  client ??= http.Client();
}

 List<Category> _categories = [];
 List<Product> _products = [];

 List<Category> get categories => _categories;
 List<Product> get products => _products;

 Future<void> getCategories() async {
  final response = await client!.get(Uri.parse(categoriesUrl));
  if (response.statusCode == 200) {
   final categoriesJson = (json.decode(response.body) as List);
   _categories = categoriesJson.map((e) => Category.fromJson(e)).toList();
   return notifyListeners();
```

```
  } else {
    throw Exception('Failed to load categories');
  }
 }
```

This function gets the list of categories from the server. It uses the http package to send requests to the server and then notifies listeners when the state changes. It throws an exception in case of unsuccessful response.

Step 7: Update The model to parse the JSON response.

Next, you need to update your data_models.dart file to implement the Category.fromJson constructor.

```
class Category {
  final int id;
  final String name;
  final IconData icon;

  Category({required this.id, required this.name, required this.icon});
  factory Category.fromJson(Map<String, dynamic> json) {
    return Category(
      id: json['id'],
      name: json['name'],
      icon: IconData(
      json['icon'],
        fontFamily: 'MaterialIcons',
      )
    );
  }
}
```

Step 8: Update AddItemPage

Next, you need to update the AddItemPage class to use the ShoppingListProvider class. Open this class in your project and update it as follows:

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../data/shopping_list_provider.dart';
import '../models/data_models.dart';

class AddItemPage extends StatefulWidget {
  const AddItemPage({super.key});

  @override
  State<AddItemPage> createState() => _AddItemPageState();
}
```

```dart
class _AddItemPageState extends State<AddItemPage> {
 final _formKey = GlobalKey<FormState>();
 final _itemNameController = TextEditingController();
 bool _nameValid = false;
 int _quantity = 1;
 Category? _selectedCategory;
 late Future<void> categoriesFuture;

 @override
 void initState() {
  super.initState();
  categoriesFuture = context.read<ShoppingListProvider>().getCategories();
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
     title: const Text('Add Item'),
    ),
    body: Padding(
     padding: const EdgeInsets.all(8.0),
     child: FutureBuilder<void>(
       future: categoriesFuture,
       builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.done) {
         if (snapshot.hasError) {
          return Center(child: Text('${snapshot.error}'));
         } else {
          return Form(
            autovalidateMode: AutovalidateMode.onUserInteraction,
            key: _formKey,
            child: Column(
              mainAxisAlignment: MainAxisAlignment.spaceAround,
              children: [
               TextFormField(
                 controller: _itemNameController,
                 decoration: InputDecoration(
                 labelText: 'Item Name',
                  hintText: 'Enter Item Name e.g. Tea Bags',
                  border: const OutlineInputBorder(),
                  focusedBorder: const OutlineInputBorder(
                  borderSide: BorderSide(
                    color: Colors.blue,
                   ),
                  ),
                  prefixIcon:
                    const Icon(Icons.emoji_food_beverage),
                 suffixIcon: _nameValid
                   ? const Icon(
                     Icons.check,
```

```
                color: Colors.green,
              )
            : null,
      ),
      onChanged: (value) {
        setState(() {
          _nameValid =
              _formKey.currentState!.validate();
        });
      },
      validator: (value) {
        if (value == null || value.length < 3) {
          return 'Please add an Item Name at least 3 characters';
        }
        return null;
      }),
    DropdownButtonFormField<Category>(
      decoration: const InputDecoration(
      labelText: 'Category',
        helperText: 'Please select a category',
        border: OutlineInputBorder(),
      ),
      value: _selectedCategory,
      items: context
          .read<ShoppingListProvider>()
          .categories
          .map<DropdownMenuItem<Category>>(
              (Category value) {
        return DropdownMenuItem<Category>(
          value: value,
          child: Text(value.name),
        );
      }).toList(),
      onChanged: (value) {
      setState(() {
          _selectedCategory = value;
        });
      },
      validator: (value) {
        if (value == null) {
          return 'Please select a category';
        }
        return null;
      }),
    TextFormField(
      decoration: const InputDecoration(
        labelText: 'Quantity',
        border: OutlineInputBorder(),
      ),
      keyboardType: TextInputType.number,
      initialValue: _quantity.toString(),
      validator: (value) {
```

```
                    if (value == null ||
                        value.isEmpty ||
                        int.tryParse(value) == null ||
                        int.parse(value) <= 0) {
                      return 'Quantity must be a number greater than 0';
                    }
                    return null;
                  },
                  onSaved: (value) {
                    _quantity = int.parse(value!);
                  }),
              ElevatedButton.icon(
                onPressed: () {
                  if (_formKey.currentState!.validate()) {
                    // Add item to shopping list
                    _formKey.currentState!.save();
                    Navigator.of(context).pop(Product(
                      id: DateTime.now().millisecondsSinceEpoch,
                      name: _itemNameController.text,
                      category: _selectedCategory!,
                      quantity: _quantity,
                    ));
                  }
                },
                icon: const Icon(Icons.add_shopping_cart),
                label: const Text('Add Item'),
              ),
            ]));
          }
        }
        return const Center(child: CircularProgressIndicator());
      }),
    ));
  }
}
```

This code uses a FutureBuilder to display a CircularProgressIndicator while the data is being loaded, a Form with form fields once the data is available, and a Text widget with an error message when an error occurs. It also uses the ShoppingListProvider instance provided by the ChangeNotifierProvider widget to interact with the server and update the state of the shopping list. Run the app and navigate to add an item, you should see a loading spinner for a moment before the form is rendered. Also observe the output of the server from the terminal. You should see a get request was sent to /categories every time you navigate to add item page.

Step 9: listing the products from the server

Now les get the list of products in a similar way to getting the categories. Start by adding a factory constructor to the Product class to create a product from a JSON map.

```
class Product {
  final int id;
  final String name;
  final Category category;
  bool isBought;
  final int quantity;

  Product({
    required this.id,
    required this.name,
    required this.category,
    required this.quantity,
    this.isBought = false,
  });

factory Product.fromJson(Map<String, dynamic> json) {
 return Product(
  id: json['id'],
  name: json['name'],
  category: Category.fromJson(json['category']),
  quantity: json['quantity'],
  isBought: json['isBought'],
 );
}
```

Then create a method to get the list of products from the server. In the ShoppingListProvider class add the following:

```
bool _isBusy = false;
bool get isBusy => _isBusy;

String? _error ;
String? get error => _error;

Future<void> getProducts() async {
 _isBusy = true;
 _error = null;
 notifyListeners();
 final response = await http.get(Uri.parse(productsUrl));
 if (response.statusCode == 200) {
  final productsJson = (json.decode(response.body) as List);
  _products = productsJson.map((e) => Product.fromJson(e)).toList();
 } else{
  _error = 'Failed to load products';
 }
 _isBusy = false;
 notifyListeners();
}
```

this code creates a function to get the products from the server, stores it in the list of `Product` objects, and updates boolean values to indicate whether the process is complete or has encountered an error.

Finally update the shoppingListPage to display the products:

```
class _ShoppingListPageState extends State<ShoppingListPage> {
  void _addItem(Product item) {
  context.read<ShoppingListProvider>().addItem(item);
  }
@override
  void initState() {
   super.initState();
   context.read<ShoppingListProvider>().getProducts();
  }
  @override
  Widget build(BuildContext context) {
   return Scaffold( //....
     body: Consumer<ShoppingListProvider>(builder: (context, state, child) {
      return  state.isBusy ? const Center(child: CircularProgressIndicator()) :
      state.error != null ? Center(child: Text(state.error!)) :
       ListView.builder(
       itemCount: state.products.length,
       itemBuilder: (context, index) {
       final item = state.products[index];
       return ListTile(
          leading: Icon(
            item.category.icon,
            color: Colors.blue,
          ),
          title: Text(item.name),
          subtitle: Text('${item.category.name} - ${item.quantity}'),
          trailing: Checkbox(
            value: item.isBought,
            onChanged: (value) {
                state.itemBoughtChanged(item);
            },
          ),
        );
       },
      );
     }),
```

Step 10: Sending Data to the server: create a product

Before sending the data to the server, it is important to format the data in the way that the server accepts. Our server is named json-server and it accepts JSON data. Les start by creating methods to convert the Dart Produc objects to JSON. Add the following methods to both the Category and Product classes respectively

```
Map<String, dynamic> toJson() {
  return {
    'id': id,
    'name': name,
    'icon': icon.codePoint
  };
}
```

```
Map<String, dynamic> toJson() {
  return {
    'id': id,
    'name': name,
    'category': category.toJson(),
    'quantity': quantity,
    'isBought': isBought
  };
}
```

Then update the addItem method in the ShoppingListProvider class:

```
void addItem(Product item) async {
  _error = null;
  _isBusy = true;
  notifyListeners();
  try {
    final response = await client!.post(Uri.parse(productsUrl),
        body: jsonEncode(item.toJson()),
        headers: {
          'Content-Type': 'application/json',
        });
    if (response.statusCode == 201) {
      _products.add(item);
    } else {
      _error = 'Failed to add item! Error: ${response.statusCode}';
    }
  } catch (e) {
    _error = e.toString();
  } finally {
    _isBusy = false;
    notifyListeners();
  }
}
```

The function takes a Product object as a parameter. It sets the _error variable to null, indicating no error initially. It sets the _isBusy variable to true, indicating that the function is currently executing. It notifies the listeners (typically widgets) that the state has

changed. It tries to make a POST request to the productsUrl with the product data encoded as JSON in the request body. If the response status code is 201 (indicating a successful creation), it adds the item to the _products list. If the response status code is not 201, it sets the _error variable with an error message. If an exception occurs during the HTTP request, it sets the _error variable with the exception message (maybe the server is unreachable). Finally, it sets _isBusy to false to indicate that the function has finished executing, and notifies the listeners again. Run the application and observe the terminal when creating a new product it should send a post request to the server.

Step 11: Update data on the server:

To update the product on the server modify the itemBoughtChanged method in the shoppingListProvider class as follows:

```
void itemBoughtChanged(Product item) async {
  _error = null;
  _isBusy = true;
  notifyListeners();
  try {
  item.isBought = !item.isBought;
    final response = await client!.patch(Uri.parse('$productsUrl/${item.id}'),
        body: jsonEncode(item.toJson()),
        headers: {
          'Content-Type': 'application/json',
        });
    if (response.statusCode == 200) {
  _doSorting();

    } else {
      _error = 'Failed to update item! Error: ${response.statusCode}';

    }
  }catch (e) {
    _error = e.toString();
  }
  _isBusy = false;
  notifyListeners();
}
```

the code sends a patch request to update the product status on the server. Run the application and observe the terminal logging a patch request when you tap the checkbox of a product.

Step 12: Making Authenticated requests

Create a new file named backend/routes.json and add the following content:

```
{
  "products": 600,
  "categories": 644
}
```

This file defines custom routes for the /categories and /products endpoints. The json-server-auth middleware uses the 600 prefix to indicate that these routes require authentication and the user must own the resource to write or read the resource. The 644 implies that the user must own the resource to write the resource and everyone can read the resource without requiring login.

Then stop the running server and restart it to apply the authorization restrictions:

```
json-server-auth --watch backend\db.json  --port 3000 --host 0.0.0.0 -
r backend\routes.json
```

if you run the application now, you should get a 401 error message. To fix that les update our application to make authenticated requests. Add the following to the ShoppinListProvider class:

```
String? _token;
int? _userId;
Future<void> _getToken() async {
  if (_token != null) {
    return;
  } else {
    _error = null;
    try {
      final response = await client!.post(
          Uri.parse(loginUrl),
          body: json.encode(
              {'email': '1@example.com', 'password': '12'}), headers: {
            'Content-Type': 'application/json',
          });
      if (response.statusCode == 200) {
        var body = json.decode(response.body);
        _token = body['accessToken'];
        _userId = body['user']['id'];
      } else {
        _error = 'Failed to login with status code ${response.statusCode}';
      }
    } catch (e) {
      _error = e.toString();
    } }
}
```

The code defines two variables _token, and _userId,, and a function _getToken() that retrieves an access token from a server using HTTP POST request. If _token is already set, the function returns early without making a request. If the response status code is 200, indicating a successful login, it parses the response body using json.decode() and assigns the access token and user ID to _token and _userId variables, respectively.

Then to make an authenticated request you need to include the token in the request header. modify the addItem method as follows:

```
Void addItem(Product item) async {
  _error = null;
  _isBusy = true;
  notifyListeners();
  try {
    await _getToken();
    final response = await client!.post(Uri.parse(productsUrl),
        body: json.encode({'userId': _userId, ...item.toJson()}),
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer $_token',
        });
}
```

The body of the request needs to include the ID of the user who added the product. We use the spread operator to combine the Json maps into one then convert it to Json string. Modify the getProducts method:

```
await _getToken();
final response = await client!.get(Uri.parse('$productsUrl?userId=$_userId'),
headers: {
  'Authorization': 'Bearer $_token',
});
```

follow a similar approach to update the other methods. Run the application, you should be able to do the same tasks as before.

Step 13 Simple optimizations:

It is important to minimize the network traffic to save the user`s time and money. One of the techniques is to cache the response and only send the request once. We used this technique in the _getToken method to check if the _token is not null, shortcut the log in connection. As an exercise, do the same for getCategories method.

Our app depends on the list of products, but we only call getProducts when we navigate to ShoppingListPage. We can save the user time if we run this method while starting the app up. In the main.dart file add a call for getProducts when creating the change notifier provider as follows:

```
create: (BuildContext context) => ShoppingListProvider()..getProducts(),
```

The reason for creating the ShoppingListPage as a stateful widget is to use the initState method to call getProducts. Since this is done in the main, the ShoppingListPage can be changed into a stateless widget as follows:

```
class ShoppingListPage extends StatelessWidget{
 const ShoppingListPage({super.key});

 @override
 Widget build(BuildContext context) {
  return Scaffold(//...
    body: Consumer<ShoppingListProvider>(builder: (context, state, child) {
...
trailing: Checkbox(
  value: item.isBought,
  onChanged: (value) {
      state.itemBoughtChanged(item);
  },
),
....

    }),
    floatingActionButton: FloatingActionButton(
     onPressed: () async {
      // Navigate to add item page
      final item =
        await Navigator.of(context).push<Product>(MaterialPageRoute(
       builder: (context) => const AddItemPage(),
      ));
      if (item != null) {
       context.read<ShoppingListProvider>().addItem(item);
      }
     },
     child: const Icon(Icons.add),
    ),
   );
 }
}
```

Exercises:

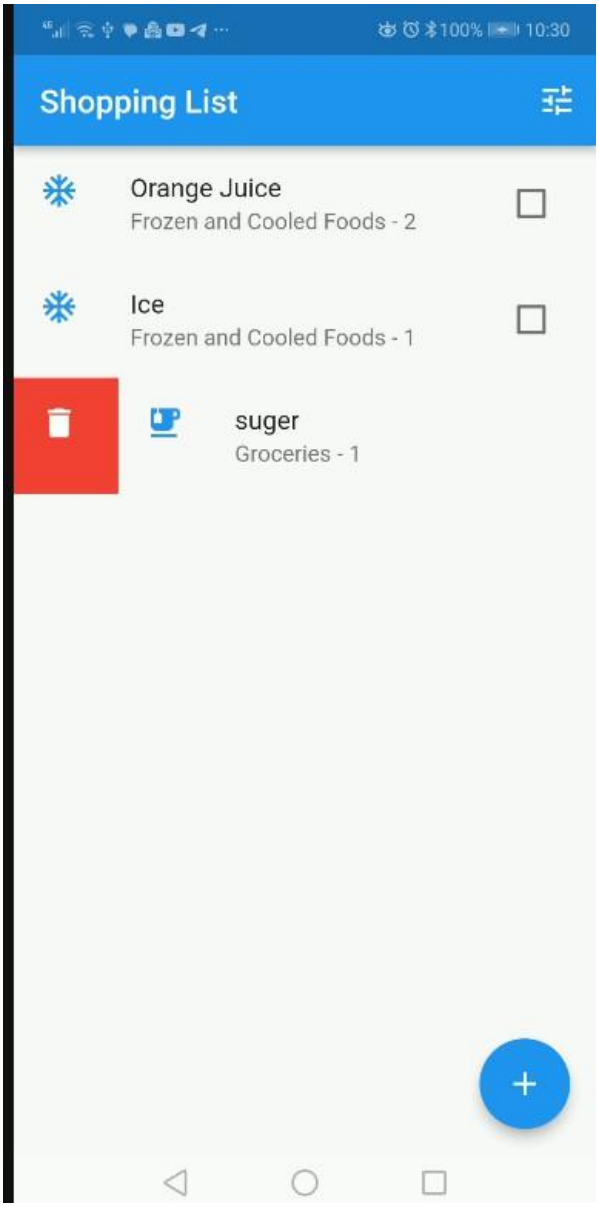
1. As an exercise, try implementing a method for deleting a product from the server in the **removeItem** method, and updating the ShoppingListPage to use a **Dismissible** widget to allow users to swipe to delete products from their shopping list by calling removeItem. Try to make the dragged Dismissible item look like the attached picture below.
2. Add a login page with a form for the user to enter email and password. Create a file `lib/widgets/login_page.dart` that includes a **LoginPage** stateful widget. The form should include an email field, a password field, login button, and login mode button with the following keys: **emailField, passwordField, loginButton, and loginModeButton** respectively. For each field add a key property like `key:Key("emailField")`. The form should have a Boolean property **loginMode** which if true, the page shows 'login' button and 'Don't have an account' button. If

false, the page shows ‘Register’ button and ‘Already have an account’ button. The default is true and toggled when taping the second button. When either the register or login buttons are pressed, validate the form then call the **`Future<bool> login(String email, String password,{bool loginMode = true})`** method on the ShoppingListProvider passing the email, password, and loginMode values. The validation should check for the @ character in the email field and the password field should be at least 3 characters long. Check the return of the login method if it is true, replace the **LoginPage** by the **ShoppingListPage** after calling getProducts. Otherwise, show an error message using a snack bar and remain in LoginPage. In the shopping_list_provider file rename the **_getToken** method to **login** and accept the form content. If the loginMode is true send a post request to ‘/login’ otherwise, send it to ‘/register’. Modify the **getProducts** and other methods to check for the value of _token not null instead of calling _getToken. In the main remove the ..getProducts call and change the home property to LoginPage().

Conclusion

In this lab, you have learned how to expand the shopping list app developed in the handling forms lab by adding networking capabilities and using the Provider package for state management. You have used the http package to interact with a server and retrieve, create, update, and delete data. You have also learned how to handle errors when interacting with a server and how to require and send authorization when accessing protected endpoints.

I hope this lab has helped you understand how to apply the concepts of networking and state management in Flutter and has given you hands-on experience working with servers, APIs, and the Provider package. 😊


100% 10:30
Shopping List
Orange Juice
Frozen and Cooled Foods - 2
Ice
Frozen and Cooled Foods - 1
suger
Groceries - 1