Entropy App

Designer: Kishan Batra

Developer: John Bensen

**Abstract**

The content that is posted on popular social media apps, such as Instagram, Tiktok, and Facebook, is typically distorted versions of reality in one way or another. Content creators typically use filters and photo-editing softwares to create a picture perfect version of their lives. Additionally, some social media platforms require a user to conform to a certain aesthetic or type of content for that user to feel rewarded by the app. The goal of Entropy is to create platform for content creators to fully express themselves

**Software Architecture**

The software architecture is based on a client-server architecture.

Data is either stored in a Postgres database or Google Firestore. Data that is updated frequently and a heavy emphasis on how it relates to other data is stored in the Postgres database. Data that is rarely updated and has very simple relations with other data is stored in Google Firestore.

The backend is currently organized into individual “apps”. Each app functions as an API that sits between the frontend and a section of the database. For example, the “user\_profile” app provides functionality to create a user account, update a user’s preferences, search for other users, etc. The frontend communicates with these apps through a REST API. The long-term goal is to break these apps up into individual microservices, each of which is deployed on a Kubernetes container.

The frontend is built with Flutter/Dart. All the code written for the frontend conforms to a specific design pattern. This design pattern ensures that the code is intuitive and consistent. The app primarily uses Provider for state management.

**Database**

**Backend**

The backend is organized into “apps”, each of which provides an API for a different part of the app. As of right now, these apps are lumped together into a monolithic architecture. However, the long-term goal is to convert each app into a microservice.

The following is a list of each app, and a description of the app and its endpoints. The symbol, “URL”, will denote the IP address of the server.

posts/

userID/posts/

**Frontend**

The following is a visualization of how the files are organized on the frontend:

Models/

models1

models2

BackendAPI/

app1

app2

Section1/

widgets/

widget1

Widget2

page1

page2

Section2/

widgets/

widget1

widget2

widget3

page1

page2

page3

Section3/

widgets/

widget2

page1

globals

Notes:

* In the directory “BackendAPI”, each file provides an API used to interact with an individual app on the backend. For example, “user\_profile.dart” contains all the API used to communicate with the “user\_profile” app on the backend.
* A “page” is everything shown on the screen at the same time. How a page should be structured is shown on the next page.
* The “widgets” directory contains all the stateless widgets that are used throughout the entire section. If a widget is used only in one page, then it should not be in this directory.
* “globals” is a dart file that contains all global variables used throughout the app.

The following is a visualization of how a page is structured:

class PageProvider extends ChangeNotifier {

// All state variables and business logic are stored in this provider.

}

class Page extends StatefulWidget {

// This widget is called to initiate the provider and build the page. All

// the logic needed to initiate this page, e.g. getting data from the

// backend, is performed here. Returns a Scaffold.

}

class \_PageState extends State<Page> {

}

class PagePage extends StatelessWidget {

// Highest widget in the page’s widget tree that doesn’t contain state.

// Controls the layout of the page and positioning of all sub widgets.

}

class PageAppBar extends PreferredSize {

}

class Widget1 extends StatelessWidget {

}

class Widget2 extends StatelessWidget {

}

class Widget2Delegate extends MultiChildLayoutDelegate {

}

class Widget2Button extends StatelessWidget {

}

Notes:

* In the above visualization, “Page” should be the name of the page.
* Every sub widget should be its own class, there should not be any function that returns a Widget.
* The above visualization is a guideline of how to structure a page. The widgets in a page should be ordered in the same way as the visualization, but a page does not need to have every widget shown above.

The following is how a widget should be structured:

class Widget extends StatelessWidget {

Widget({@required this.var1, @required this.var1, @required this.var3});

final var1;

final var2;

final var3;

@override

void initState() {

}

@override

Widget build(BuildContext context) {

return Widget();

}

void performFunction1() {

}

int performFunction2() {

}

}