SEE U

Old Code Review

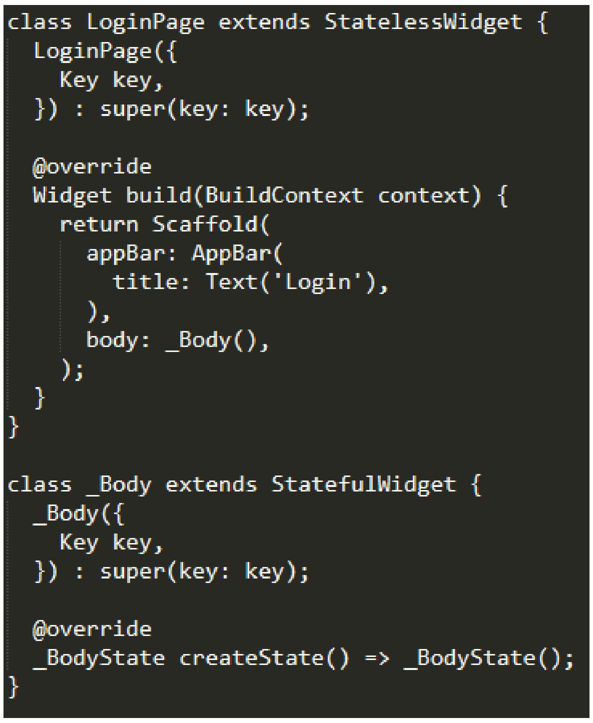
Hang Xie 1025910
Jiasheng Yu 1025957
Rui Chen 1025567
Jinhang Wu 1025891

Version: (1.0) **Date: (05/15/2020)**

Table Content

Login/Register/Login

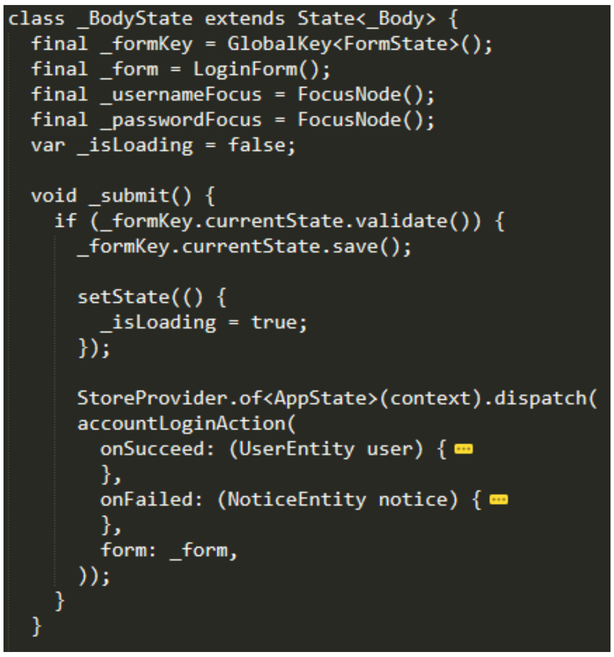
Normally, the user interface of login page should be shown as Figure.1. The users can input their username and password, and then login the system. In the meanwhile, when users type the username and click the “Next” button on the virtual keyboard, the cursor will automatically change to password input box. Additionally, the users can register new accounts by clicking the link “Register New Account”. This page is developed at “login.dart” which is under the library “pages/account”.



```
class LoginPage extends StatelessWidget {
  LoginPage({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Login'),
      ),
      body: _Body(),
    );
  }
}

class _Body extends StatefulWidget {
  _Body({
    Key key,
  }) : super(key: key);

  @override
  _BodyState createState() => _BodyState();
}
```

Comment: This part of codes mainly defines the several basic elements of login page including title and body. The title is text “Login”, and a function _Body() would be loaded in the body of page. In the meanwhile, the function _Body() is the construction method of class _Body. Additionally, the _Body() requires a new method _BodyState() to support.

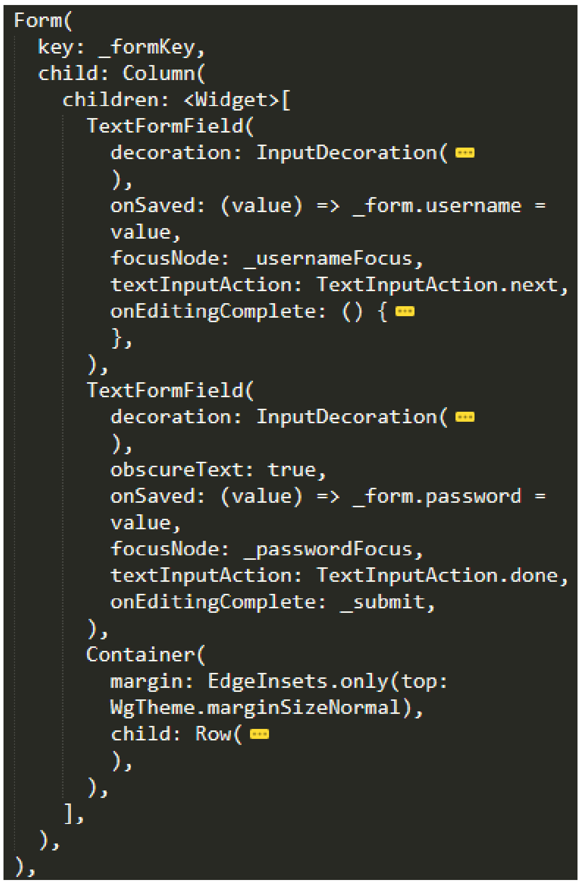
Figure 1. LogIn Page



```
class _BodyState extends State<_Body> {
  final _formKey = GlobalKey<FormState>();
  final _form = LoginForm();
  final _usernameFocus = FocusNode();
  final _passwordFocus = FocusNode();
  var _isLoading = false;

  void _submit() {
    if (_formKey.currentState.validate()) {
      _formKey.currentState.save();

      setState(() {
        _isLoading = true;
      });

      StoreProvider.of<AppState>(context).dispatch(
      accountLoginAction(
        onSucceed: (UserEntity user) {⋯
        },
        onFailed: (NoticeEntity notice) {⋯
        },
        form: _form,
      ));
    }
  }
```

Comment: This part of codes describes the class _BodyState. In this class, it defines some variables in order to manage the form and page. In the meanwhile, it provides a function _submit() which set the action of form and the page. When users click the "Login" button, the form which contains username and password information will be uploading and page will display the uploading state for users.

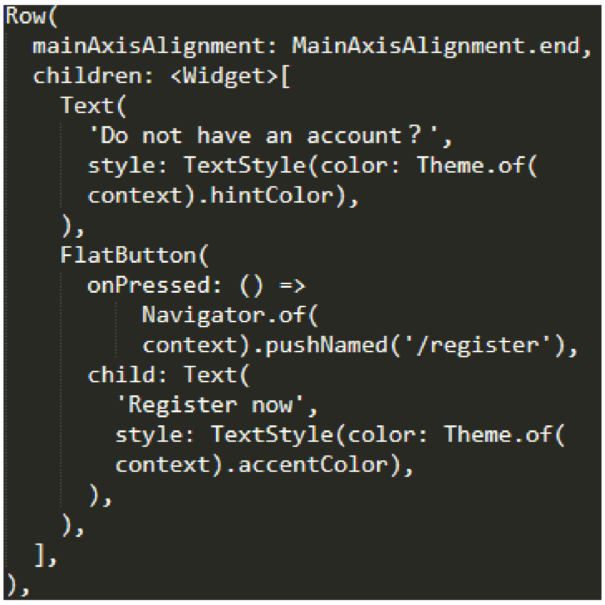
Figure 2: _BodyState class



```
Form(
  key: _formKey,
  child: Column(
    children: <Widget>[
      TextFormField(
        decoration: InputDecoration(⋯
        ),
        onSaved: (value) => _form.username =
        value,
        focusNode: _usernameFocus,
        textInputAction: TextInputAction.next,
        onEditingComplete: () {⋯
        },
      ),
      TextFormField(
        decoration: InputDecoration(⋯
        ),
        obscureText: true,
        onSaved: (value) => _form.password =
        value,
        focusNode: _passwordFocus,
        textInputAction: TextInputAction.done,
        onEditingComplete: _submit,
      ),
      Container(
        margin: EdgeInsets.only(top:
        WgTheme.marginSizeNormal),
        child: Row(⋯
        ),
      ),
    ],
  ),
),
```

Comment: This part of codes work for designing the input boxes and login button. The first TextFormField represents the username input box; the second TextFormField works for the password input box; the Container defines the attributes of login button. When clicking the button, the program will jump to the _submit() function which is in the _BodyState class.

Figure 3: Form defining the input box and login button

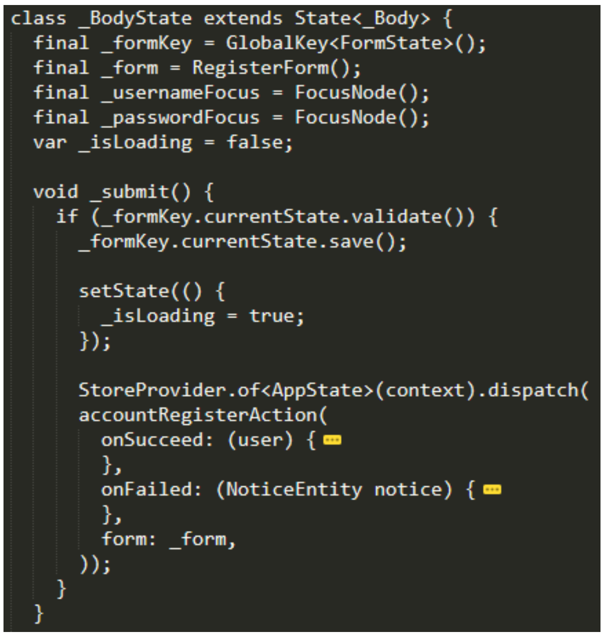


```
Row(
  mainAxisAlignment: MainAxisAlignment.end,
  children: <Widget>[
    Text(
      'Do not have an account？',
      style: TextStyle(color: Theme.of(
      context).hintColor),
    ),
    FlatButton(
      onPressed: () =>
          Navigator.of(
          context).pushNamed('/register'),
      child: Text(
        'Register now',
        style: TextStyle(color: Theme.of(
        context).accentColor),
      ),
    ),
  ],
),
```

Comment: This part of codes work for designing the register link. The Text is a message which reminds the users. The FlatButton provides a link which will jump to Register page.

Figure 4: Router for registration

Similar to Login page, the Register page has the same user interface. The only difference is the action after clicking the Register button.



```
class _BodyState extends State<_Body> {
  final _formKey = GlobalKey<FormState>();
  final _form = RegisterForm();
  final _usernameFocus = FocusNode();
  final _passwordFocus = FocusNode();
  var _isLoading = false;

  void _submit() {
    if (_formKey.currentState.validate()) {
      _formKey.currentState.save();

      setState(() {
        _isLoading = true;
      });

      StoreProvider.of<AppState>(context).dispatch(
      accountRegisterAction(
        onSucceed: (user) {⋯
        },
        onFailed: (NoticeEntity notice) {⋯
        },
        form: _form,
      ));
    }
  }
```

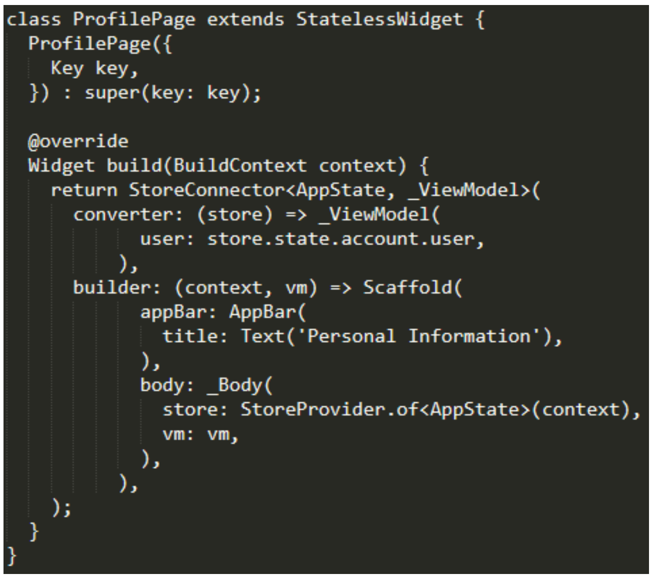
Comment: This part of codes defines the action after clicking the Register button. Compare with Login page, the action accountLoginAction() is replaced by accountRegisterAction(). Both of two action is defined in the account.dart which is under library "action"

Figure 5: Action after clicking the Register button

When users log in the system successfully, they can logout the system. The Logout button is displayed in the "me.dart". After clicking it, the application will do the function accountLoginAction() which is in the file account.dart.

User Information Edit

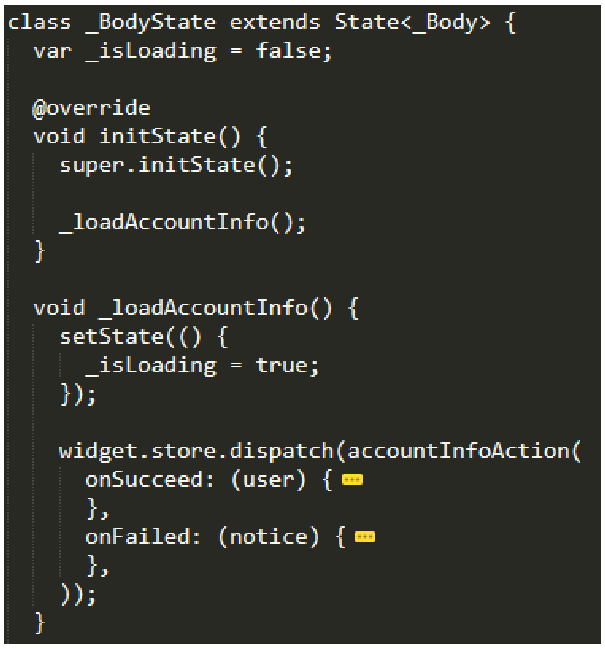
The user information has only two parts in this system which are username and phone number. It will display these information in the page “profile.dart” which is under library “pages/account”. However, this system only provides the function to edit one information once. It does not support change several things in the same time.



```
class ProfilePage extends StatelessWidget {
  ProfilePage({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return StoreConnector<AppState, _ViewModel>(
      converter: (store) => _ViewModel(
            user: store.state.account.user,
          ),
      builder: (context, vm) => Scaffold(
            appBar: AppBar(
              title: Text('Personal Information'),
            ),
            body: _Body(
              store: StoreProvider.of<AppState>(context),
              vm: vm,
            ),
          ),
    );
  }
}
```

Comment: Similar to the Login and Register pages. The user information page defines several basic elements and the body of page is displayed by the function _Body(). Similarly, the function _Body() is the construction method of class _Body. Additionally, the _Body() requires a new method _BodyState() to support.

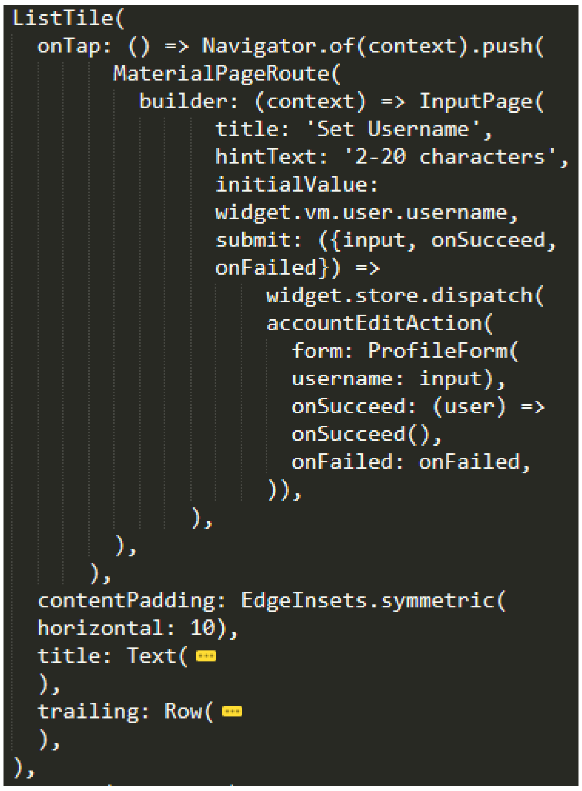
Figure 6: User Information page.



```
class _BodyState extends State<_Body> {
  var _isLoading = false;

  @override
  void initState() {
    super.initState();

    _loadAccountInfo();
  }

  void _loadAccountInfo() {
    setState(() {
      _isLoading = true;
    });

    widget.store.dispatch(accountInfoAction(
      onSucceed: (user) {⋯
      },
      onFailed: (notice) {⋯
      },
    ));
  }
```

Comment: This part of codes mainly describes the class _BodyState. When users enter the user information page, the username and phone number will be loaded through the function accountInfoAction() which is defined in the file “account.dart”.

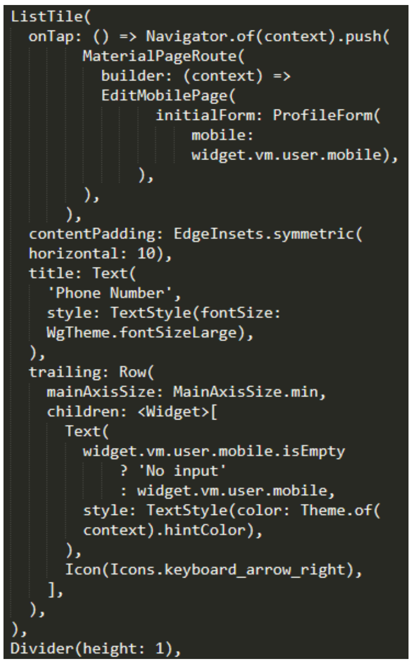
Figure 7: _BodyState class



```
ListTile(
  onTap: () => Navigator.of(context).push(
        MaterialPageRoute(
          builder: (context) => InputPage(
                title: 'Set Username',
                hintText: '2-20 characters',
                initialValue:
                widget.vm.user.username,
                submit: ({input, onSucceed,
                onFailed}) =>
                    widget.store.dispatch(
                    accountEditAction(
                      form: ProfileForm(
                      username: input),
                      onSucceed: (user) =>
                      onSucceed(),
                      onFailed: onFailed,
                    )),
              ),
        ),
      ),
  contentPadding: EdgeInsets.symmetric(
  horizontal: 10),
  title: Text( ⋯
  ),
  trailing: Row( ⋯
  ),
),
```

Figure 8: Design the User Information page

Comment: This part of codes works for design the user interface of the user information page. In this part, it displays the phone number. However, different from changing username, the phone number will be passed to EditMobilePage which is in the file "edit_mobile.dart" under library "pages/account". After that, the users can input a new mobile number and the system will send verify code to the phone. This function is defined in "edit_mobile.dart". After that, if the verify code is correct, the number will be changed through the method accountEditAction().



```
ListTile(
  onTap: () => Navigator.of(context).push(
        MaterialPageRoute(
          builder: (context) =>
          EditMobilePage(
                initialForm: ProfileForm(
                  mobile:
                  widget.vm.user.mobile),
              ),
        ),
      ),
  contentPadding: EdgeInsets.symmetric(
  horizontal: 10),
  title: Text(
    'Phone Number',
    style: TextStyle(fontSize:
    WgTheme.fontSizeLarge),
  ),
  trailing: Row(
    mainAxisSize: MainAxisSize.min,
    children: <Widget>[
      Text(
        widget.vm.user.mobile.isEmpty
            ? 'No input'
            : widget.vm.user.mobile,
        style: TextStyle(color: Theme.of(
        context).hintColor),
      ),
      Icon(Icons.keyboard_arrow_right),
    ],
  ),
),
Divider(height: 1),
```

Figure 9: User Information Page

Design

We think the original design for posting moments, exhibiting moments, like posts and deleting posts is proper. The interactions are meaningful and the changes are codebase. Honestly, this application can be run on both Android and IOS devices. Because our group cannot implement the Flutter environment properly, so we think this is not the good time for us to add new functions. Moreover, the original code has nearly accomplished all the functions.

However, because we changed the IDE and developmental environment, we have to change the design of sending the data request to the back-end.

Functionality

Post moments

We think this function is developed successfully. Although our group cannot implement the code under Flutter developmental environment, the original group gives us the .apk file and we installed it. From the perspective of programmers, the code is not friendly to us but from the perspective of users, the application can get high ranks because they don't need to see the source code.

The edge case we met is the developmental environment issue. Our group cannot implement the environment properly so we cannot test the code but the mobile application works well.

Because we cannot revise, we want to explain some details of the original source code to let you have a brief idea about this function.

This function was developed at "publish.dart". The library "image_picker" is used to access users' photos and videos on their mobile phones. The below code is one of the key parts of the function which is called the StoreConnector. This generic object can protect the data if the app is closed accidentally.

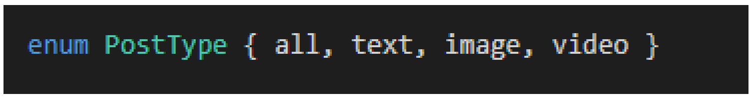
```
StoreConnector<AppState, _ViewModel>(
    converter: (store) => _ViewModel(
        type: store.state.publish.type,
        text: store.state.publish.text,
        images: store.state.publish.images,
        videos: store.state.publish.videos,
      ),
```

Figure 11: StoreConnector code

Another import component in this function is PopupMenuButton. This one can help users change the type of moments they want to post. switchType() method is responsible for this function and text, image, and video types are defined.

```
PopupMenuButton<String>(
          onSelected: (value) => _bodyKey.currentState.switchType(
             PostType.values.firstWhere(
              (v) => v.toString() == value,
             ),
            ),
          initialValue: PostType.image.toString(),
          itemBuilder: (context) => PostType.values
            .skip(1)
            .map<PopupMenuEntry<String>>((v) => PopupMenuItem<String>(
               value: v.toString(),
               child: Text(PostEntity.typeNames[v]),
              ))
            .toList(),
```

Figure 12: PopupMenuButton code



```
enum PostType { all, text, image, video }
```

Figure 13: PostType Defination

Next, I want to talk about the _removeFile method. This method can dynamically decide the moment type and you can click the right corner cross to delete it.

```
_removeFile(File file) {
  if (widget.vm.type == PostType.image) {
   widget.store.dispatch(PublishRemoveImageAction(
    image: file.path,
   ));
  } else if (widget.vm.type == PostType.video) {
   widget.store.dispatch(PublishRemoveVideoAction(
    video: file.path,
   ));
  }
 }
```

Figure 14: removeFile code

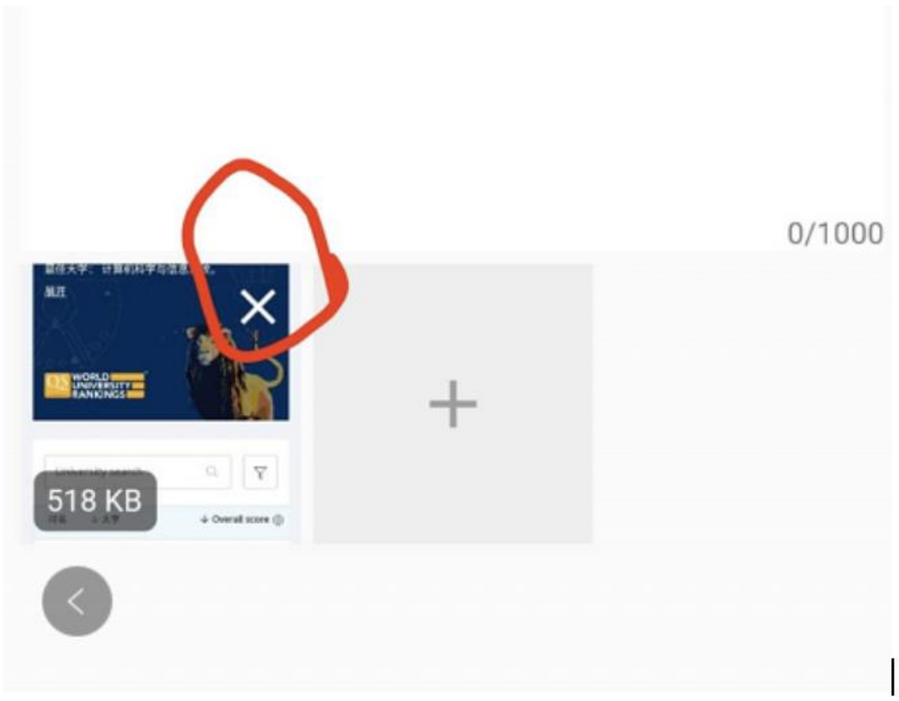


Figure 15: Remove the selected file

Also, in the old source old, the maximum photos are restricted within 6.

```
if (widget.vm.images.length < 6) {
        children.add(GestureDetector(
          onTap: Feedback.wrapForTap(_addFile, context),
          child: Container(
            width: width,
            height: height,
            color: WgTheme.greyLight,
            child: Center(
              child: Icon(
                Icons.add,
                color: WgTheme.greyNormal,
                size: 32,
              ),
            ),
          ),
        ));
      }
```

Figure 16: The code for restriction of the number of images

Exhibiting moments

For the same reason, I want to show you some functions built by the original code.

CircleAvatar() function is used to display the user’s avatar. Users can choose the avatar on their own or use the default one.

```
child: CircleAvatar(
          radius: 15,
          backgroundImage: vm.creator.avatar == ''
              ? null
              : CachedNetworkImageProvider(vm.creator.avatar),
          child: vm.creator.avatar == '' ? Icon(Icons.person) : null,
        ),
```

Figure 17: The code for user avatar

This part is for username, "5" means the largest digit that can be shown next to the avatar.
If the number of username digits beyond 5, the exceeded part will be shown like “….”

```
padding: EdgeInsets.all(5),
            child: Text(
              vm.creator.username,
              overflow: TextOverflow.ellipsis,
              style: TextStyle(
                color: Theme.of(context).accentColor,
                fontSize: WgTheme.fontSizeLarge,
```

Figure 18: The code for username

The below part is for showing the time.

```
Text(
      widget.post.createdAt.toString().substring(0, 16),
      style: TextStyle(color: Theme.of(context).hintColor),
    ),
```

Figure 19: The code for showing the post time

The following part is for image or video display.

```
return Wrap(
    spacing: margin,
    runSpacing: margin,
    children: images
        .asMap()
        .entries
        .map<Widget>((entry) => GestureDetector(
            onTap: Feedback.wrapForTap(
              () => Navigator.of(context).push(MaterialPageRoute(
                  builder: (context) => ImagesPlayerPage(
                      images: images
                          .map<ImageEntity>((image) => image.original)
```

```
                    .toList(),
                  initialIndex: entry.key,
                ),
            )),
          context,
        ),
```

Figure 20: The code for showing the videos or images

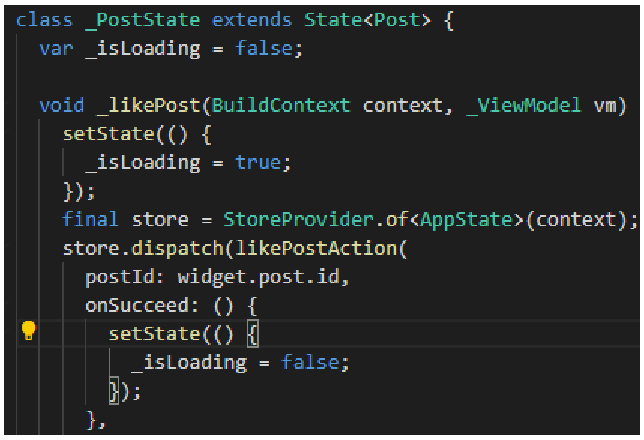
Likes posts

Once the likes button is hit, a heart icon will become solid. “widget.post.isLiked” is used to check the status of likes button. When the user clicks the likes button again, the liked will be removed and this is what “unlikePost” do.

```
widget.post.isLiked
          ? GestureDetector(
              onTap: Feedback.wrapForTap(
                () => _unlikePost(context, vm),
                context,
              ),
              child: Container(
                padding: EdgeInsets.all(5),
                child: Icon(
                  Icons.favorite,
                  size: 20,
                  color: WgTheme.redLight,
                ),
              ),
            )
```

Figure 21: The code for showing the likes button

Once the moment is liked, an asynchronous method “_likePost” is called. And the application will show you the loading information.



```
class _PostState extends State<Post> {
  var _isLoading = false;

  void _likePost(BuildContext context, _ViewModel vm)
    setState(() {
      _isLoading = true;
    });
    final store = StoreProvider.of<AppState>(context);
    store.dispatch(likePostAction(
      postId: widget.post.id,
      onSucceed: () {
        setState(() {
          _isLoading = false;
        });
      },
```

Figure 22: The asynchronous method "_likePost" to react to the operation of the like

The storeConnector in pages/post/posts_liked.dart, shows different users' liked moments. If some users don't like any moment before, it will return null.

```
return StoreConnector<AppState, _ViewModel>(
    converter: (store) => _ViewModel(
        userId: userId,
        postsLiked: (store.state.post.postsLiked[userId.toString()] ?? [])
            .map<PostEntity>((v) => store.state.post.posts[v.toString()])
            .toList(),
      ),
```

Figure 23: StoreConnecttor code

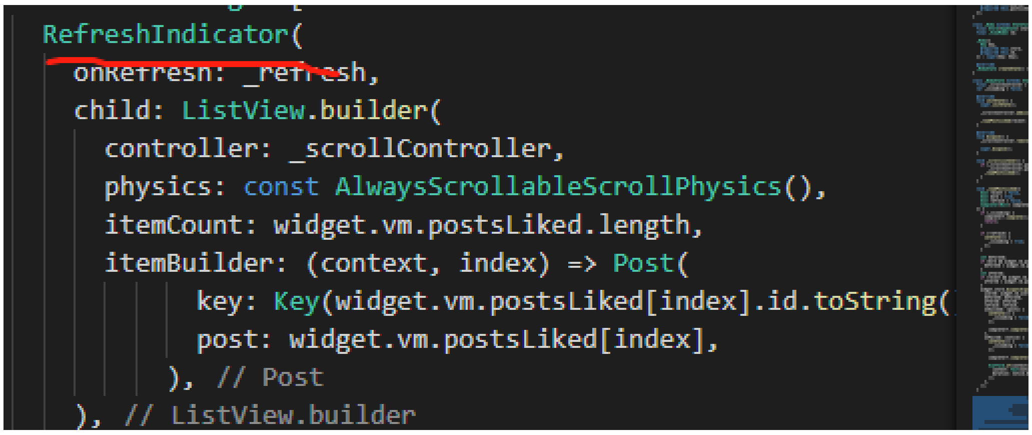
_scrollListener can help user scroll down the list.

```
void _scrollListener() {
    if (_scrollController.position.pixels ==
        _scrollController.position.maxScrollExtent) {
      _loadPostsLiked();
    }
  }
```

Figure 24: _scrollListener code

_refresh method helps the user refresh the moments when they scroll down the page. The completer.future object will be sent to RefreshIndicator and the ListView will show the new posts.

```
 Future<Null> _refresh() {
   final completer = Completer<Null>();
   _loadPostsLiked(
     more: false,
     refresh: true,
     completer: completer,
   );
   return completer.future;
 }
```



```
RefreshIndicator(
  onRefresh: _refresh,
  child: ListView.builder(
    controller: _scrollController,
    physics: const AlwaysScrollableScrollPhysics(),
    itemCount: widget.vm.postsLiked.length,
    itemBuilder: (context, index) => Post(
          key: Key(widget.vm.postsLiked[index].id.toString(
          post: widget.vm.postsLiked[index],
        ), // Post
  ), // ListView.builder
```

Figure 25: The refresh method

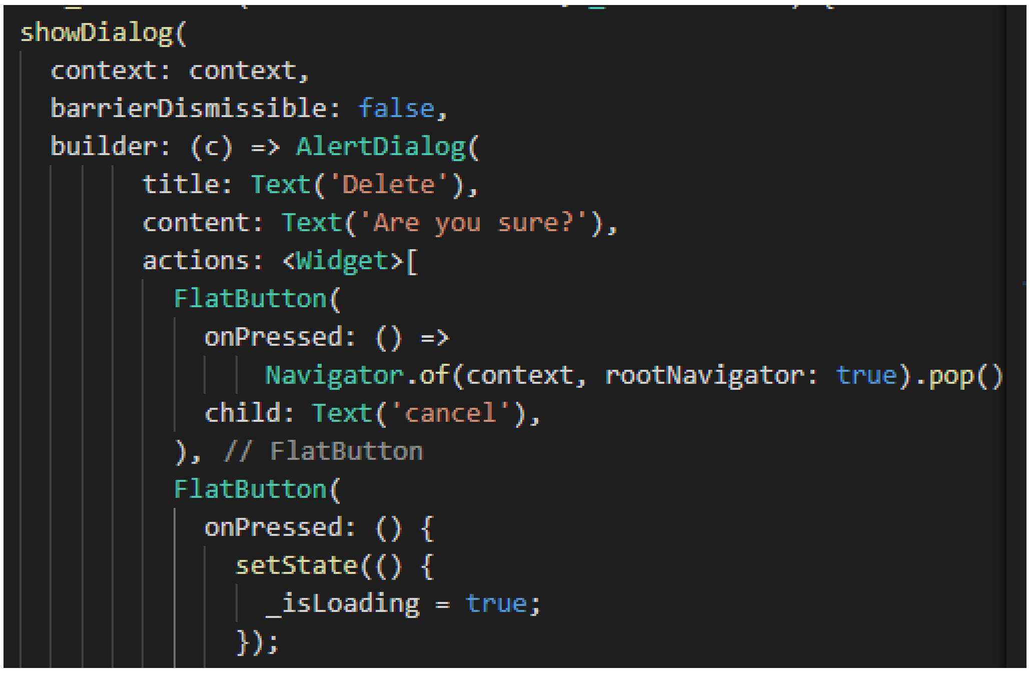
Delete posts

The following function is used to display the delete model. We want to mention that only if the creatorID == user.id, the delete model can be visible.

```
Visibility(
          visible: widget.post.creatorId == vm.user.id,
          child: GestureDetector(
            onTap: Feedback.wrapForTap(
              () => _deletePost(context, vm),
              context,
            ),
            child: Container(
              padding: EdgeInsets.symmetric(horizontal: 10, vertical: 5),
              child: Icon(
                Icons.delete_outline,
                size: 20,
                color: Theme.of(context).hintColor,
              ),
```

Figure 26: The visible method

Once the _deletePost method is called, it will go to the method body.

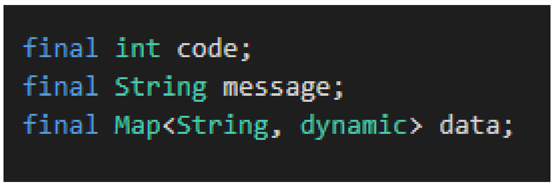


```
showDialog(
  context: context,
  barrierDismissible: false,
  builder: (c) => AlertDialog(
        title: Text('Delete'),
        content: Text('Are you sure?'),
        actions: <Widget>[
          FlatButton(
            onPressed: () =>
                Navigator.of(context, rootNavigator: true).pop()
            child: Text('cancel'),
          ), // FlatButton
          FlatButton(
            onPressed: () {
              setState(() {
                _isLoading = true;
              });
```

Figure 27: _deletePost method body

Ask for the reply from servlet

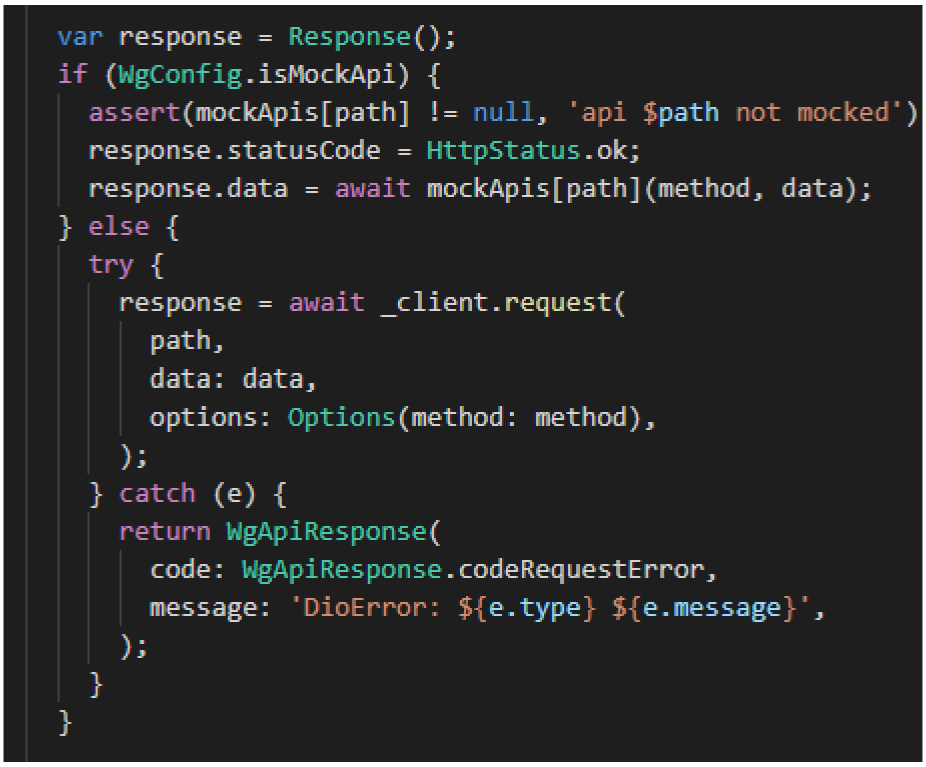
The basic part of the back-end response is designed as the class "WgApiResponse”. Integer code is for status code, the message is used for description, and data is for business data.



```
final int code;
final String message;
final Map<String, dynamic> data;
```

Figure 28: Data fields for back-end response

When the original group test the code the above part of the below code was called. When the apk is export, the “try” part is used.



```
var response = Response();
if (WgConfig.isMockApi) {
  assert(mockApis[path] != null, 'api $path not mocked')
  response.statusCode = HttpStatus.ok;
  response.data = await mockApis[path](method, data);
} else {
  try {
    response = await _client.request(
      path,
      data: data,
      options: Options(method: method),
    );
  } catch (e) {
    return WgApiResponse(
      code: WgApiResponse.codeRequestError,
      message: 'DioError: ${e.type} ${e.message}',
    );
  }
}
```

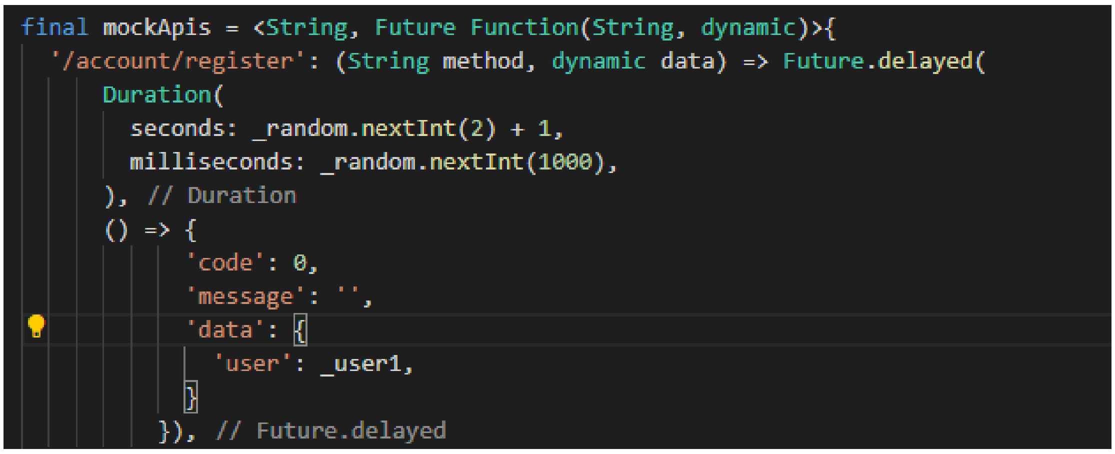
Figure 29: response part

The below methods are predefined when the requests are sent. It will automatically decide which method should be invoked.

```
Future<WgApiResponse> get(String path, {Map<String, dynamic> data}) async {
    return request('GET', path, data: data);
  }

  Future<WgApiResponse> post(String path, {Map<String, dynamic> data}) async {
    return request('POST', path, data: data);
  }

  Future<WgApiResponse> postForm(String path, {FormData data}) async {
    return request('POST', path, data: data);
  }
```

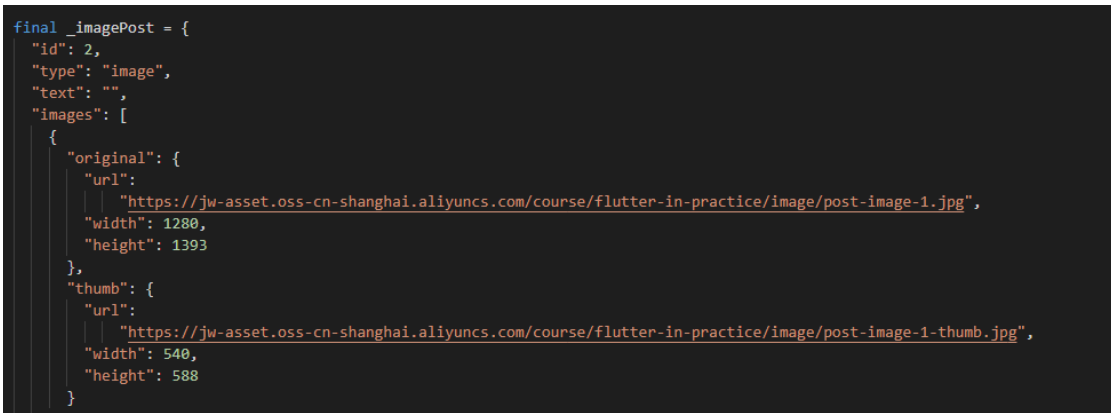
Figure 30: Encapsulation method

The basic structure of one method “register”. Future.delayed() is used to simulate the time that real user needs to wait when they register.



```
final mockApis = <String, Future Function(String, dynamic)>{
  '/account/register': (String method, dynamic data) => Future.delayed(
      Duration(
        seconds: _random.nextInt(2) + 1,
        milliseconds: _random.nextInt(1000),
      ), // Duration
      () => {
            'code': 0,
            'message': '',
            'data': {
              'user': _user1,
            }
          }), // Future.delayed
```

Figure 31: Encapsulated API structure



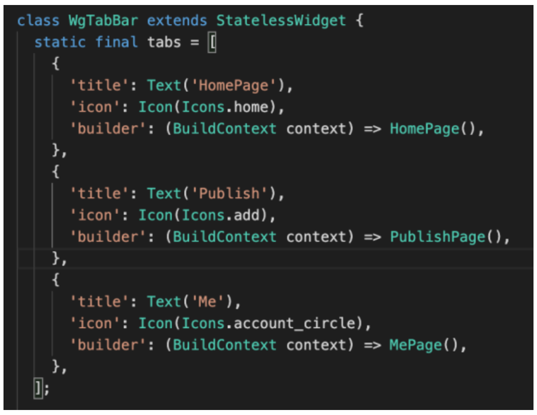
```
final _imagePost = {
  "id": 2,
  "type": "image",
  "text": "",
  "images": [
    {
      "original": {
        "url":
            "https://jw-asset.oss-cn-shanghai.aliyuncs.com/course/flutter-in-practice/image/post-image-1.jpg",
        "width": 1280,
        "height": 1393
      },
      "thumb": {
        "url":
            "https://jw-asset.oss-cn-shanghai.aliyuncs.com/course/flutter-in-practice/image/post-image-1-thumb.jpg",
        "width": 540,
        "height": 588
      }
```

Figure 32: Test example of image post

Routers

We all know that for a software, it is important to have a router to navigate between different pages. In the old SeeU code, we are glad to know that it adopts several strategies to navigate.

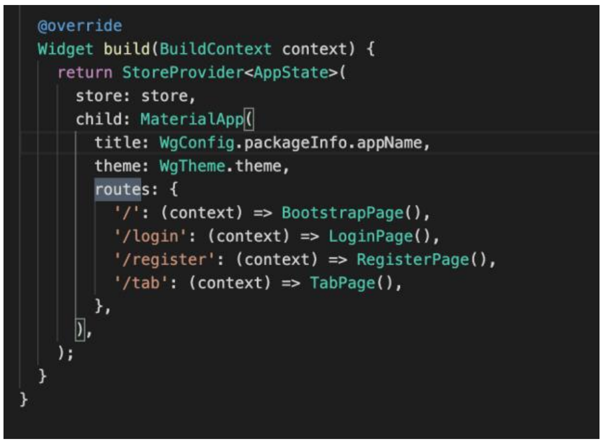
For pages that requires navigation between different tabs, this group creates a component called WgTabBar that defines the route to different pages.



```
class WgTabBar extends StatelessWidget {
  static final tabs = [
    {
      'title': Text('HomePage'),
      'icon': Icon(Icons.home),
      'builder': (BuildContext context) => HomePage(),
    },
    {
      'title': Text('Publish'),
      'icon': Icon(Icons.add),
      'builder': (BuildContext context) => PublishPage(),
    },
    {
      'title': Text('Me'),
      'icon': Icon(Icons.account_circle),
      'builder': (BuildContext context) => MePage(),
    },
  ];
```

Figure 33: WgTabBar Tabs

Besides, another approach that this group adopts is the traditional routing, where developers manually define the path, as Figure 34 shows.



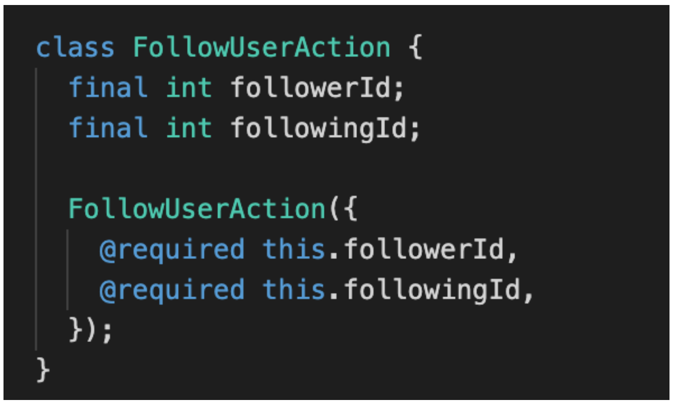
```
@override
Widget build(BuildContext context) {
  return StoreProvider<AppState>(
    store: store,
    child: MaterialApp(
      title: WgConfig.packageInfo.appName,
      theme: WgTheme.theme,
      routes: {
        '/': (context) => BootstrapPage(),
        '/login': (context) => LoginPage(),
        '/register': (context) => RegisterPage(),
        '/tab': (context) => TabPage(),
      },
    ),
  );
}
}
```

Figure 34: Traditional Routers

From the code they developed, we know that they have a great grasp of how router works and could use the most accurate routing method under different situation.

Follow User Function

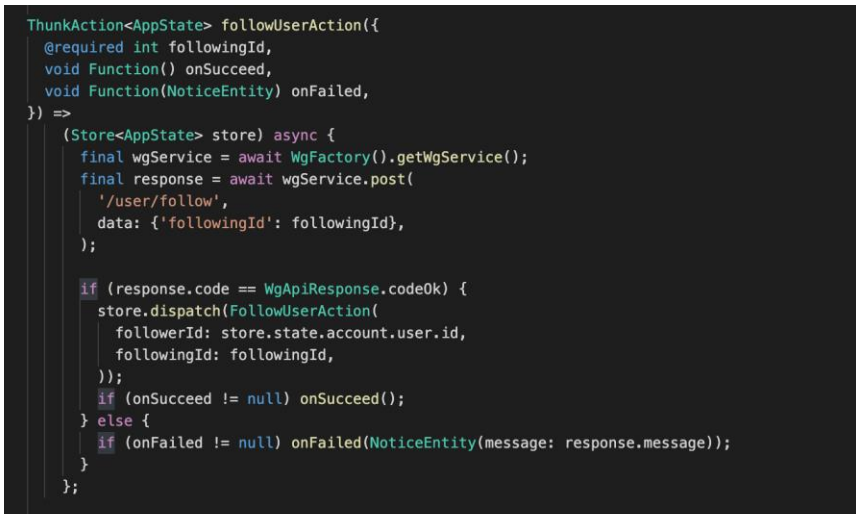
SeeU also provides the function for a user to follow another user. The logic behind this function is simple, where they define a class called FollowUserAction which requires two parameters, followerId and followingId, like Figure 35 shows.



```
class FollowUserAction {
  final int followerId;
  final int followingId;

  FollowUserAction({
    @required this.followerId,
    @required this.followingId,
  });
}
```

Figure 35: Follow User Function

After, when a user is pressed the follow button, the system will call the followUserAction, where it takes the current user's id and the following id, waits till the response concerning whether this process succeeds or not, and provide notification to the user. Although this process is relatively simple, previous group did a great job illustrating their logic and didn't write redundant code. For people who is interested in the implementation, they can check out the Figure 26 for detail.

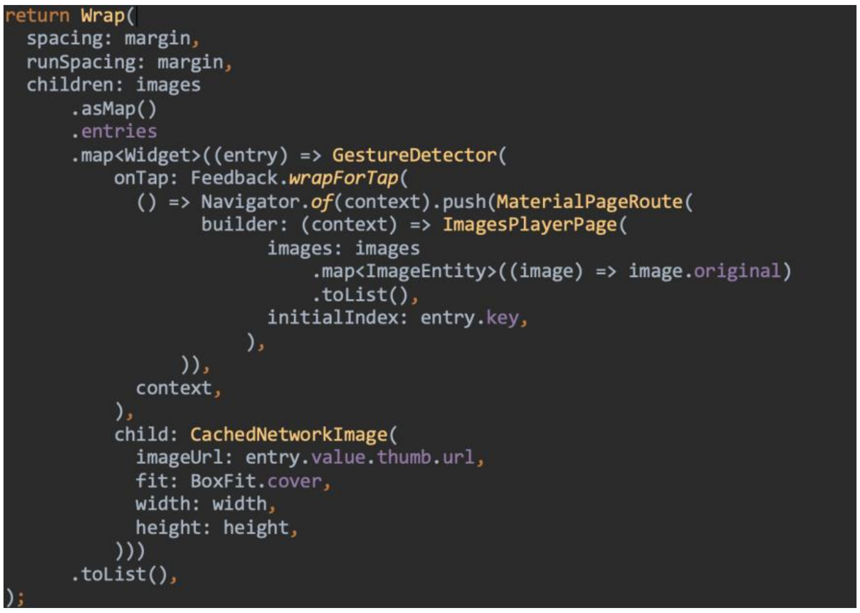


```
ThunkAction<AppState> followUserAction({
  @required int followingId,
  void Function() onSucceed,
  void Function(NoticeEntity) onFailed,
}) =>
    (Store<AppState> store) async {
      final wgService = await WgFactory().getWgService();
      final response = await wgService.post(
        '/user/follow',
        data: {'followingId': followingId},
      );

      if (response.code == WgApiResponse.codeOk) {
        store.dispatch(FollowUserAction(
          followerId: store.state.account.user.id,
          followingId: followingId,
        ));
        if (onSucceed != null) onSucceed();
      } else {
        if (onFailed != null) onFailed(NoticeEntity(message: response.message));
      }
    };
```

Figure 36: The Implementation of Follow a User

The implementation of Unfollowing function is basically the same as the Following function. Due to the space limitation, we are not going to review this part.

Picture Slideshow Function



```
return Wrap(
  spacing: margin,
  runSpacing: margin,
  children: images
      .asMap()
      .entries
      .map<Widget>((entry) => GestureDetector(
          onTap: Feedback.wrapForTap(
            () => Navigator.of(context).push(MaterialPageRoute(
                  builder: (context) => ImagesPlayerPage(
                        images: images
                            .map<ImageEntity>((image) => image.original)
                            .toList(),
                        initialIndex: entry.key,
                      ),
                )),
            context,
          ),
          child: CachedNetworkImage(
            imageUrl: entry.value.thumb.url,
            fit: BoxFit.cover,
            width: width,
            height: height,
          )))
      .toList(),
);
```

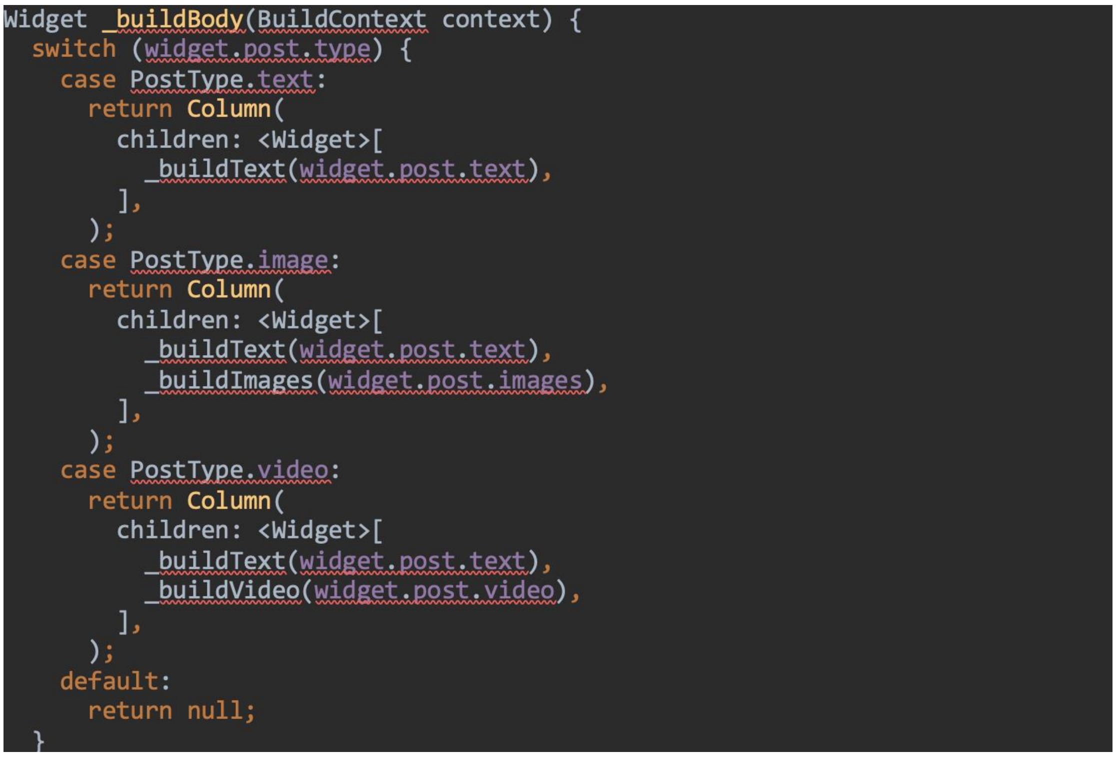
Figure 37: The implementation of Picture Slideshow

It is good to use a third-party repository "cached_network_image" ('package:cached_network_image/cached_network_image.dart';) to implement the function of picture slideshow so that we can save the data and improve the performance.

But some specific situation needs to be cared, such as there are too many pictures shall be showed and each of them is big. It is better to cache the picture and use carousel_slider to show the thumbnail by default. And it shall make a judgement when user import some images into _buildImages.

Video Player Function

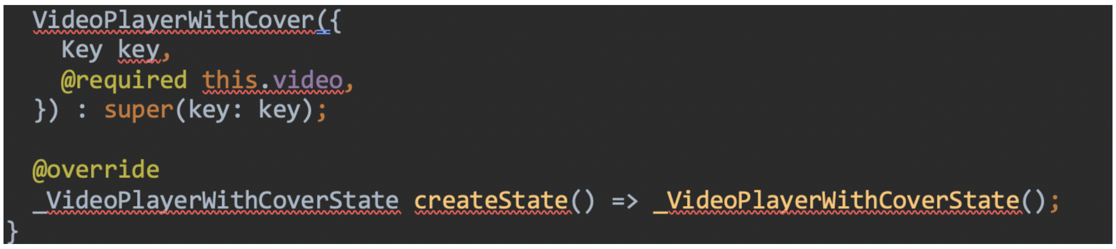
Using a third-party plugin video_player ('../common/video_player.dart') to implement the function of a video player is a good idea, which makes it easier to play the video without spending extra time to build a player, and this plugin is maintained by the official flutter team, which means it can have a better compatibility.



```
Widget _buildBody(BuildContext context) {
  switch (widget.post.type) {
    case PostType.text:
      return Column(
        children: <Widget>[
          _buildText(widget.post.text),
        ],
      );
    case PostType.image:
      return Column(
        children: <Widget>[
          _buildText(widget.post.text),
          _buildImages(widget.post.images),
        ],
      );
    case PostType.video:
      return Column(
        children: <Widget>[
          _buildText(widget.post.text),
          _buildVideo(widget.post.video),
        ],
      );
    default:
      return null;
  }
```

Figure 38: Implementation of the video player

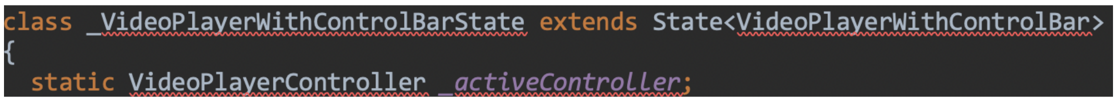
However, as Figure 39 shows, the name _VideoPlayerWithCoverState is too long. Although it fully communicates what the item is or does, it is not a good name for other developer to read and remember.



```
  VideoPlayerWithCover({
    Key key,
    @required this.video,
  }) : super(key: key);

  @override
  _VideoPlayerWithCoverState createState() => _VideoPlayerWithCoverState();
}
```

Figure 39: Naming Issues

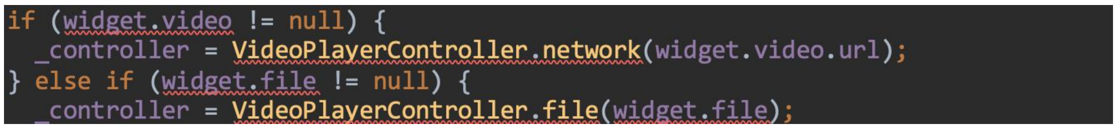
And it needs to be concerned that when user try to open more than one video, the player shall cache the last video's state and then open the new video.



```
class _VideoPlayerWithControlBarState extends State<VideoPlayerWithControlBar>
{
  static VideoPlayerController _activeController;
```

Figure 40: An issue in the code

The if… else if… judgement here is adding complexity to the system without any actual performance benefit that we can see. There is little difference between online videos and local videos. And there's no extra performance benefit to use two kinds of VideoPlayerController, it's better to combine these two codes and add an extra piece of codes to get the meta data for the online videos.



```
if (widget.video != null) {
  _controller = VideoPlayerController.network(widget.video.url);
} else if (widget.file != null) {
  _controller = VideoPlayerController.file(widget.file);
```

Figure 41: An issue in the code

Complexity

We should tell the truth that the Flutter framework is really complex but the old group has tried best to make each function developed in the easiest way.

As we mentioned before, our group cannot implement the environment properly so it's likely to introduce bugs when we try to call or modify this code.

We don't think this project is over-engineering. All the functions and codes are needed. So far, the only issue is our group cannot revise the code. No other issues are found here.

Tests

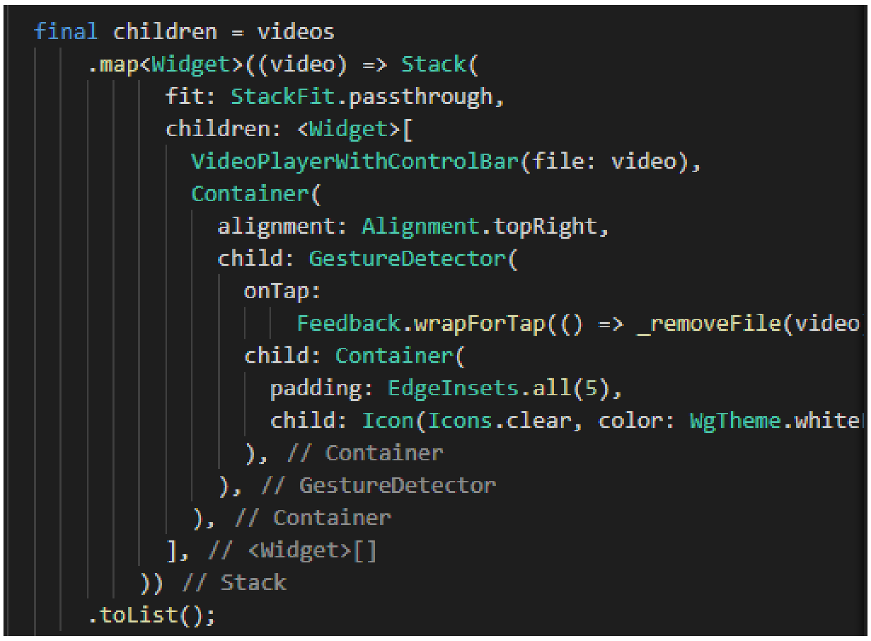
We perform α test, β test, integration test, and end-to-end test. The application performs well but we need to connect our Tumblr account.

Naming

All variables are named formally.

Comments

The comments are not good. Although there are many English comments, very few comments explain why the codes are needed somewhere. I think lots of comments are not necessary, such as the below one:



```
final children = videos
    .map<Widget>((video) => Stack(
          fit: StackFit.passthrough,
          children: <Widget>[
            VideoPlayerWithControlBar(file: video),
            Container(
              alignment: Alignment.topRight,
              child: GestureDetector(
                onTap:
                    Feedback.wrapForTap(() => _removeFile(video
                child: Container(
                  padding: EdgeInsets.all(5),
                  child: Icon(Icons.clear, color: WgTheme.white
                ), // Container
              ), // GestureDetector
            ), // Container
          ], // <Widget>[]
        )) // Stack
    .toList();
```

Figure 42: Comments are not necessary

Actually, we hope the comments can provide some explanation about what does some functions do and what is the meaning of some functions?

Style

We think the code style of Flutter they used is based on Google standard.

Consistency

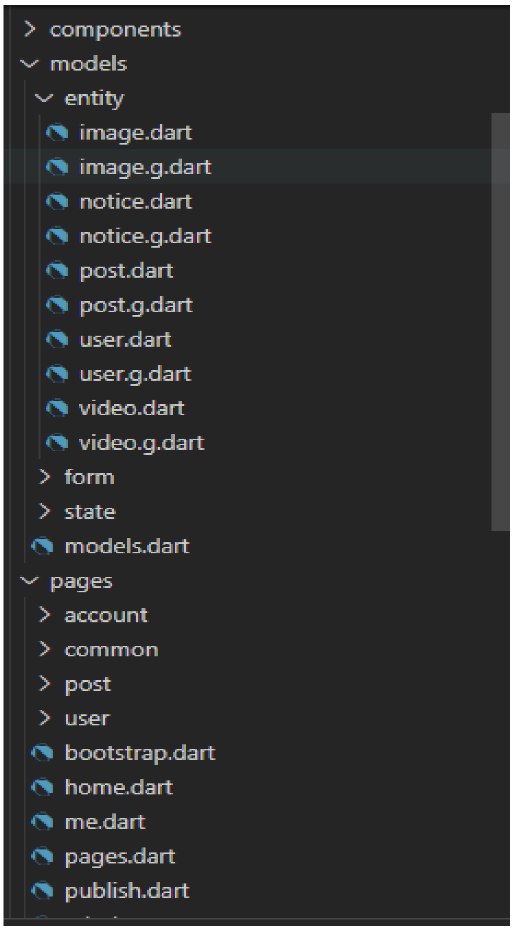
The style guide is based on https://flutter.dev/docs/resources/design-docs

Documentation

For related documents please see https://github.com/Sven97/SeeU.

Good Things

We think the original group has already tried their best to develop the application. We think they used all possible libraries to implement and this is the best thing. The structure is also the best we have seen from this course. Flutter is a really difficult framework and we think what they have done is the best among all groups. Deploy the Flutter environment is also tough, so we think they surprise me with the project integrity. We can run the application and it works well, although we cannot test their code. All functions are valid.



```
> components
v models
  v entity
      image.dart
      image.g.dart
      notice.dart
      notice.g.dart
      post.dart
      post.g.dart
      user.dart
      user.g.dart
      video.dart
      video.g.dart
  > form
  > state
    models.dart
v pages
  > account
  > common
  > post
  > user
    bootstrap.dart
    home.dart
    me.dart
    pages.dart
    publish.dart
```

Figure 43: Structure of the Flutter project