Deliverable 2b

Johanna Bell
Luca Bosch
Niklas Maier
Sebastian Schwarz
Simon Vogelbacher
Yasmin Hoffman

Konstanz, July 26, 2020

Advised by Till Niese

Contents

List of Figures

Part I

System Design

Part II

Object Design

1 Introduction

1.1 Guidelines of interface documentation

We, Group 12, use the following naming conventions that will help structuring and writing clear and readable code:

- Facades will be named with the suffix "Facade" and a prefix which describes what the Facade is for.
- Every class which includes logic will be named with the suffix "Handler" and a prefix which describes what the Handle handles.
- Classes which interact with the database will be named with the suffix "Manager" and a prefix which describes which data the Manager handles.
- Class names will be combinations of nouns.
- All parameters and methods will be given names which describes their purpose.

1.2 Literature references

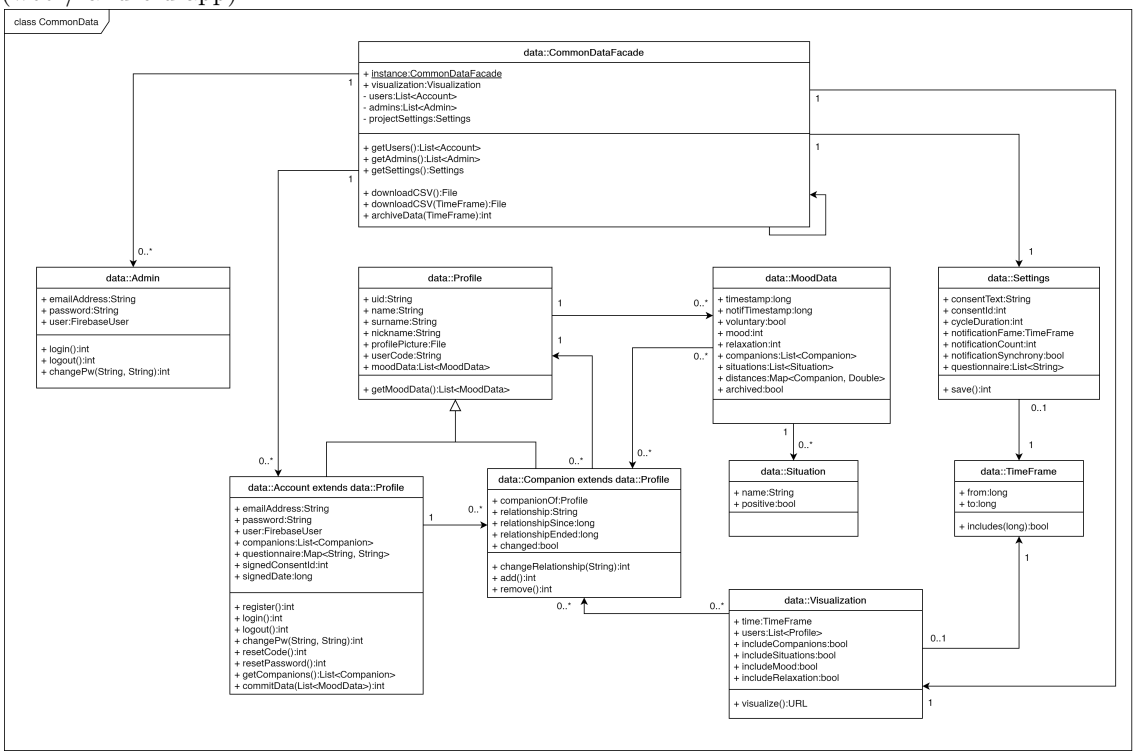
We will lightly use the example submission file "Example_D2b.pdf" as a template and our previously created documentation as reference.

2 Packages

This section defines a list of data and class models to be implemented to form the proposed system.

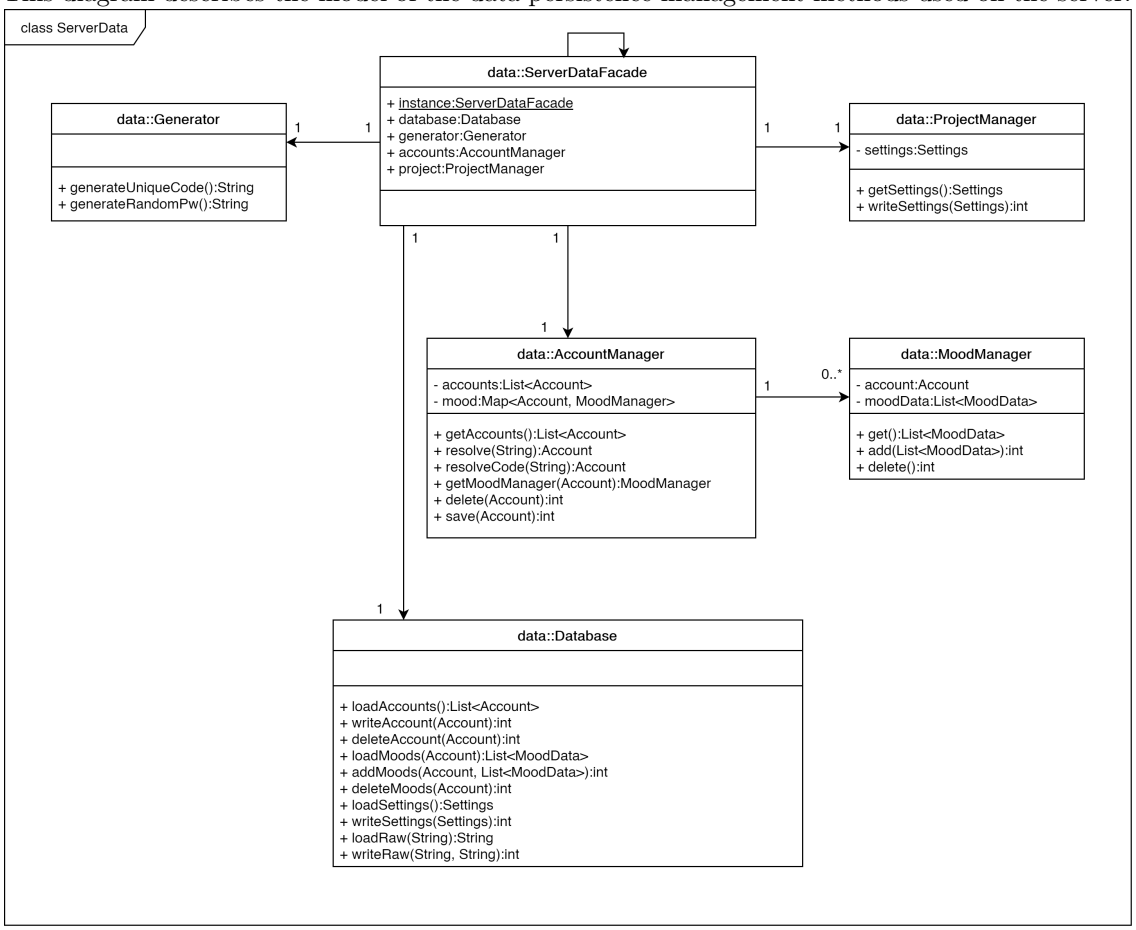
2.1 CommonData

This diagram describes the model of custom data types used both on the server and the clients (web / android app).



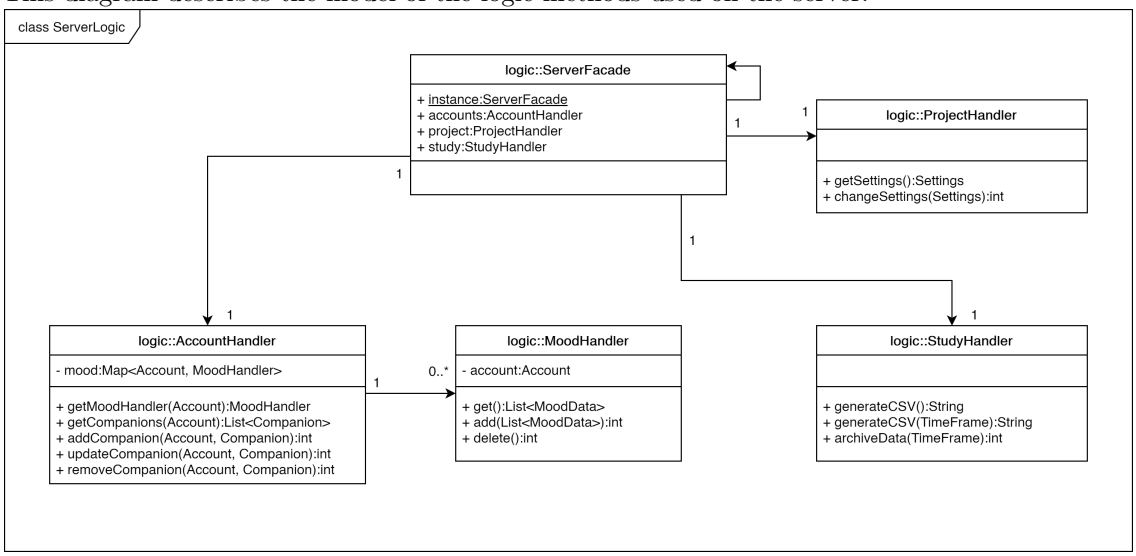
2.2 ServerData

This diagram describes the model of the data persistence management methods used on the server.



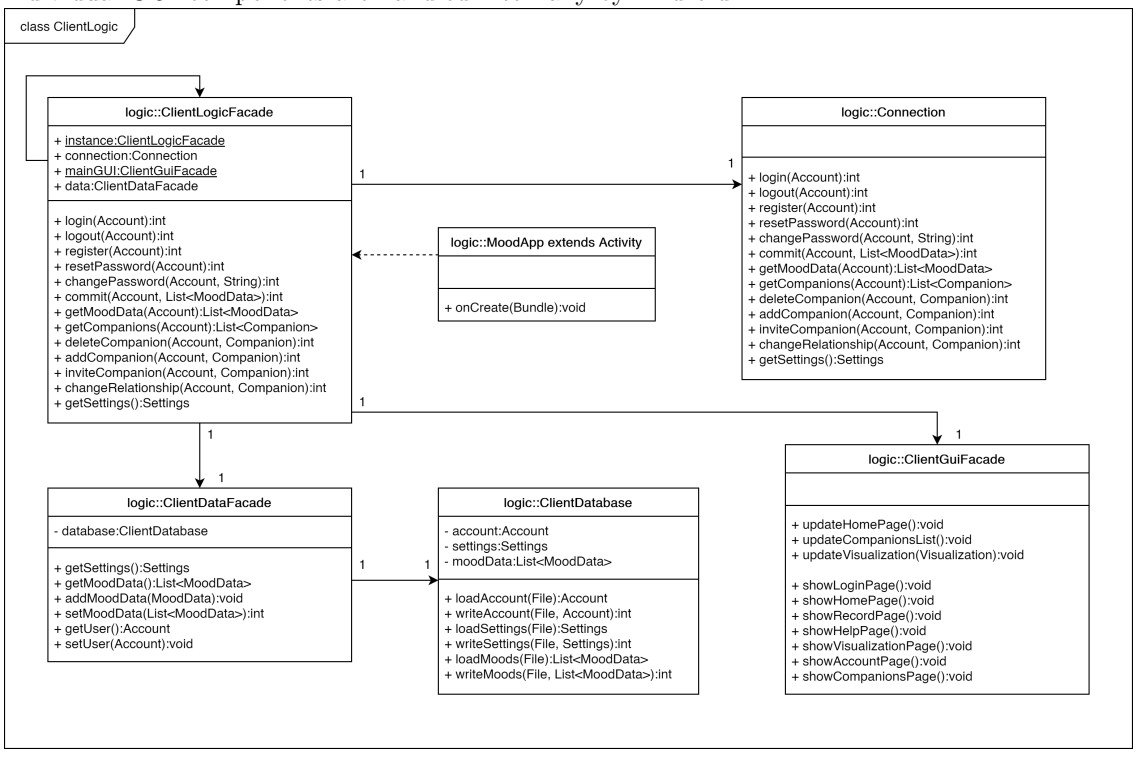
2.3 ServerLogic

This diagram describes the model of the logic methods used on the server.



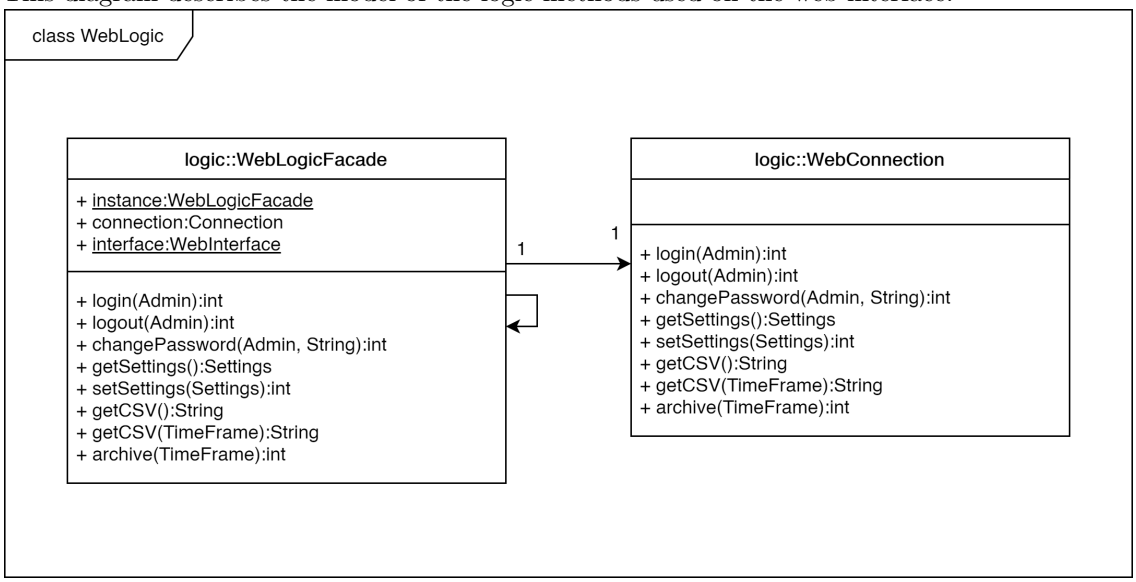
2.4 ClientLogic

This diagram describes the model of the logic methods used on the client. Additionally, since not much data is saved on the client, this diagram also describes the client database.
Furthermore, this diagram also describes the basic navigation methods for the GUI since the individual GUI components are handled internally by Android.



2.5 WebLogic

This diagram describes the model of the logic methods used on the web interface.



3 Class Interfaces

The algorithms in the following interfaces are written in Java-like pseudo code.
Many of the algorithms or methods we use are implemented by Firebase itself. These are not documented here.

3.1 CommonDataFacade

Methods within the CommonDataFacade or classes which are part of it merely call the corresponding methods on either the ClientLogic or ServerLogic, depending on which instance they belong to, with `this` as self-reference if applicable.

3.1.1 TimeFrame

Algorithm 1 includes(long time) : bool

Constraints:
none

Procedure:
return this.from <= time && this.to >= time;

3.2 ServerData

3.2.1 Database

The Database class is responsible for interacting with the Firebase Real-Time-Database. Given methods simply perform get or set operations on the corresponding fields.

3.2.2 Generator

Algorithm 2 generateUniqueCode() : String

Constraints:
post: returned value is a unique code not assigned to any account yet

Procedure:

```
String code = this.generateRandomPw();
while ServerDataFacade.instance.accounts.resolveCode(code) != null do
  code = this.generateRandomPw();
end while
return code
```

Algorithm 3 generateRandomPw() : String

Constraints:
none

Procedure:

```
StringBuilder sb ← new StringBuilder();
for int i←0; i<12; i++ do
  sb.append((char)(Math.random() * 93 + 33));
end for
return sb.toString();
```

3.2.3 AccountManager

Algorithm 4 getAccounts() : List<Account>

Constraints:
none

Procedure:
```
if this.accounts = null then
  this.accounts ← ServerDataFacade.instance.database.loadAccounts();
end if
return this.accounts;
```

Algorithm 5 resolve(String id) : Account

Constraints:
none

Procedure:
```
Iterator<Account> it ← this.getAccounts().iterator();
while it.hasNext() do
  Account current ← it.next();
  if current.email = id || current.uid = id then
    return current;
  end if
end while
```

Algorithm 6 resolveCode(String code) : Account

Constraints:
none

Procedure:
```
Iterator<Account> it ← this.getAccounts().iterator();
while it.hasNext() do
  Account current ← it.next();
  if current.userCode = code then
    return current;
  end if
end while
```

Algorithm 7 getMoodManager(Account account) : MoodManager

Constraints:
none

Procedure:
```
if this.mood.get(account) = null then
  this.mood.put(account, new MoodManager(account));
end if
return this.mood.get(account);
```

Algorithm 8 delete(Account account) : int

Constraints:
pre: the account is known (in the list)
post: the account is removed from the list

Procedure:
```
int index ← this.getAccounts().indexOf(account);
this.accounts.remove(index);
this.getMoodManager(account).delete();
return ServerDataFacade.instance.database.deleteAccount(Account);
```

Algorithm 9 save(Account account) : int

Constraints:
none

Procedure:
```
int index ← this.getAccounts().indexOf(account);
if index < 0 then
  this.accounts.add(account);
else
  this.accounts.set(index, account);
end if
return ServerDataFacade.instance.database.writeAccount(account);
```

Algorithm 10 delete(Account account) : int

Constraints:
none

Procedure:
```
int index ← this.getAccounts().indexOf(account);
if index < 0 then
  return -1;
end if
this.accounts.remove(index);
return ServerDataFacade.instance.database.deleteAccount(account);
```

3.2.4 MoodManager

Algorithm 11 get() : List<MoodData>

Constraints:
none

Procedure:
```
if this.moodData = null then
  this.moodData ← ServerDataFacade.instance.database.loadMoods(this.account);
end if
return this.moodData;
```

Algorithm 12 add(List<MoodData> moods) : List<MoodData>

Constraints:
none

Procedure:
```
this.get().addAll(moods);
return ServerDataFacade.instance.database.addMoods(this.account, moods);
```

Algorithm 13 delete() : int

Constraints:
none

Procedure:
```
for int i←this.get().size(); i>0; i– do
  this.moodData.remove(0);
end for
return ServerDataFacade.instance.database.deleteMoods(this.account);
```

3.2.5 ProjectManager

Algorithm 14 getSettings() : Settings

Constraints:
none

Procedure:

```
if this.settings = null then
  this.settings ← ServerDataFacade.instance.database.loadSettings();
end if
return this.settings;
```

Algorithm 15 writeSettings(Settings settings) : int

Constraints:
none

Procedure:

```
Settings old ← this.getSettings();
Database db ← ServerDataFacade.instance.database
for int i=1; i<=old.consentId; i++ do
  if db.loadRaw("consent_" + i) = settings.consentText then
    settings.consentId ← i;
    break;
  else if i = old.consentId then
    settings.consentId ← i + 1;
    db.writeRaw("consent_" + (i + 1), settings.consentText);
  end if
end for
this.settings ← settings;
db.writeSettings(this.settings);
return 20;
```

3.3 ServerLogic

3.3.1 AccountHandler

Algorithm 16 getMoodHandler(Account account) : MoodHandler

Constraints:
none

Procedure:
```
if this.mood.get(account) = null then
    this.mood.put(account, new MoodHandler(account));
end if
return this.mood.get(account);
```

Algorithm 17 getCompanions(Account account) : List<Companion>

Constraints:
none

Procedure:
```
return ServerDataFacade.instance.accounts.resolve(account.uid).companions;
```

Algorithm 18 addCompanion(Account account, Companion comp) : int

Constraints:
none

Procedure:
```
Account current ← ServerDataFacade.instance.accounts.resolve(account.uid);
current.companions.add(comp);
return ServerDataFacade.instance.database.writeAccount(crrent);
```

Algorithm 19 updateCompanion(Account account, Companion comp) : int

Constraints:
none

Procedure:

```
Account current ← ServerDataFacade.instance.accounts.resolve(account.uid);
int index ← current.companions.indexOf(comp);
if index < 0 then
    return -1;
end if
current.companions.set(index, comp);
return ServerDataFacade.instance.database.writeAccount(crrent);
```

Algorithm 20 removeCompanion(Account account, Companion comp) : int

Constraints:
none

Procedure:

```
if comp.relationshipEnded ≥ 0 then
    return -1;
end if
comp.relationshipEnded ← Date.now();
return ServerDataFacade.instance.database.writeAccount(acount);
```

3.3.2 MoodHandler

Algorithm 21 get() : List<MoodData>

Constraints:
none

Procedure:

```
return ServerDataFacade.instance.accounts.getMoodManager(this.account).get();
```

Algorithm 22 add(List<MoodData> moods) : int

Constraints:
none

Procedure:
return ServerDataFacade.instance.accounts.getMoodManager(this.account).add(moods);

Algorithm 23 delete() : int

Constraints:
none

Procedure:
return ServerDataFacade.instance.accounts.getMoodManager(this.account).delete();

3.3.3 ProjectHandler

Algorithm 24 getSettings() : Settings

Constraints:
none

Procedure:
return ServerDataFacade.instance.project.getSettings();

Algorithm 25 changeSettings(Settings settings) : int

Constraints:
none

Procedure:
return ServerDataFacade.instance.project.writeSettings(settings);

3.3.4 StudyHandler

Algorithm 26 generateCSV(TimeFrame time) : String

Constraints:
post: Returned string contains all mood data for the selected time frame with one line per user.

Definition:
This algorithm returns recorded mood data for a selected time frame with one line per user.
Each line uses the following comma-separated entries:

- ID of user
- Join date of user
- Signed consent ID of user
- Questionnaire answers in the format q1:a1&q2:a2&...
- Relationships in the format r1id:r1type:r1start:r1end&r2id:r2type:r2start:r2end&...
- Mood data block 1 in the format m1time|m1notifTime|m1mood|m1relax|m1volunt|m1archived
  |selected companions in the format c1id&c2id&...
  |GPS companion distances in the format c1id:c1distance&c2id:c2distance&...
- Mood data block 2 in the above format
- ...

Algorithm 27 generateCSV() : String

Constraints:
none

Procedure:
return this.generateCSV(new TimeFrame(0, Long.MAX_VALUE));

3.4 ClientLogic

3.4.1 Connection

The connection is responsible for calling the corresponding methods on the server.

3.4.2 ClientDataFacade

3.4.3 ClientLogicFacade

Algorithm 28 register(Account account) : int

Constraints:
pre: user has no account, has filled in the signup form (consent form, questionnaire, personal information, login credentials)
post: user is registered

Procedure:

```
firebaseAuth.createUserWithEmailAndPassword(account.email, account.password);
if task.isSuccessful() then
    updateUI(account);
    return ServerDataFacade.instance.accounts.save(account);
else
    return -1;
end if
```

Algorithm 29 login(Account account) : int

Constraints:
pre: user is logged out
post: user is logged in

Procedure:

```
firebaseAuth.signInWithEmailAndPassword(account.email, account.password);
if task.isSuccessful() then
    updateUI(account);
    return 1;
else
    log("Authentication failed");
    return -1;
end if
```

Algorithm 30 resetPassword(Account account) : int

Constraints:
pre: user is logged in
post: password reset email is sent

Procedure:

```
firebaseAuth.sendPasswordResetEmail(account.email);
if task.isSuccessful() then
  log("Email has been sent!");
  return 1;
else
  return -1;
end if
```

Algorithm 31 changePassword(Account account, String newPassword) : int

Constraints:
pre: user is logged in
post: password is changed

Procedure:

```
firebaseUser.updatePassword(newPassword);
if task.isSuccessful() then
  log("Password has been changed!");
  return 1;
else
  return -1;
end if
```

Algorithm 32 logOut(Account account) : int

Constraints:
pre: user is logged in
post: user is logged out

Procedure:

```
firebaseAuth.signOut();
if task.isSuccessful() then
  return 1;
else
  return -1;
end if
```

Algorithm 33 getMoodData(Account account) : List<MoodData> data

Constraints:
post: mood data is sent to the app

Procedure:

```
return getMoodHandler(account).get();
```

Algorithm 34 getCompanions(Account account) : List<Companion> companionsList

Constraints:
post: companion data is sent to the app

Procedure:
return ServerLogicFacade.instance.accounts.getCompanions(account);

Algorithm 35 deleteCompanion(Account account, Companion companion) : int

Constraints:
post: companion is deleted from users companion list

Procedure:
return ServerLogicFacade.instance.accounts.removeCompanion(account, companion);

Algorithm 36 addCompanion(Account account, Companion companion) : int

Constraints:
post: companion added to users companion list

Procedure:
return ServerLogicFacade.instance.accounts.addCompanion(account, companion);

Algorithm 37 addCompanionByCode(Account account, String friendcode, String relationship) : int

Constraints:
pre: user wants to invite the companion with the friend code
post: user with the friendcode is invited and added to the friend list

Procedure:
companion ← new Companion(ServerDataFacade.instance.accounts.resolveCode(friendcode), relationship);
return ServerLogicFacade.instance.accounts.addCompanion(account, companion);

Algorithm 38 inviteCompanion(Account account) : int

Constraints:
pre: user wants to invite a companion by email

Method:
This method generates an invite email draft containing an invite link which can be used to download the app if the recipient doesn't have the app installed yet and otherwise add the invitee as a companion. The user is prompted to send the generated email from their primary installed email client.

Algorithm 39 changeRelationship(Account account, Companion companion) : int

Constraints:
post: users relationship to companion is changed

Procedure:
return ServerLogicFacade.instance.accounts.updateCompanion(account, companion);

Algorithm 40 getSettings() : settings

Constraints:
post: user get's the settings

Procedure:
return ServerLogicFacade.instance.project.getSettings();

3.4.4 ClientDatabase

The ClientDatabase class is responsible for storing data on the user devices (phones). This data includes the currently logged in account, cached project settings and mood data.

3.4.5 ClientGuiFacade

The ClientGuiFacade class is responsible for updating and presenting the user interface of the Android app.

Algorithm 41 updateHomePage() : void

Constraints:
post: the calendar on the homepage reflects the most up-to-date data

Method:
The calendar on the homepage displays an overview of the signed in user's mood of the current month. This method reads the mood data of the user and updates the displayed calendar.

Algorithm 42 updateCompanionsList() : void

Constraints:
post: the companions list contains a list of GUI elements for each current companion

Method:
The companions of the signed in user are read from the server database if possible, else from the client cache and the GUI list is updated to contain an element for each known companion. The design will be according to the corresponding UI prototype from the first Milestone.

Algorithm 43 createVisualization() : void

Constraints:
post: the visualization frame contains a visualization matching the current options

Method:
The visualization is fetched in form of a web page and displayed in a WebView frame.
We use D3 to generate the visualizations on the page.

Algorithm 44 updateVisualization() : void

Constraints:
post: the visualization frame contains a visualization matching the current options

Method:
The visualization is fetched in form of a web page and displayed in a WebView frame.
We use D3 to generate the visualizations on the page.

Algorithm 45 showLoginPage() : void

Constraints:
pre: user is not logged in

Method:
Displays the login page, where the user can decide whether to log in to an existing account or register a new account. If they pick to create a new account, they will have to accept the consent form and fill the questionnaire first.

Algorithm 46 showHomePage() : void

Constraints:
pre: user is logged in

Method:
Displays the home page with calendar and overview.

Algorithm 47 showRecordPage() : void

Constraints:
pre: user is logged in

Method:
Displays the mood recording screen where the user can enter their current mood.

Algorithm 48 showHelpPage() : void

Constraints:
None

Method:
Displays the help page with information about the app.

Algorithm 49 showRecordPage() : void

Constraints:
pre: user is logged in

Method:
Displays the mood recording screen where the user can enter their current mood.

Algorithm 50 showVisualizationPage() : void

Constraints:
pre: user is logged in

Method:
Displays the visualization screen where the user can change parameters to view different visualizations of their own and their companions' mood history.

Algorithm 51 showAccountPage() : void

Constraints:
pre: user is logged in

Method:
Displays the account management screen where the user can change their password or log out.

Algorithm 52 showCompanionPage() : void

Constraints:
pre: user is logged in

Method:
Displays the companion list screen where the user can add, invite, change and remove companions.

3.5 WebLogic

3.5.1 WebConnection

Algorithm 53 login(Admin admin) : int

Constraints:
pre: Admin is not logged in.
post: Admin is logged in.

Procedure:

```
try{
await firebase.auth().signInWithEmailAndPassword(admin.email, admin.password);
return 1;
}
catch(Exception e){
return -1;
}
```

Algorithm 54 logout(Admin admin) : int

Constraints:
pre: Admin is logged in.
post: Admin is not logged in.

Procedure:

```
try{
await firebase.auth().signOut();
return 1;
}
catch(Exception e){
return -1;
}
```

Algorithm 55 changePassword(Admin admin, String newPw) : int

Constraints:
pre: Admin is logged in.
post: Password is changed.

Procedure:

```
try{
await firebase.auth().currentUser.updatePassword(newPw);
return 1;
}
catch(Exception e){
return -1;
}
```

Algorithm 56 getSettings() : String

Constraints:
pre: Admin is logged in.
post: Settings in the DB are synchronized with the ones shown in the Web App (overwrite local data).

Procedure:

```
firebase.database().ref('Settings').once('value').then(;
return snapshot.val();
);
```

Algorithm 57 setSettings(setting, value) : int

Constraints:
pre: Admin is logged in.
post: Settings in the DB are synchronized with the ones entered in the Web App (overwrite server data).

Procedure:

```
await firebase.database().ref('Settings').update(
setting: value
return 1;
);
catch(Exception e){
return -1;
}
```

Algorithm 58 getCSV() : String

The following algorithm will be implemented with Firebase web functions:
Constraints:
pre: Admin is logged in.
post: Computer gets CSV data from cloud function page.

Procedure:

```
window.open('https://us-central1-angrynerds-dac9e.cloudfunctions.net/getCSV');
```

Algorithm 59 getCSV(timeframeStart, timeframeEnd) : void

The following algorithm will be implemented with Firebase web functions:

Constraints:

pre: Admin is logged in.

post: Computer gets CSV data from cloud function page.

post: Only data between the selected dates is returned.

Procedure:

```
window.open('https://us-central1-angrynerds-dac9e.cloudfunctions.net/getCSV?timestamp1=' +
timeframeStart + '&timestamp2=' + timeframeEnd);
```

Algorithm 60 archive(timeframeStart, timeframeEnd) : String

The following algorithm will be implemented with Firebase web functions:

Constraints:

pre: Admin is logged in.

post: Entries within the specified timeframe get an archived flag.

Procedure:

```
window.open('https://us-central1-angrynerds-dac9e.cloudfunctions.net/archiveData?timestamp1='
+ timeframeStart + '&timestamp2=' + timeframeEnd');
```

4 Third-party libraries

- Firebase
  - Authorization
  - Database
  - Functions
- Node.js
- jQuery
- d3
- java.util.*
- Bootstrap

5 Glossary

RMI
Remote method invocation: RMI resembles a method call on the server being invoked by a client system.

Admin/Administrator
The person who will run the experiment and is able to access the data generated by the application and change its settings.

Mood data
Mood data refers to the data input by the user, including mood level, relaxation level, near companions and special situations.

GPS
Global Positioning System is used to determine the users position.

One-time consent form
The one time consent form can be edited by the admin and has to be accepted once by every users before he can start use the app.

CSV
Comma-Separated Values. A comma separated value (csv) file with all the data of each participant. Each row stands for one single user. Each row includes: an identifier that links the person to a consent form, the answers of the user's questionnaires, each relationship established (with ID to the other user and timestamp on when relationship started and/or ended), all entry log info (timestamp, voluntary flag), in case it was elicited: notification time, and data input time; mood, stress level, companions, special situations and GPS.
If a user didn't click on a notification,the entry only contains a timestamp,notification label,notification time, and a minus 1.

Experiment cycle
: After establishing a relation with a companion, the *experiment cycle* specifies the time frame which has to elapse before the corresponsive participants can view each other's data for said time frame.

Participant
: With participant, we refer to the people who will be partaking in the experiment, the users of our application.

Logged in/out
: A user can only log in with his credentials, this assures that only people that are supposed to participate in the experiment are able to do so. A user can log out at any time, but will not be able to use the application.

Voluntarity flag
: Indicates whether the user entered the mood after receiving a notification prompting them to record their mood or on their own.

User ID
: Every user will have a unique *User ID*, usually a string of letters and numbers which allows a program to identify him easily, even if there are multiple users with the same names, etc.

DB : *Database*

Part III

Integration Tests

We will approach the integration testing via bottom-up technique : we will first test the data layers, then the logic layers and lastly the GUI.

5.1 CommonData

5.1.1 Testing of the CommonDataFacade Class includes:

- get*(); * stands for Users,Admins,Settings

The tests are successful if the correct values (Accounts,Admins and Settings) are returned.

- archiveData(TimeFrame)

The test is successful if all data of the correct time frame is stored in the database.

- downloadCSV();with or without a TimeFrame

The tests are successful if a correct CSV file for the correct time frame is downloaded.

5.1.2 Testing of the Admin Class includes:

- login()

The test is successful if the admin is logged in to the system with the correct credentials.

- logout()

The test is successful if the admin is logged out of the system.

- changePw(String,String)

The test is successful if,given a correct old and valid new password, the admin password is changed.

5.1.3 Testing of the Account class includes:

- register()

The test is successful if the user is registered in the system.

- login()

The test is successful if the user is logged in to the system with the correct credentials.

- logout()

The test is successful if the user is logged out of the system.

- changePw(String,String)

The test is successful if, given a correct old and valid new password, the user password is changed.

- resetCode()

The test is successful if the user's unique code is resetted and replaced with a new one.

- resetPassword()

The test is successful if an email with a password reset link is sent to the users email.

- commitData(List<MoodData>)

The test is successful if the list of MoodData is commited to the server.

- getCompanions()

The test is successful if the correct companion list is returned.

5.1.4 Testing of the Profile class includes:

- getMoodData()

The test is successful if the correct MoodData list is returned.

5.1.5 Testing of the Companion class includes:

- changeRelationship(String)

The test is successful if the relationship status in the databse is changed correctly.

- add()

The test is successful if the companion is added to the users companions list.

- remove()

The test is successful if the selected companion is flagged, as not being a companion anymore, in the users companions list in the database.

5.1.6 Testing of the Visualization class includes:

- visualize()

The test is successful if an URL with the visualization is returned. The visualization itself should be correct too.

5.1.7 Testing of the Settings class includes:

- save()

The test is successful if the settings are correctly stored in the database.

5.1.8 Testing of the TimeFrame class includes:

- includes(long)

The test is successful if the test returns the correct boolean value for the given time frame.

5.2 ServerData

5.2.1 Testing of the Database Class includes:

- load*(); *stands for: Accounts, Moods, Settings, Raw
- write*(); *stands for: Account, Settings, Raw
- delete*(); *stands for: Account,Moods
- addMoods()

The tests on the database are successful if the values of the objects are loaded, written, deleted or added correctly.

5.2.2 Testing of the AccountManager Class includes:

- get*(); *stands for: Accounts, MoodManager
- resolve(String);
- resolveCode(String)
- save(Account)

The tests on the AccountManager are successful if the correct values (Account and MoodManager) are returned and the correct Account is saved.

5.2.3 Testing of the MoodManager Class includes:

- get()
- add(List<MoodData>)
- delete()

The tests on the MoodManager are successful if the correct values (Moods) are returned, added to the database or deleted.

5.2.4 Testing of the ProjectManager Class includes:

- getSettings()
- writeSettings(Settings)

The tests on the ProjectManager are successful if the correct setting values are returned, or written to the database.

5.2.5 Testing of the Generator Class includes:

- generateUniqueCode()
- generateRandomPw()

The tests on the Generator are successful if a unique user code or random password are generated correctly.

5.3 ServerLogic

5.3.1 Testing of the AccountHandler Class includes:

- get*(Account); *stands for: MoodHandler, Companions

The tests are successful if the correct values are returned.

- addCompanion(Account, Companion)

The test is successful if the requested companion object is added to the correct user account.

- updateCompanion(Account,Companion)

The test is successful if all the values of the user's companion object are updated correctly.

- removeCompanion(Account,Companion)

The test is successful if the selected companion is flagged as not being a companion anymore in the users companions list in the database.

5.3.2 Testing of the MoodHandler Class includes:

- get()
- add()
- delete()

The tests on the MoodHandler are successful if the correct values (Moods) are returned, added to the database or deleted.

5.3.3 Testing of the StudyHandler Class includes:

- generateCSV(); with selected time frame or without.

The test is successful if a correct CSV file with all user data for the correct time frame is generated.

- archiveData(TimeFrame)

The test is successful if all data of the correct time frame is stored in the database.

5.3.4 Testing of the ProjectHandler Class includes:

- getSettings()

The test is successful if the current project settings are retrieved from the database.

- changeSettings(Settings)

The test is successful if the project settings are changed.

5.4 ClientLogic

5.4.1 Testing of the ClientLogicFacade Class includes:

- login(Account)

The test is successful if the account is logged in and a FirebaseUser is assigned.

- logout(Account)

The test is successful if the account is logged out.

- register(Account)

The test is successful if the account is registered and saved on the database.

- resetPassword(Account)

The test is successful if an email containing a reset link was sent to the account's email address.

- changePassword(Account, String)

The test is successful if the account's password was changed successfully.

- commit(Account, List¡MoodData¿)

The test is successful if the list of mood data is saved on the database and removed from local cache.

- getMoodData(Account)

The test is successful if a list of mood data is retrieved from the database.

- getCompanions(Account)

The test is successful if a list of companions for the account is retrieved from the database.

- deleteCompanion(Account, Companion)

The test is successful if the relationship between the account and the companion is marked as ended on the database.

- addCompanion(Account, Companion)

The test is successful if a companion relationship between the account and the companion is created and saved on the database and the companion received a notification.

- inviteCompanion(Account, Companion)

The test is successful if the user is prompted to send a generated email draft in their email app.

- changeRelationship(Account, Companion)

The test is successful if the desired relationship changes are saved on the database, or the user is shown a warning if they have previously changed their relationship with this companion.

- getSettings()

The test is successful if the current project settings are retrieved from the database.

Testing of the ClientLogicFacade Class inherit the testing of the Connection Class.

5.4.2 Testing of the ClientDataFacade Class includes:

- getSettings()

The test is successful if cached project settings are retrieved from local storage.

- getMoodData()

The test is successful if cached mood data is retrieved from local storage.

- addMoodData(MoodData)

The test is successful if the mood data was added to the local cache storage.

- setMoodData(List¡MoodData¿)

The test is successful if the list of mood data was saved on the local storage.

- getUser()

The test is successful if the currently logged in user is retrieved from local storage, if given.

- setUser(Account)

The test is successful if the account is saved as currently logged in account on the local storage.

Testing of the ClientDataFacade Class inherits testing of the ClientDatabase Class, which is responsible for reading and storing this information on the phone's local storage.

5.4.3 ClientGuiFacade

- updateHomePage()

The test is successful if the home page content reflects the up-to-date data of the logged in user.

- updateCompanionsList()

The test is successful if the companion list displays the list of the current companions of the logged in user.

- updateVisualization()

The test is successful if the visualization displays data corresponding to the logged in user and their visualization preferences.

- show*Page(); * stands for Login, Home, Record, Help, Visualization, Account, Companions

The tests are successful if the corresponding page is shown on the device.

5.5 WebLogic

5.5.1 Testing of the WebConnection Class includes:

1. Operations on the admin account

    - login(Admin)
      The test is successful, if, after providing valid credentials, the admin is logged in with Firebase.

    - logout(Admin)
      The test is successful if the admin is logged in with Firebase afterwards.

    - changePassword(Admin, String)
      The test is successful, if the admins password is changed to the specified value in Firebase.

2. Interaction with the project settings

    - getSettings()
      The test is successful if the settings are correctly loaded from the realtime database.

    - setSetting(Settings)
      The test is successful if the settings are correctly written to the realtime database.

3. Interaction with the user data in the DB

    - getCSV()
      The test is successful if the function returns a String in CSV format containing all of the required information.

    - getCSV(TimeFrame)
      The test is successful if the function returns a String in CSV format containing all of the required information, but only in the specified timeframe.

    - archive(TimeFrame)
      The test is successful if all accounts in the specified timeframe are flagged as archived in the database.

Part IV

Documentation of Group Work for D2b

6 General organization of group

- Project Manager - Niklas Maier
- Customer Relationship Officer - Johanna Bell
- Documentation Lead - Luca Bosch
- Technical Lead - Sebastian Schwarz
- Repository Lead - Simon Vogelbacher
- Quality Assurance Manager - Yasmin Hoffmann

7 Individual tasks of group members

In this deliverable we had to coordinate our work a lot, since the functions and diagrams are heavily interwoven. Thus we did all of the work together, there were not really any *individual* Tasks.

7.1 Johanna Bell

- Sections Part II - 1, 2, 3, 4, 5, Part III
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests

7.2 Luca Bosch

- Sections Part II - 1, 2, 3, 4, 5, Part III, Part IV
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests, Documentation of Group Work

7.3 Niklas Maier

- Sections Part II - 1, 2, 3, 4, 5, Part III
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests

7.4 Sebastian Schwarz

- Sections Part II - 1, 2, 3, 4, 5, Part III
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests

7.5 Simon Vogelbacher

- Sections Part II - 1, 2, 3, 4, 5, Part III
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests

7.6 Yasmin Hoffman

- Sections Part II - 1, 2, 3, 4, 5, Part III
- Introduction, Packages, Class Interfaces, Third-Party Libraries, Glossary, Integration Tests

8 Meeting protocols

8.1 Protocol on 26. June 2020

Datum: 11. June 2020 **Zeit:** 11:00-13:30

- Discussed Topics
  - After getting feedback on the updated diagrams on D2a, we realized that we changed the wrong things and discussed how to finally get them right
  - After getting feedback on the updated diagrams on D2a, we realized that we changed the wrong things and discussed how to finally get them right
  - We brainstormed about where to start with the next deliverable, using the existing class diagrams
  - We thought about which set of deadlines we are going to stick to
- Conclusions
  - We wrote some of the pseudocode in D2b for methods that were easily understandable and clearest in their function
  - We decided to try and get back some of the time we had to invest into D2a by working even harder on D2b and finishing it quickly

8.2 Protocol on 29. June 2020

Datum: 11. June 2020 **Zeit:** 11:00-13:30

- Discussed Topics
  - After getting feedback on the updated diagrams on D2a, we realized that we changed the wrong things and discussed how to finally get them right
  - We needed to make a new work plan and schedule our advisor meeting
  - Yesterday, we found out that we misunderstood the functionality of Firebase and hat to adjust our documents accordingly
- Conclusions
  - We scheduled an advisor meeting for Wed. 10:30
  - We tried to understand the firebase login process together
  - We created the new work plan and uploaded it to git
  - We decided to meet again after the advisor meeting to discuss how we want to proceed

8.3 Protocol on 7. July 2020

Datum: 11. June 2020 **Zeit:** 13:00-17:30

- Discussed Topics
  - We discussed how we want to modify the class diagrams so they would be as accurate as possible compared with the final code
  - We figured out how the realtime database can be accessed via functions and how the functions there can be called dynamically
- Conclusions
  - We removed most of the authentication parts of the diagrams since those things will be handled by Firebase
  - We decided to meet again tomorrow to finish the deliverable, since we will be studying for the exams the next days