

Tests: 27 total, 27 passed

1.67 s

[Collapse](#) | [Expand](#)

bonfirechat

1.67 s

BonfireAPITest

89 ms

[testEncode](#)

passed 89 ms

DateHelperTest

104 ms

[testParse](#)

passed 101 ms

[testFormat](#)

passed 3 ms

RingBufferTest

10 ms

[testEmpty](#)

passed 9 ms

[testContains](#)

passed 0 ms

[testOverflow](#)

passed 1 ms

StreamHelperTest

38 ms

[testStreamToString](#)

passed 36 ms

[testByteArrayToHexString](#)

passed 2 ms

StringHelperTest

12 ms

[testRegexMatch](#)

passed 12 ms

ContactTest

1.28 s

[testSetFirstName](#)

passed 1.28 s

[testToString](#)

passed 1 ms

[testSetLastName](#)

passed 0 ms

[testGetLastName](#)

passed 0 ms

[testGetFirstName](#)

passed 0 ms

[testSetNickname](#)

passed 0 ms

[testGetNickname](#)

passed 1 ms

ConversationTest

112 ms

[testAddMessages](#)

passed 28 ms

[testGetLastMessage](#)

passed 1 ms

testFromCursor	passed	83 ms
EnvelopeTest		6 ms
testHasFlag	passed	3 ms
testFromMessage	passed	3 ms
MessageTest		17 ms
testToString	passed	0 ms
testDirection	passed	5 ms
testHasFlag	passed	11 ms
testSetTransferProtocol	passed	1 ms
MyPublicKeyTest		1 ms
testGet	passed	1 ms
testAsByteArray	passed	0 ms

Generated by Android Studio on 18.09.15 03:07

[all classes]

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	6,9% (15/ 216)	5,5% (43/ 781)	3,4% (138/ 4043)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
android.support.v7.appcompat	0% (0/ 13)	0% (0/ 14)	0% (0/ 39)
com.github.curran.showcaseview	0% (0/ 9)	0% (0/ 10)	0% (0/ 12)
com.google.android.gms	0% (0/ 10)	0% (0/ 11)	0% (0/ 13)
com.google.android.gms.gcm	0% (0/ 10)	0% (0/ 11)	0% (0/ 13)
com.google.android.gms.maps	0% (0/ 10)	0% (0/ 11)	0% (0/ 13)
de.tudarmstadt.informatik.bp.bonfirechat	0% (0/ 20)	0% (0/ 22)	0% (0/ 52)
de.tudarmstadt.informatik.bp.bonfirechat.data	33,3% (1/ 3)	4,3% (2/ 46)	0,7% (2/ 307)
de.tudarmstadt.informatik.bp.bonfirechat.helper	33,3% (4/ 12)	13,6% (9/ 66)	9,1% (27/ 298)
de.tudarmstadt.informatik.bp.bonfirechat.helper.zxing	0% (0/ 7)	0% (0/ 61)	0% (0/ 342)
de.tudarmstadt.informatik.bp.bonfirechat.location	0% (0/ 1)	0% (0/ 11)	0% (0/ 52)
de.tudarmstadt.informatik.bp.bonfirechat.models	85,7% (6/ 7)	40,6% (26/ 64)	33,5% (77/ 230)
de.tudarmstadt.informatik.bp.bonfirechat.network	0% (0/ 30)	0% (0/ 134)	0% (0/ 784)
de.tudarmstadt.informatik.bp.bonfirechat.routing	33,3% (4/ 12)	10% (6/ 60)	12,9% (32/ 248)
de.tudarmstadt.informatik.bp.bonfirechat.stats	0% (0/ 7)	0% (0/ 21)	0% (0/ 121)
de.tudarmstadt.informatik.bp.bonfirechat.ui	0% (0/ 65)	0% (0/ 239)	0% (0/ 1519)

generated on 2015-09-18 03:04

Code Review



TECHNISCHE
UNIVERSITÄT
DARMSTADT

BonfireChat
Bachelorpraktikum 2015

1. Ist die Funktionalität korrekt?
2. Sind die Klassen, Funktionen und Variablen angemessen benannt?
3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?
4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?
5. Gibt es unnötig überladene Funktionen oder Konstruktoren?
6. Wurde die Datenbankstruktur gut entworfen?
7. Enthält das Codeteil Funktionen, die wiederverwendet werden können? Sind diese in einer Helper-Klasse untergebracht?
8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?
9. Werden alle Eingabeparameter validiert?
10. Wie beeinflusst die Funktionalität die Performance der App - Stromverbrauch, Rechenzeit und Speicherbedarf?
11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?
12. Werden alle möglichen Fehlschläge behandelt?
13. Findet eine "Graceful Degradation statt?
14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?
15. Welche Teile können parallel ausgeführt werden?
16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?
17. Ist das Layout passend für alle Bildschirmdimensionen?
18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

Code-Review vom 22.06.2015

Durchgeführt von: Jens Heuschkel, Max Weller

Geprüfte Codeteile: RingBuffer

Änderungen vorgenommen in Commit: ed7607da504e27a40cf2c99841c539be1d60c29a

1. Ist die Funktionalität korrekt?

Die contains-implementierung funktioniert nicht immer korrekt. TODO: Warum?

2. Sind die Klassen, Funktionen und Variablen angemessen benannt?

Ja.

3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?

Die Struktur ist gut.

4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?

Nein.

5. Gibt es unnötig überladene Funktionen oder Konstruktoren?

Nein.

6. Wurde die Datenbankstruktur gut entworfen?

Der Codeteil besitzt keine Datenbankstruktur.

7. Enthält das Codeteil Funktionen, die wiederverwendet werden können? Sind diese in einer Helper-Klasse untergebracht?

Es sind keine wiederverwendbaren Funktionen vorhanden.

8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?

Es wurde keine doppelte Funktionalität implementiert.

9. Werden alle Eingabeparameter validiert?

Das ist für diesen Codeteil nicht sinnvoll/relevant.

10. Wie beeinflusst die Funktionalität die Performance der App - Stromverbrauch, Rechenzeit und Speicherbedarf?

Da dies davon abhängig ist, wie der RingBuffer verwendet wird, kann keine Aussage darüber getroffen werden. Bei normaler Nutzung sollte es aber keine negativen Auswirkungen geben.

11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?

Nicht relevant.

12. Werden alle möglichen Fehlschläge behandelt?

Ja.

13. Findet eine "Graceful Degradation" statt?

Nein, aber für diesen Codeteil sind unbekannte Fehler fast ausgeschlossen. Daher ist es nicht sinnvoll, dies zu implementieren.

14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?

Ja.

15. Welche Teile können parallel ausgeführt werden?

Keine.

16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?

Müssen sie nicht.

17. Ist das Layout passend für alle Bildschirmdimensionen?

Diese Funktionalität hat kein Layout.

18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

- die Attribute content, length und insertPosition sollten alle Private sein, sind sie aber nicht
- content und length sollten beide final sein, sind sie aber nicht

Diese Fehler wurden korrigiert.

Code-Review vom 13.07.2015

Durchgeführt von: Max Weller, Alexander Frömmgen, Jens Heuschkel

Geprüfte Codeteile: models-Package, ConnectionManager, EchoProtocol, GcmProtocol, TracerouteHandler

Änderungen vorgenommen in Commit: da815895bedcd619aaf928dc29c80a433cdc0ba4

1. Ist die Funktionalität korrekt?

- Im ConnectionManager in der SENDMESSAGE_ACTION wird eine RuntimeException nicht geworfen, sondern einfach einer Variable zugewiesen. Dies wurde geändert, sodass die Exception nun geworfen wird.

2. Sind die Klassen, Funktionen und Variablen angemessen benannt?

Ja

3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?

Einige Konstanten und einige Funktionen für die HTTP-API sind nicht an einer sinnvollen Stelle definiert, sonder immer in der Klasse, in der sie gerade benutzt wurden. Als Konsequenz wurde eine eigene Klasse für die API angelegt (BonfireAPI.java), in die diese Konstanten und Funktionen verlagert wurden.

4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?

Nein

5. Gibt es unnötig überladene Funktionen oder Konstruktoren?

Nein

6. Wurde die Datenbankstruktur gut entworfen?

Ist nicht relevant für die geprüften Codeteile.

7. Enthält das Codeteil Funktionen, die wiederverwendet werden können? Sind diese in einer Helper-Klasse untergebracht?

Nein

8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?

Es wurde keine doppelte Funktionalität implementiert.

9. Werden alle Eingabeparameter validiert?

Ja

10. Wie beeinflusst die Funktionalität die Performance der App - Stromverbrauch, Rechenzeit und Speicherbedarf?

Allein aus dem Grund, dass viele Netzwerkressourcen angefordert werden, wird der Stromverbrauch negativ beeinflusst, dies liegt aber nicht an der Implementierung, sondern an der Natur der Sache.

11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?

Nicht relevant

12. Werden alle möglichen Fehlschläge behandelt?

Ja

13. Findet eine "Graceful Degradation" statt?

Wird nicht betrachtet.

14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?

Ja

15. Welche Teile können parallel ausgeführt werden?

Ist für diesen Codeteil nicht relevant.

16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?

Ist für diesen Codeteil nicht relevant.

17. Ist das Layout passend für alle Bildschirmdimensionen?

Ist für diesen Codeteil nicht relevant.

18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

Sehr viele Variablen, die final sein sollten, sind nicht final deklariert. Dies wurde korrigiert.

Code-Review vom 3. August 2015

Durchgeführt von:

Geprüfte Codeteile: Die Datenbankfunktionalität (Im Wesentlichen die Klasse BonfireData)

Änderungen vorgenommen in Commit: 5a26429def4f8278980232cac29e2cf2d6afe32

1. Ist die Funktionalität korrekt?

Ja, abgesehen von der onUpgrade-Methode, die aber gewollt so implementiert ist, um die Entwicklung zu vereinfachen. Sie wird richtig implementiert werden, wenn die App veröffentlicht wird.

2. Sind die Klassen, Funktionen und Variablen angemessen benannt?

Ja.

3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?

Keine Verbesserungen nötig.

4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?

Nein.

5. Gibt es unnötig überladene Funktionen oder Konstruktoren?

Nein.

6. Wurde die Datenbankstruktur gut entworfen?

Ja.

7. Enthält das Codeteil Funktionen, die wiederverwendet werden

können? Sind diese in einer Helper-Klasse untergebracht?

Da es nur Methoden gibt, die Dinge in die Datenbank schreiben oder aus ihr lesen, kann keine Funktion wiederverwendet werden.

8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?

Es wurde keine doppelte Funktionalität implementiert.

9. Werden alle Eingabeparameter validiert?

Die Eingabeparameter werden ausreichen überprüft.

10. Wie beeinflusst die Funktionalität die Performance der App - Stromverbrauch, Rechenzeit und Speicherbedarf?

Sehr einfache Funktionalität, fast keine Auswirkungen.

11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?

Nicht relevant für diese Funktionalität.

12. Werden alle möglichen Fehlschläge behandelt?

Ja.

13. Findet eine "Graceful Degradation" statt?

Ja.

14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?

Nein, folgende Fehler wurden gefunden und behoben:

- Methode deleteContact gab immer true zurück. Gibt nun bei Fehlschlag false zurück.
- Das Feld helper wurde nicht benutzt und deshalb entfernt
- Die Methode getContactByXmppId wurde nicht benutzt und deshalb entfernt
- Der import von org.abstractj.kalium.keys.PublicKey wurde nicht benötigt und deshalb entfernt
- Diverse ungenutzte Variablen wurden entfernt

15. Welche Teile können parallel ausgeführt werden?

Keine

16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?

Müssen sie nicht.

17. Ist das Layout passend für alle Bildschirmdimensionen?

Es gibt kein Layout.

18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

- Alle Methoden müssen public sein, da sie von anderen Klassen aus verwendet werden.

Code-Review vom 17. August 2015

Durchgeführt von:

Geprüfte Codeteile: ConnectionManager

Änderungen vorgenommen in Commit:

1. Ist die Funktionalität korrekt?

Ja.

2. Sind die Klassen, Funktionen und Variablen angemessen benannt?

Ja.

3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?

Ja sie ist sinnvoll erstellt und es sind keine Verbesserungen bekannt.

4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?

Nein.

5. Gibt es unnötig überladene Funktionen oder Konstruktoren?

Die Methode storeAndDisplayMessage kann in eine storeMessage und in ein displayMessage aufgeteilt werden.

6. Wurde die Datenbankstruktur gut entworfen?

Ja.

7. Enthält das Codeteil Funktionen, die wiederverwendet werden können? Sind diese in einer Helper-Klasse untergebracht?

Die Klasse ist dafür zuständig Nachrichten zu versenden und diese weiterzuleiten. Immer wenn dies geschieht erledigt das der ConnectionManager. Insofern gibt es keine Funktionen die später wiederverwendet werden können.

8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?

Es gibt keine bereits existierenden Helper-Methoden.

9. Werden alle Eingabeparameter validiert?

Es gibt keine Eingabeparameter im klassischen Sinne. Besitzt ein Handy keine Wifi oder Bluetooth Funktionalität, dann unterliegt es den Protokollen dieser Klassen dies zu überprüfen. Werden Nachrichten an benachbarte Peers verschickt so ist es nicht möglich diese zu validieren, da diese sich außer Reichweite bewegen können.

10. Wie beeinflusst die Funktionalität die Performance der App - Stromverbrauch, Rechenzeit und Speicherbedarf?

Der Connection Manager startet die Protokolle Wifi, Bluetooth und GCM. Diese überprüfen häufig welche Geräte sich in der unmittelbaren Nähe befinden und verbrauchen damit Strom. Da dies jedoch eine wichtige Funktionalität ist für die Funktionalität der App, kann dies nicht vermieden werden. Will ein User nur eins der Protokole benutzen so kann er dies in den Einstellungen auswählen.

11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?

Der Großteil des Codes im ConnectionManager läuft bereits in einem Hintergrundthread. Nur einige statische Methoden dienen als Helper für UI-Klassen, diese müssen daher im UI Thread laufen.

12. Werden alle möglichen Fehlschläge behandelt?

Es gibt noch eine Methode in der mit TODO gekennzeichnet ist, ob hier nicht ein Fehler geworfen werden soll. Um alle anderen Fehler wird sich gekümmert.

13. Findet eine "Graceful Degradation" statt?

Bei dem Handlen mit der Datenbank kann ein try-catch Block eingefügt werden.

14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?

Nein, folgende Fehler wurden gefunden und werden behoben:

- Fehler wurden nicht userfreundlich dargestellt
- Es befinden sich TODO Blöcke im Code, die abgearbeitet werden müssen.
- Die CanSend Funktionalität returned immer true und ist noch nicht implementiert -> BluetoothProtokol & WifiProtokol
- Die sendMessage Methode ruft nur die sendEnvelope Methode auf. Überbleibsel von altem Code.
- Ungenutzte Imports entfernen.

15. Welche Teile können parallel ausgeführt werden?

Die Parallelisierung findet in den aufgerufenen Klassen statt und ist somit bereits vorhanden.

16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?

Müssen sie nicht.

17. Ist das Layout passend für alle Bildschirmdimensionen?

Es gibt kein Layout.

18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

- routingManager und peers sollten nicht public sein, alle Zugriffe von außen sind zur Generierung von Debugausgaben bzw. zum manuellen Setzen einer Route zum Debuggen. Die Methode zur Generierung von Debugausgaben kann in den ConnectionManager verschoben werden.
- registeredProtocols, getOrCreateConnection muss nicht public sein (wird nicht extern verwendet)
- connections muss nicht public sein, wird von außen nur zur Generierung von Debugausgaben verwendet. Die Methode zur Generierung von Debugausgaben kann in den ConnectionManager verschoben werden.

Code-Review vom 31.08.2015

Durchgeführt von: Jens Heuschkel, Jonas Mönnig

Geprüfte Codeteile: GpsTracker

Änderungen vorgenommen in Commit: Im Zuge dieses Code-Reviews wurden keine Änderungen vorgenommen.

1. Ist die Funktionalität korrekt?

Ja.

2. Sind die Klassen, Funktionen und Variablen angemessen benannt?

Ja.

3. Wurde die Klassenstruktur gut entworfen und erfüllt sie alle Anforderungen, oder sind Verbesserungen nötig?

Ja.

4. Gibt es Klassen, die aufgrund neuer Implementierungen überflüssig geworden sind?

Nein.

5. Gibt es unnötig überladene Funktionen oder Konstruktoren?

Nein.

6. Wurde die Datenbankstruktur gut entworfen?

Für dieses Codeteil gibt es keine Datenbankstruktur.

7. Enthält das Codeteil Funktionen, die wiederverwendet werden können? Sind diese in einer Helper-Klasse untergebracht?

Die Klasse enthält keinen wiederverwendbaren Funktionen.

8. Wurden existierende Helper-Funktionen benutzt? Ist keine doppelte Funktionalität implementiert?

Es wurde keine doppelte Funktionalität implementiert.

9. Werden alle Eingabeparameter validiert?

Es gibt keine Eingabeparameter.

10. Wie beeinflusst die Funktionalität die Performance der App -

Stromverbrauch, Rechenzeit und Speicherbedarf?

Da GPS mit der maximalen Genauigkeit verwendet wird, wird der Stromverbrauch stark erhöht. Die hohe Genauigkeit wird aber benötigt, darum wird sie beibehalten.

11. Ist es unbedingt nötig, den Code in einem UI-Thread laufen zu lassen oder würde ein Background-Thread ausreichen?

Für diesen Codeteil nicht relevant.

12. Werden alle möglichen Fehlschläge behandelt?

Ja.

13. Findet eine "Graceful Degradation" statt?

Ja.

14. Werden die Best Practices zur Appentwicklung, laut Android Lint, befolgt?

Ja.

15. Welche Teile können parallel ausgeführt werden?

Für diesen Codeteil nicht relevant.

16. Sind die Operationen, bei denen Threadsicherheit benötigt wird, threadsicher?

Müssen sie nicht.

17. Ist das Layout passend für alle Bildschirmdimensionen?

Diese Funktionalität hat kein Layout.

18. Haben alle Methoden und Felder die richtigen Zugriffsmodifier?

Ja.

Da uns der Aufwand für die Nutzung eines Instrumented-Test-Frameworks für das recht einfache User Interface der App unverhältnismäßig erscheint, werden manuelle Tests anhand der folgenden Testpläne vorgenommen.

Die einzelnen Testabläufe sind nach funktionalen Bereichen der App aufgeteilt.

I) Darstellung

1. Darstellung der Kontaktübersicht

Die Kontaktübersicht(ContactsActivity) soll so übersichtlich wie möglich gehalten sein. Dazu sind einige Dinge zu beachten.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer öffnet die Kontaktübersicht	Dem Benutzer werden alle Kontakte aufsteigend alphabetisch angezeigt, also oben beginnend mit Kontaktnamen die mit A anfangen usw.

2. Darstellung der Unterhaltungsübersicht

Die Unterhaltungsübersicht(ConversationsActivity) soll so übersichtlich wie möglich gehalten sein. Dazu sind einige Dinge zu beachten.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer öffnet die Unterhaltungsübersicht	Dem Benutzer werden alle Unterhaltungen angezeigt, und zwar geordnet nach der Zeit der letzten Nachricht, beginnend mit der Unterhaltung mit der neuesten Nachricht. Die aktuellste Nachricht wird unter dem Namen der Unterhaltung angezeigt

3. Darstellung der Chatansicht

Der Benutzer soll möglichst übersichtlich Nachrichten lesen können. Dementsprechend muss die Chatansicht (MessagesActivity) gestaltet sein.

Allgemeine Darstellung der Nachrichten

Nachrichten in der Chatansicht werden grundsätzlich wie hier beschrieben dargestellt. Die neuesten Nachrichten sind unten zu finden, die ältesten oben. Nachrichten des Benutzers werden rechts angezeigt, die seines Chatpartners links. Neben der Nachricht wird das Kontaktbild des Absenders angezeigt. Unter dem Nachrichtentext sind durch folgende Symbole

die Nachrichtendetail dargestellt:

- Ein Schlosssymbol, wenn die Nachricht verschlüsselt ist
- Das Bluetooth-Symbol, wenn die Nachricht über Bluetooth versendet wurde
- Das WiFi-Symbol, wenn die Nachricht über WiFi peer-to-peer versendet wurde
- Eine Wolke, wenn die Nachricht über Google-Cloud-Messaging versendet wurde
- Ein rotes Dreieck mit Ausrufezeichen darin, wenn die Nachricht nicht angekommen ist

Außerdem wird noch der genaue Sendezeitpunkt angezeigt.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer öffnet eine Unterhaltung	Die Tastatur öffnet sich und der Benutzer kann einen Text eingeben. Gleichzeitig werden die vergangenen Nachrichten des Chats angezeigt wie in Allgemeine Darstellung von Nachrichten (Manueller-Test-UI#allgemeine-darstellung-der-nachrichten) beschrieben

II) Erster Start und Einstellungen

1. Willkommensseite

Benutzeraktion	Erwartetes Verhalten der App
Die App wird nach Neuinstallation / Löschen der App-Daten geöffnet.	Im Hintergrund wird ein Schlüsselpaar für das Gerät generiert. Es öffnet sich eine Begrüßungsseite, auf der der Nutzer zur Eingabe eines Namens (Nickname) aufgefordert wird.
Es wird ein Nickname eingetragen, der bereits vergeben ist. Der Bestätigungsbutton wird betätigt.	Es wird ein HTTP-Post-Request an das Server-API gesendet, welcher Public-Key, Nickname und optional Handynummer enthält.
	Es wird eine Fehlermeldung angezeigt, die den Nutzer auffordert, einen anderen Nickname zu wählen. Die Begrüßungsseite bleibt sichtbar.
Es wird ein neuer Nickname eingetragen. Der Bestätigungsbutton wird betätigt.	Die Begrüßungsseite schließt sich und der Benutzer gelangt in die Haupt-Activity der App, wobei der Navigations-Drawer geöffnet ist.

Benutzeraktion	Erwartetes Verhalten der App
Die App wird durch Betätigen der Back-Taste verlassen.	

2. Automatisches Öffnen des NavigationDrawer

Um dem Benutzer die Navigation in der App nahezubringen, soll, wie in den Android-UI-Richtlinien empfohlen, das Navigationsmenü (NavigationDrawer) so lange beim App-Start automatisch eingeblendet werden, bis der Benutzer es einmal selbstständig geöffnet hat.

Benutzeraktion	Erwartetes Verhalten der App
Die App wird aus der App-Liste gestartet	Die MainActivity der App öffnet sich, der Menüpunkt "Unterhaltungen" ist ausgewählt, der NavigationDrawer ist geöffnet.
Schließen des NavigationDrawer durch Klick auf "Unterhaltungen". Wieder Öffnen des NavigationDrawer durch Hereinwischen vom linken Bildschirmrand.	Der NavigationDrawer schließt sich und öffnet sich wieder.
Beenden der App durch Betätigen der Back-Taste	
Die App wird erneut aus der App-Liste gestartet	Die MainActivity der App öffnet sich, der Menüpunkt "Unterhaltungen" ist sichtbar, der NavigationDrawer ist nicht geöffnet.
Beenden der App durch Betätigen der Back-Taste	

3. Tutorial beim ersten Start

Um dem Benutzer den Einstieg in die Benutzung der App zu vereinfachen, wird er beim ersten Start der App durch ein interaktives Tutorial geführt. Dieses muss einwandfrei funktionieren, damit der Nutzer nicht verwirrt wird.

Benutzeraktion	Erwartetes Verhalten der App
----------------	------------------------------

Benutzeraktion	Erwartetes Verhalten der App
Die App wird nach Neuinstallation / Löschen der App-Daten geöffnet. Der Benutzer gibt einen noch freien Benutzernamen und seine Telefonnummer ein und drückt auf OK	Dem Benutzer wird der NavigationDrawer gezeigt und darauf hingewiesen, dass er mit diesem in der App navigieren kann. Des Weiteren wird ein Knopf eingeblendet, mit dem der Benutzer zum nächsten Teil des Tutorials springen kann
Der Benutzer drückt auf den eben beschriebenen Knopf	Es wird der Share-Button zum Teilen der Kontaktinformationen hervorgehoben, und dem Benutzer seine Funktionsweise beschrieben. Des Weiteren wird ein Knopf eingeblendet, mit dem der Benutzer zum nächsten Teil des Tutorials springen kann
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer wird aufgefordert, auf Unterhaltungen zu klicken
Der Benutzer klickt auf Unterhaltungen	Die Unterhaltungsübersicht öffnet sich, und es ist bereits eine Unterhaltung zu sehen. Der Benutzer wird aufgefordert, die Unterhaltung anzuklicken
Der Benutzer klickt die Unterhaltung an	Die Unterhaltung öffnet sich und der Benutzer wird als erstes auf die Titel-ändern-Funktion hingewiesen. Auch wird ihm wieder ein Weiter-Knopf angezeigt
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer wird nun auf die Bilder-schicken-Funktion hingewiesen. Auch wird ihm wieder ein Weiter-Knopf angezeigt
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer wird auf die Standort-Teilen-Funkiton hingewiesen. Auch wird ihm wieder ein Weiter-Knopf angezeigt
Der Benutzer drückt auf den eben beschriebenen Knopf	Eine Nachricht wird hervorgehoben und dem Benutzer erklärt, dass er durch Tippen auf die Nachricht in die Nachrichtendetailansicht gelangen kann. Auch wird wieder ein Weiter-Knopf angezeigt
Der Benutzer drückt auf den eben beschriebenen Knopf	Der NavigationDrawer wird wieder eingeblendet und der Benutzer wird aufgefordert, auf Kontakte zu klicken.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer klickt auf Kontakte	Die Kontaktübersicht öffnet sich und der Benutzer wird auf die Kontaktdaten-teilen-Funktion hingewiesen. Auch wird wieder ein Weiter-Knopf angezeigt.
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer wird auf die QR-Code-Scannen-Funktion hingewiesen. Auch wird wieder ein Weiter-Knopf angezeigt.
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer wird auf die Kontakte-Suchen-Funktion hingewiesen. Nun wird ein "Los gehts"-Knopf angezeigt.
Der Benutzer drückt auf den eben beschriebenen Knopf	Der Benutzer gelangt wieder in die Unterhaltungsübersicht. Das Tutorial ist beendet.

III) Senden und Empfangen von Nachrichten

1. Korrektes Anzeigen von Benachrichtigungen

Benachrichtigungen über neue Nachrichten sollen nur angezeigt werden, wenn das entsprechende Chatfenster nicht geöffnet ist. Außerdem sollen diese Benachrichtigungen automatisch verschwinden, wenn die betreffende Nachricht gelesen wurde.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer befindet sich auf dem Startbildschirm der App, eine Nachricht trifft ein	Es wird eine Benachrichtigung in der Benachrichtigungsleiste angezeigt, beim Klicken auf die Benachrichtigung gelangt der Benutzer zum Chat, in dem die Nachricht gesendet wurde, die Benachrichtigung verschwindet
Der Benutzer befindet sich in einem Chat (in der MessagesActivity), eine Nachricht trifft in diesem Chat ein.	Es wird keine Benachrichtigung angezeigt

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer befindet sich in einem Zustand, in dem eine Benachrichtigung über eine Nachricht angezeigt wird. Dann liest der die entsprechende Nachricht, indem er direkt - ohne anklicken der Benachrichtigung - in den entsprechenden Chat geht	Die Benachrichtigung verschwindet

2. Senden und empfangen von normalen Nachrichten

Benutzeraktion	Erwartetes Verhalten der App
Benutzer befindet sich in der Chatansicht und tippt einen Nachrichtentext in das Textfeld ein. Dann drückt er die Sendetaste	Die Nachricht wird sofort an den Chatpartner gesendet. Die Nachricht wird sofort angezeigt, wie in Allgemeine Darstellung von Nachrichten (Manueller-Test-UI#allgemeine-darstellung-der-nachrichten) beschrieben.
Der Benutzer empfängt eine Nachricht und geht dann in die entsprechende Chatansicht oder ist dort bereits	Alle Nachrichten werden wie in Allgemeine Darstellung von Nachrichten (Manueller-Test-UI#allgemeine-darstellung-der-nachrichten) angezeigt. Die Empfangene Nachricht ist bereits enthalten.

3. Senden und empfangen von Bildern

Das Senden von Bildern soll möglichst einfach in der Bedienung sein und den Benutzer nicht verwirren. Vor allem soll die App nicht bei unerwarteten Eingaben abstürzen.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer klickt in der Chatansicht auf das Bild-Senden-Symbol	Es öffnet sich eine von Android bereitgestellte Liste mit versendbaren Bildern.
Der Benutzer klickt auf ein Bild in der eben beschriebenen Liste	Das Bild wird sofort an den Chatpartner gesendet. Die App kehrt zu betreffenden Chat zurück. Dort ist nun das Bild zu sehen.
Der Benutzer wählt in der Liste kein Bild aus und kehrt mittels des Zurück-Knopfs zum Chat zurück	Es passiert nichts, insbesondere stürzt die App nicht ab.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer versucht, ein nicht darstellbares Bild zu versenden	Es wird eine Fehlermeldung angezeigt, dass das Bild nicht versendet werden kann. Das Bild wird nicht versendet. Die App kehrt zur MessagesActivity zurück.
Der Benutzer bekommt in einem Chat ein Bild gesendet. Daraufhin wechselt er in diesen Chat	Der Benutzer sieht das ihm gesendete Bild

4. Teilen des aktuellen Standorts

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer befindet sich in der Chatansicht, er tippt auf das Kompass-Symbol in der Menüleiste. Sein Standort kann bestimmt werden	Der Standort wird zum Chatpartner gesendet. Es ist eine Google-Maps-Vorschau des Standorts zu sehen
Der Benutzer befindet sich in der Chatansicht, er tippt auf das Kompass-Symbol in der Menüleiste. Sein Standort kann nicht bestimmt werden.	Es wird eine Fehlermeldung eingeblendet, die aussagt, dass der Standort nicht bestimmt werden kann und deswegen auch nicht gesendet werden kann
Der Benutzer empfängt eine Standortteilung	Die App verhält sich wie bei normalen Nachrichten. In der Chatansicht sieht er eine Google-Maps-Vorschau des Standorts
Der Benutzer tippt auf eine Vorschau eines Standorts	Eine neue Ansicht öffnet sich, in der der Standort auf einer Karte in Großansicht angezeigt wird.

IV) Löschen von Elementen

1. Löschen von Kontakten

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer ist in der Kontaktansicht und wählt durch langes Tippen auf einen Kontakt denselben aus	Die Menüleiste wird durch eine auswahlspezifische Menüleiste ersetzt, in der auch ein Müllimer als Lösch-Symbol zu sehen ist

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer wählt einen Kontakt aus und löscht ihn durch Tippen auf das Mülleimer-Symbol	Der Kontakt wird aus der Datenbank gelöscht und verschwindet sofort aus der Ansicht. Mit dem Kontakt werden auch sämtliche mit dem Kontakt geführte Unterhaltungen gelöscht, die ab sofort nicht mehr in der Unterhaltungsübersicht zu sehen sind
Nach dem Löschen eines spezifischen Kontakts wechselt der Benutzer auf eine andere Ansicht und öffnet dann die Kontaktansicht erneut.	Der vorher gelöschte Kontakt ist immer noch nicht zu sehen

2. Löschen von Unterhaltungen

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer ist in der Unterhaltungsansicht und wählt durch langes Tippen auf eine Unterhaltung dieselbe aus	Die Menüleiste wird durch eine auswahlspezifische Menüleiste ersetzt, in der auch ein Mülleimer als Lösch-Symbol zu sehen ist
Der Benutzer wählt eine Unterhaltung aus und löscht sie durch Tippen auf das Mülleimer-Symbol	Die Unterhaltung wird aus der Datenbank gelöscht und verschwindet sofort aus der Ansicht.
Nach dem Löschen einer spezifischen Unterhaltung wechselt der Benutzer auf eine andere Ansicht und öffnet dann die Unterhaltungsansicht erneut.	Die vorher gelöschte Unterhaltung ist immer noch nicht zu sehen

3. Löschen von Nachrichten

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer ist in der Chatansicht und wählt durch langes Tippen auf eine Nachricht dieselbe aus	Die Menüleiste wird durch eine auswahlspezifische Menüleiste ersetzt, in der auch ein Mülleimer als Lösch-Symbol zu sehen ist

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer wählt eine Nachricht aus und löscht sie durch Tippen auf das Müllsymbol	Die Nachricht wird aus der Datenbank gelöscht und verschwindet sofort aus der Ansicht.
Nach dem Löschen einer spezifischen Nachricht wechselt der Benutzer auf eine andere Ansicht und öffnet dann die Chatansicht erneut.	Die vorher gelöschte Nachricht ist immer noch nicht zu sehen

V) Diverses

1. Korrekte Funktionsweise der regelmäßigen Standortteilung

Die Ansicht, unter der man den dauerhaft geteilten Standort eines Chatpartners sehen kann, soll automatisch aktualisiert werden, wenn ein Standortupdate eintrifft.

Benutzeraktion	Erwartetes Verhalten der App
Der Benutzer ist in der Kontaktansicht und wählt durch langes Tippen auf einen Kontakt denselben aus	Die Menüleiste wird durch eine auswahlspezifische Menüleiste ersetzt, in der auch ein Standort-Symbol zu sehen ist
Der Benutzer b1 hat den Kontakt b2 ausgewählt und tippt auf das Standort-Symbol. Der b2 hat seinen Standort noch nicht geteilt	Es wird dem Benutzer b1 eine Ansicht angezeigt, die ihm sagt, dass kein Standort vorliegt
Benutzer b1 befindet sich in der eben beschriebenen Ansicht. b2 geht nun in das Kontakt-bearbeiten-Menü von b1 und setzt das "Diesem Kontakt meinen Standort mitteilen"-Häkchen	Die von b1 geöffnete Ansicht zeigt nun eine Karte, auf der der Standort von b2 zu sehen ist.
b1 befindet sich in der Ansicht, in der der Standort von b2 zu sehen ist. b2 entfernt sich um mindestens 50m von seinem Ausgangspunkt.	b1 sieht nun innerhalb des aktuellen Aktualisierungintervalls des Standorts eine veränderte Position von b2 auf der Karte

Durchführung des manuellen UI-Testplans vom 19.08.15

Durchgeführt von: Jonas Mönnig

Änderungen vorgenommen in Commits: 111c37d20d97f5314f23ff38bf3a204624e44986, 029b846b7423357bc05f1ca0bbf71e39bb7b7f43, d4dd283de2fc9cccd82fe93d0858d2f7816cd669

Hinweis: Diese Testplanausführung wurde mit einer älteren Version des Tesplans durchgeführt, die an einigen Stellen vom aktuellen Testplan abweicht. An den entsprechenden Stellen sind Hinweise angebracht. Außerdem wurden die Testnummerierungen so angepasst, dass sie der aktuellen Nummerierung entsprechen.

Teil I: Darstellung

Hinweis: Test I.1 und Test I.2 beinhalteten damals noch nicht, dass Kontakte bzw. Unterhaltungen sortiert sein müssen, weil dies damals noch keine Anforderung war.

Test I.1

Bestanden.

Test I.2

Bestanden.

Test I.3

Bestanden.

Teil II: Erster Start und Einstellungen

Hinweis: Test II.3 gab es damals noch nicht, da es auch kein Tutorial gab.

Test II.1

Bestanden.

Test II.2

Der NavigationDrawer ist beim Start nicht geöffnet. (Update 26.08.15: Dies wurde von Max Weller behoben)

Teil III: Senden und Empfangen von Nachrichten

Hinweis: Es gab damals noch keinen Standtaustausch und keinen Bilderversand. Darum

fehlen auch die entsprechenden Tests.

Test III.1

Nicht bestanden, da die Benachrichtigungen einfach immer angezeigt werden und auch nicht verschwinden, wenn die Nachricht gelesen wurde. (Update 01.09.15: Dies wurde von Jonas Mönnig behoben)

Test III.2

Bestanden.

Teil IV: Löschen von Elementen

Test IV.1

Nicht bestanden. Konversationen mit gelöschten Kontakten blieben weiterhin sichtbar. (Update 30.08.15: Dies wurde von Johannes Lauinger behoben)

Test IV.2

Bestanden.

Test IV.3

Bestanden.

Hinweis: Die Standortteilen-Funktion gab es damals noch nicht, daher auch keinen Test.

Durchführung des manuellen UI-Testplans vom 08.09.2015

Durchgeführt von: Jonas Mönnig

Änderungen vorgenommen in Commit: TODO: Commit einfügen

Teil I: Darstellung

Test I.1

Bestanden

Test I.2

Bestanden

Test I.3

Bestanden

Teil II: Erster Start und Einstellungen

Test II.1

Bestanden

Test II.2

Bestanden

Test II.3

Nicht bestanden. Die Benutzerführung ist eine völlig andere. Johannes Lauinger wird beauftragt, den Fehler zu beheben.

Teil III: Senden und Empfangen von Nachrichten

Test III.1

Bestanden

Test III.2

Bestanden

Test III.3

Bestanden

Test III.4

Bestanden

Teil IV: Löschen von Elementen

Test IV.1

Bestanden

Test IV.2

Bestanden

Test IV.3

Bestanden

Teil V: Diverses

Test V.1

Bestanden

BonfireChat Dokumentation

- [Datenstrukturen](#)
 - [Identität](#)
 - [IPublicIdentity](#)
 - [Public Key Infrastruktur](#)
 - [Identity](#)
 - [Contact](#)
 - [Nachrichten](#)
 - [Unterhaltungen](#)
- [Nachrichtenübertragung](#)
 - [Senden und Empfangen](#)
 - [ConnectionManager](#)
 - [SocketProtocol](#)
 - [Datenstruktur Umschlag](#)
 - [verschlüsselt übertragene Daten](#)
 - [Traceroute](#)
- [Kryptographie: Bibliothek libkalium](#)
- [Protokolle](#)
 - [BluetoothProtocol](#)
 - [WiFiProtocol](#)
 - [GcmProtocol: Client-Server-Architektur](#)
- [Routing](#)
 - [Flooding](#)
 - [Geplante Mechanismen](#)
- [Kontaktaustausch](#)
 - [serverbasierte Suche](#)
 - [direkter Austausch \(NFC oder QR-Code\)](#)
- [lokale Datenhaltung: SQLite](#)
- [serverseitige Datenhaltung](#)
 - [MySQL-Datenbank](#)
 - [API](#)

- [POST /register.php](#)
- [GET /search.php](#)
- [POST /traceroute.php](#)
- [GET /traceroute.php](#)

- [User Interface Design](#)

- [Menü und Activities](#)
- [Nachrichten](#)
- [kontakte bearbeiten](#)
- [eigene Identität teilen](#)

Datenstrukturen

Im Package `bonfirechat.models` befinden sich alle Klassen, mit denen die verwendeten Datenstrukturen dargestellt werden. Diese werden im Folgenden vorgestellt.

Identität

Zunächst wird eine Person beschrieben. Dabei handelt es sich momentan auch um ein eindeutig identifiziertes Gerät. Eine Person kann also die gleiche Identität nicht auf mehreren Geräten verwenden. Dies könnte in Zukunft noch geändert werden. Zur Darstellung einer Identität existieren die folgenden Klassen:

`IPublicIdentity`

Das Interface `IPublicIdentity` fasst alle Daten zusammen, die zur Identifikation einer Person unbedingt erforderlich sind. Es abstrahiert von Daten, die für die konkrete Speicherung der Person auf einem Gerät notwendig sind, und enthält keinerlei private Informationen. Daher kann eine Instanz von `IPublicIdentity` problemlos zusammen mit einer Nachricht verschickt werden, reicht aber aus um keine Doppeldeutigkeiten bei der Zuordnung der Person zu verursachen.

Folgende Funktionen sind vorhanden:

```
MyPublicKey getPublicKey();
String getNickname();
String getXmppId();
String getPhoneNumber();
```

Public Key Infrastruktur

Unbedingt notwendig zur Identifikation einer Person ist der Public Key. Wir setzen Public-Key-Kryptographie ein, um Nachrichten verschlüsselt zu übertragen und die Authentizität des Absenders zu verifizieren. Für jeden neuen Benutzer wird daher beim Erstellen der Identität ein

Schlüsselpaar generiert. Genaue Informationen zur Erstellung finden sich im Abschnitt *Kryptographie*. Es handelt sich um einen 32 Byte Schlüssel für Curve25519.

Die Klasse `MyPublicKey` erweitert die Basisklasse `PublicKey` um einige nützliche Funktionen. Dazu gehören:

```
public byte[] asByteArray();
public String asBase64();
public static MyPublicKey deserialize(byte[] publicKey);
public static MyPublicKey deserialize(String base64publicKey);
```

Identity

Um abzubilden, dass der Benutzer über seine eigene Identität mehr Informationen verwalten muss als über fremde Kontakte, wird `IPublicIdentity` von zwei Klassen implementiert, die diese beiden Fälle abdecken. Die Klasse `Identity` stellt dabei die eigene Identität dar. Ein Objekt dieser Klasse wird beim ersten Start von BonfireChat erzeugt und serverseitig registriert.

Folgende Felder sind vorhanden:

```
public String nickname, server, username, password, phone;
public MyPublicKey publicKey;
public PrivateKey privateKey;
public long rowid; // Zeilen-ID in der Datenbank
```

Da insbesondere der `PrivateKey` in dieser Klasse gespeichert wird, darf eine Instanz von `Identity` natürlich nicht versendet werden.

Contact

Für fremde Kontakte, mit denen Nachrichten ausgetauscht werden, gibt es die Klasse `Contact`. Auch `Contact` implementiert `IPublicIdentity`. In `Contact`-Instanzen können auch Daten darüber gespeichert werden, wie der Kontakt momentan erreichbar ist. Dazu gehören z.B. WiFi- und Bluetooth-Adressen.

Eine `Contact`-Instanz bietet folgende Felder:

```
private String nickname, firstName, lastName, phoneNumber;
public MyPublicKey publicKey;
public String xmppId; // Daten über die Erreichbarkeit des Kontakts. Diese werden momentan nicht verwendet.
public String wifiMacAddress;
public String bluetoothMacAddress;
public long rowid; // Zeilen-ID in der Datenbank
```

Nachrichten

Eine einzelne Nachricht wird als Instanz der Klasse `Message` dargestellt. Dabei stellt das Message-Objekt die Nachricht dar, nachdem sie empfangen oder bevor sie gesendet wurde. Es handelt sich also um die Repräsentation auf einem Gerät, nicht um die Daten, die versendet werden. Für diesen Zweck würde eine Instanz der Klasse `Envelope` verwendet werden, welche genauer im Kapitel *Nachrichtenübertragung* beschrieben wird.

Eine Nachricht besteht aus den folgenden Feldern:

```
public UUID uuid;
public Date sentTime;
public IPublicIdentity sender;
public List<Contact> recipients;
public String body;
public String transferProtocol;
public String error;
public int flags;
```

Die `uuid` identifiziert eine einzelne Nachricht eindeutig. Diese wird insbesondere verwendet, um zu überprüfen, ob eine Nachricht schon mehrfach empfangen und/oder gesendet wurde, und als eindeutige Kennzeichnung der Nachricht in der Datenbank. Die UUID wird beim Erstellen einer neuen Nachricht mit `UUID.randomUUID()` zufällig generiert.

Der `sender` ist als Instanz von `IPublicIdentity` implementiert, da hier je nachdem, ob die Nachricht versendet oder empfangen wurde, unterschiedliche Dinge eingetragen werden: ist man selbst der Absender der Nachricht, steht das eigene `Identity` Objekt im Absender. Empfängt man die Nachricht, wird aus dem `sender` Feld des empfangenen `Envelope` Objekts eine Instanz von `Contact` erzeugt.

Mit diesem Mechanismus wird auch festgestellt, ob eine Nachricht empfangen oder gesendet wurde. Das ist unter Anderem für die visuelle Darstellung in den UI Klassen notwendig. Für diesen Zweck existiert die folgende Funktion:

```
public MessageDirection direction() {
    return (sender instanceof Identity) ? MessageDirection.Sent :
MessageDirection.Received;
}
```

Folgende Werte sind für `MessageDirection` möglich:

```
public enum MessageDirection {
    Unknown, // wird momentan nicht verwendet
    Sent,
    Received
}
```

Im Feld `transferProtocol` wird das verwendete Protokoll in Textform gespeichert, um den

Benutzer zu informieren, wie seine Nachricht angekommen ist. Tritt beim Versenden eine Exception auf, wird die Fehlermeldung im Feld `error` abgelegt und als Nachrichtentext angezeigt. Das ist vor Allem während der Entwicklung zur Fehlersuche sehr hilfreich, wenn keine Zugriff auf die ADB-Logs vorhanden ist.

Im Integer `flags` werden verschiedene Informationen zur Nachricht als Flags kodiert. Folgende Flags können dabei gesetzt werden:

```
public static final int FLAG_ENCRYPTED = 4;
public static final int FLAG_PROTO_BT = 16;
public static final int FLAG_PROTO_WIFI = 32;
public static final int FLAG_PROTO_CLOUD = 64;
```

Das Flag `FLAG_ENCRYPTED` wird gesetzt, wenn die Übertragung mit Ende-zu-Ende-Verschlüsselung stattfindet. Die anderen Flags markieren, über welche Protokolle die Nachricht gesendet wurde. Dies können auch mehrere sein. Damit werden unter Anderem in den UI Klassen kleine Icons angezeigt.

Unterhaltungen

Nachrichten werden in BonfireChat, ähnlich wie in anderen Instant Messengern, in Unterhaltungen mit Kontakten gruppiert. Momentan werden nur Einzelchats unterstützt, es ist also nicht möglich, mehrere Kontakte als Unterhaltungsteilnehmer einzutragen. Falls in der Zukunft Mehrbenutzerchats unterstützt werden sollen, müsste dies hier erweitert werden. Die anderen Datenstrukturen, insbesondere die Repräsentation einer Nachricht beim Verschicken, sind teilweise bereits darauf vorbereitet, Mehrbenutzerchats darzustellen.

Eine Unterhaltung wird als Instanz von `Conversation` dargestellt. Diese besteht aus folgenden Feldern:

```
private Contact peer;
private ArrayList<Message> messages;
public String title;
public ConversationType conversationType = ConversationType.Single;
public long rowid; // Zeilen-ID in der Datenbank
```

Die Nachrichten, die zu der Unterhaltung gehören, werden als `Message` Objekte in einer `ArrayList` gespeichert. Diese Liste enthält dabei sowohl die gesendeten als auch die empfangenen Nachrichten.

Es ist möglich, eine Unterhaltung zu benennen. Der Name wird dann im Feld `title` gespeichert. Standardmäßig wird als Titel der Name des Kontaktes, mit dem die Unterhaltung geführt wird, verwendet.

Falls Mehrbenutzerchats unterstützt werden, würde der das Feld `conversationType` auf einen anderen Wert als `Single` gesetzt werden. Die folgende möglichen Werte existieren:

```
public enum ConversationType {  
    Single,  
    GroupChat  
}
```

Nachrichtenübertragung

Die bis jetzt behandelten Datenstrukturen haben sich alle mit der Speicherung der Daten auf einem lokalen Gerät befasst. Im Folgenden soll es nun um die tatsächliche Übertragung von Daten gehen.

Senden und Empfangen

Da BonfireChat verschiedene Protokolle unterstützt, gibt es einige Abstraktionsebenen, die die Teile der Nachrichtenübermittlung bündeln, die bei allen verwendeten Technologien gleich bleiben.

ConnectionManager

Es gibt eine Instanz der Klasse `bonfirechat.network.ConnectionManager`. Diese kümmert sich darum, die einzelnen Protokolle zu verwalten und stellt die Schnittstelle zwischen den Protokollen, also der Netzwerkschicht, und der Darstellung, also den UI Klassen, dar.

Der `ConnectionManager` ist ein `NonStopIntentService`. Um eine Nachricht zu versenden, können neue Intent-Objekte erstellt und an den `ConnectionManager` gesendet werden. Dafür gibt es die folgenden statischen Wrapper-Methoden:

```
public static void sendEnvelope(Context ctx, Envelope envelope);  
public static void sendMessage(Context ctx, Message message);
```

Diese erstellen den Intent und starten ihn. In der `onHandleIntent` Methode bearbeitet der `ConnectionManager` ihn anschließend. Dies stellt eine FIFO-Queue dar. Neue zu sendende Nachrichten werden hinten eingereiht und von einem separaten Thread abgeholt und bearbeitet.

Der `ConnectionManager` sucht sich ein verfügbares Protokoll aus und überprüft in den App-Einstellungen, ob dieses verwendet werden soll. Anschließend wird die Nachricht an dieses Protokoll zum Senden übergeben. Genauere Informationen zu den Protokollen folgen im Kapitel *Protokolle*.

Neben dem Senden kümmert sich der `ConnectionManager` auch um das Empfangen von Nachrichten. Dafür implementiert er eine Instanz des Interfaces `OnMessageReceivedListener` und übergibt eine Referenz darauf den Protokollklassen. Wird von einem Protokoll nun eine Nachricht empfangen, kann es diese Instanz als Callback verwenden.

In der Methode `onMessageReceived` wird anschließend vom `ConnectionManager` geprüft, ob die Nachricht bereits bearbeitet wurde. Dies ist notwendig, da es beim Mesh-Netz-Routing dazu kommen kann, dass die gleiche Nachricht mehrmals ankommt, oder eine bereits gesendete Nachricht wieder zurückkommt. Dafür wird eine Implementierung der Datenstruktur Ringpuffer, nämlich `bonfirechat.helper.RingBuffer` verwendet. Die UUIDs aller Nachrichten, die bereits bearbeitet (also entweder empfangen oder gesendet) wurden, werden an den Ringpuffer angehängt. Beim Empfangen wird nun überprüft, ob die UUID der eingehenden Nachricht im Puffer existiert, und in diesem Fall nichts getan. Der Puffer hat eine Größe von 250 Einträgen. Wird er voll, werden alte Nachrichten einfach überschrieben und damit vergessen. Dies ist kein Problem, da diese Nachrichten mit sehr großer Wahrscheinlichkeit bereits angekommen sind oder wegen großem `hopCount` verworfen wurden.

Bei neuen empfangene Nachrichten wird überprüft, ob der eigene Public Key in der Liste der Empfänger vorhanden ist. Wenn das der Fall ist, wird die Nachricht in der Datenbank gespeichert, wenn nötig ein neues `Conversation` oder sogar `Contact` Objekt erstellt und die Nachricht zur Darstellung mit einem neuen Intent an die UI-Klassen übergeben.

Zusätzlich wird überprüft, ob weitere Empfänger vorhanden sind oder die Nachricht nicht für das Gerät bestimmt war. In diesem Fall wird die Nachricht weiter verbreitet, falls sie (angezeigt durch ihren `hopCount`) noch nicht mehr als 20 Hops durchlaufen hat. Mehr zu dieser Technik wird im Kapitel *Routing* behandelt. Zum erneuten Senden erstellt der `ConnectionManager` einfach einen neuen Intent, sodass die Nachricht in die Sendewarteschlange eingereiht wird.

SocketProtocol

Auch die Kodierung von Nachrichten von Nachrichten in eine Form, die zum Senden geeignet ist, ist für alle Protokolle gleich. Daher existiert die abstrakte Basisklasse `SocketProtocol`, welche diese Operationen kapselt. Die notwendigen Funktionen, um einem Protokoll die eigene Identität und das Callback-Objekt für eingehende Nachrichten zuzuweisen, werden hier bereits ausformuliert.

Daneben sind die beiden Funktionen `sendEnvelope` und `receiveEnvelope` implementiert. Damit wird der Transport von `Envelope` Objekten über Streams abstrahiert. Die konkreten Protokolle müssen also nur noch ein Socket zum Empfänger öffnen und einen `OutputStream` bzw. `InputStream` erzeugen.

Das eigentliche `Envelope` Objekt wird mit `writeObject` bzw. `readObject` geschrieben und gelesen. Es wird also die Standard-Java-Serialisierung verwendet.

Datenstruktur Umschlag

Um Nachrichten auf dem Transportweg zu repräsentieren, werden sie in einen Umschlag eingepackt. Dieser wird durch ein Objekt der Klasse `Envelope` dargestellt. Damit das `Envelope`-Objekt im Zusammenhang mit Streams verwendet werden kann, implementiert die Klasse das Interface `Serializable`.

Die folgenden Felder sind vorhanden:

```
public UUID uuid;
public int hopCount;
public Date sentTime;
public ArrayList<byte[]> recipientsPublicKeys;
public String senderNickname; // eventuell entfernen, für
Entwicklungszwecke aber sehr praktisch
public byte[] senderPublicKey;
public byte[] encryptedBody; // verschlüsselt
public byte[] nonce;
public int flags;
```

Zusätzlich zur `uuid`, welche die Nachricht und damit den Umschlag eindeutig kennzeichnet, wird das Feld `hopCount` geführt. Dieses wird bei jedem Empfangen des Umschlags hochgezählt, sodass damit festgestellt werden kann, über wie viele Geräte die Nachricht an ihr Ziel gelangt ist.

Von den Empfängern werden nur die Public Keys übertragen, da diese zur Identifikation ausreichen. Dabei wird auch nicht das vollständige `MyPublicKey` Objekt übertragen, sondern nur der Schlüssel selbst als Byte-Array.

Der Absender wird ebenfalls durch seinen Public Key gekennzeichnet. Zusätzlich wird vom Absender der Nickname übertragen. Das ist zwar nicht unbedingt nötig, aber sehr praktisch, um in Logs oder beim Empfänger ohne Anfragen an einen Server anzuzeigen, von wem die Nachricht ist.

Im Integer `flags` werden, ähnlich wie bei `Message` Objekten, verschiedene Daten zum Umschlag abgelegt. Folgende Flags sind möglich:

```
public static final int FLAG_ENCRYPTED = 4;
public static final int FLAG_TRACEROUTE = 8;
```

`FLAG_ENCRYPTED` zeigt an, dass der Nachrichteninhalt verschlüsselt im Umschlag steht. `FLAG_TRACEROUTE` markiert eine spezielle Traceroute-Nachricht. Mehr dazu wird im Abschnitt *Traceroute* beschrieben.

verschlüsselt übertragene Daten

Das einzige Feld, das im `Envelope` Objekt verschlüsselt übertragen wird, ist `encryptedBody`. Dieses ist daher auch direkt ein Byte-Array und kein String. In BonfireChat werden alle Strings mit UTF-8 Kodierung zu Byte-Arrays und zurück transformiert.

Für die Verschlüsselung notwendig ist außerdem das Feld `nonce`. Genaue Informationen, wie die Verschlüsselung funktioniert, folgen im Kapitel *Kryptographie*.

Die Felder `uuid`, `hopCount` und `recipientsPublicKeys` zu verschlüsseln, macht keinen

Sinn, da dann Routing nicht mehr möglich wäre und die Nachrichten nicht ankommen könnten. nonce und flags sind für die Entschlüsselung notwendig und müssen daher ebenfalls im Klartext übertragen werden.

Die Felder sentTime, senderPublicKey und senderNickname könnten verschlüsselt werden, da nur der Empfänger wissen muss, von wann und wem die Nachricht stammt. Dies ist momentan aber noch nicht umgesetzt.

Traceroute

Um nachvollziehen zu können, über welche Geräte und Protokolle eine Nachricht versendet wurde, gibt es den speziellen Nachrichtentyp Traceroute. Ein Umschlag wird mit FLAG_TRACEROUTE als solche markiert. Beim Senden und Empfangen eines Umschlags übergibt der ConnectionManager diesen an die statische Klasse TracerouteHandler. Dieser überprüft, ob das Flag gesetzt ist, und hängt in diesem Fall eine Zeile an den Nachrichteninhalt an. Traceroute Nachrichten werden stets unverschlüsselt übertragen, damit dies möglich ist.

In der angehängten Zeile ist vermerkt, ob der Umschlag empfangen oder gesendet wurde, über welches Protokoll dies geschah und auf welchem Gerät die Aktion stattgefunden hat.

Schließlich wird beim Empfangen einer Traceroute-Nachricht die Funktion TracerouteHandler.publishTraceroute aufgerufen. Diese übermittelt den Nachrichteninhalt an einen Server und ersetzt den lokalen Inhalt durch einen Link auf eine Seite, auf der man die Route anschauen kann. Auf diese Weise werden Traceroute-Nachrichten beim Empfänger sinnvoll dargestellt und auch der Absender kann sich unter dem Link die Route anschauen.

Kryptographie: Bibliothek libkalium

Alle kryptographischen Operationen werden durch die Bibliothek Kalium von AbstractJ (<https://github.com/abstractj/kalium>) unterstützt. Dabei handelt es sich um Java-Bindings und das JNI Objekt libkaliumjni.so.

Kalium ist eine Portierung der NaCl Bibliothek von Daniel J. Bernstein auf Android. Diese stellt Funktionen zur Generierung von Schlüsselpaaren und zufälligen Nonces sowie zur sicheren Ver- und Entschlüsselung der Nachrichteninhalte bereit. In der Envelope Klasse wird eine Nachricht auf die folgende Weise verschlüsselt:

```
if (encrypt) {  
    Identity sender = (Identity) message.sender;  
    Box crypto = new Box(new PublicKey(publicKeys.get(0)),  
    sender.privateKey);  
    envelope.nonce = CryptoHelper.generateNonce();  
    envelope.encryptedBody = crypto.encrypt(envelope.nonce,  
    envelope.encryptedBody);
```

```
    envelope.flags |= FLAG_ENCRYPTED;
}
```

Die CryptoBox ist ein Konzept von NaCl, welche die Verschlüsselung und Signierung einer Nachricht kapselt. Dabei wird die elliptische Kurve Curve25519, die sehr gute Streamverschlüsselung xSalsa20 und Poly1305 für Einmal-Authentifizierung verwendet. Mehr dazu unter <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>

Protokolle

BonfireChat unterstützt diverse Protokolle zur Nachrichtenübertragung. Um diese abstrakt zu verwenden, gibt es das Interface `IProtocol`. Diese wird von der abstrakten Basisklasse `SocketProtocol` implementiert und bietet die folgenden minimalen Anforderungen an ein Protokoll:

```
void sendMessage(Envelope envelope);
void setIdentity(Identity identity);
void setOnMessageReceivedListener(OnMessageReceivedListener
listener);
boolean canSend();
```

Mit `setIdentity` wird dem Protokoll die eigene Identität bekannt gemacht, falls diese für den Sendevorgang benötigt wird. `setOnMessageRecievedListener` erhält das Callback-Objekt, um eingehende Nachrichten an den `ConnectionManager` zu melden. Beide Funktionen werden bereits in `SocketProtocol` implementiert.

`canSend` gibt zurück, ob mit dem Protokoll momentan Daten versendet werden können. Das kann zum Beispiel nicht der Fall sein, wenn keine Geräte in der Nähe sind, zu denen eine Verbindung aufgebaut werden kann.

BluetoothProtocol

Im `BluetoothProtocol` wird zunächst die Module ID definiert, welche Bonfire als Bluetooth-Dienst kennzeichnet. Diese wurde zufällig erstellt und sollte vermutlich noch geändert werden.

```
private static final UUID BTMODULEUUID = UUID.fromString("D5AD0434-
34AA-4B5C-B100-4964BFE3E739");
```

Nach Erstellung einer Instanz startet das `BluetoothProtocol` zunächst einen Intent, das Handy unendlich lang sichtbar zu stellen. Dieser muss vom Benutzer akzeptiert werden. Anschließend werden mit einiger Zeitverzögerung die Bluetooth Threads initialisiert.

Der `searchDevicesThread` startet alle zwei Minuten die Suche nach Geräten in der Umgebung neu. Dabei wird eine Liste aller gefundenen Geräte zunächst geleert, und anschließend mit dem `BroadcastReceiver onDeviceFoundReceiver` wieder nach und nach befüllt.

Der `listeningThread` lauscht mit `listenUsingInsecureRfcommWithServiceRecord` nach eingehenden Verbindungen ohne vorheriges Pairing. Dafür wird in einer unendlichen while-Schleife `accept()` aufgerufen und das daraus resultierende Socket an einen neuen Thread übergeben, welcher für die Dauer der Verbindung läuft.

Dieser `IncomingConnectionHandler` erstellt aus dem Socket einen `InputStream`, empfängt mit `receiveEnvelope` den Umschlag und übergibt ihn an das `listener` Callback-Objekt.

Wenn eine Nachricht gesendet werden soll, wird in der Methode `connect` zunächst der evtl. laufende Discovery-Mode gestoppt. Anschließend wird zu jedem Gerät in der Liste der gefundenen benachbarten Geräte ein Socket aufgebaut. Auftretende `IOExceptions` werden dabei ignoriert, da sie sehr oft auftreten. Zum Beispiel wird eine solche geworfen, wenn die App versucht, sich mit einem Gerät zu verbinden, auf dem BonfireChat nicht ausgeführt wird, da dann der passende Port nicht geöffnet ist.

Nachdem die Verbindungen aufgebaut wurden, wird mit `sendEnvelope` der Umschlag in jeden der erzeugten `OutputStreams` geschrieben. Daraufhin wird noch 50 Millisekunden gewartet, damit das Socket nicht direkt nach Ende der Daten geschlossen wird und die Daten sicher ankommen können. Anschließend werden die Verbindungen wieder getrennt und der `DiscoveryMode` wieder gestartet.

WiFiProtocol

TODO: nach erfolgreicher Implementierung dokumentieren

GcmProtocol: Client-Server-Architektur

Das `GcmProtocol` implementiert Kommunikation über ein Client-Server-Modell. Dabei wird Google Cloud Messaging verwendet, um gezielt Verbindungen zum Zielgerät über das Internet aufzubauen. Über ein solches Socket wird dann der Umschlag mit den vorgesehenen Methoden übertragen.

TODO: Dokumentation erweitern.

Routing

Falls Nachrichten zu Empfängern verschickt werden sollen, die nicht direkt benachbart sind und mit einer Verbindung erreicht werden können, muss Routing eingesetzt werden.

Flooding

Momentan werden Nachrichten über ein Protokoll an alle verfügbaren Empfänger geschickt. Das `BluetoothProtocol` zum Beispiel baut also zu allen Geräten in der Nähe eine Verbindung auf und verschickt den Umschlag. Geplant ist weiterhin, nicht nur ein Protokoll zu verwenden,

sondern die Nachricht über alle verfügbaren Protokolle zu schicken.

Wenn eine Nachricht ankommt, die nicht für das eigene Gerät bestimmt ist oder noch weitere Empfänger hat, wird sie erneut versendet. Dabei wird sie mit der gleichen Technik an die Warteschlange der zu sendenden Nachrichten angehängt und somit also wieder an alle verfügbaren Empfänger geschickt. Außerdem wird der hopCount hochgezählt, und sichergestellt, dass eine Nachricht nur 20 mal weiterversendet wird. Damit wird vermieden, dass eine Nachricht ewig im Kreis gesendet wird.

Bei diesem Flooding ist dann zu hoffen, dass die Nachricht früher oder später mindestens einmal beim korrekten Empfänger ankommt.

geplante Mechanismen

Stumpfes Broadcasting der Nachrichten führt zu einem unnötig hohen Datenaufkommen. Insbesondere über Bluetooth dauert das Versenden einer Nachricht recht lange, und es können auch nicht mehrere Nachrichten gleichzeitig versendet werden. Daher ist es wünschenswert, ein klügeres Routingverfahren einzusetzen.

Probleme bereitet dabei, dass kein Knoten Kenntnis über den momentanen Aufbau des Netzes hat und sich das Netz außerdem stetig verändert. Das liegt daran, dass die Personen sich umher bewegen und die Signalqualität der eingesetzten Funktechnologien schwankt.

Kontaktaustausch

Um eine Nachricht zu verschicken, muss dem Sender der Empfänger bekannt sein. Das ist nötig, um den korrekten Public Key in die Nachricht einzutragen und die Nachricht zu verschlüsseln. Zum Austausch von Kontaktdaten stehen mehrere Methoden zur Verfügung:

serverbasierte Suche

Neu angelegte Kontakte werden auf einem Server gespeichert. Dieser bietet auch eine Suche nach Kontakten. Der Benutzer kann mit der `SearchUserActivity` nach Nicknames anderer Kontakte suchen.

In einem `AsyncTask` wird anschließend eine HTTP-Anfrage an den Server gestellt. Der folgende Aufruf wird abgesetzt:

```
BonfireData.API_ENDPOINT + "/search.php?nickname=" +
URLEncoder.encode(keyword, "UTF-8")
```

Der Server antwortet daraufhin mit einem JSON-Array. Dieses enthält für jeden gefundenen Kontakt ein Objekt, welches die folgenden Eigenschaften bietet:

```
{
  "nickname": "",
```

```

"phone": "",
"publickey": "",
"xmppid": "" // wird nicht mehr verwendet
}

```

Wählt der Benutzer daraus einen Kontakt aus, wird ein entsprechendes Contact Objekt erstellt und in der Datenbank gespeichert.

direkter Austausch (NFC oder QR-Code)

Falls kein Internet vorhanden ist, oder die beiden Benutzer sich ohnehin gegenüber stehen, können die Kontaktdaten auch direkt untereinander ausgetauscht werden. Dafür können zum Einen die eigenen Kontaktdaten als QR-Code angezeigt werden. Mittels `startActivityForResult` und einer Integration der BarcodeScanner App kann dieser QR-Code auf einem anderen Handy eingescannt werden.

Zusätzlich dazu können die Kontaktdaten auch über NFC übertragen werden. Dafür wird `Android Beam` verwendet.

In beiden Fällen wird ebenfalls ein Contact Objekt erstellt und in die Datenbank geschrieben.

lokale Datenhaltung: SQLite

BonfireChat speichert einige Dinge lokal auf dem Gerät. Dazu gehören die eigene Identität, Kontakte, Unterhaltungen und Nachrichten. Diese Daten werden in einer SQLite Datenbank verwaltet. Diese verwendet das folgende Schema:

```

CREATE TABLE if not exists contacts (
    nickname TEXT,
    firstName TEXT,
    lastName TEXT,
    phoneNumber TEXT,
    publicKey TEXT,
    xmppId TEXT,
    wifiMacAddress TEXT,
    bluetoothMacAddress TEXT
);
CREATE TABLE if not exists conversations (
    peer INT,
    conversationType INT,
    title TEXT
);
CREATE TABLE if not exists messages (
    uuid TEXT NOT NULL PRIMARY KEY,
    conversation INT NOT NULL,
    ...
);

```

```

    sender INT NOT NULL,
    flags INTEGER NOT NULL,
    protocol TEXT,
    body TEXT,
    sentDate TEXT,
    insertDate INT
);
CREATE TABLE if not exists identities (
    nickname TEXT,
    privatekey TEXT,
    publickey TEXT,
    server TEXT,
    username TEXT,
    password TEXT,
    phone TEXT
);

```

In der Klasse `bonfirechat.data.BonfireData` befinden sich Methoden, um Objekte der Klassen `Identity`, `Contact`, `Conversation` oder `Message` zu speichern und zu laden.

serverseitige Datenhaltung

Einige Daten, insbesondere die registrierten Benutzer, werden auf einem Server gespeichert, auf den die App zugreift. Die Adresse ist momentan
<https://bonfire.projects.teamwiki.net/>.

MySQL-Datenbank

Auf dem Server werden die Kontakte in der MySQL Datenbank gespeichert. Diese hat das folgende Schema:

```

CREATE TABLE `users` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `nickname` varchar(100) DEFAULT NULL,
    `xmppid` varchar(100) DEFAULT NULL,
    `publickey` varchar(200) DEFAULT NULL,
    `phone` varchar(200) DEFAULT NULL,
    `gcmid` varchar(200) DEFAULT NULL,
    `last_updated` datetime DEFAULT NULL,
    `created` datetime DEFAULT NULL,
    `ip` varchar(40) DEFAULT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `nickname` (`nickname`)
) ENGINE=InnoDB AUTO_INCREMENT=69 DEFAULT CHARSET=utf8;
CREATE TABLE `traceroutes` (

```

```
`id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
`uuid` varchar(200) DEFAULT NULL,  
`traceroute` text DEFAULT NULL,  
PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

API

TODO: Genauer dokumentieren, insbesondere erwartete Encodings.

Der Server bietet mehrere API-Endpunkte, die Antworten im JSON Format zurückgeben:

POST /register.php

Mit diesem Endpunkt lassen sich neue Benutzer registrieren. Folgende Daten werden als multipart/formdata erwartet:

```
postdata: {  
  "nonce": "",  
  "publickey": "",  
  "data": {  
    "xmppid": "",  
    "nickname": "",  
    "phone": "",  
    "gcmid": ""  
  }  
}
```

Gibt bei Erfolg ein 200 OK zurück, ansonsten den Fehlercode 400 mit Fehlerbeschreibung.

Die Daten im Feld data müssen dabei mit dem korrekten Schlüssel signiert werden, was serverseitig mit dem übermittelten publickey geprüft wird.

GET /search.php

Sucht nach passenden Kontakten und gibt gefundene als JSON-Array zurück.

Dieser Endpunkt erwartet nickname als GET-Paramter.

Die Ausgabe sieht folgendermaßen aus:

```
[  
 {  
   "nickname": "",  
   "xmppid": "",  
   "publickey": "",  
   "phone": ""  
 }
```

```
    "last_updated": "",  
},  
...  
]
```

POST /traceroute.php

Dieser Endpunkt dient zum Veröffentlichen einer Route, welche mit einer Traceroute-Nachricht bestimmt wurde. Genauere Informationen hierzu stehen im Abschnitt *traceroute*.

Der Endpunkt erwartet folgende Daten als `multipart/formdata`:

```
postdata: {  
  "uuid": "",  
  "traceroute": ""  
}
```

Anschließend wird der entsprechende Tracroute in die Datenbank eingetragen.

GET /traceroute.php

Mit GET kann eine eingetragene Route abgerufen und angezeigt werden. Der Endpunkt erwartet `uuid` als GET-Parameter und zeigt die Route einfach im Response Body an.

User Interface Design

Abschließend folgt nun noch eine Dokumentation der Designentscheidungen hinsichtlich der Benutzererfahrung. Alle UI-Klassen befinden sich im Package `bonfirechat.ui`.

Menü und Activities

Die oberste Ebene in der Hierarchie bildet die `MainActivity`. Diese wird beim Start standardmäßig ausgewählt. Sie bietet links ein Menü, welches sich durch einen Button oder durch Streichen ein- und ausblenden lässt. Mit diesem Menü können die drei Ansichten für Unterhaltungen (`ConversationsFragment`), Kontakte (`ContactsFragment`) und Einstellungen (`SettingsFragment`) angezeigt werden.

`ConversationsFragment` und `ContactsFragment` zeigen jeweils Unterhaltungen bzw. Kontakte in einer Liste an. Dies geschieht über den jeweiligen `ConversationsAdapter` bzw. `ContactsAdapter`. Durch längeres Tippen auf einen Listeneintrag lassen sich ein oder mehrere Einträge auswählen. In der Menüleiste werden dann angepasste Aktionen angeboten. Zu diesem Zweck implementieren die Klassen einen `MultiChoiceModeListener`.

Nachrichten

Durch einfaches Tippen auf eine Unterhaltung oder das Kontextmenü der Kontakte kann man die `MessagesActivity` öffnen, welche die Nachrichten der Unterhaltung anzeigt. Es handelt sich dabei auch um eine Liste, wofür die Klasse den `MessagesAdapter` nutzt.

Nachrichten werden, je nachdem ob sie empfangen oder gesendet wurden, auf der linken oder rechten Bildschirmseite angezeigt. Unter dem Nachrichtentext befindet sich die Sendezeit sowie kleine Icons. Diese zeigen an, ob die Nachricht verschlüsselt übertragen wurde und welches Protokoll genutzt wurde.

Kontakte bearbeiten

Im Kontextmenü der Kontakte lassen sich diese bearbeiten. Dafür wird die `ContactDetailsActivity` gestartet. Damit lassen sich Namen und Telefonnummer ändern. Der Public Key eines Kontakts kann nicht verändert werden, da dann die Nachrichten nicht mehr entschlüsselt werden könnten.

Auch die eigene Identität lässt sich (in den Einstellungen) bearbeiten, hierzu wird die (etwas abgewandelte) `IdentityActivity` genutzt.

eigene Identität teilen

Um die eigene Identität zu teilen, klickt man im `NavigationDrawer` der `MainActivity` ganz unten auf einen entsprechenden Button, oder nutzt den Menüeintrag in der `ContactsActivity`. Es wird daraufhin die `ShareMyIdentityActivity` gestartet, welche den eigenen QR-Code anzeigt und auf NFC-Kontakte lauscht.