

Entwurfsdokument en**Courage**

Cole Bailey - Dominik Doerner - René Brandel - Tobias Röddiger

Inhaltsverzeichnis

1.	Einleitung.....	3
2.	Systemaufbau.....	4
2.1.	Systemarchitektur.....	4
2.2.	Subsysteme	5
2.3.	Subsystem-Schnittstellen.....	10
3.	Paketführer.....	17
3.1.	Paketbeschreibungen.....	17
3.2.	Benutzt-Relation	25
3.3.	Feinentwurf.....	26
4.	Klassenbeschreibungen.....	28
4.1.	Klassen des Views.....	28
4.2.	Klassen des UIControllers.....	35
4.3.	Klassen des Models.....	36
4.4.	Klassen des ServerControllers.....	48
4.5.	Klassen des Servers	58
5.	Daten	64
5.1.	Lokale Speicherung	64
5.2.	Aggregate	64
5.3.	Serverseitige Speicherung.....	65
6.	Dynamik und Ablauf	67
6.1.	Aktionen (lokal)	67
6.2.	Aktionen (serverseitig).....	73
6.3.	UI-Navigation	74
6.4.	UI-Seiten.....	75
7.	Klassendiagramm	79
8.	Anhang.....	80
8.1.	Abbildungsverzeichnis.....	80
8.2.	Klassenverzeichnis.....	82
8.3.	Glossar.....	84

1. Einleitung

Dieses Dokument für den Entwurf der Applikation enCourage – entstanden im Rahmen des Softwarepraktikums PSE – wurde nach dem Top-Down Prinzip, gemäß der Vorlesung Softwaretechnik I (SS 2014) am Karlsruhe Institut für Technologie, erstellt. Es wird zunächst die Systemarchitektur vorgestellt, welche sich in weitere funktionale Subsysteme unterteilt. Jedes Subsystem enthält wiederum Pakete, in denen die einzelnen konkreten Klassen enthalten sind.

Alle Teile des Systems werden dabei von der Architektur (*Abschnitt 2*) über die Pakete (*Abschnitt 3*) bis zu den Klassen (*Abschnitt 4*) genau beschrieben. Im selben Maße wird auch das gesamte Klassendiagramm den Systemteilen entsprechend in immer kleineren Ausschnitten dargestellt. Hyperlinks (am Computer) und Seitenzahlen sollen dabei die Navigation durch das Dokument erleichtern.

Auch die Speicherung und Verwendung von Daten wird dabei in *Abschnitt 5* genauer erklärt. Es wird dabei zwischen lokaler (Serialisierung) und serverseitiger (Datenbank) Datenspeicherung unterschieden. Auch das besondere Prinzip der Aggregate ist hier beschrieben.

Ebenso wichtig ist die Dynamik der Applikation, welche im nachfolgenden *Abschnitt 6* in mehreren Sequenz-, Aktivitäts- und Zustandsdiagrammen dargestellt und beschrieben wird.

Nach der Ansicht des gesamten Klassendiagramms in *Abschnitt 7* bildet der Anhang in *Abschnitt 8* das Ende des Dokumentes einschließlich verschiedener Verzeichnisse, inklusive Klassenverzeichnis und Glossar.

Der hier präsentierte Entwurf soll den Anforderungen, die im Pflichtenheft aufgestellt wurden, entsprechen und ist in Anbetracht auf hohe Wart-, Erlern- und Erweiterbarkeit erstellt worden.

Hinweis für die Klassendiagramme:

Alle grau markierten Klassen sind an anderer Stelle (z.B. in einer externen API) beschrieben und hier nur der Beziehung zu anderen Klassen wegen enthalten. Diese sind auch in den Klassenbeschreibungen nicht zu finden.

2. Systemaufbau

2.1. Systemarchitektur

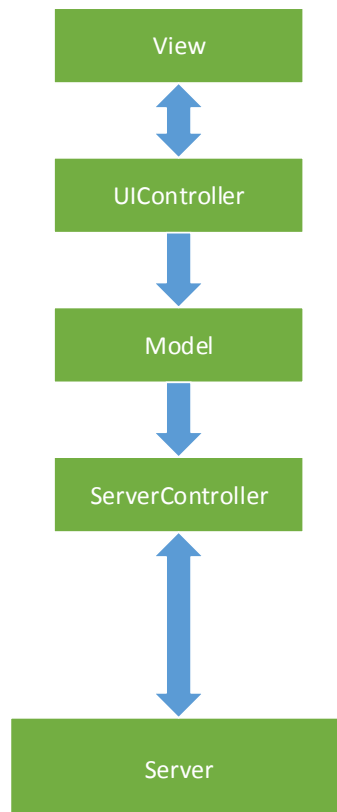


Abbildung 2.1 – Server-Architektur

Die Logik der Applikation verteilt sich auf einen zentralen Server und beliebig viele identische Clients. Diese Verteilung soll im Folgenden beschrieben werden:

Der Server ist das zentrale Bindeglied der Applikation. Er verwaltet die Datenbank und somit alle relevanten Daten, kennt alle aktiven Benutzer und speichert ihre Position in regelmäßigen Zeitabständen. Der Großteil der Serverlogik bedient sich dabei dem Application Programming Interface (API) der Azure Mobile Services von Microsoft, wodurch nur wenige weitere Klassen zwischen Serverschnittstelle und Datenbank benötigt werden, die die Kontrolle der übermittelten Daten, Benachrichtigungen sowie Hintergrundaufgaben übernehmen.

Für den Server steht keine Fassade zur Verfügung, da die benutzte API durch Abstraktion der Datenbanktabellen den Datenaustausch erleichtert und eine eigens geschriebene API überflüssig macht.

Der Client der Applikation ist hingegen fest an ein Gerät gebunden. Alle Einstellungen und Profildaten sind mit der Geräte-ID verknüpft und nicht übertragbar.

Die Architektur trennt sich dabei in fünf linear angeordnete Subsysteme, die so unabhängig wie möglich interagieren und höchstens mit ihrem direkten Nachfolger/Vorgänger kommunizieren.

Der Teil der Anwendungslogik, der von den im Server gespeicherten Daten Gebrauch macht, folgt dabei der asynchronen, Task-getriebenen Programmierung, wobei die Weitergabe von Daten über diese Tasks hauptsächlich zwischen ServerController und Model benutzt wird. Für den UIController (und damit der Aktualisierung des Views) steht hierbei vor allem ein Beobachter-Prinzip zur Verfügung.

Alle Aufrufe an eine tiefere Ebene erfolgen stets über (zumeist asynchrone) Methodenaufrufe, weswegen jede Schicht einen Verweis auf ihren Nachfolger besitzt (dargestellt durch die Pfeilspitzen im oberen

Diagramm). Sowohl das Model als auch der ServerController agieren dabei unabhängig von den Schichten über ihnen.

Eine Ausnahme bildet hierbei der UIController. Als Verbindungsstück zwischen View und Model kennt und kommuniziert dieser mit beiden Partnern. Dies ist vor allem wichtig, da viele Aufrufe vom View an den UIController beinahe direkt zum Model durchgegeben werden.

2.2. Subsysteme

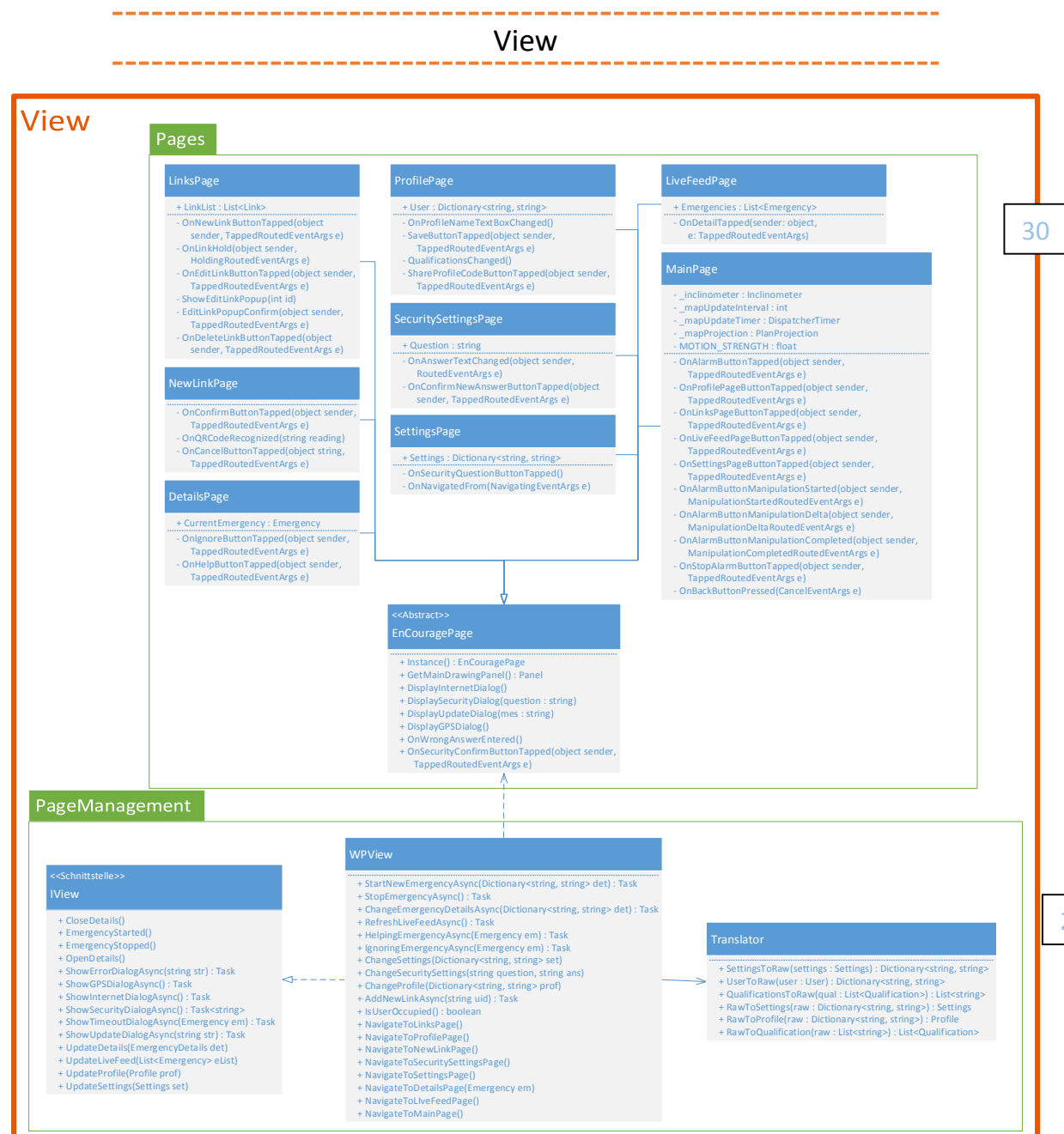


Abbildung 2.2 – Klassen des Views

Der View dient als Eingabe- und Ausgabeschnittstelle für die Applikation.

Das Paket **PageManagement**, zuständig für die Verwaltung der Seiten, nimmt sowohl Instruktionen des UIControllers (Anfrage zu Veränderung der Anzeige) als auch Eingaben von den Pages (Daten für die unteren

Schichten der Applikation) entgegen. Es verbindet somit den UIController mit der direkten Benutzerinteraktion und steuert ebenso die Navigation zu anderen Seiten.

Das Paket **Pages** hingegen beruht auf der Idee, einzelne UI-Zustände über Klassen zu kapseln und somit einen besseren Überblick über die einzelnen dargestellten Seiten zu geben. Alle diese Seiten erben dabei der Kapselung wegen von einer gemeinsamen Oberklasse (EnCouragePage).

Statische Animationen werden hierbei als Methoden gekapselt, dynamische Animationen von Anzeigeelementen hingegen ändern sich durch jeden Methodenaufruf.

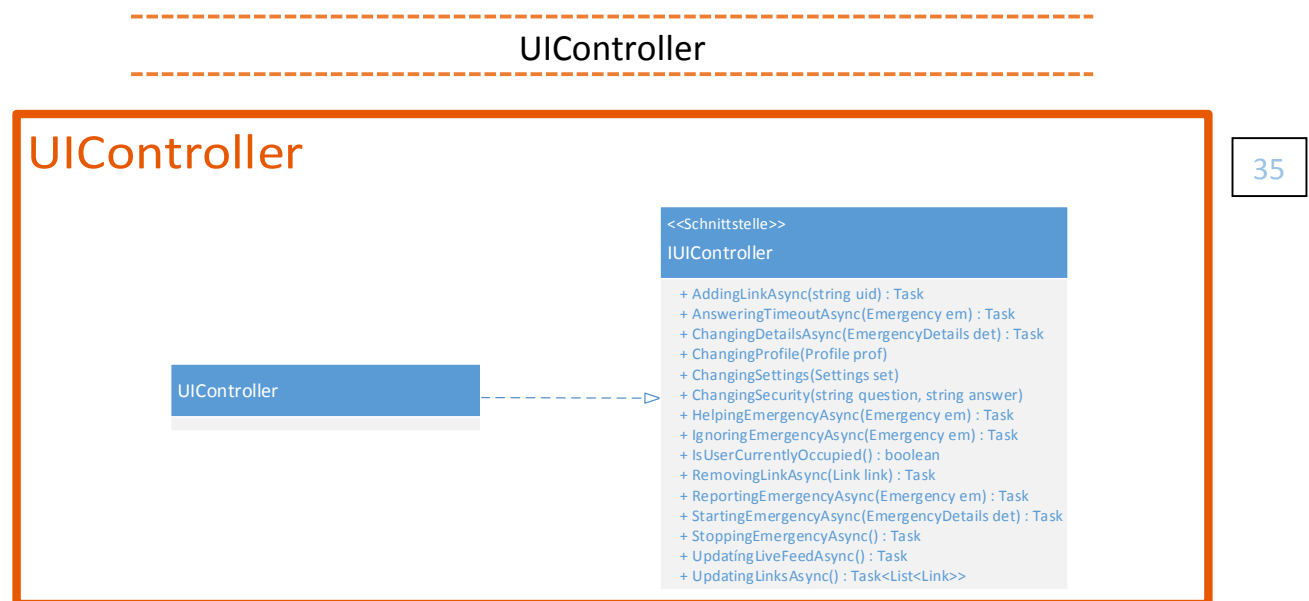


Abbildung 2.3 – Klassen des UIControllers

Der UIController ist das zentrale Bindeglied zwischen View und Model und dient als Adapter für die Kommunikation zwischen den beiden Subsystemen.

Er leistet die Vorverarbeitung und Kontrolle der Daten des Views, stößt neue Anzeigen an und ermöglicht es dem Model, komplett unabhängig zu agieren.

View und UIController kennen sich dabei als einzige Subsysteme des Clients gegenseitig und rufen Methoden auf dem jeweils anderen Subsystem auf. Die Aufrufe des Views am UIController dienen dabei der Anforderung einer Aktualisierung der angezeigten Daten (Anfrage für die Zukunft und damit ohne Rückgabewert) und die Aufrufe des UIControllers am View wiederum stoßen diese Anzeige an. Auch die Aufrufe am Model sind zum Großteil als asynchrone Anforderung neuer Daten zu verstehen, besonders da sich der Großteil dieser angeforderten Daten auf Notfälle bezieht. Aus diesem Grund kann sich der UIController als Beobachter anmelden, um ohne Kenntnis der Herkunft (Push-Benachrichtigung oder Anfrage) über diese informiert zu werden, weswegen diese Anforderungen keinen Rückgabewert besitzen.

Da die Anwendungslogik so minimal wie möglich gehalten werden soll und der Hauptzweck dieses Subsystems in der Verbindung der anderen Subsysteme liegt, kommt der UIController ohne weitere Pakete aus.

Model

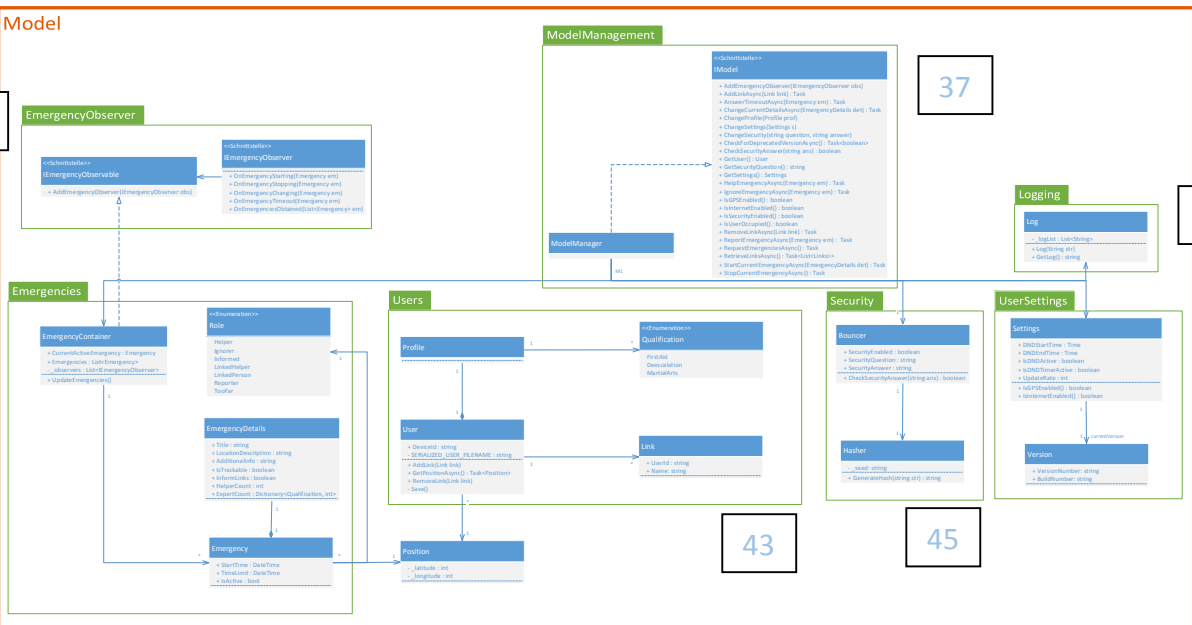


Abbildung 2.4 – Klassen des Models

Das Model mit der Fassaden-Implementierung `ModelManager` enthält alle Klassen und Methoden zum Speichern, Laden und Ändern der lokalen Daten sowie der dahinterliegenden Serverschnittstelle.

Für Operationen, die Daten vom Server benötigen, arbeitet das Model mit dem Subsystem ServerController. All diese an den ServerController weitergegeben Aufrufe sind asynchron, wodurch auch alle Methoden am Model asynchron sind, die diese benötigen.

Alle Anfragen im Bezug auf Notfalldaten werden dabei im Paket Emergencies gespeichert und nach einer Serveranfrage dort aktualisiert. Um über neue Daten in diesem Paket informiert zu werden, müssen sich Beobachter gemäß der Logik aus dem Paket **EmergencyObserver** anmelden.

Die Pakete Users, Security und UserSettings enthalten dabei Objekte für die Sicherung des Benutzers, der Sicherheitsfrage und der Einstellungen. In erster Linie dienen diese Objekte der Speicherung von Daten und weniger der Ausführung von Anwendungslogik und werden allesamt im ModelManagement verwaltet.

Zu guter Letzt dient das Paket Logging der Protokollierung wichtiger Nachrichten und Aktionen, um die Verwendung der Applikation verstehen und Fehler zurückverfolgen zu können.

ServerController

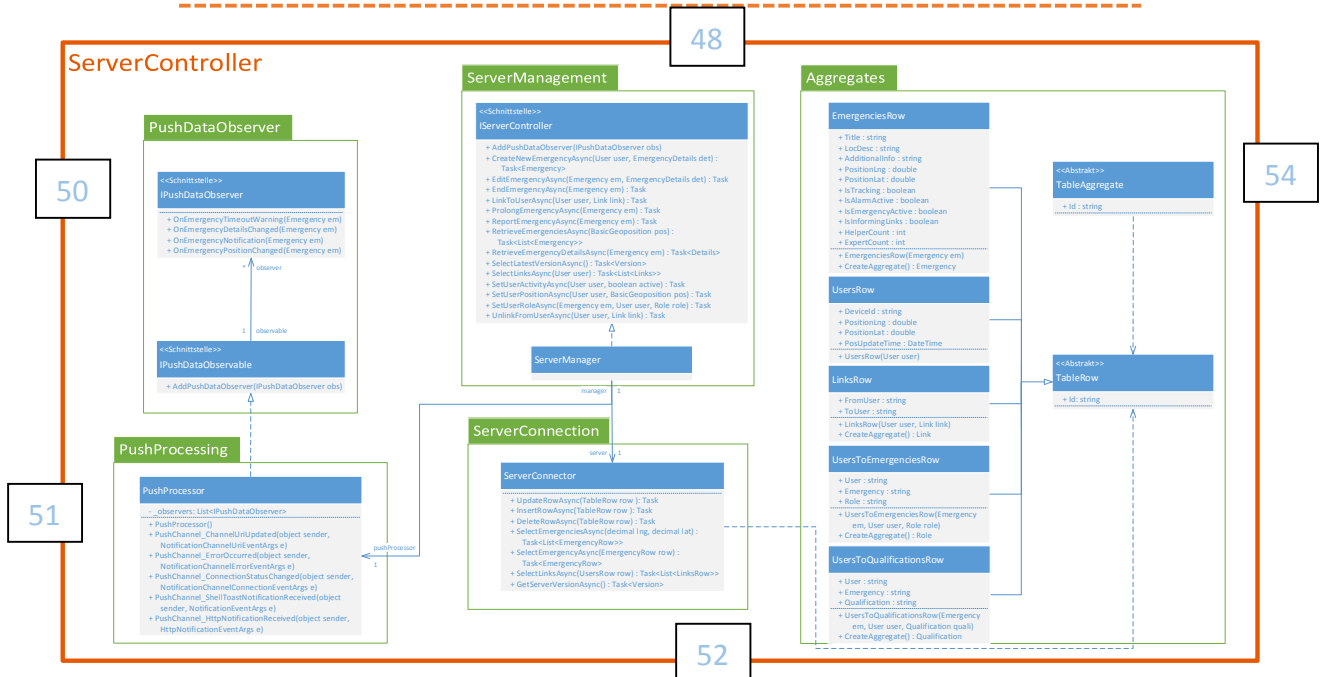


Abbildung 2.5 – Klassen des ServerControllers

Der ServerController mit der Fassaden-Implementierung ServerManager dient dem Client als Schnittstelle zum Zugriff auf die Daten im Hintergrundspeicher, hier dem Server.

Alle Aufrufe auf dem ServerController sind wegen der benötigten Zeit für den Netzwerkkontakt grundsätzlich asynchron.

Der ServerController arbeitet ohne Kenntnis der anderen Subsysteme und speichert selbst keine Daten, verwendet aber die Datenbank über die von der Azure Mobile Services API gelieferte Datenbanktabellenabstraktion wie eine lokale Datenstruktur. Die Wahl von Server, Datenbank und Tabellen ist dabei im Paket **ServerConnection** gekapselt, über die der Server angesprochen wird. Hierzu ist die Verwendung von speziellen Variationen für die zu sichernden Objekte erforderlich, die dem Format der Datenbank ähneln, um über das Netzwerk versendet werden zu können. Diese Servervariationen sowie die Regeln zum Überführen der server- und clientseitigen Variationen sind dabei im Paket **Aggregates** definiert.

Zusätzlich übernimmt der ServerController im Paket **PushProcessing** auch die Entgegennahme und Verarbeitung der Push-Benachrichtigungen, die vom Server versendet werden. Nach der Entschlüsselung der erhaltenen Daten bedient sich dieses Paket der Beobachter-Logik im **PushDataObserver**-Paket zum Benachrichtigen der interessierten Objekte.

Server

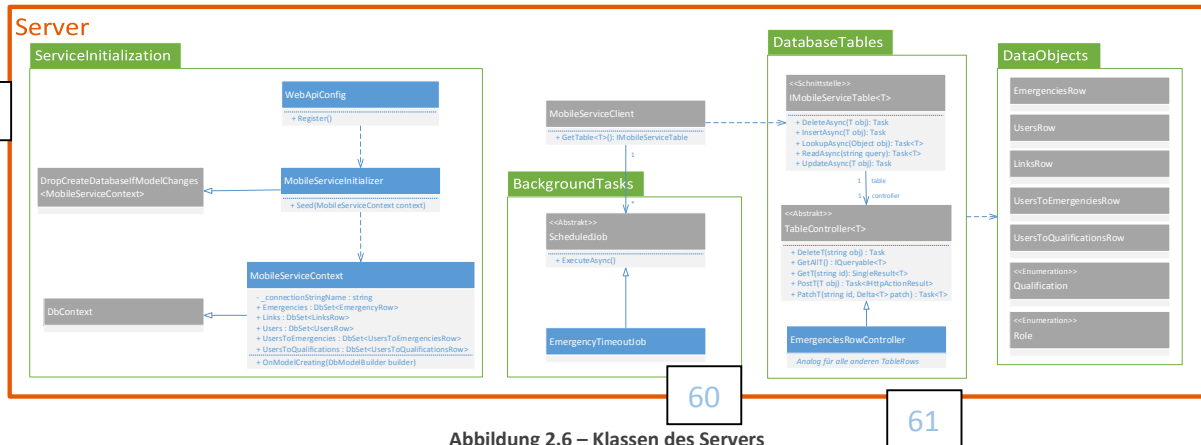


Abbildung 2.6 – Klassen des Servers

Der Server ist der zentrale Schnittpunkt, der alle Clients miteinander verbindet. Er speichert alle für die Notfälle relevanten Daten ab, benachrichtigt Personen und übernimmt die Verwaltung.

Als Microsoft Azure Mobile Service gehostet, liefert die dazugehörige API (Microsoft.WindowsAzure.MobileServices) Klassen und Methoden für die direkte Interaktion zwischen Client und Server. Von zentraler Bedeutung sind dabei die `IMobileServiceTable`-Objekte (Datenbanktabellenabstraktion) eines `IMobileServiceClient` (Serverabstraktion), über die vom Gerät des Benutzers direkt mit der Serverdatenbank interagiert werden kann. Aus Gründen der Sicherheit und zusätzlicher Funktionalität werden dabei serverseitig Tabellenkontrollklassen (Paket `DatabaseTables`) zwischengeschaltet, die die Daten auswerten und verarbeiten können.

Zur einfachen Versendung der Daten stehen die Tabellenzeilenabstraktionsklassen aus dem Paket `Aggregates` des ServerControllers ebenso auf dem Server zur Verfügung (Paket `DataObjects`). Das bringt auch weitere Vorteile mit sich, denn die oben genannte Azure API ermöglicht es ebenso, die Datenbanktabellen (Paket `DatabaseTables`) nach dem Schema dieser Klasse automatisch zu erstellen, wodurch eine Pflege von Hand nicht notwendig ist.

Abgesehen von dem Speichern und Abrufen von Daten aus der Datenbank ermöglicht der Server ebenso das Versenden von Push-Benachrichtigungen über die Windows Push Notification Services (WNS) sowie die Ausführung wiederholt auszuführender Hintergrundaufgaben (Paket `BackgroundTasks`), die etwa das Zeitlimit für Notfälle realisieren können.

2.3. Subsystem-Schnittstellen

Schnittstelle View

```
<<Schnittstelle>>
View

+ CloseDetails()
+ EmergencyStarted()
+ EmergencyStopped()
+ OpenDetails()
+ ShowErrorDialogAsync(string str) : Task
+ ShowGPSDialogAsync() : Task
+ ShowInternetDialogAsync() : Task
+ ShowSecurityDialogAsync() : Task<string>
+ ShowTimeoutDialogAsync(Emergency em) : Task
+ ShowUpdateDialogAsync(string str) : Task
+ UpdateDetails(EmergencyDetails det)
+ UpdateLiveFeed(List<Emergency> eList)
+ UpdateProfile(Profile prof)
+ UpdateSettings(Settings set)
```

Abbildung 2.7 – Subsystem-Schnittstelle des Views

CloseDetails()

Schließt die Detail-Seite in der Benutzeroberfläche und kehrt zur vorherigen Seite zurück.

EmergencyStarted()

Ändert den Zustand der Hauptseite, um anzuzeigen, dass ein Notfall des Benutzers erfolgreich gemeldet wurde und momentan aktiv ist.

EmergencyStopped()

Ändert den Zustand der Hauptseite, um anzuzeigen, dass der Notfall des Benutzers beendet wurde und kehrt zum Hauptbildschirm zurück.

OpenDetails()

Öffnet die Detail-Seite in der Benutzeroberfläche.

ShowErrorDialogAsync(string str) : Task

Öffnet ein Pop-Up über der aktuellen Seite in der Benutzeroberfläche, das die übergebene Fehlermeldung *str* anzeigt.

ShowGPSDialogAsync() : Task

Öffnet ein Pop-Up über der aktuellen Seite der Benutzeroberfläche, das den Benutzer auffordert, die GPS-Funktionalität in den Einstellungen seines Gerätes einzuschalten.

ShowInternetDialogAsync() : Task

Öffnet ein Pop-Up über der aktuellen Seite der Benutzeroberfläche, das den Benutzer auffordert, eine Internetverbindung in den Einstellungen seines Gerätes einzuschalten.

ShowSecurityDialogAsync() : Task<string>

Öffnet ein Pop-Up über der aktuellen Seite der Benutzeroberfläche, das den Benutzer auffordert, die Antwort auf die in den Einstellungen eingestellte Sicherheitsfrage einzugeben, bevor eine privilegierten Aktion ausgeführt werden kann. Die eingegebene Antwort wird im zurückgegebenen Task übermittelt.

ShowTimeoutDialogAsync(Emergency em) : Task

Öffnet ein Pop-Up über der aktuellen Seite der Benutzeroberfläche, das den Benutzer darüber informiert,

dass das Zeitlimit des Notfalls *em* bald abläuft und gibt ihm die Möglichkeit, das besagte Zeitlimit zu verlängern.

ShowUpdateDialogAsync(string str) : Task

Öffnet ein Pop-Up über der aktuellen Seite der Benutzeroberfläche, das den Benutzer darüber informiert, dass die Version der Applikation auf seinem Gerät nicht aktuell ist und zeigt die Update-Informationen in *str* an.

UpdateDetails(EmergencyDetails det)

Übergibt der Detail-Seite der Benutzeroberfläche neue Daten in *det*, um die Anzeige zu aktualisieren.

UpdateLiveFeed(List<Emergency> eList)

Übergibt der LiveFeed-Seite der Benutzeroberfläche neue Daten in *eList*, um die Anzeige zu aktualisieren.

UpdateProfile(Profile prof)

Übergibt der Profile-Seite der Benutzeroberfläche neue Daten in *prof*, um die Anzeige zu aktualisieren.

UpdateSettings(Settings set)

Übergibt der Settings-Seite der Benutzeroberfläche neue Daten in *set*, um die Anzeige zu aktualisieren.

Schnittstelle UIController

<<Schnittstelle>>

UIController

```
+ AddingLinkAsync(string uid) : Task
+ AddingLinkAsync(string uid, string name) : Task
+ AnsweringTimeoutAsync(Emergency em) : Task
+ ChangingDetailsAsync(EmergencyDetails det) : Task
+ ChangingLink(Link link, string name) : Task
+ ChangingProfile(Profile prof)
+ ChangingSettings(Settings set)
+ ChangingSecurity(string question, string answer)
+ HelpingEmergencyAsync(Emergency em) : Task
+ IgnoringEmergencyAsync(Emergency em) : Task
+ IsUserCurrentlyOccupied() : boolean
+ RemovingLinkAsync(Link link) : Task
+ ReportingEmergencyAsync(Emergency em) : Task
+ StartingEmergencyAsync(EmergencyDetails det) : Task
+ StoppingEmergencyAsync() : Task
+ UpdatingLiveFeedAsync() : Task
+ UpdatingLinksAsync() : Task<List<Link>>
+ VerifyForPrivilegedActionAsync() : Task<boolean>
```

Abbildung 2.8 – Subsystem-Schnittstelle des UIControllers

AddingLinkAsync(string uid) : Task

Fügt den Benutzer mit der User-ID *uid* zu den verknüpften Personen hinzu

AddingLinkAsync(string uid, string name) : Task

Fügt den Benutzer mit der User-ID *uid* zu den verknüpften Personen hinzu und benennt die Verknüpfung mit *name*..

AnsweringTimeoutAsync(Emergency em) : Task

Verlängert das Zeitlimit für den Notfall *em*.

ChangingDetailsAsync(EmergencyDetails det) : Task

Aktualisiert die Details des aktuellen Notfalls mit den Daten in *det*.

ChangingLink(Link link, string name)

Bennent den Link *link* um zu *name*.

ChangingProfile(Profile prof)

Aktualisiert das Profil des aktuellen Benutzers mit den Daten in *prof*.

ChangingSettings(Settings set)

Aktualisiert die Einstellungen des aktuellen Benutzers mit den Daten in *set*.

ChangingSecurity(string question, string answer)

Ändert die festgelegte Sicherheitsfrage zu *question* sowie die Antwort darauf zu *answer*. Dies gilt als privilegierte Aktion.

HelpingEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, bei dem Notfall *em* zu helfen. Andere Betroffene werden darüber informiert.

IgnoringEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, den Notfall *em* zu ignorieren. Dieser erscheint ab jetzt nicht mehr im LiveFeed.

IsUserCurrentlyOccupied() : boolean

Gibt „wahr“ zurück, falls der Benutzer momentan einen Notfall gemeldet hat und auf Hilfe wartet oder selbst als Helfer zu einem Notfall unterwegs ist.

RemovingLinkAsync(Link link) : Task

Entfernt die Verknüpfung *link* aus der Liste der verknüpften Personen.

ReportingEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, den Notfall *em* als Missbrauch zu melden. Missbrauchsdaten werden im Server gesammelt.

StartingEmergencyAsync(EmergencyDetails det) : Task

Versucht, für den aktuellen Benutzer einen neuen Notfall mit den Details *det* zu melden. Die Anfrage wird an den Server weitergeleitet und dort verarbeitet. Eine Nachricht im Falle der erfolgreichen Ausführung wird nicht zurückgegeben.

StoppingEmergencyAsync() : Task

Versucht, den Notfall des aktuellen Benutzers zu beenden. Die Anfrage wird an den Server weitergeleitet und dort verarbeitet. Eine Nachricht im Falle der erfolgreichen Ausführung wird nicht zurückgegeben.

UpdatingLiveFeedAsync() : Task

Gibt eine Anfrage zum Aktualisieren des LiveFeeds weiter. Nur EmergencyObserver werden über das Eintreffen dieser Daten informiert.

UpdatingLinksAsync() : Task<List<Link>>

Gibt eine Anfrage zum Aktualisieren der verknüpften Personen weiter. Im Erfolgsfall wird die aktuelle Liste zurückgegeben.

VerifyForPrivilegedActionAsync() : Task<boolean>

Methode für die Überprüfung des Benutzers vor Ausführung einer privilegierten Aktion. Der UIController überprüft dabei, ob diese durch eine Sicherheitsfrage geschützt sind und weist in diesem Falle den Benutzer über ein Pop-Up auf, die Antwort darauf einzugeben. Falls der Benutzer die richtige Antwort eingegeben hat, oder gar keine Sicherheitssperre eingerichtet ist, gibt der Task „wahr“ zurück.

Schnittstellen Model

<<Schnittstelle>>

IModel

```
+ AddEmergencyObserver(IEmergencyObserver obs)
+ AddLinkAsync(Link link) : Task
+ AnswerTimeoutAsync(Emergency em) : Task
+ ChangeCurrentDetailsAsync(EmergencyDetails det) : Task
+ ChangeProfile(Profile prof)
+ ChangeSettings(Settings s)
+ ChangeSecurity(string question, string answer)
+ CheckForDeprecatedVersionAsync() : Task<boolean>
+ CheckSecurityAnswer(string ans) : boolean
+ GetUser() : User
+ GetSecurityQuestion() : string
+ GetSettings() : Settings
+ HelpEmergencyAsync(Emergency em) : Task
+ IgnoreEmergencyAsync(Emergency em) : Task
+ IsGPSEnabled() : boolean
+ IsInternetEnabled() : boolean
+ IsSecurityEnabled() : boolean
+ IsUserOccupied() : boolean
+ RemoveLinkAsync(Link link) : Task
+ ReportEmergencyAsync(Emergency em) : Task
+ RequestEmergenciesAsync() : Task
+ RetrieveLinksAsync() : Task<List<Links>>
+ StartCurrentEmergencyAsync(EmergencyDetails det) : Task
+ StopCurrentEmergencyAsync() : Task
```

Abbildung 2.9 – Subsystem-Schnittstelle des Models

AddEmergencyObserver(IEmergencyObserver obs)

Fügt einen neuen IEmergencyObserver *obs* hinzu, der über jede Änderung im Bezug auf die Notfälle in der näheren Umgebung informiert wird, wie etwa im Falle geänderter Details, einer neuer Position oder dem Ende eines Notfalls.

AddLinkAsync(Link link) : Task

Fügt den Benutzer mit der User-ID *uid* zu den verknüpften Personen hinzu.

AnswerTimeoutAsync(Emergency em) : Task

Verlängert das Zeitlimit für den Notfall *em*.

ChangeCurrentDetailsAsync(EmergencyDetails det) : Task

Aktualisiert die Details des aktuellen Notfalls mit den Daten *det*.

ChangeProfile(Profile prof)

Aktualisiert das Profil des aktuellen Benutzers mit den Daten *prof*.

ChangeSettings(Settings s)

Aktualisiert die Einstellungen des aktuellen Benutzers mit den Daten *s*.

ChangeSecurity(string question, string answer)

Ändert die festgelegte Sicherheitsfrage zu *question* sowie die Antwort darauf zu *answer*.

CheckForDeprecatedVersionAsync() : Task<boolean>

Überprüft, ob die aktuelle Version des Clients veraltet ist und gibt, falls veraltet, „wahr“ zurück.

CheckSecurityAnswer(string ans) : boolean

Überprüft, ob die gegebene Antwort *ans* die korrekte Antwort auf die in den Einstellungen festgelegte Sicherheitsfrage ist und gibt in diesem Fall „wahr“ zurück.

GetProfile() : Profile

Gibt das Profil des aktuellen Benutzers zurück.

GetSecurityQuestion() : string

Gibt die eingestellte Sicherheitsfrage zurück.

GetSettings() : Settings

Gibt die aktuellen Applikation-Einstellungen des Benutzers zurück.

HelpEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, bei dem Notfall *em* zu helfen. Andere Betroffene werden darüber informiert.

IgnoreEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, den Notfall *em* zu ignorieren. Dieser erscheint ab jetzt nicht mehr im LiveFeed.

IsGPSEnabled() : boolean

Gibt „wahr“ zurück, falls die GPS-Funktionalität in den Einstellungen des Gerätes aktiviert ist.

IsInternetEnabled() : boolean

Gibt „wahr“ zurück, falls eine Internetverbindung auf dem Gerät aktiv ist.

IsSecurityEnabled() : boolean

Gibt „wahr“ zurück, falls der Benutzer in den Applikation-Einstellungen angegeben hat, privilegierte Aktionen der Applikation mit einer Sicherheitsfrage zu schützen.

IsUserOccupied() : boolean

Gibt „wahr“ zurück, falls der Benutzer momentan einen Notfall gemeldet hat und auf Hilfe wartet oder selbst als Helfer zu einem Notfall unterwegs ist.

RemoveLinkAsync(Link link) : Task

Entfernt die Verknüpfung *link* aus der Liste der verknüpften Personen.

ReportEmergencyAsync(Emergency em) : Task

Verarbeitet die Wahl des Benutzers, den Notfall *em* als Missbrauch zu melden. Missbrauchsdaten werden im Server gesammelt.

RequestEmergenciesAsync() : Task

Fordert die Menge aller Notfälle an, die sich gerade in der größeren Umgebung des Gerätes befinden. Um über das Eintreffen dieser Daten informiert zu werden, muss ein Objekt zuvor als *IEmergencyObserver* am Model angemeldet sein.

RetrieveLinksAsync() : Task<List<Links>>

Fordert die Menge aller Verknüpfungen des Benutzers an und gibt diese zurück.

StartCurrentEmergencyAsync(EmergencyDetails det) : Task

Versucht, für den aktuellen Benutzer einen neuen Notfall mit den Details *det* zu melden. Die Anfrage wird an den Server weitergeleitet und dort verarbeitet. Es gibt keine Garantie, dass die Anfrage erfolgreich ist.

StopCurrentEmergencyAsync() : Task

Versucht, den Notfall des aktuellen Benutzers zu beenden. Die Anfrage wird an den Server weitergeleitet und dort verarbeitet. Es gibt keine Garantie, dass die Anfrage erfolgreich ist.

Schnittstelle ServerController

<<Schnittstelle>>

IServerController

```
+ AddPushDataObserver(IPushDataObserver obs)
+ CreateNewEmergencyAsync(User user, EmergencyDetails det) : Task<Emergency>
+ EditEmergencyAsync(Emergency em, EmergencyDetails det) : Task
+ EndEmergencyAsync(Emergency em) : Task
+ LinkToUserAsync(User user, Link link) : Task
+ ProlongEmergencyAsync(Emergency em) : Task
+ ReportEmergencyAsync(Emergency em) : Task
+ RetrieveEmergenciesAsync(BasicGeoposition pos) : Task<List<Emergency>>
+ RetrieveEmergencyDetailsAsync(Emergency em) : Task<Details>
+ SelectLatestVersionAsync() : Task<Version>
+ SelectLinksAsync(User user) : Task<List<Links>>
+ SetUserActivityAsync(User user, boolean active) : Task
+ SetUserPositionAsync(User user, BasicGeoposition pos) : Task
+ SetUserRoleAsync(Emergency em, User user, Role role) : Task
+ UnlinkFromUserAsync(User user, Link link) : Task
```

Abbildung 2.10 – Subsystem-Schnittstelle des ServerControllers

AddPushDataObserver(IPushDataObserver obs)

Fügt einen neuen IPushDataObserver *obs* hinzu, der über jede eingehende Push-Benachrichtigung informiert wird.

CreateNewEmergencyAsync(User user, EmergencyDetails det) : Task<Emergency>

Sendet eine Anfrage an den Server, einen neuen Notfall für den Benutzer *user* mit den Details *det* zu erstellen. Der neu erstellte Notfall wird dann zurückgegeben.

EditEmergencyAsync(Emergency em, EmergencyDetails det) : Task

Aktualisiert den Notfall *em* auf dem Server mit den Daten in *det*.

EndEmergencyAsync(Emergency em) : Task

Sendet eine Anfrage an den Server, den Notfall *em* zu beenden.

LinkToUserAsync(User user, Link link) : Task

Fügt die Verknüpfung *link* auf dem Server als neue Verknüpfung des Benutzers *user* hinzu.

ProlongEmergencyAsync(Emergency em) : Task

Sendet eine Anfrage an den Server, das Zeitlimit des Notfalls *em* zu verlängern.

ReportEmergencyAsync(Emergency em) : Task

Teilt dem Server mit, dass der Notfall *em* als Missbrauch gemeldet wurde. Die daraus resultierenden Schritte liegen vollständig in der Verantwortung der serverseitigen Logik.

RetrieveEmergenciesAsync(BasicGeoposition pos) : Task<List<Emergency>>

Fordert die Menge aller Notfälle an, die in der größeren Umgebung der Position *pos* liegen. Der Server legt den genauen Radius fest und gibt eine Liste dieser Daten zurück.

RetrieveEmergencyDetailsAsync(Emergency em) : Task<Details>

Fordert die aktuellen Details des Notfalls *em* vom Server an.

SelectLatestVersionAsync() : Task<Version>

Fordert die neueste Release-Version der Applikation vom Server an.

SelectLinksAsync(User user) : Task<List<Links>>

Fordert die Menge aller Verknüpfungen des Benutzers *user* vom Server an.

SetUserActivityAsync(User user, boolean active) : Task

Registriert den Benutzer *user* beim Server als aktive (inaktiven) Person, wenn *active* mit „wahr“ („falsch“) belegt ist.

SetUserPositionAsync(User user, BasicGeoposition pos) : Task

Aktualisiert den aktuellen Standort des Benutzers *user* zur Position *pos*.

SetUserRoleAsync(Emergency em, User user, Role role) : Task

Teilt dem Server mit, dass der Benutzer *user* beim Notfall *em* die Rolle *role* einnimmt. Die Rolle kann jederzeit mit dieser Methode überschrieben werden, der Standard ist „TooFar“.

UnlinkFromUserAsync(User user, Link link) : Task

Entfernt die Verknüpfung *link* aus der Liste der verknüpften Personen des Benutzers *user*.

3. Paketführer

3.1. Paketbeschreibungen

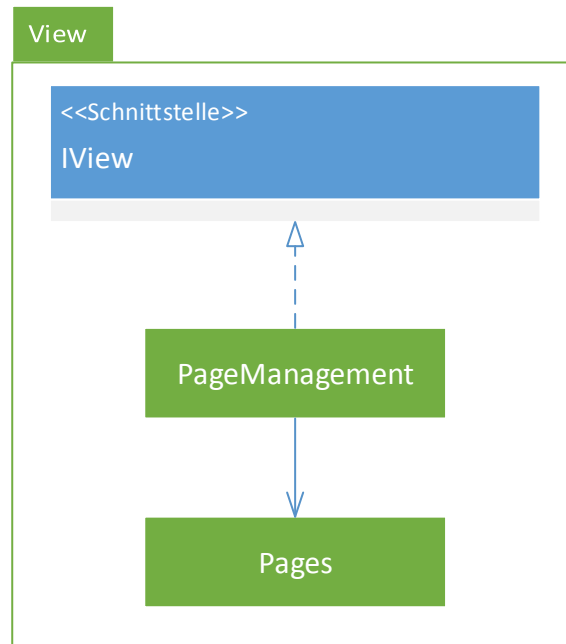


Abbildung 3.1 – Pakete des Subsystems View

PageManagement

Funktion: Das **PageManagement** nimmt Anfragen des UIControllers entgegen und verwaltet die einzelnen Seiten der Applikation sowie die Anzeige der Daten auf diesen. Ereignisse, die auf den Seiten ausgelöst werden, werden wiederum zum UIController heruntergereicht. Als einheitliches Verbindungsstück erleichtert dies die Abänderung einzelner Seiten und die Weitergabe von Informationen.

Ebenso werden in diesem Paket die Formulardaten des Views in Objekte gekapselt sowie eingehende Objekte in anzeigbare Formate übersetzt.

Entwurfsentscheidung: Verwaltung und Kapselung der Seiten

Subsystem: View

Klassendiagramm: siehe S. 28

Enthaltene Klassen: IView, WPView, Translator

Pages

Funktion: Dieses Paket enthält die Seiten, die die verschiedenen Elemente der Anzeige auf dem physischen Gerät kapselt. Animationen und Dialoge werden über Methoden gestartet, während Benutzereingaben asynchrone Ereignisse auslösen, die der Seitenverwaltung im **PageManagement** übergeben werden.

Durch die Vererbung aller Seiten von einer einzelnen Superklasse können gemeinsame Methoden herausgezogen und der Austausch dieser Seiten erleichtert werden.

Entwurfsentscheidung: Auftrennung der Funktionalitäten in separate Seiten

Subsystem: View

Klassendiagramm: siehe S. 30

Enthaltene Klassen: EnCouragePage, MainPage, SettingsPage, ProfilePage, LiveFeedPage, LinksPage, NewLinkPage, DetailsPage, SecuritySettingsPage

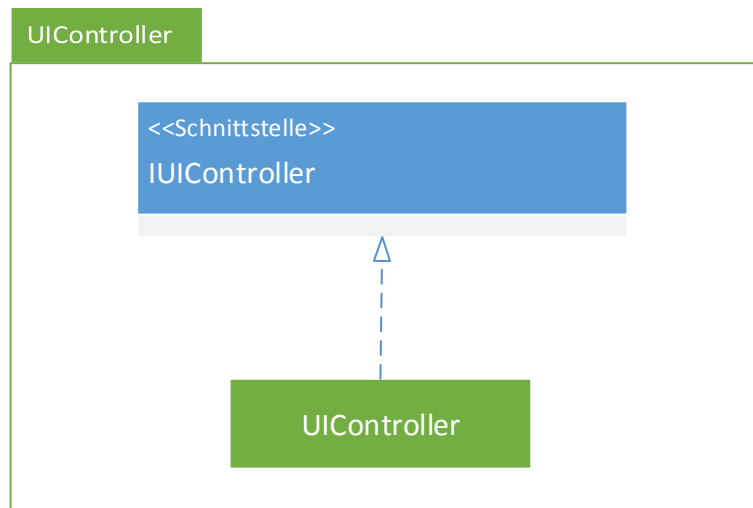


Abbildung 3.2 – Pakete des Subsystems UIController

UIController

Funktion: Der **UIController** dient als Verbindungsstück zwischen View und Model. Der View stellt dabei Anfragen an den **UIController**, die dieser zum Großteil an das Model weiterreicht. Die Art der Rückgabe der angeforderten Informationen richtet sich dabei nach der Art der Daten (siehe 2.2 Subsysteme). Da fast alle Anfragen durchgereicht werden, enthält der **UIController** nur einen kleinen Teil der Anwendungslogik und dient vor allem der Entkopplung der beiden Partner. Nur die Prüfung der Anfragen sowie die Aktualisierung des Views sind Aufgaben, die der **UIController** selbst übernimmt.

Entwurfsentscheidung: Implementierung des Verbindungsstücks zwischen Model und View

Subsystem: UIController

Klassendiagramm: siehe S. 35

Enthaltene Klassen: IUIController, UIController

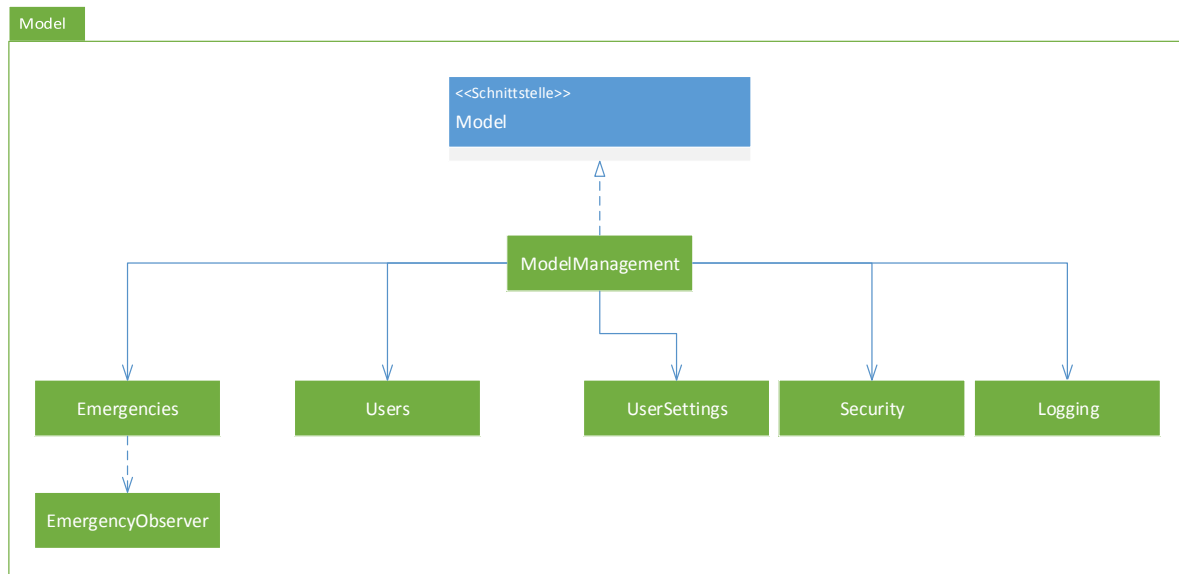


Abbildung 3.3 – Pakete des Subsystems Model

ModelManagement

Funktion: Dieses Paket ist die direkte Implementierung der Subsystem-Schnittstelle IModel und delegiert die Aufgaben sowie Beobachter-Anmeldungen innerhalb des Subsystems, Der ModelManager besitzt Zugriff auf das Subsystem ServerController, über welchen er neue und aktualisierte Daten bezieht, sowie Änderungen weitergibt. Abgesehen davon agiert er unabhängig von den anderen Subsystemen.

Entwurfsentscheidung: Organisation der Klassen, Weitergabe der Informationen

Subsystem: Model

Klassendiagramm: siehe S. 37

Enthaltene Klassen: IModel, ModelManager

EmergencyObserver

Funktion: Dieses Paket verbindet eine Datenstruktur für Notfälle mit einer daran interessierten Klasse, die nach dem Beobachter-Entwurfsmuster über Änderungen informiert wird. Die Benachrichtigung erfolgt dabei über das Push-Prinzip (Versendung der neuen Daten mit der Benachrichtigung) und unterscheidet die verschiedenen Arten von Daten über verschiedene Methodenaufrufe am Beobachter.

Entwurfsentscheidung: Benachrichtigung zwischen Daten und Verarbeitung von Notfällen

Subsystem: Model

Klassendiagramm: siehe S. 42

Enthaltene Klassen: IEmergencyObserver, IEmergencyObservable

Emergencies

Funktion: Dieses Paket kapselt eine Datenstruktur als Puffer für alle relevanten Notfälle, was die nahen Notfälle für den LiveFeed sowie den Notfall mit einschließt, bei dem Benutzer gerade aktiv ist. Die Datenstruktur selbst kann sich beim ServerController für die Versorgung mit per Push-Benachrichtigung erhaltenen Daten anmelden (**PushDataObserver**), wird vom **ModelManagement** mit angeforderten Daten gefüllt und informiert ihrerseits interessierte Klassen aus anderen Systemen (**EmergencyObserver**).

Entwurfsentscheidung: Kapselung der nahen und aktiven Notfälle.

Subsystem: Model

Klassendiagramm: siehe S. 39

Enthaltene Klassen: EmergencyContainer, Emergency, EmergencyDetails, Role

Users

Funktion: Repräsentiert den Benutzer des Gerätes inklusive seines angegebenen Profils, welches die eigenen Qualifikationen und verknüpften Personen enthält. Es kann nur eine Instanz dieser Klasse geben, die hauptsächlich nur durch Interaktionen im View beeinflusst und im Model verändert wird. Zur Sicherung der Anonymität und Einzigartigkeit wird dieses Objekt stets serialisiert lokal gespeichert und beim Starten der Applikation ausgelesen.

Entwurfsentscheidung: Kapselung aller Informationen über den Benutzer

Subsystem: Model

Klassendiagramm: siehe S. 43

Enthaltene Klassen: User, Qualification, Profile, Link

UserSettings

Funktion: Datenstruktur für die Speicherung der aktuellen, vom Benutzer vorgenommenen Einstellungen, die das Verhalten der Applikation steuern, wie z.B. den Ruhemodus.

Entwurfsentscheidung: Kapselung aller Einstellungen

Subsystem: Model

Klassendiagramm: siehe S. 46

Enthaltene Klassen: Settings, Version

Security

Funktion: Verwaltung und Verarbeitung aller sicherheitsrelevanten Daten, mit denen über eine eingestellte Sicherheitsfrage privilegierte Aktionen geschützt sowie sensible Daten verschlüsselt werden können.

Entwurfsentscheidung: Verwendung und Speicherung der Autorisierung

Subsystem: Model

Klassendiagramm: siehe S. 45

Enthaltene Klassen: Bouncer, Hasher

Logging

Funktion: Verwaltet ein Protokoll für Daten, die der Qualitätssicherung der Applikation und dem Verstehen der Benutzung dienen. Sowohl Fehlermeldungen als auch Aufrufe zentraler Methoden können hier gespeichert und später ausgelesen werden.

Das Eintragen und Auslesen der Daten des Logs obliegt dabei allein dem **ModelManagement**.

Entwurfsentscheidung: Protokollierung des Systems

Subsystem: Model

Klassendiagramm: siehe S. 47

Enthaltene Klassen: Log

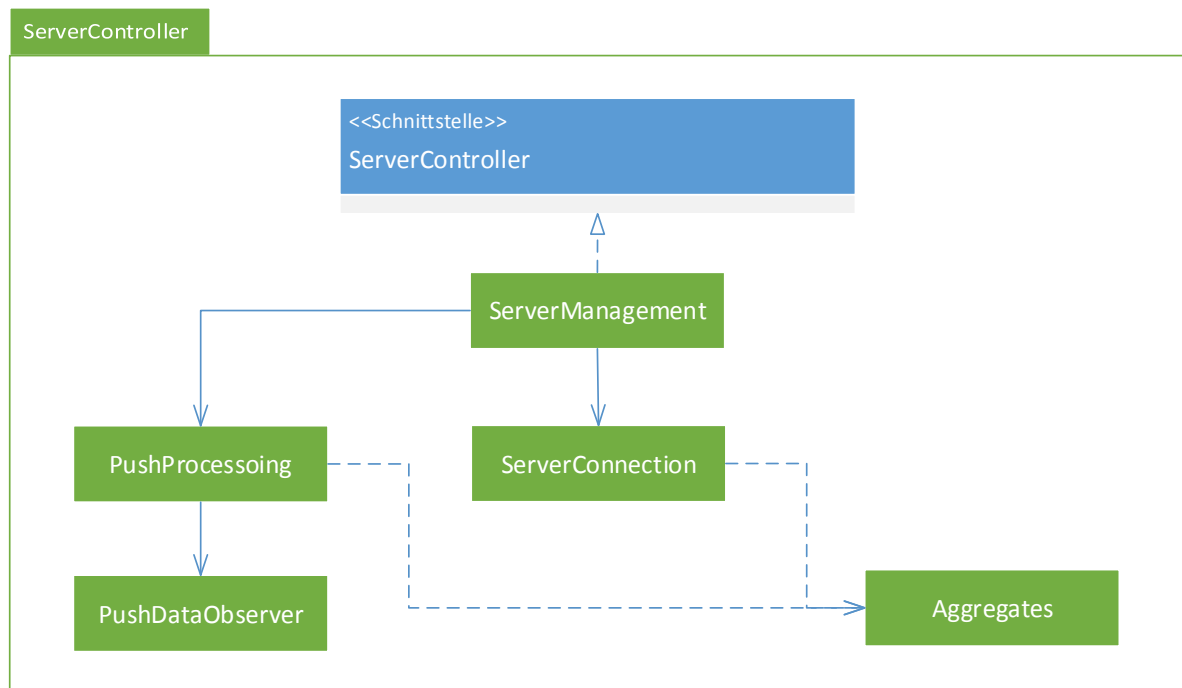


Abbildung 3.4 – Pakete des Subsystems ServerController

ServerManagement

Funktion: Das **ServerManagement** ist die direkte Implementierung der Subsystem-Schnittstelle **IServerController** und delegiert die Aufgaben innerhalb des Subsystems. Datenanfragen des Models werden an die Datenbank (Paket **ServerConnection**) weitergeleitet, während Push-Benachrichtigungen vom Paket **PushProcessing** übernommen werden.

Dieses Paket agiert dabei unabhängig von den anderen Subsystemen und nimmt nur Anfragen von außen entgegen.

Entwurfsentscheidung: Verwendung der Pakete im ServerController

Subsystem: ServerController

Klassendiagramm: siehe S. 48

Enthaltene Klassen: **IServerController**, **ServerManager**

Aggregates

Funktion: Aggregat-Objekte kapseln Daten, die in der Datenbank in eigenen Tabellen hinterlegt sind. Um die Benutzung eines solchen Datenobjekts in der Applikation vom Aufbau der Datenbank zu trennen, existieren sowohl servernahe, referenzlose Versionen (**TableRow**), die übers Netzwerk versendet werden können, als auch besser objektorientierte Versionen für die Verwendung im Client (**TableAggregate**).

Im Server existieren dabei zwei Arten von Datenbanktabellen: Reine Datentabellen (Notfälle, Verknüpfungen, Benutzer), deren Server- und Clientinstanzen ineinander überführt werden können, sowie Verbindungstabellen (Benutzerrollen, Benutzerqualifikationen), deren Clientversionen als Enumerationen definiert und von der Serverinstanz auf den jeweiligen Wert übertragen werden können, bei der Rücküberführung aber viele weitere Daten benötigen.

Entwurfsentscheidung: Unabhängigkeit der Datenspeicherung auf Server und Client

Subsystem: ServerController

Klassendiagramm: *siehe S. 54*

Enthaltene Klassen: TableRow, EmergenciesRow, UsersRow, LinksRow, UsersToEmergenciesRow, UsersToQualificationsRow, TableAggregate

ServerConnection

Funktion: Dieses Paket kapselt die Verbindungsdetails zum Server, der die Datenbank enthält. Über dieses Paket können Datenbankeinträge eingefügt, bearbeitet und angefordert, sowie weitere serverseitige Funktionen ausgeführt werden. Alle Anfragen auf diesem Paket sind rein asynchron. Weiterhin werden in diesem Paket die server- und clientseitigen Versionen der zu übertragenden Objekte (Paket **Aggregates**) ineinander umgewandelt.

Entwurfsentscheidung: Wahl des Servers und Verbindungsaufbau zu diesem

Subsystem: ServerController

Klassendiagramm: *siehe S. 52*

Enthaltene Klassen: ServerConnector

PushProcessing

Funktion: Der PushProcessor empfängt und verarbeitet die erhaltenen Push-Benachrichtigungen und ändert sie in TableAggregate-Objekte, damit andere Pakete mit den Daten arbeiten können. Nach Ankunft und Verarbeitung einer Benachrichtigung werden angemeldete IPushDataObserver (Paket **PushDataObserver**) über die Daten informiert.

Entwurfsentscheidung: Format der Push-Benachrichtigungen

Subsystem: ServerController

Klassendiagramm: *siehe S. 51*

Enthaltene Klassen: PushProcessor

PushDataObserver

Funktion: Dieses Paket dient der Realisierung eines Beobachter-Entwurfsmusters, um es einer Klasse zu ermöglichen, andere interessierte Objekte über eintreffende Push-Benachrichtigungen zu informieren. Die Benachrichtigung folgt dabei dem Push-Prinzip.

Entwurfsentscheidung: Benachrichtigung Anderer über Push-Benachrichtigungen

Subsystem: ServerController

Klassendiagramm: siehe S. 50

Enthaltene Klassen: IPushDataObserver, IPushDataObservable

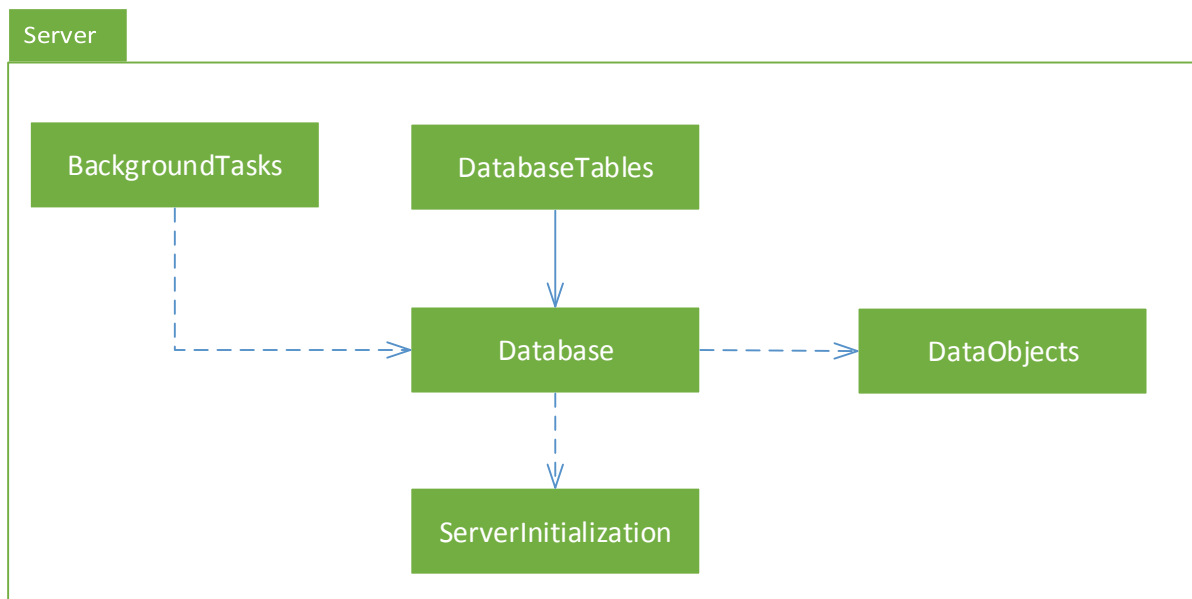


Abbildung 3.5 – Pakete des Subsystems Server

ServerInitialization

Funktion: Die Klassen des Pakets **ServerInitialization** dienen der Konfiguration des Servers sowie dem automatischen Aufbau der Datenbank anhand dem Schema der Objekte aus dem Paket **DataObjects**.

Entwurfsentscheidung: Initialisierung des Server-Dienstes

Subsystem: Server

Klassendiagramm: siehe S. 58

Enthaltene Klassen: WebApiConfig, **MobileServiceInitializer**, MobileServiceContext

DataObjects

Funktion: Dieses Paket enthält eine Kopie der Klassen aus dem Paket **Aggregates** sowie der Enumerationen **Role** und **Qualification**, um alle zu übertragenden Objekte sowohl auf dem Client als auch dem Server definiert zu haben.

Entwurfsentscheidung: keine

Subsystem: Server

Enthaltene Klassen: **EmergenciesRow**, **UsersRow**, **LinksRow**, **UsersToEmergenciesRow**, **UsersToQualificationsRow**, **Qualification**, **Role**

DatabaseTables

Funktion: Dieses Paket enthält die von der Azure API angebotenen Datenbanktabellenabstraktionen sowie die selbst geschriebenen Kontrollklassen, die die Dateneinspeisung und –abrufung des Clients von der Datenbank überprüfen und zusätzliche Funktionalität ausführen können.

Entwurfsentscheidung: Anwendungslogik bei Datenbankverwendung

Subsystem: Server

Klassendiagramm: *siehe S. 61*

Enthaltene Klassen: **EmergenciesRowController**, **UsersRowController**, **LinksRowController**, **UsersToEmergenciesRowController**, **UsersToQualificationsRowController**

BackgroundTasks

Funktion: Dieses Paket enthält alle Aufgaben, die auf dem Server in regelmäßigen Abständen ausgeführt werden müssen, wie etwa die Verwaltung von Zeitbegrenzungen für Notfälle.

Entwurfsentscheidung: Terminplanung und Ausführung zeitgebundener Aufgaben

Subsystem: Server

Klassendiagramm: *siehe S. 60*

Enthaltene Klassen: **EmergencyTimeoutJob**

3.2. Benutzt-Relation

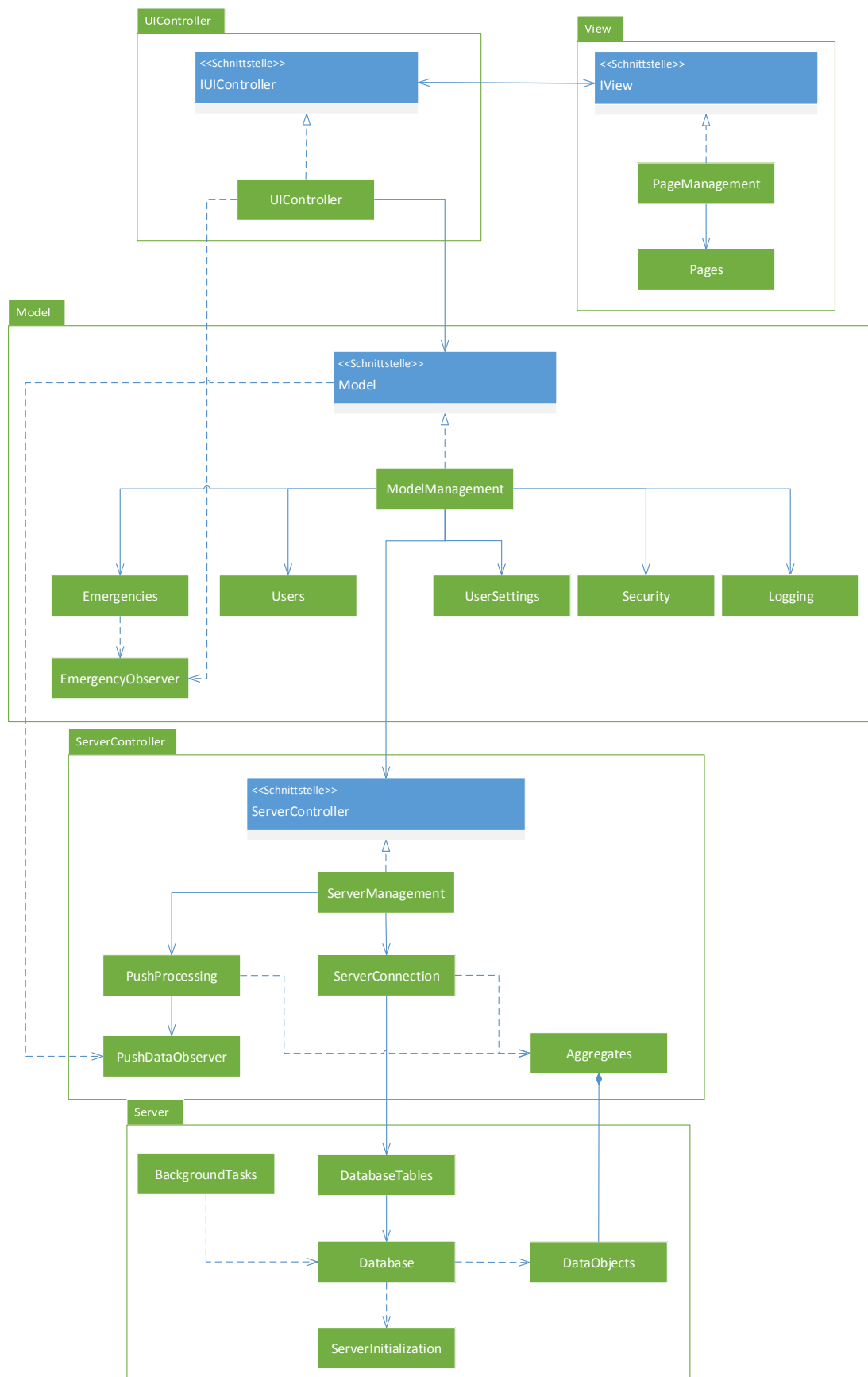
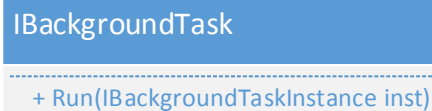


Abbildung 3.6 – Benutzrelation des Systems

Bis auf den gegenseitigen Kontakt zwischen `UIController` und `View` und die Beziehung zwischen `Aggregates` und `DataObjects` sind alle Beziehungen zwischen Paketen einseitig. Dies ermöglicht eine strikte Bottom-To-Top Implementierung des Systems.

Zudem wird die Benutzthierarchie dadurch vereinfacht, dass innerhalb des Clients alle Subsysteme (mit Ausnahme der Beobachter-Klassen) nur über die definierten Fassaden interagieren (siehe 2.3 *Subsystem-Schnittstellen*).

3.3. Feinentwurf



```
classDiagram
    class IBackgroundTask {
        +Run(IBackgroundTaskInstance inst)
    }
```

Abbildung 3.7 – Vorlage eines Hintergrundauftrags

Regelmäßige Aktualisierung der Position aller Benutzer

Um die Position des Benutzers im Hintergrund in regelmäßigen Abständen zu aktualisieren, wird der bestehenden Solution ein „Windows Runtime Component“ hinzugefügt. Dem Vordergrund-Projekt wird dann eine Referenz zu der Komponente hinzugefügt, welche beim Starten der Applikation beim Betriebssystem registriert wird.

Ein `IBackgroundTask` besitzt die Methode `OnInvoke`, welche vom Betriebssystem aufgerufen wird, wenn der Hintergrundauftrag vom Betriebssystem gestartet wird. In dieser wird dann das User-Objekt des aktuellen Benutzers aus dem Hintergrundspeicher deserialisiert, die aktuelle Position des Gerätes bestimmt und die neuen Daten an den Server gesendet.



```
classDiagram
    class GeolocatorObserver {
        -Geolocator : Geolocator
        -MOVEMENT_THRESHOLD : int
        +geolocator_StatusChanged(sender : Geolocator, args : StatusChangedEventArgs)
        +geolocator_PositionChanged(sender : Geolocator, args : PositionChangedEventArgs)
        -Application_RunningInBackground(sender : object, args : RunningInBackgroundEventArgs)
    }
```

Abbildung 3.8 – Vorlage einer Klasse zum Überwachen der Position

Stetige Aktualisierung der Position des Melders

Da der Melder eines Notfalls gegebenenfalls auch stetig verfolgt werden muss, enthält der `ModelManager` eine `Geolocator`-Instanz. Beim Melden des Notfalls wird der `Geolocator` initialisiert und die „StatusChanged“- und „PositionChanged“-Ereignisse bei dem `Geolocator` registriert.

Beim Verändern des Status des `Geolocators` wird das Model darüber informiert und im Fall eines Fehlers aufgefordert den zu beheben. Sobald die neue Position des Geräts den eingestellten Abstand `MOVEMENT_THRESHOLD` übersteigt, wird das Model über die „PositionChanged“-Ereignis von der Veränderung benachrichtigt. Beim Erhalten einer aktualisierten Position fordert das Model den `ServerController` auf, die neuen Daten an den Server weiterzuleiten.

Um die Position des Benutzers auch im Hintergrund stetig verfolgen zu können, wird die ID_CAP_LOCATION-Einstellung in der Manifest-XML-Datei hinzugefügt und die Manifest-Datei der Applikation für die Erlaubnis der Ausführung im Hintergrund modifiziert. Beim Schließen der Applikation werden über die "RunningInBackground"-Ereignis anderen unnötigen Prozessen unterbrochen und nur noch die Position stetig verfolgt.

Sobald der Notfall beendet wird, werden die "StatusChanged"- und "PositionChanged"-Ereignisse bei dem **Geolocator** deregistriert, und der **Geolocator** auf null gesetzt, um die stetige Verfolgung des Benutzers zu beenden.

4. Klassenbeschreibungen

Hinweis: Alle hier angezeigten Klassendiagramme sind Ausschnitte des Systemklassendiagramms. Viele Attribute sind deswegen nicht in der jeweiligen Klasse enthalten, sondern als Relationen zwischen den Klassen angegeben. Diese sind deswegen nicht in allen Diagrammen zu finden, sondern nur in den Attributsauflistungen.

4.1. Klassen des Views



Abbildung 4.1 – Paket PageManagement



Klasse 1 - IView

IView

Beschreibung: Die Schnittstelle **IView** bildet die Fassade des Subsystems View und definiert alle Methoden, die der UIController auf dem Subsystem aufrufen kann. Eine Beschreibung aller Methoden ist in 2.3 *Subsystem-Schnittstellen* zu finden.

Paket: PageManagement

WPView

```
+ StartNewEmergencyAsync(Dictionary<string, string> det) : Task
+ StopEmergencyAsync() : Task
+ ChangeEmergencyDetailsAsync(Dictionary<string, string> det) : Task
+ RefreshLiveFeedAsync() : Task
+ HelpingEmergencyAsync(Emergency em) : Task
+ IgnoringEmergencyAsync(Emergency em) : Task
+ ChangeSettings(Dictionary<string, string> set)
+ ChangeSecuritySettings(string question, string ans)
+ ChangeProfile(Dictionary<string, string> prof)
+ AddNewLinkAsync(string uid) : Task
+ RemoveLinkAsync(Link link) : Task
+ UpdateLink(Link link, string name)
+ IsUserOccupied() : boolean
+ NavigateToLinksPage()
+ NavigateToProfilePage()
+ NavigateToNewLinkPage()
+ NavigateToSecuritySettingsPage()
+ NavigateToSettingsPage()
+ NavigateToDetailsPage(Emergency em)
+ NavigateToLiveFeedPage()
+ NavigateToMainPage()
```

Klasse 2 - WPView

Translator

```
+ SettingsToRaw(Settings settings) : Dictionary<string, string>
+ UserToRaw(User user) : Dictionary<string, string>
+ QualificationsToRaw(List<Qualification> qual) : List<string>
+ RawToSettings(Dictionary<string, string> raw) : Settings
+ RawToProfile(Dictionary<string, string> raw) : Profile
+ RawToQualification(List<string> raw) : List<Qualification>
```

Klasse 3 - Translator

WPView : IView

Beschreibung: Die Klasse **WPView** ist eine Implementierung der Fassadenschnittstelle IView und ist zuständig für die Steuerung der Seiten (*NavigateTo...*), die Anzeige unabhängiger UI-Elemente und Dialoge und der Weiterleitung der Benutzereingaben an den UIController (*Change...*).

Attribute:

- + *Controller* : *UIController* – Der UIController, bei dem Daten angefordert werden können.
- + *Translator* : *Translator* – Der benutzte Translator für die Übersetzung der Formulardaten in Objekte und umgekehrt

Paket: **PageManagement**

Translator

Beschreibung: Der **Translator** konvertiert primitive Rohdaten der angezeigten Formulare zu gekapselten Aggregaten und umgekehrt. Er unterstützt dabei die Übersetzung von Qualification-, User- und Settings-Objekten.

Attribute:

Paket: **PageManagement**

Pages

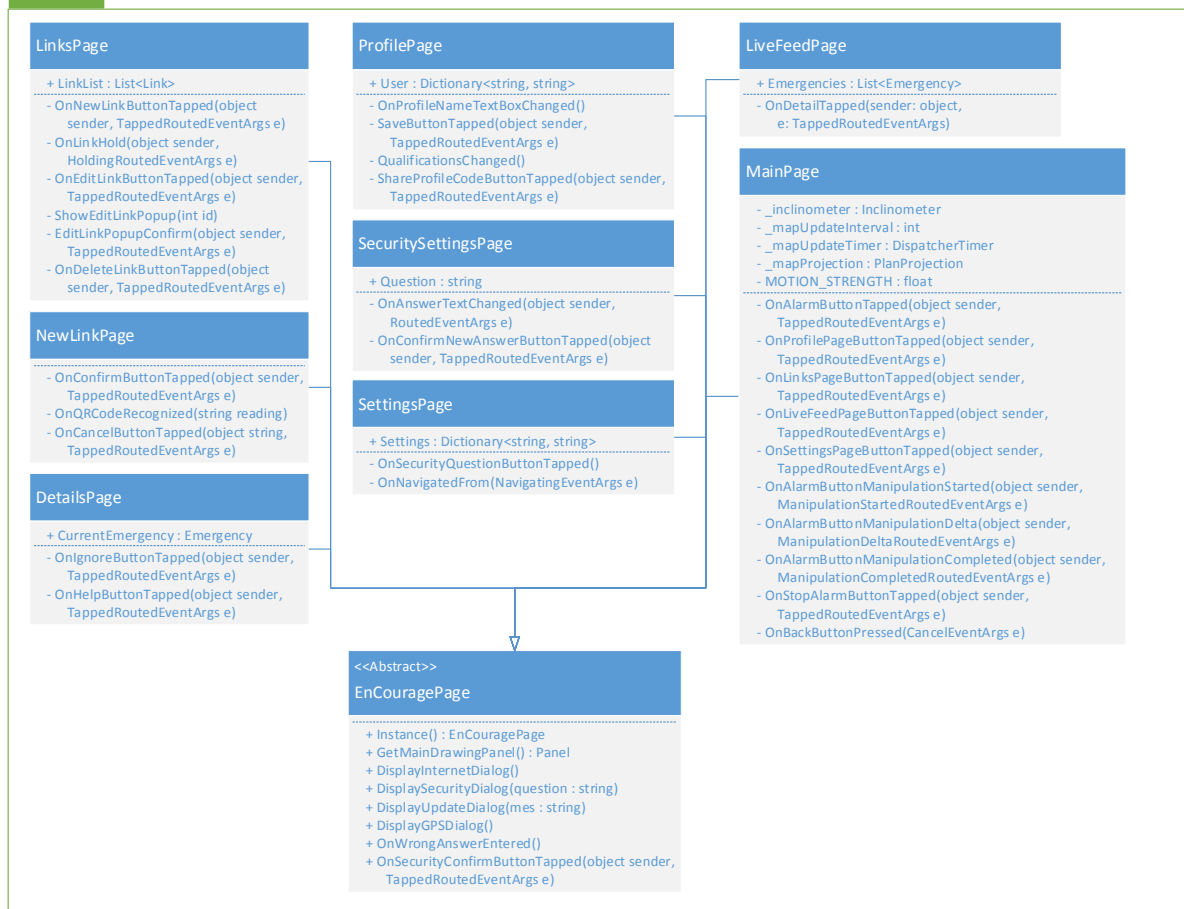


Abbildung 4.2 – Paket Pages

<<Abstract>>

EnCouragePage

```
+ Instance() : EnCouragePage
+ GetMainDrawingPanel() : Panel
+ DisplayInternetDialog()
+ DisplaySecurityDialog(question : string)
+ DisplayUpdateDialog(mes : string)
+ DisplayGPSDialog()
+ OnWrongAnswerEntered()
+ OnSecurityConfirmButtonTapped(object sender,
  TappedRoutedEventArgs e)
```

Klasse 4 - EnCouragePage

EnCouragePage : Page, ICommonAction

Beschreibung: **EnCouragePage** kapselt als Oberklasse aller Seiten gemeinsame Methoden der verschiedenen Anzeigen sowie Dialoge, die unabhängig von der aktuellen Seite verwendet werden können. So werden hier etwa Konstruktoren und einheitliche Darstellungsparameter definiert.

Verfügbare Dialoge sind Benachrichtigungen zum Aktivieren des Internets (*DisplayInternetDialog*) oder der GPS-Funktionalität (*DisplayGPSDialog*), dem Aufruf zum Aktualisieren der Applikation (*DisplayUpdateDialog*) und dem Anzeigen der Sicherheitsfrage (*DisplaySecurityDialog*). Für die Sicherheitsfrage sind hier ebenfalls Methoden für das Bestätigen (*OnSecurityConfirmButtonTapped*) oder eine Reaktion auf eine inkorrekte Antwort enthalten (*OnWrongAnswer*).

Um den Zugriff auf die korrekte Instanz der angezeigten Seite zu sichern, wird das Singleton-Entwurfsmuster verwendet (*Instance*).

Attribute:

Paket: **Pages**

MainPage

```
- _inclinometer : Inclinometer
- _mapUpdateInterval : int
- _mapUpdateTimer : DispatcherTimer
- _mapProjection : PlanProjection
- MOTION_STRENGTH : float
- OnAlarmButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnProfilePageButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnLinksPageButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnLiveFeedPageButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnSettingsPageButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnAlarmButtonManipulationStarted(object sender,
    ManipulationStartedRoutedEventArgs e)
- OnAlarmButtonManipulationDelta(object sender,
    ManipulationDeltaRoutedEventArgs e)
- OnAlarmButtonManipulationCompleted(object sender,
    ManipulationCompletedRoutedEventArgs e)
- OnStopAlarmButtonTapped(object sender,
    TappedRoutedEventArgs e)
- OnBackButtonPressed(CancelEventArgs e)
```

Klasse 5 - MainPage

MainPage : EnCouragePage

Beschreibung: Die Seite **MainPage** bildet die Haupt- und Startseite der Applikation, von der alle anderen Seiten erreichbar sind. Sie enthält den Notfallmeldeknopf und die Möglichkeit, einen Notfall sowohl vor als auch nach dem Melden zu spezifizieren.

Den Hintergrund der Seite bildet eine Kartenansicht (**MapControl**), zentriert auf den Benutzer, die der Ausrichtung des Gerätes über einen **Inclinometer** folgt.

Die Klasse **MainPage** reagiert dabei auf...

... den Druck auf den Meldeknopf (*OnAlarmButtonTapped*)

... die Bewegung des Knopfes für weitere Aktionen

(*OnAlarmButtonManipulationStarted* /
OnAlarmButtonManipulationDelta /
OnAlarmButtonManipulationCompleted)

... die Navigation zu anderen Seiten (*OnProfilePageButtonTapped* /
OnLinksPageButtonTapped /

OnLiveFeedPageButtonTapped /
OnSettingsPageButtonTapped / *OnBackButtonPressed*)

... das Beenden eines Notfalls (*OnStopAlarmButtonTapped*)

Attribute:

- *_inclinometer* : *Inclinometer* – Der Inclinometer, der den Drehwinkel des Gerätes erfasst

- *_mapUpdateInterval* : *int* – Gibt das Intervall in Millisekunden an, in der *_mapUpdateTimer* ausgeführt wird

- *_mapUpdateTimer* : *DispatcherTimer* – Kümmert sich um die Verschiebung der Hintergrundkarte mit den Daten von *_inclinometer*

- *_mapProjection* : *PlaneProjection* – Repräsentiert die Verschiebung der Hintergrundkarte

- *MOTION_STRENGTH* : *float* – Gibt die Stärke der Bewegung der Hintergrundkarte an

Paket: **Pages**

LiveFeedPage

```
+ Emergencies : List<Emergency>
- OnDetailTapped(sender: object,
  e: TappedRoutedEventArgs)
```

Klasse 6 - LiveFeedPage

LiveFeedPage : EnCouragePage

Beschreibung: Die Seite **LiveFeedPage** zeigt alle Notfälle in der größeren Umgebung des Benutzers an. Durch ein Tippen auf einer dieser Notfälle (*OnDetailTapped*) kann der Benutzer zu der Detailansicht (**DetailsPage**) des jeweiligen Notfalls navigieren. Ebenso stellt eine Kartenansicht (**MapControl**) die Positionen der Notfälle grafisch dar.

Attribute:

+ *Emergencies : List<Emergency>* - Eine Liste aller Notfälle in der näheren Umgebung

Paket: **Pages**

DetailsPage

```
+ CurrentEmergency : Emergency
- OnIgnoreButtonTapped(object sender,
  TappedRoutedEventArgs e)
- OnHelpButtonTapped(object sender,
  TappedRoutedEventArgs e)
```

Klasse 7 - DetailsPage

DetailsPage : EnCouragePage

Beschreibung: Auf der Seite **DetailsPage** werden alle Details und Informationen über einen Notfall angezeigt. Ein bislang nicht involvierter Benutzer hat hier ebenfalls die Möglichkeit, anzugeben, bei einem Notfall zur Hilfe zu eilen (*OnHelpButtonTapped*) oder diesen zu ignorieren (*OnIgnoreButtonTapped*).

Attribute:

+ *CurrentEmergency : Emergency* – Der aktuell angezeigte Notfall

Paket: **Pages**

SettingsPage

```
+ Settings : Dictionary<string, string>
- OnSecurityQuestionButtonTapped()
- OnNavigatedFrom(NavigatingEventArgs e)
```

Klasse 8 - SettingsPage

SettingsPage : EnCouragePage

Beschreibung: Auf der Seite **SettingsPage** kann der Benutzer Einstellungen für das Verhalten der Applikation vornehmen. Bis auf die Sicherheitsfrage in **SecuritySettingsPage** (*OnSecurityQuestionButtonTapped*) sind hier alle Einstellungen in einem Formular bearbeitbar. Beim Verlassen der Seite wird dann die Speicherung veranlasst (*OnNavigatedFrom*).

Attribute:

+ *Settings : Dictionary<string, string>* - Enthält alle aktuellen Einstellungen im Format <Einstellung, Wert>

Paket: **Pages**

SecuritySettingsPage

- + Question : string
- OnAnswerTextChanged(object sender, RoutedEventArgs e)
- OnConfirmNewAnswerButtonTapped(object sender, TappedRoutedEventArgs e)

Klasse 9 - SecuritySettingsPage

SecuritySettingsPage : EnCouragePage

Beschreibung: Auf der Seite [SecuritySettingsPage](#) kann der Benutzer die eingestellte Sicherheitsfrage sowie die Antwort darauf spezifizieren oder ändern. Für das Speichern der Änderungen auf dieser Seite muss aber zudem die korrekte Antwort auf die alte Sicherheitsfrage eingegeben werden.

Attribute:

+ Question : string – Die alte festgelegte Sicherheitsfrage

Paket: [Pages](#)

LinksPage

- + LinkList : List<Link>
- OnNewLinkButtonTapped(object sender, TappedRoutedEventArgs e)
- OnLinkHold(object sender, HoldingRoutedEventArgs e)
- OnEditLinkButtonTapped(object sender, TappedRoutedEventArgs e)
- ShowEditLinkPopup(int id)
- EditLinkPopupConfirm(object sender, TappedRoutedEventArgs e)
- OnDeleteLinkButtonTapped(object sender, TappedRoutedEventArgs e)

Klasse 10 - LinksPage

LinksPage : EnCouragePage

Beschreibung: Die Seite [LinksPage](#) zeigt eine Liste aller verknüpften Personen des Benutzers an. Durch ein längeres Gedrückt Halten (*OnLinkHold* -> *ShowEditLinkPopup*) auf eine Verknüpfung kann auf zusätzliche Funktionen wie das Löschen (*OnDeleteLinkButtonTapped*) oder Modifizieren des Namens (*OnEditLinkButtonTapped*) zugegriffen werden. Ebenso kann zur Seite [NewLinkPage](#) zum Hinzufügen einer neuen Verknüpfung navigiert werden (*OnNewLinkButtonTapped*).

Attribute:

+ LinkList : List<Link> – Eine Liste aller Verknüpfungen des Benutzers

Paket: [Pages](#)

NewLinkPage

- OnConfirmButtonTapped(object sender, TappedRoutedEventArgs e)
- OnQRCodeRecognized(string reading)
- OnCancelButtonTapped(object sender, TappedRoutedEventArgs e)

Klasse 11 - NewLinkPage

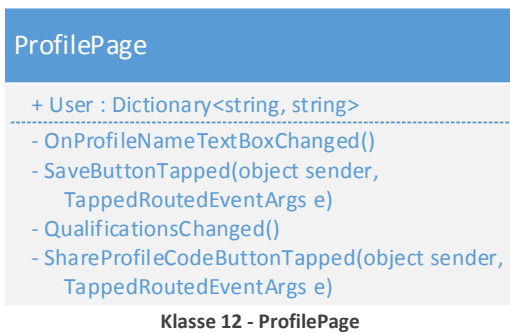
NewLinkPage : EnCouragePage

Beschreibung: Die Seite [NewLinkPage](#) enthält die Möglichkeit, eine neue Verknüpfung über das Scannen des Profil-QR-Codes (*OnQRCodeRecognized*) eines anderen Benutzers hinzuzufügen. Ein Name für die neue Verknüpfung kann in einem Textfeld angegeben werden.

Vor dem Speichern der neuen Verknüpfung wird dem Benutzer zudem eine Möglichkeit zum Bestätigen (*OnConfirmButtonTapped*) oder Abbrechen (*OnCancelButtonTapped*) gegeben.

Attribute:

Paket: [Pages](#)



ProfilePage : EnCouragePage

Beschreibung: Auf der Seite **ProfilePage** kann der Benutzer seine persönliche Daten einsehen und verändern (*OnProfileNameTextBoxChanged* / *QualificationsChanged*). Vor der Speicherung hat der Benutzer die Möglichkeit, die Änderungen zu bestätigen (*SaveButtonTapped*) oder abubrechen.

Unter anderem ist hier auch der QRCode zu finden, mit dem andere Benutzer dieses Gerät als Verknüpfung hinzufügen können. Dieser kann auch über soziale Medien geteilt werden (*ShareProfileCodeButtonTapped*).

Attribute:

+ *User* : *Dictionary<string, string>* – Enthält alle Daten des Benutzers im Format <Datum, Wert>

Paket: **Pages**

4.2. Klassen des UIControllers

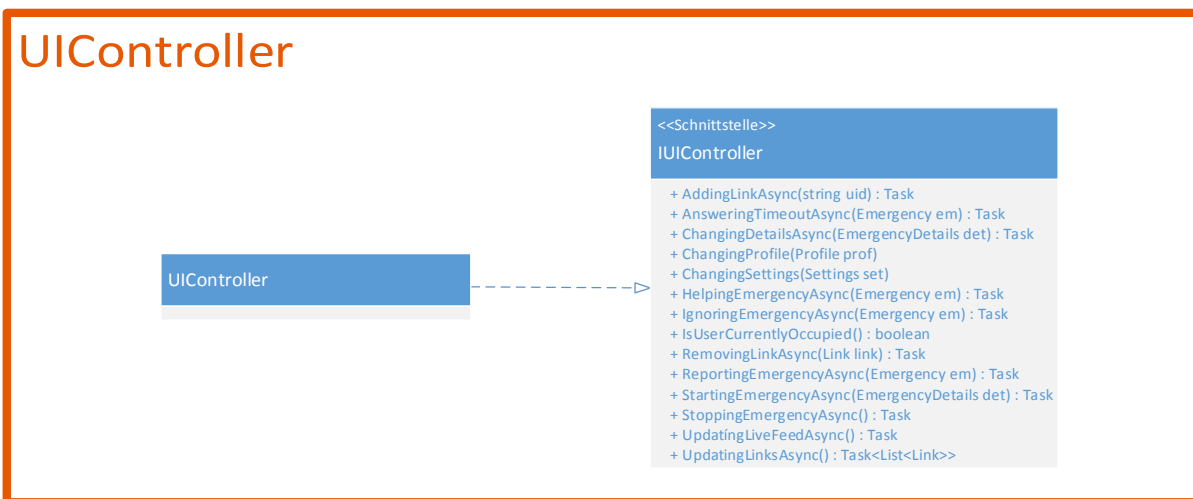


Abbildung 4.3 – Paket UIController

<<Schnittstelle>>

IUIController

- + AddingLinkAsync(string uid) : Task
- + AnsweringTimeoutAsync(Emergency em) : Task
- + ChangingDetailsAsync(EmergencyDetails det) : Task
- + ChangingProfile(Profile prof)
- + ChangingSettings(Settings set)
- + ChangingSecurity(string question, string answer)
- + HelpingEmergencyAsync(Emergency em) : Task
- + IgnoringEmergencyAsync(Emergency em) : Task
- + IsUserCurrentlyOccupied() : boolean
- + RemovingLinkAsync(Link link) : Task
- + ReportingEmergencyAsync(Emergency em) : Task
- + StartingEmergencyAsync(EmergencyDetails det) : Task
- + StoppingEmergencyAsync() : Task
- + UpdatingLiveFeedAsync() : Task
- + UpdatingLinksAsync() : Task<List<Link>>
- + VerifyForPrivilegedActionAsync() : Task<boolean>

Klasse 13 - IUIController

UIController

Klasse 14 - UIController

IUIController

Beschreibung: Diese Schnittstelle dient als Fassade für das gesamte Subsystem des UIControllers und definiert alle Aufrufe, die andere Subsysteme darauf ausführen können. Eine Beschreibung aller Methoden ist in 2.3 *Subsystem-Schnittstellen* zu finden.

Paket: **UIController**

UIController : IUIController

Beschreibung: Der UIController implementiert die Subsystem-Fassade IUIController und ist die einzige implementierte Klasse des Subsystems. Er nimmt Aufrufe des Views entgegen und gibt die meisten Anfragen nach Prüfung der korrekten Umstände (wie etwa aktive Verbindungen oder erlaubte Aktionen) an *model* weiter. Ebenso ist er dafür zuständig, die Anzeigen in *view* zu aktualisieren und Dialoge aufzurufen.

Attribute:

+ *model* : *IModel* – Das Model der Applikation

+ *view* : *IView* – Der View der Applikation

Paket: **UIController**

4.3. Klassen des Models

ModelManagement

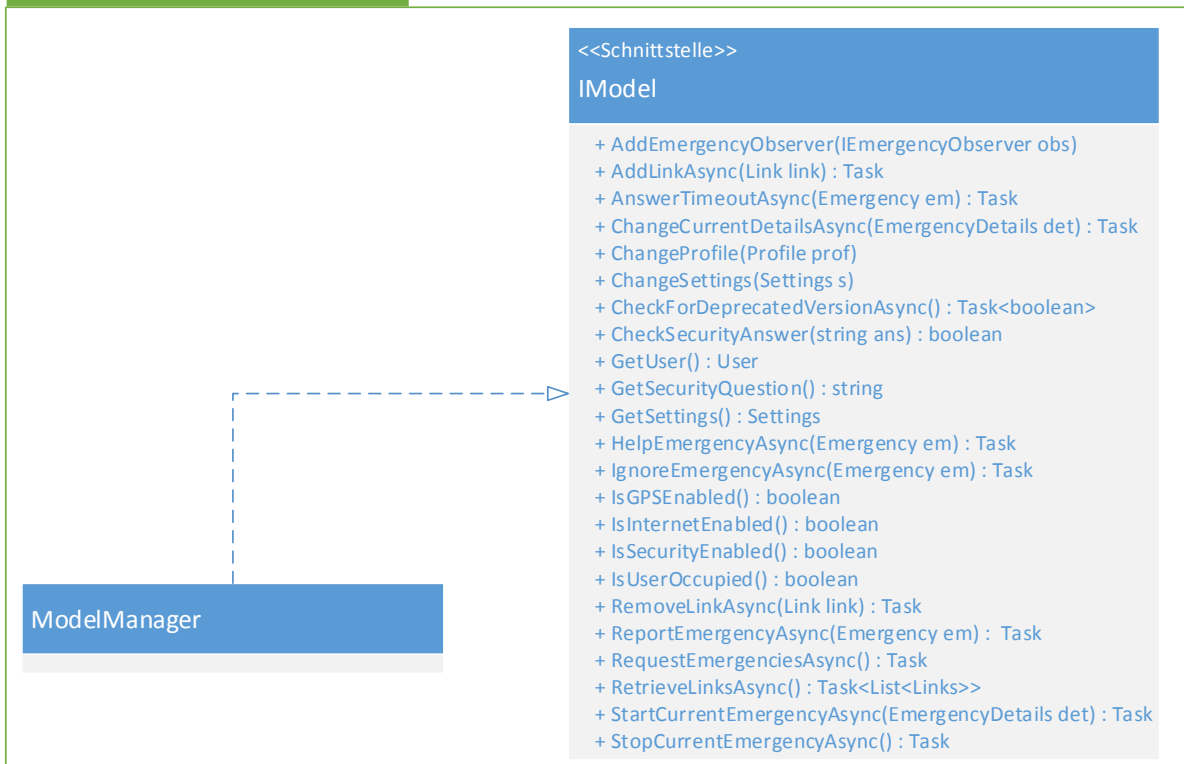
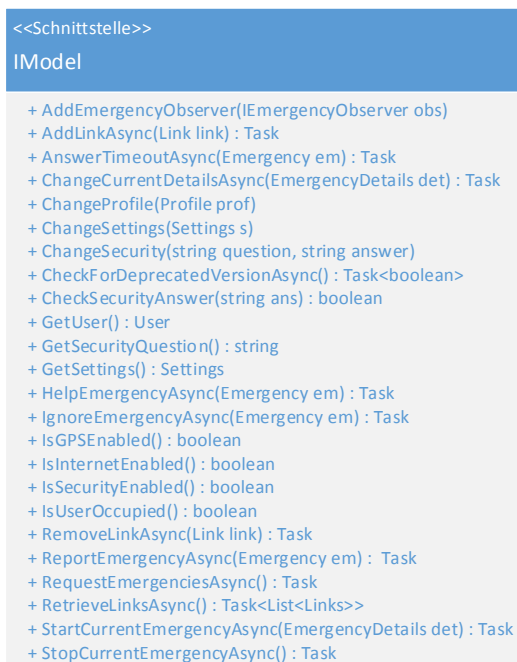


Abbildung 4.4 – Paket ModelManagement



Klasse 15 - IModel

IModel

Beschreibung: Diese Schnittstelle dient als Fassade für das gesamte Subsystem des Models und definiert alle Aufrufe, die der UIController darauf ausführen kann. Eine Beschreibung aller Methoden ist in 2.3 *Subsystem-Schnittstellen* zu finden.

Paket: ModelManagement

ModelManager : IModel, IPushDataObserver

Beschreibung: Der **ModelManager** implementiert die Fassade-Schnittstelle IModel und alle ihre Methoden. Er delegiert dabei die Aufrufe auf dem Subsystem an die anderen Klassen und Pakete und verknüpft die Anwendungslogik. Ebenso ist er als Beobachter (IPushDataObserver) beim ServerController angemeldet, um über ankommende Push-Benachrichtigungen informiert zu werden.

Attribute:

- + *user* : *User* – Der aktuelle Benutzer der Applikation
- + *serverController* : *IServerController* – Die benutzte Serverschnittstelle
- + *emergencies* : *EmergencyContainer* – Datenstruktur für alle für den Benutzer relevanten Notfälle
- + *security* : *Bouncer* – Enthält die Sicherheitsfrage
- + *settings* : *Settings* – Die aktuellen Einstellungen des Benutzers
- + *log* : *Log* – Der verwendete Log

Paket: **ModelManagement**

Emergencies

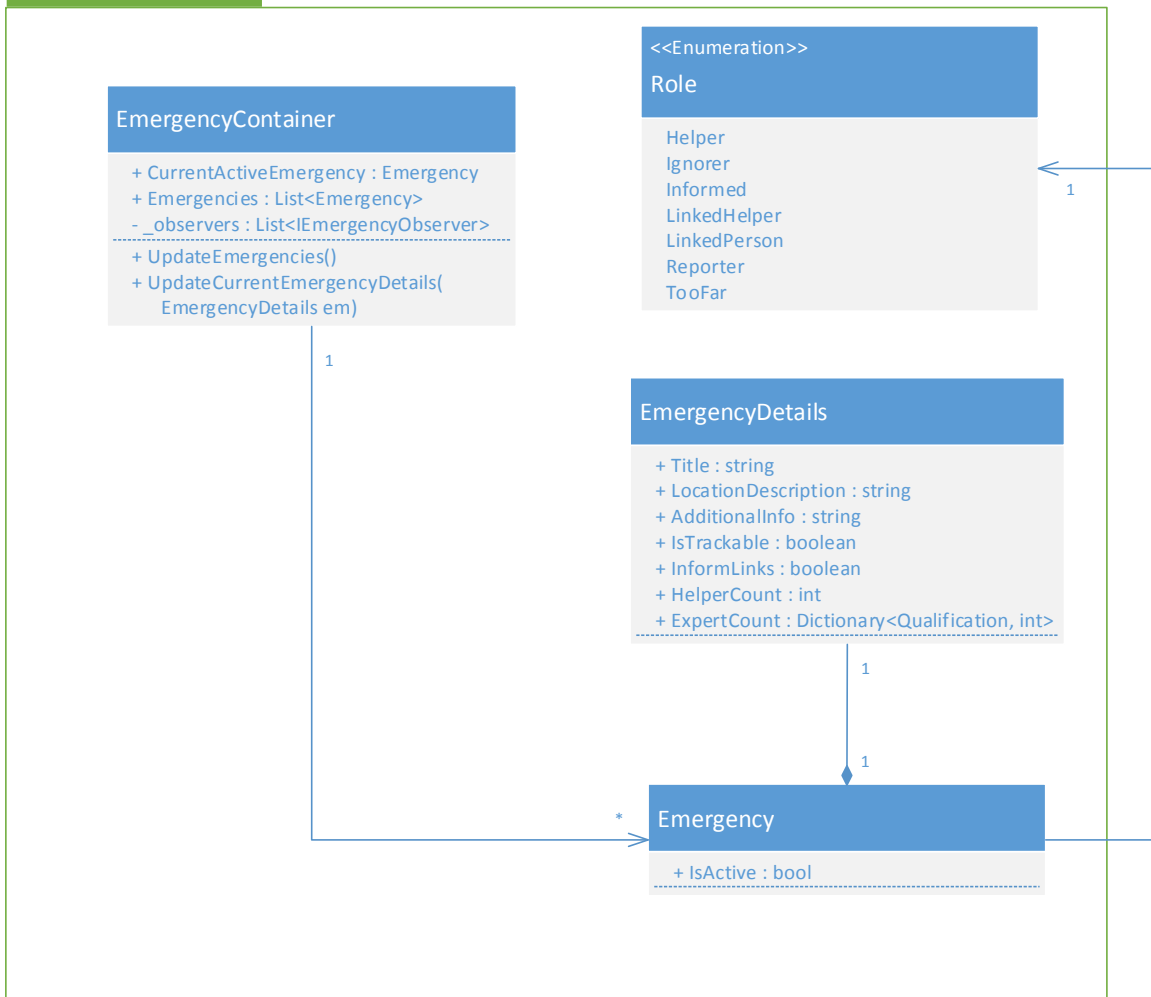


Abbildung 4.5 – Paket Emergencies

EmergencyContainer

```
+ CurrentActiveEmergency : Emergency
+ Emergencies : List<Emergency>
- _observers : List<IEmergencyObserver>
+ UpdateEmergencies()
+ UpdateCurrentEmergencyDetails(
    EmergencyDetails em)
```

Klasse 17 - EmergencyContainer

EmergencyContainer : IEmergencyObservable

Beschreibung: Die Klasse **EmergencyContainer** kapselt alle für den Benutzer relevanten Notfälle (Emergency-Objekte) in *Emergencies* sowie den aktuell aktiven Notfall in *CurrentActiveEmergency*. Der **EmergencyContainer** implementiert dabei die Schnittstelle **IEmergencyObservable**, um angemeldete Beobachter über Änderungen der Daten informieren zu können. Durch diese Implementierungen dient er dem Model als zentrales Modul zur Weitergabe von Daten an den UIController und damit den View.

Attribute:

- + *CurrentActiveEmergency* : *Emergency* – Der aktive Notfall, in dem sich der Benutzer befindet
- + *Emergencies* : *List<Emergency>* - Eine Liste aller relevanten Notfälle
- *_observers* : *List<IEmergencyObserver>* - Eine Liste aller angemeldeten Beobachter

Paket: **Emergencies**

Emergency

```
+ IsActive : bool
```

Klasse 18 - Emergency

Emergency : TableAggregate

Beschreibung: Die Klasse **Emergency** kapselt die Informationen über einen Notfall. Ein **Emergency** besitzt *EmergencyDetails*, welche die genauen Informationen über diesen **Emergency** kapseln. Außerdem besitzt ein **Emergency** eine **Fehler! erweisquelle konnte nicht gefunden werden.** die den geographischen Standort des Notfalls repräsentiert. Die *Role* eines **Emergency** steht für die Rolle des lokalen Benutzers in diesem **Emergency**.

Attribute:

- + *IsActive* : *boolean* – gibt an, ob dieser **Emergency** aktiv ist
- + *Position* : *BasicGeoposition* – die Position des Notfalls
- + *Details* : *EmergencyDetails* – die Notfalldetails
- + *Role* : *Role* – die Rolle des aktuellen Benutzers bei dem Notfall

Paket: **Emergencies**

EmergencyDetails

- + Title : string
- + LocationDescription : string
- + AdditionalInfo : string
- + IsTrackable : boolean
- + InformLinks : boolean
- + HelperCount : int
- + ExpertCount : Dictionary<Qualification, int>

Klasse 19 - EmergencyDetails

EmergencyDetails

Beschreibung: Die Klasse **EmergencyDetail** kapselt die Details eines Emergency. Ein **EmergencyDetail** Objekt hat ohne einen Emergency keine Daseinsberechtigung.

Attribute:

- + Title : *String* – der Titel des zugehörigen Emergency
- + FurtherLocation : *String* – eine genauere Ortbeschreibung des zugehörigen Emergency
- + OtherDetails : *String* – sonstige Details über den zugehörigen Emergency
- + IsTrackable : *bool* – gibt an, ob die Position des zugehörige Emergency stetig verfolgbar ist
- + InformLinks : *bool* – gibt an, ob die verknüpften Personen über den Emergency informiert werden
- + HelperCount : *int* – repräsentiert die Anzahl der Helfer bei diesem Emergency
- + ExpertCount : *Dictionary<Qualification, int>* - die Anzahl der Experten einer Qualifikation eines Emergency

Paket: **Emergencies**

<<Enumeration>>

Role

- Helper
- Ignorer
- Informed
- LinkedHelper
- LinkedPerson
- Reporter
- TooFar

Klasse 20 - Role

Role

Beschreibung: Enumeration, welche die verschiedenen Rollen die ein Benutzer einnehmen kann enthält.

Werte:

Reporter – Melder eines Notfalls
LinkedPerson – Verknüpfte Person des Melders des Notfalls
LinkedHelper – Verknüpfte Person, die helfen will
Informed – Informierte Person in der nahen Umgebung des Notfalls
Helper – Benutzer hilft bei einem Notfall
Ignorer – Benutzer ignoriert den Notfall
TooFar – Person in der größeren Umgebung des Notfalls (Standard)

Paket: **Emergencies**

EmergencyObserver

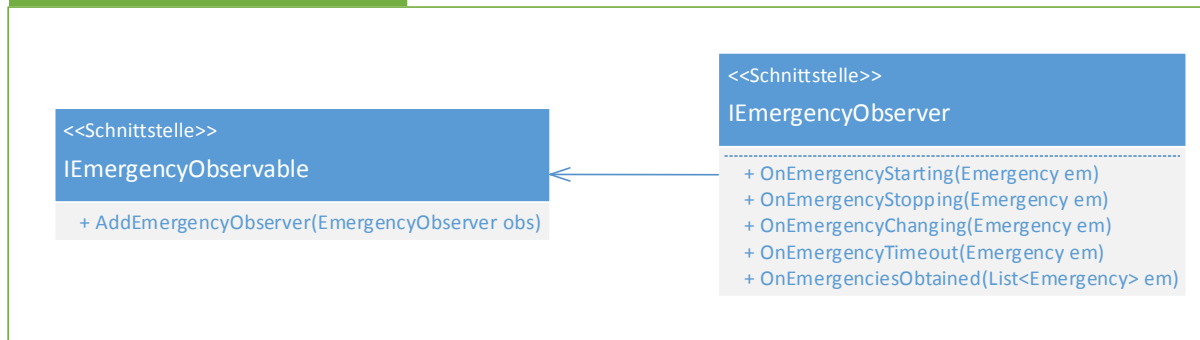


Abbildung 4.6 – Paket EmergencyObserver

<<Schnittstelle>>

IEmergencyObservable

+ AddEmergencyObserver(EmergencyObserver obs)

Klasse 21 - IEmergencyObservable

IEmergencyObservable

Beschreibung: Schnittstelle, um angemeldete Beobachter (*AddEmergencyObserver*) über Änderungen an Datenstrukturen zu informieren.

Paket: EmergencyObserver

<<Schnittstelle>>

IEmergencyObserver

+ OnEmergencyStarting(Emergency em)
+ OnEmergencyStopping(Emergency em)
+ OnEmergencyChanging(Emergency em)
+ OnEmergencyTimeout(Emergency em)
+ OnEmergenciesObtained(List<Emergency> em)

Klasse 22 - IEmergencyObserver

IEmergencyObserver

Beschreibung: Schnittstelle, um IEmergencyObservable-Objekte beobachten zu können. Je nach Art der Datenänderung können auf dem Beobachter verschiedene Methoden aufgerufen werden, denen jeweils die geänderten Daten mitgegeben werden. Es stehen dabei Methoden im Falle des Startens (*OnEmergencyStarting*) und Beendens (*OnEmergencyStopping*) eines Notfalls sowie des Änderns der Notfalldetails (*OnEmergencyChanging*), eines nahenden Zeitlimits (*OnEmergencyTimeout*) oder einer aktualisierten Notfallliste (*OnEmergenciesObtained*) zur Verfügung.

Paket: EmergencyObserver

Users

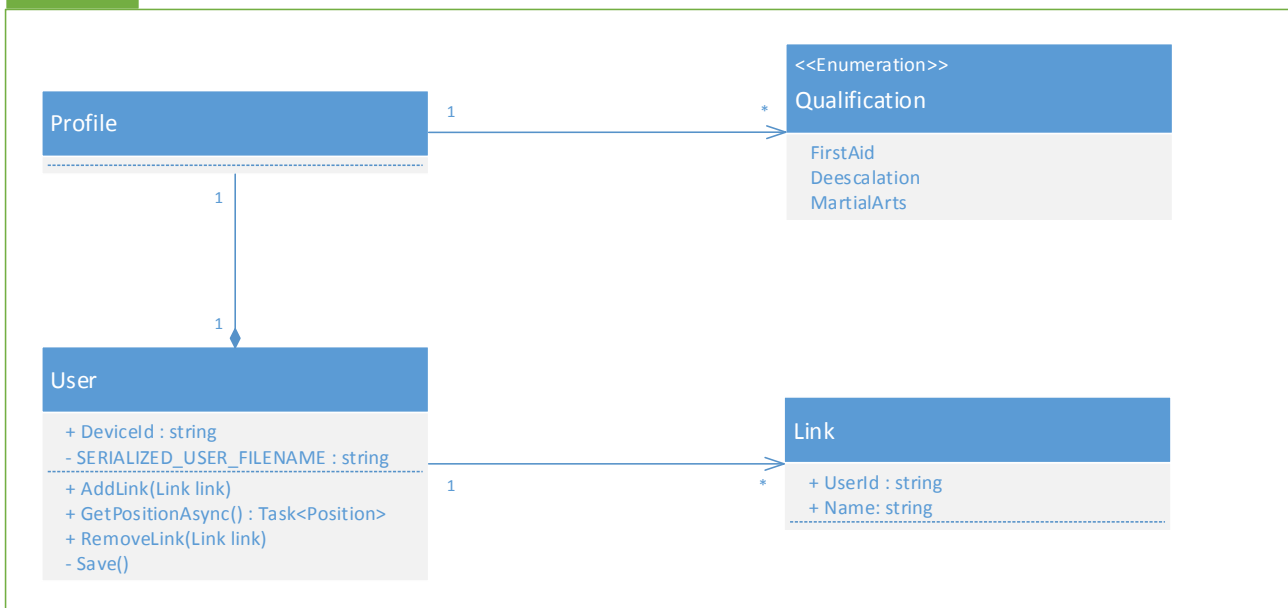


Abbildung 4.7 – Paket Users



Klasse 23 - User

User : TableAggregate

Beschreibung: Die Klasse **User** kapselt die Informationen über den aktuellen Benutzer der Applikation. Zu einem **User** können Links hinzugefügt und entfernt werden. Ein **User** besitzt außerdem ein **Profile** in dem weitere Informationen über den **User** gekapselt sind. Dieses **Profile** kann geändert werden. Da das **User**-Objekt im Hintergrundspeicher gesichert werden muss, kümmert sich die Methode **Save** nach jeder Änderung um die Serialisierung.

Attribute:

- `+ DeviceId : string` – Die Geräte-ID des Benutzers, die im Server für die Push-Benachrichtigungen benötigt wird
- `+ Links : List<Link>` – Die Liste aller verknüpften Personen
- `+ Position : BasicGeoposition` – Die aktuelle Position des Benutzers
- `+ Profile : Profile` – Das Profil des Benutzers
- `- SERIALIZED_USER_FILENAME : string` – Konstante, die den Speicherort des Objektes für die Serialisierung angibt

Paket: **Users**

Link

+ UserId : string
+ Name: string

Klasse 24 - Link

Link : TableAggregate

Beschreibung: Die Klasse **Link** repräsentiert eine Verbindung zu einem anderen Benutzer.

Attribute:

+ *UserId* : string – die ID des Benutzers auf den dieser **Link** zeigt
+ *Name* : string – der Bezeichner des **Link**

Paket: **Users**

<<Enumeration>>

Qualification

FirstAid
Deescalation
MartialArts

Klasse 25 - Qualification

Qualification

Beschreibung: Enumeration der verschiedenen Qualifikationen, die ein User in seinem Profil (Profile) angeben kann.

Werte:

FirstAid – Erste Hilfe

Deescalation – Deeskalation und Massenkontrolle

MartialArts – Kampfsport

Paket: **Users**

Profile

Klasse 26 - Profile

Profile

Beschreibung: Das Profil bildet den Teil der Benutzerinformationen, der „öffentlich“ auch anderen zur Verfügung gestellt werden kann. Momentan sind nur Qualifikationen darin enthalten, dies kann als separates Objekt aber in Zukunft erweitert werden.

Attribute:

+ *Qualifications* : Set<Qualification> - Die Menge aller Qualifikationen des Benutzers

Paket: **Users**

Security

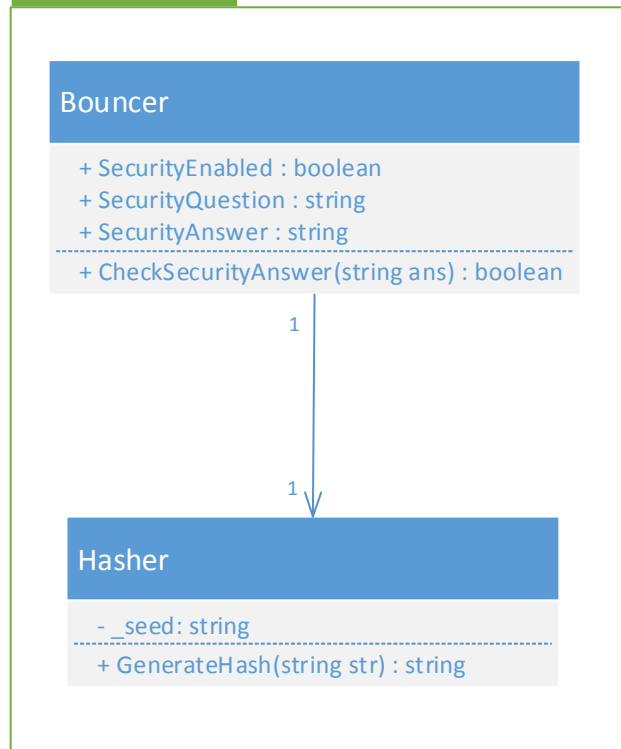
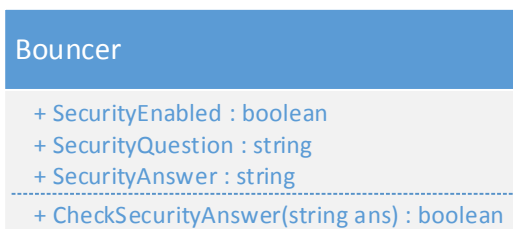


Abbildung 4.8 – Paket Security



Klasse 27 - Bouncer

Bouncer

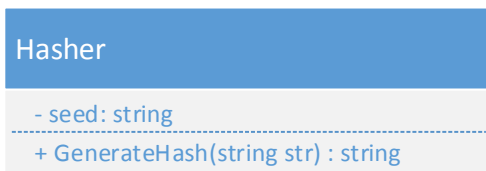
Beschreibung: Die Klasse **Bouncer** kümmert sich um sicherheitsrelevante Informationen. Sie benutzt den Hasher zum hashen der *SecurityAnswer*.

Eine von außen eingegebene Antwort kann vom **Bouncer** über *CheckSecurityAnswer* überprüft werden, sodass ein Herausgeben der Antwort nicht nötig ist.

Attribute:

- + *SecurityEnabled* : *bool* – gibt an, ob die Sicherheitsfrage aktiv ist
- + *SecurityQuestion* : *string* – die Sicherheitsfrage
- + *SecurityAnswer* : *string* – ein Hash der Antwort auf die Sicherheitsfrage
- + *Hasher* : *Hasher* – Der benutzte **Hasher**

Paket: **Security**



Klasse 28 - Hasher

Hasher

Beschreibung: Erstellt Hashes von strings mit einem festgelegten Schlüssel (seed).

Attribute:

- *_seed* : *string* – der Seed zum hashen des Strings

Paket: Security

UserSettings

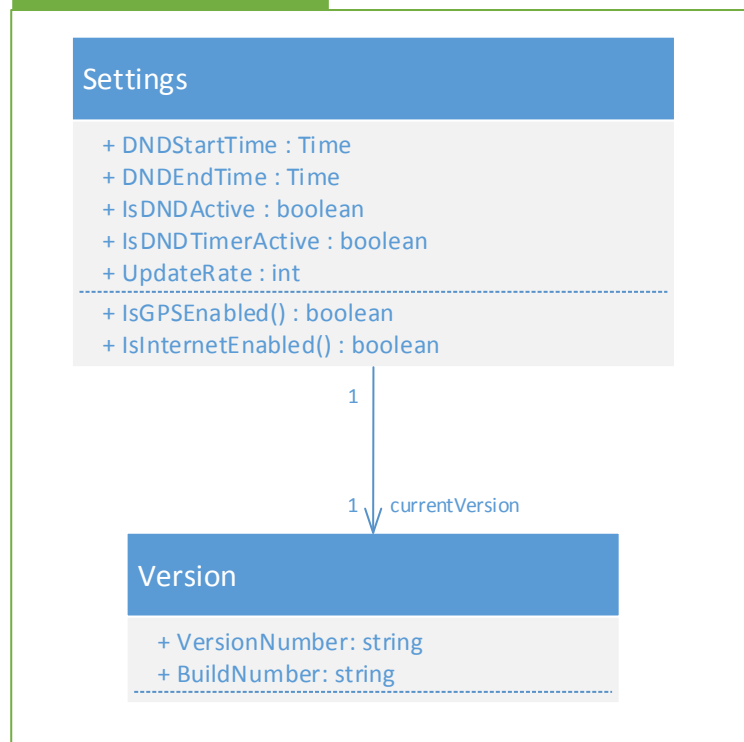


Abbildung 4.9 – Paket UserSettings

Settings
<ul style="list-style-type: none"> + DNDStartTime : Time + DNDEndTime : Time + IsDNDActive : boolean + IsDNDTimerActive : boolean + UpdateRate : int
<ul style="list-style-type: none"> + IsGPSEnabled() : boolean + IsInternetEnabled() : boolean

Klasse 29 - Settings

Settings

Beschreibung: Die Klasse **Settings** kapselt alle Einstellungen des Benutzers, die das Verhalten der Applikation steuern. Um Inkonsistenzen zu vermeiden, werden alle Daten im Hintergrundspeicher gesichert und die Parameter der Klasse dienen nur als Fassade für den Zugriff auf diese gemeinsame, synchronisierte Datenstruktur (siehe 5.1 Lokale Speicherung).

Attribute:

- + *CurrentVersion* : *Version* – Die aktuell installierte Version der Applikation
- + *DNDStartTime* : *Time* – Der festgelegte Startzeitpunkt des Ruhemodus
- + *DNDEndTime* : *Time* – Der festgelegte Endzeitpunkt des Ruhemodus
- + *IsDNDActive* : *bool* – “Wahr”, falls der Ruhemodus manuell aktiviert wurde
- + *IsDNDTimeActive* : *bool* – “Wahr”, falls die Zeitschaltung des Ruhemodus aktiviert wurde
- + *UpdateRate* : *int* – Die festgelegte Aktualisierungsrate der Position des Benutzers (Ausführungsrate des BackgroundAgent)

Paket: **UserSettings**

Version
<ul style="list-style-type: none"> + VersionNumber: string + BuildNumber: string

Klasse 30 - Version

Version

Beschreibung: Die Klasse **Version** kapselt die Version der aktuell installierten Applikation.

Attribute:

- + *VersionNumber* : *string* – die aktuelle Version der Applikation
- + *BuildNumber* : *string* – die aktuelle Buildnummer der Applikation

Paket: **UserSettings**

Logging

Log
<ul style="list-style-type: none"> - _logList : List<String>
<ul style="list-style-type: none"> + Log(String str) + GetLog() : List<String>

Abbildung 4.10 – Paket Logging

Log

```
- _logList : List<String>
+ Log(String str)
+ GetLog() : string
```

Klasse 31 - Log

Log

Beschreibung: Die Klasse **Log** kümmert sich um das Aufzeichnen des Verhaltens des Benutzers und der Kommunikation der Applikation mit dem Server. Sie soll dabei helfen, Fehler und abnormales Verhalten aufzudecken.

Attribute:

- *_logList* : *string* – Ein String mit allen Log-Einträgen.

Paket: **Logging**

4.4. Klassen des ServerControllers

ServerManagement

<<Schnittstelle>>

IServerController

```
+ AddPushDataObserver(IPushDataObserver obs)
+ CreateNewEmergencyAsync(User user, EmergencyDetails det) : Task<Emergency>
+ EditEmergencyAsync(Emergency em, EmergencyDetails det) : Task
+ EndEmergencyAsync(Emergency em) : Task
+ LinkToUserAsync(User user, Link link) : Task
+ ProlongEmergencyAsync(Emergency em) : Task
+ ReportEmergencyAsync(Emergency em) : Task
+ RetrieveEmergenciesAsync(Position pos) : Task<List<Emergency>>
+ RetrieveEmergencyDetailsAsync(Emergency em) : Task<Details>
+ SelectLatestVersionAsync() : Task<Version>
+ SelectLinksAsync(User user) : Task<List<Links>>
+ SetUserActivityAsync(User user, boolean active) : Task
+ SetUserPositionAsync(User user, Position pos) : Task
+ SetUserRoleAsync(Emergency em, User user, Role role) : Task
+ UnlinkFromUserAsync(User user, Link link) : Task
```



ServerManager

Abbildung 4.11 – Paket ServerManagement

<<Schnittstelle>>

IServerController

```
+ AddPushDataObserver(IPushDataObserver obs)
+ CreateNewEmergencyAsync(User user, Details det) : Task<Emergency>
+ EditEmergencyAsync(Emergency em, Details det) : Task
+ EndEmergencyAsync(Emergency em) : Task
+ LinkToUserAsync(User user, Link link) : Task
+ ProlongEmergencyAsync(Emergency em) : Task
+ ReportEmergencyAsync(Emergency em) : Task
+ RetrieveEmergenciesAsync(Position pos) : Task<List<Emergency>>
+ RetrieveEmergencyDetailsAsync(Emergency em) : Task<Details>
+ SelectLatestVersionAsync() : Task<Version>
+ SelectLinksAsync(User user) : Task<List<Links>>
+ SetUserActivityAsync(User user, boolean active) : Task
+ SetUserPositionAsync(User user, Position pos) : Task
+ SetUserRoleAsync(Emergency em, User user, Role role) : Task
+ UnlinkFromUserAsync(User user, Link link) : Task
```

Klasse 32 - IServerController

ServerManager

Klasse 33 - ServerManager

IServerController

Beschreibung: Diese Schnittstelle dient als Fassade für das gesamte Subsystem des ServerControllers und definiert alle Aufrufe, die das Model darauf ausführen kann. Eine Beschreibung aller Methoden ist in 2.3 *Subsystem-Schnittstellen* zu finden.

Paket: **ServerManagement**

ServerManager : IServerController

Beschreibung: Der **ServerManager** implementiert die Fassaden-Schnittstelle **IServerController** und alle ihre Methoden. Er delegiert dabei die Aufrufe auf dem Subsystem an die anderen Klassen und Pakete und verknüpft die Anwendungslogik. **IPushDataObserver** werden zum *pushProcessor* weitergegeben, die meisten anderen Aufrufe hingegen benötigen die Serververbindung in *server*. Zu den Methodenbeschreibungen siehe 2.3 *Subsystem-Schnittstellen*.

Attribute:

- *server* : *ServerConnector* – Enthält Serververbindung und – aufrufe
- *pushProcessor* : *PushProcessor* – Verwaltet Push-Benachrichtigungen

Paket: **ServerManagement**

PushDataObserver

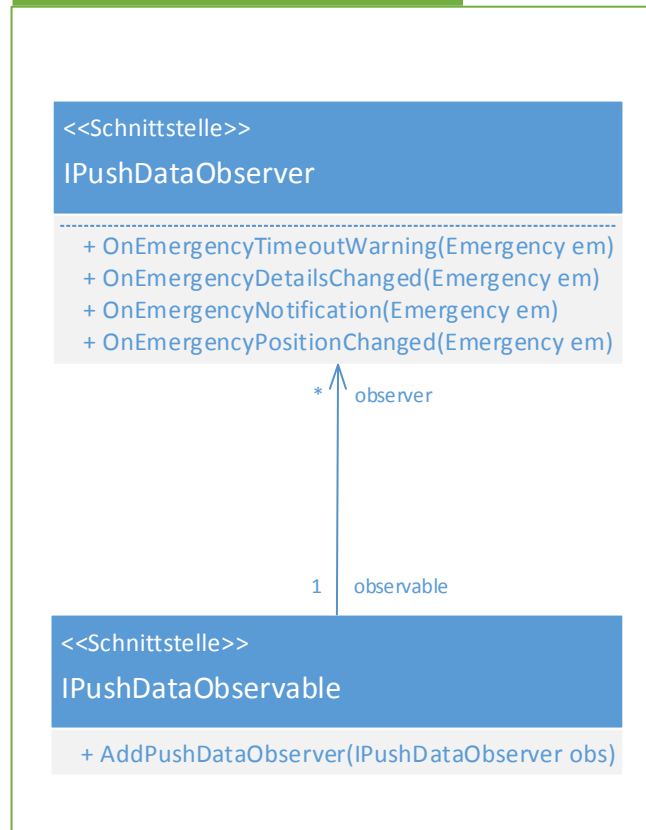


Abbildung 4.12 – Paket PushDataObserver



Klasse 34 - IPushDataObserver

IPushDataObserver

Beschreibung: Schnittstelle, um Methoden zum Beobachten neu eintreffender Push-Benachrichtigungen an einem `IPushDataObservable` definieren. Durch Wahl der aufzurufenden Methode kann die Art der Benachrichtigung unterschieden werden. Dabei stehen Methoden zum Informieren über nahende Zeitlimits (*OnEmergencyTimeoutWarning*), geänderte Notfalldetails (*OnEmergencyDetailsChanged*), neue Notfälle in der näheren Umgebung (*OnEmergencyNotification*) oder eine geänderte Notfallposition (*OnEmergencyPositionChanged*) bereit. Der aktuelle Notfall wird allen Methodenaufrufen mit übergeben.

Paket: `PushDataObserver`

<<Schnittstelle>>

IPushDataObservable

+ AddPushDataObserver(IPushDataObserver obs)

Klasse 35 - IPushDataObservable

IPushDataObservable

Beschreibung: Schnittstelle, die von IPushDataObserver-Objekten beobachtet werden kann. Diese müssen sich zuerst über *AddPushDataObserver* anmelden und werden beim Eintreffen neuer Push-Benachrichtigungen informiert.

Paket: PushDataObserver

PushProcessing

PushProcessor

```
- _observers: List<IPushDataObservers>
+ PushProcessor()
+ PushChannel_ChannelUriUpdated(object sender,
  NotificationChannelUriEventArgs e)
+ PushChannel_ErrorOccurred(object sender,
  NotificationChannelErrorEventArgs e)
+ PushChannel_ConnectionStatusChanged(object sender,
  NotificationChannelConnectionEventArgs e)
+ PushChannel_ShellToastNotificationReceived(object
  sender, NotificationEventArgs e)
+ PushChannel_HttpNotificationReceived(object sender,
  HttpNotificationEventArgs e)
```

Abbildung 4.13 – Paket PushProcessing

PushProcessor

```
- _observers: List<IPushDataObservers>
+ PushProcessor()
+ PushChannel_ChannelUriUpdated(object sender,
  NotificationChannelUriEventArgs e)
+ PushChannel_ErrorOccurred(object sender,
  NotificationChannelErrorEventArgs e)
+ PushChannel_ConnectionStatusChanged(object sender,
  NotificationChannelConnectionEventArgs e)
+ PushChannel_ShellToastNotificationReceived(object
  sender, NotificationEventArgs e)
+ PushChannel_HttpNotificationReceived(object sender,
  HttpNotificationEventArgs e)
```

Klasse 36 - PushProcessor

PushProcessor : IPushDataObservable

Beschreibung: enCourage nutzt die „Raw Notifications“ der „Windows Push Notification Services“ (WNS), um Daten in der Applikation ohne Anfrage zu aktualisieren und der **PushProcessor** kümmert sich um den Erhalt und die Verarbeitung dieser Benachrichtigungen. Diese Klasse meldet sich dabei für diese an und erhält die empfangenen Daten direkt vom Betriebssystem. Die aufzurufenden Methoden sind dabei vom der WNS API vordefiniert.

Um andere Klassen über diese erhaltenen Daten zu informieren, implementiert der **PushProcessor** die Schnittstelle **IPushDataObservable** und informiert nach jeder Nachricht alle observers.

Attribute:

- *_observers* : *List<IPushDataObserver>* – Eine Liste aller angemeldeten Beobachter

Paket: PushProcessing

ServerConnection

ServerConnector

```
+ UpdateRowAsync(TableRow row ): Task  
+ InsertRowAsync(TableRow row ): Task  
+ DeleteRowAsync(TableRow row ) : Task  
+ SelectEmergenciesAsync(decimal lng, decimal lat) :  
    Task<List<EmergencyRow>>  
+ SelectEmergencyAsync(EmergencyRow row) :  
    Task<EmergencyRow>  
+ SelectLinksAsync(UsersRow row) : Task<List<LinksRow>>  
+ GetServerVersionAsync() : Task<Version>
```

Abbildung 4.14 – Paket ServerConnection

ServerConnector

```
+ UpdateRowAsync(TableRow row) : Task
+ InsertRowAsync(TableRow row) : Task
+ DeleteRowAsync(TableRow row) : Task
+ SelectEmergenciesAsync(decimal lng, decimal lat) :
  Task<List<EmergencyRow>>
+ SelectEmergencyAsync(EmergencyRow row) :
  Task<EmergencyRow>
+ SelectLinksAsync(UsersRow row) : Task<List<LinksRow>>
+ GetServerVersionAsync() : Task<Version>
```

Klasse 37 - ServerConnector

ServerConnector

Beschreibung: Der **ServerConnector** stellt die Verbindung zwischen Client und Server her und fungiert als zentrales Bindeglied zwischen den Systemen. Er kapselt dabei sowohl die Serververbindungsdaten als auch die Datenbankwahl und -verbindung. Damit dient er auch dem Client als Fassade für alle Serveranfragen.

Ebenso besitzt der **ServerConnector** mithilfe der Azure Mobile Services API direkten Kontakt zu den Datenbanktabellen und kann Operationen wie das Aktualisieren (*UpdateRowAsync*), Einfügen (*InsertRowAsync*) und Löschen (*DeleteRowAsync*) von Datenbankzeilen auf ihnen wie auf lokalen Datenstrukturen ausführen, auch wenn im Server noch Anwendungscode über Controller zwischengeschaltet werden kann. Auch das Anfordern einer (*SelectEmergencyAsync*) oder mehrerer (*SelectEmergenciesAsync*) Notfall-Datenzeilen oder Verknüpfungs-Datenzeilen (*SelectLinksAsync*) ist in kompakter Form ohne äußerliches Wissen über Datenbank und Server möglich.

Attribute:

- *mobileService* : *MobileServiceClient* – Der verwendete Azure Mobile Service
- *emergenciesTable* : *IMobileServiceTable* – Die Datenbanktabelle für die *EmergenciesRow*-Objekte
- *usersTable* : *IMobileServiceTable* – Die Datenbanktabelle für die *UsersRow*-Objekte
- *linksTable* : *IMobileServiceTable* – Die Datenbanktabelle für die *LinksRow*-Objekte
- *uteTable* : *IMobileServiceTable* – Die Datenbanktabelle für die *UsersToEmergenciesRow*-Objekte
- *utqTable* : *IMobileServiceTable* – Die Datenbanktabelle für die *UsersToQualificationsRow*-Objekte

Paket: **ServerConnection**

Aggregates

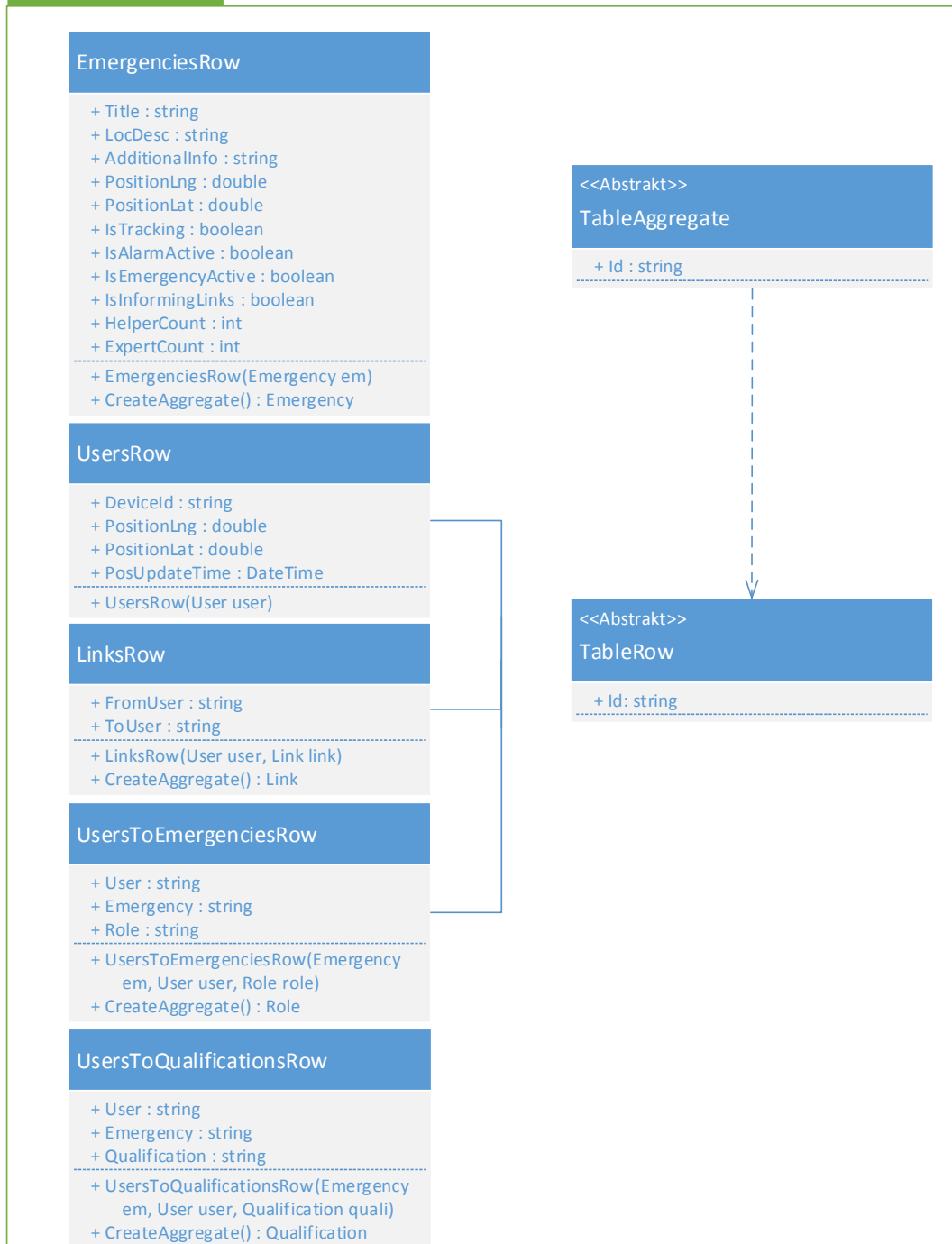


Abbildung 4.15 – Paket Aggregates

<<Abstrakt>>

TableRow

+ Id: string

Klasse 38 - TableRow

EmergenciesRow

+ Title : string
+ LocDesc : string
+ AdditionalInfo : string
+ PositionLng : double
+ PositionLat : double
+ IsTracking : boolean
+ IsAlarmActive : boolean
+ IsEmergencyActive : boolean
+ IsInformingLinks : boolean
+ HelperCount : int
+ ExpertCount : int

+ EmergenciesRow(Emergency em)
+ CreateAggregate() : Emergency

Klasse 39 - EmergenciesRow

TableRow

Beschreibung: Abstrakte Oberklasse aller Servervarianten der Aggregat-Klassen. Sie definiert dabei keine gemeinsamen Methoden zur Verwendung, sondern nur eine für die Datenbank eindeutige Identifikation und dient primär der gebündelten Verwendung der Unterklassen.

Attribute:

+ Id : string – Eindeutiger Identifikationshash für die Datenbank

Paket: **Aggregates**

EmergenciesRow : TableRow

Beschreibung: Serverseitige Version der Klasse Emergency, die alle Attribute enthält, die in der Datenbank gespeichert werden müssen.

Zusätzlich existieren zum Konvertieren der server- und clientseitigen Versionen ein parametrisierter Konstruktor (*EmergenciesRow*) für die eine Richtung, sowie eine Fabrikmethode (*CreateAggregate*) für die andere.

Attribute:

+ Title : string – Titel des Notfalls

+ LocDesc : string – Zusätzliche Ortbeschreibung

+ AdditionalInfo : string – Zusätzliche Informationen

+ PositionLng : double – Längengrad der Position

+ PositionLat : double – Breitengrad der Position

+ IsTracking : boolean – Wahr, falls die Notfallposition kontinuierlich verfolgt wird

+ IsAlarmActive : boolean – Wahr, falls der Alarm noch aktiv ist

+ IsEmergencyActive : boolean – Wahr, falls der Notfall noch nicht beendet wurde

+ IsInformingLinks : boolean – Wahr, wenn die Verknüpfungen des Melders benachrichtigt werden sollen

+ HelperCount : int – Die Anzahl der helfenden Personen

+ ExpertCount : int – Die Anzahl der helfenden Experten

Paket: **Aggregates**

UsersRow

```
+ DeviceId : string
+ PositionLng : double
+ PositionLat : double
+ PosUpdateTime : DateTime
+ UsersRow(User user)
```

Klasse 40 - UsersRow

UsersRow : TableRow

Beschreibung: Serverseitige Version der Klasse User, die alle Attribute enthält, die in der Datenbank gespeichert werden müssen.

Zusätzlich existiert zum Konvertieren der client- in die serverseitige Version ein parametrisierter Konstruktor (*UsersRow*), eine Umwandlung in die andere Richtung ist nicht möglich, da das User-Objekt Gerät-gebunden ist.

Attribute:

- + *DeviceId* : *string* – Geräte-ID des Benutzers
- + *PositionLng* : *double* – Längengrad der Position
- + *PositionLat* : *double* – Breitengrad der Position
- + *PosUpdateTime* : *DateTime* – Zeitstempel der letzten Positionsaktualisierung

Paket: *Aggregates*

LinksRow

```
+ FromUser : string
+ To User : string
+ LinksRow(User user, Link link)
+ CreateAggregate() : Link
```

Klasse 41 - LinksRow

LinksRow : TableRow

Beschreibung: Serverseitige Version der Klasse Link, die alle Attribute enthält, die in der Datenbank gespeichert werden müssen.

Zusätzlich existieren zum Konvertieren der server- und clientseitigen Versionen ein parametrisierter Konstruktor (*LinksRow*) für die eine Richtung, sowie eine Fabrikmethode (*CreateAggregate*) für die andere.

Attribute:

- + *FromUser* : *string* – ID des Benutzers, von dem die Verknüpfung ausgeht (Benachrichtigt)
- + *ToUser* : *string* – ID des Benutzers, der verknüpft ist (Wird benachrichtigt)

Paket: *Aggregates*

UsersToEmergenciesRow

```
+ User : string
+ Emergency : string
+ Role : string
-----
+ UsersToEmergenciesRow(Emergency
    em, User user, Role role)
+ CreateAggregate() : Role
```

Klasse 42 - UsersToEmergenciesRow

UsersToQualificationsRow

```
+ User : string
+ Emergency : string
+ Qualification : string
-----
+ UsersToQualificationsRow(Emergency
    em, User user, Qualification quali)
+ CreateAggregate() : Qualification
```

Klasse 43 - UsersToQualificationsRow

UsersToEmergenciesRow : TableRow

Beschreibung: Serverseitige Darstellung der Beziehung zwischen einem Benutzer und einem Notfall durch eine in der Enumeration Role definierten Rolle. Diese Klasse enthält alle Attribute, die in der Datenbank gespeichert werden müssen.

Durch einen parametrisierter Konstruktor (*UsersToEmergenciesRow*) kann eine Datenbankzeile mit der aktuellen Rolle neu erzeugt werden, für das Auslesen der Rolle (die andere Richtung) existiert eine Fabrikmethode (*CreateAggregate*).

Attribute:

- + *User : string* – Benutzer, der bei dem Notfall eine Rolle einnimmt
- + *Emergency : string* – Notfall, bei dem der Benutzer eine Rolle einnimmt
- + *Role : string* – Rolle, die der Benutzer bei dem Notfall einnimmt

Paket: *Aggregates*

UsersToQualificationsRow : TableRow

Beschreibung: Serverseitige Darstellung der besonderen Qualifikationen eines Benutzers bei einem Notfall durch einen in der Enumeration Qualification definierten Wert. Diese Klasse enthält alle Attribute, die in der Datenbank gespeichert werden müssen.

Durch einen parametrisierter Konstruktor (*UsersToQualificationsRow*) kann eine Datenbankzeile mit der aktuellen Rolle neu erzeugt werden, für das Auslesen der Rolle (die andere Richtung) existiert eine Fabrikmethode (*CreateAggregate*).

Attribute:

- + *User : string* – Benutzer, der bei dem Notfall eine Qualifikation besitzt
- + *Emergency : string* – Notfall, bei dem der Benutzer eine Qualifikation besitzt
- + *Qualification : string* – Qualifikation, die der Benutzer bei einem Notfall besitzt

Paket: *Aggregates*



TableAggregate

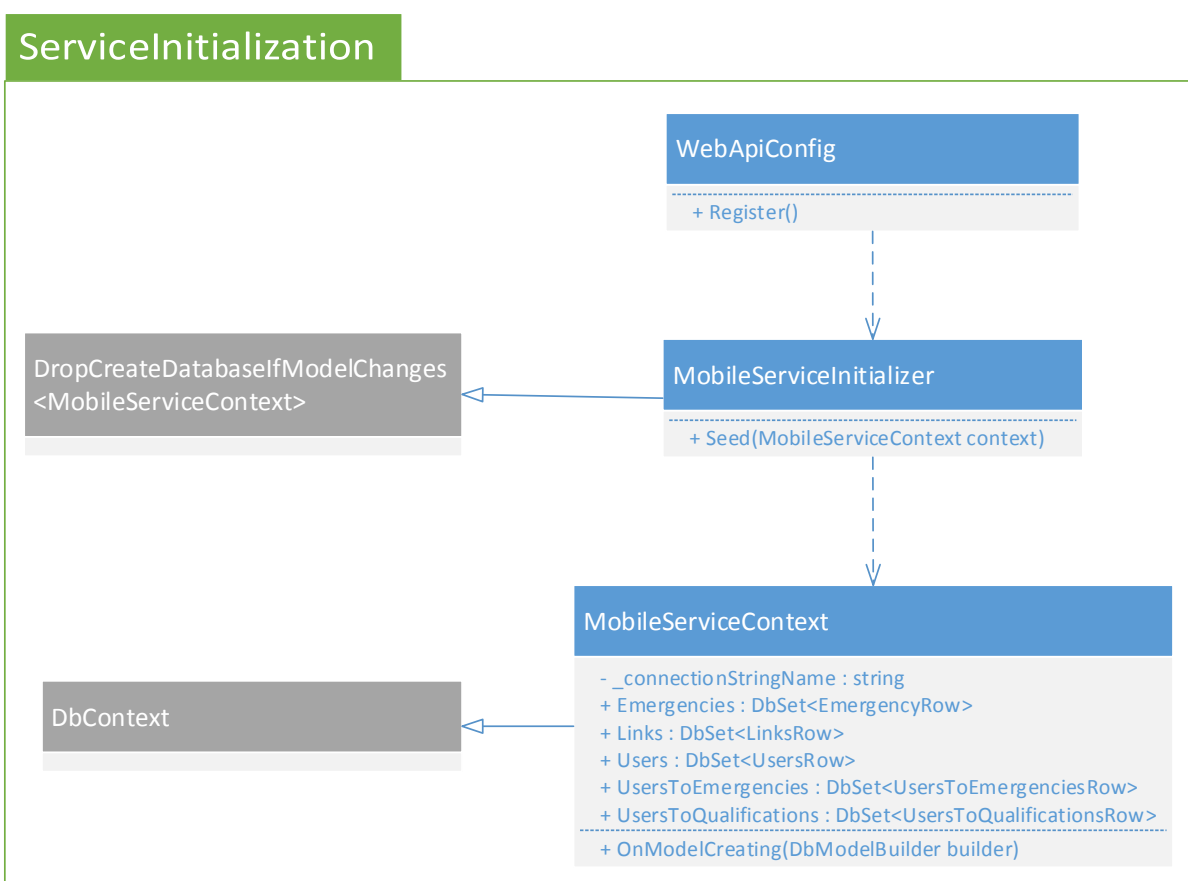
Beschreibung: Abstrakte Oberklasse aller clientseitigen Varianten der Aggregat-Klassen. Sie definiert dabei keine gemeinsamen Methoden zur Verwendung, sondern nur eine für die Datenbank eindeutige Identifikation und dient primär der gebündelten Verwendung der Unterklassen.

Attribute:

+ Id : string – Eindeutiger Identifikationshash für die Datenbank

Paket: **Aggregates**

4.5. Klassen des Servers



WebApiConfig

+ Register()

Klasse 45 - WebApiConfig

WebApiConfig

Beschreibung: Diese Klasse dient dem Konfigurieren des Servers und leitet die Initialisierung des Dienstes und der Datenbank an. Beim Starten des Servers wird hierbei die Methode *Register* aufgerufen, in der diese Vorarbeit geleistet werden kann.

Attribute:

Paket: **ServerInitialization**

MobileServiceInitializer

+ Seed(MobileServiceContext context)

Klasse 46 - MobileServiceInitializer

MobileServiceInitializer : DropCreateDatabaseIfModelChanges <MobileServiceContext>

Beschreibung: Der **MobileServiceInitializer** startet die Initialisierung der Datenbank und erstellt die Datenbanktabellen anhand eines gegebenen Kontextes (MobileServiceContext). Beim Starten wird *Seed* aufgerufen, worin anhand des gegebenen Kontextes die Datenbank erstellt und mit ersten, vordefinierten Daten gefüllt wird.

Attribute:

Paket: **ServerInitialization**

MobileServiceContext

- _connectionStringName : string
+ Emergencies : DbSet<EmergencyRow>
+ Links : DbSet<LinksRow>
+ Users : DbSet<UsersRow>
+ UsersToEmergencies : DbSet<UsersToEmergenciesRow>
+ UsersToQualifications : DbSet<UsersToQualificationsRow>
+ OnModelCreating(DbModelBuilder builder)

Klasse 47 - MobileServiceContext

MobileServiceContext : DbContext

Beschreibung: In dieser Klasse werden die Datenbanktabellen anhand gegebener Model-Objekte (*DataObjects*) erstellt sowie die Zugriffsparameter auf diese definiert. Jede Tabelle wird dabei abstrakt durch ein **DbSet**-Objekt dargestellt.

Attribute:

- *_connectionStringName* : *string* – Enthält den Namen der Datenbankverbindung
+ *Emergencies* : *DbSet<EmergencyRow>* - Die Datenbanktabelle der **EmergencyRow**-Objekte
+ *Links* : *DbSet<LinksRow>* - Die Datenbanktabelle der **LinksRow** - Objekte
+ *Users* : *DbSet<UsersRow>* - Die Datenbanktabelle der **UsersRow** - Objekte
+ *UsersToEmergencies* : *DbSet<UsersToEmergenciesRow>* - Die Datenbanktabelle der **UsersToEmergenciesRow** -Objekte
+ *UsersToQualifications* : *DbSet<UsersToQualificationsRow>* - Die Datenbanktabelle der **UsersToQualificationsRow** -Objekte

Paket: **ServerInitialization**

BackgroundTasks

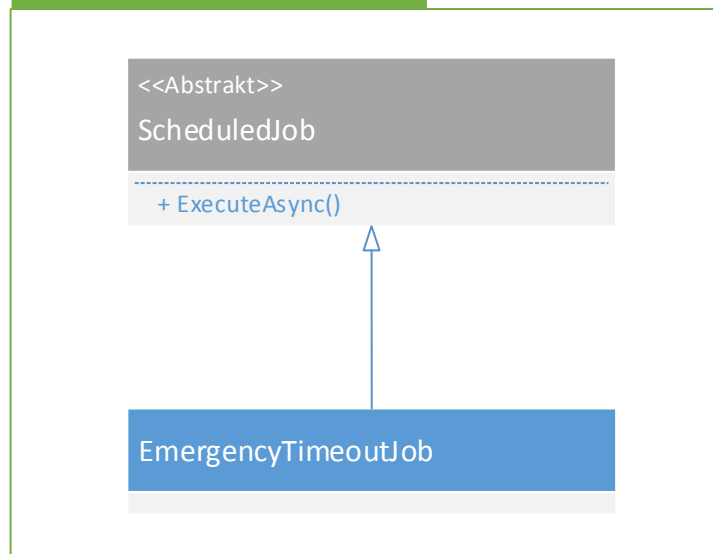


Abbildung 4.17 – Paket BackgroundTasks

EmergencyTimeoutJob

Klasse 48 - EmergencyTimeoutJob

EmergencyTimeoutJob : ScheduledJob

Beschreibung: Diese Klasse kapselt den geplanten Auftrag, der kurz vor Ablauf des Zeitlimits eines Notfalls aufgerufen wird, und den Melder sowie dessen verknüpfte Personen über den Ablauf des Zeitlimits informiert. Die Startmethode des Auftrags bildet dabei die von `ScheduledJob` geerbte Methode `ExecuteAsync()`.

Attribute:

Paket: BackgroundTasks

DatabaseTables



Abbildung 4.18 – Paket DatabaseTables

<<Abstrakt>>

TableController<T>

```
+ DeleteT(string obj) : Task
+ GetAllT() : IQueryable<T>
+ GetT(string id): SingleResult<T>
+ PostT(T obj) : Task<IHttpActionResult>
+ PatchT(string id, Delta<T> patch) : Task<T>
```

Klasse 49 – TableController<T>

EmergenciesRowController : TableController <EmergenciesRow>

*Bemerkung: Anhand der generischen Natur der TableController-Klassen wird hier die Verwendung der abstrakten Oberklasse **TableController<T>** beschrieben. **EmergenciesRowController** (sowie analog jeder andere TableController) ist dabei nur eine Implementierung für ein konkretes Objekt $T = \text{EmergenciesRow}$. T wird in der implementierten Klasse dabei ebenfalls in den Methodennamen ersetzt.*

Beschreibung: Diese Klasse wird als Kontroll-Klasse von der Azure Mobile Services API vor jeder Ausführung einer Anfrage an die Tabellenabstraktion **IMobileServicesTable** eingebunden, um vor der eigentlichen Anfrage an die Datenbank noch zusätzliche Funktionalität auszuführen. Diese endgültige Datenbankanfrage muss schlussendlich ebenfalls vom Controller ausgeführt werden. Es existiert hierbei für jede Art von Anfrage an die Datenbank eine Methode, die in diesem Fall aufgerufen wird.

Das wären das Anfragen einer Tabellenzeile anhand ihrer ID (GetT), das Anfragen aller Tabellenzeilen (GetAllT), das Löschen einer Zeile anhand ihrer ID (DeleteT), das Einfügen einer neuen Zeile (PostT) sowie das Aktualisieren einer Zeile anhand der ID sowie der Änderungen (PatchT).

Attribute:

Paket: **DatabaseTables**

UsersRowController : TableController<UsersRow>

Siehe **EmergenciesRowController**.

LinksRowController : TableController<LinksRow>

Siehe **EmergenciesRowController**.

UsersToEmergenciesRowController : TableController <UsersToEmergenciesRow>

Siehe **EmergenciesRowController**.

UsersToQualificationsRowController :
TableController
<UsersToQualificationsRow>

Siehe [EmergenciesRowController](#).

5. Daten

5.1. Lokale Speicherung

Zur Serialisierung der Einstellungen (Settings), die der Benutzer in der Applikation vornimmt, verwenden wir die `LocalSettings` des „Windows Phone“ Betriebssystems.

Da der Background Agent ebenfalls Zugriff auf die Einstellungen benötigt, um im Ruhemodus zu verhindern, dass neue Standortdaten an den Server gesendet werden, sollen Synchronisationsprobleme zwischen Vordergrund-Applikation und Background Agent mit Hilfe des Synchronisationsobjektes `Mutex` verhindert werden. `Mutex` ist ein primitiver Synchronisierungstyp, welcher es erlaubt, prozessübergreifende Synchronisierung sicherzustellen.

Da das User-Objekt nach dem Beenden der Applikation wieder zur Verfügung stehen muss und der Background Agent zum Aktualisieren der Position ebenfalls das User-Objekt benötigt, wird dieses mit Hilfe einer `XmlSerializer`-Instanz serialisiert und im Hintergrundspeicher lokal gesichert. Ebenso hilft der `XmlSerializer` dabei das serialisierte User-Objekt wieder zu deserialisieren.

Hierbei wird beim ersten Erstellen des User-Objekts beim Start der Applikation überprüft, ob die XML-Datei mit dem Namen entsprechend aus `SERIALIZED_USER_FILENAME` bereits existiert. Ist dies der Fall, so werden alle Informationen aus dem Hintergrundspeicher geladen. Andernfalls werden alle zu dem Benutzer gehörigen Informationen neu initialisiert und generiert und dann im Hintergrundspeicher gesichert. Nach jeder Veränderung am User-Objekt, wird dieses neu serialisiert und gesichert.

Auch hier wird zum Verhindern von Synchronisationsproblemen `Mutex` verwendet.

5.2. Aggregate

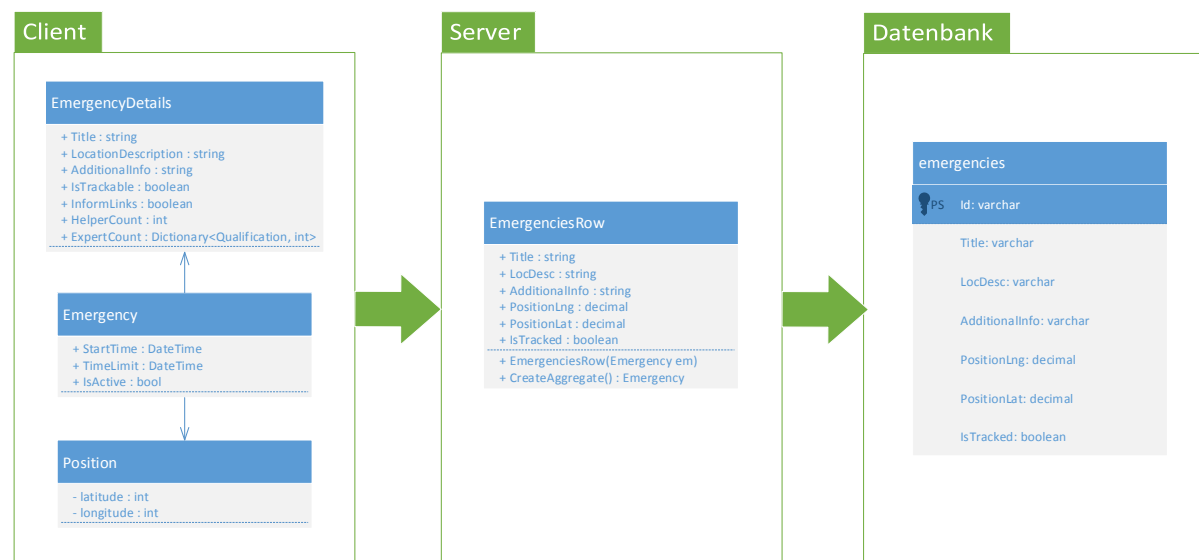


Abbildung 5.1 – Versionen der Aggregate

„Aggregat“ ist die Oberbezeichnung für alle Ansammlungen an Daten, die in der Datenbank des Servers gespeichert werden müssen, als auch die Klassen, die diese Daten enthalten. Dies ist dabei nicht mit dem Paket `Aggregates` zu verwechseln, in welchem viele (aber nicht alle) dieser Aggregat-Klassen enthalten sind.

Generell gesprochen existieren für jedes konzeptionelle Aggregat-Objekt (im Beispiel einen Notfall) mehrere Arten von Klassen, die die enthaltenen Daten speichern und verwendet werden können. Hier ist zwischen der clientseitigen Version, der serverseitigen Version sowie der Datenbankzeile zu unterscheiden (von links nach rechts).

Die clientseitige Version ist die eigentliche Klassendarstellung nach dem objektorientierten Prinzip, die in der Applikation Verwendung findet. Sie enthält primitive Datentypen sowie Objekte und definiert wichtige Methoden auf den Daten. Deswegen kann sie auch aus mehr als einer Klasse bestehen.

Die serverseitige Version wird auf dem Client nur im ServerController verwendet. Sie existiert in derselben Definition auf Client und Server und dient der einfacheren Versendung von Objekten über das Netzwerk, unterstützt durch die Azure Mobile Services API. Aus diesem Grund dient sie als reines Speicherobjekt, das nicht verändert wird, ohne viele Methoden auskommt und nur primitive Datentypen verwendet. Für die Umwandlung der client- und serverseitigen Versionen existieren hierbei besondere Konstruktoren und Fabrikmethoden, die aber von den konkreten Aggregaten abhängen.

Im Server wird durch das Schema der serverseitigen Version dann schließlich die Datenbank für dieses Objekt automatisch generiert. Das wird durch die primitiven Attributtypen vereinfacht. Dadurch sind die Instanzen der einzelnen serverseitigen Versionen als Datenbankzeilen zu verstehen und können durch geeignete Methoden auch so verwendet werden.

Ausnahmen zu dem vorher genannten Prinzip bilden dabei die beiden Aggregate für die Rollen und Qualifikationen eines Benutzers bei einem Notfall. Als Beziehung zwischen Objekten statt alphanumerischer Werte werden diese auf dem Client durch Enumerationen statt Klasseninstanzen ausgedrückt und benötigen damit für die Umwandlung in die serverseitige Version einen Kontext.

5.3. Serverseitige Speicherung

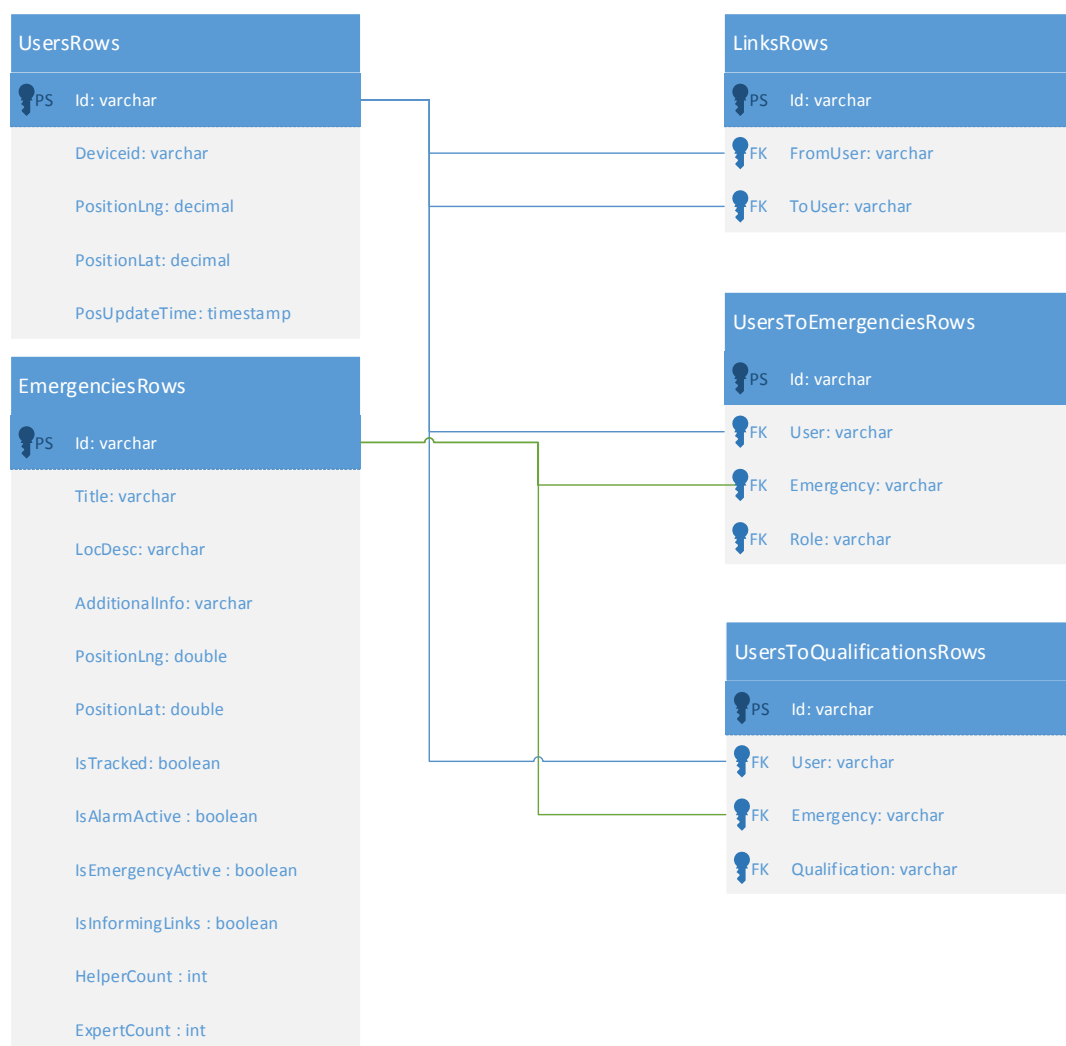


Abbildung 5.2 – Aufbau der Datenbank

Auf dem Server wird eine relationale, von Azure vorkonfigurierte SQL-Datenbank bereitgestellt (Azure SQL Database V12), die über Transact-SQL verwendet werden kann. Für die Applikation ist dabei eine Verwaltung der Datenbank sowie deren Tabellen nicht erforderlich, da die Azure Mobile Services API Möglichkeiten bietet, um viele Aufgaben zu automatisieren oder zu vereinfachen.

So sind sowohl Verbindungsaufbau, als auch Datenbankansprache über den Mobile Service Client abstrahiert als Objekte verfügbar, welche im Server- als auch Client-Code besser zu verwenden sind. Auch Datenbankabfragen können leicht per LINQ formuliert und ausgeführt werden. Die Performance der Abfragen wird über das „Lazy load“-Prinzip (möglichst minimale und späte Ausführung einer Anfrage) verbessert.

Weiterhin ermöglicht die API, Objekte auf dem Server per Vererbung der DataEntity-Klasse als Datenbankobjekte zu klassifizieren. Dadurch kann eine Datenbanktabelle für eine konkrete Klasse anhand des Schemas der Attribute automatisch erstellt werden. Mit richtiger Pflege des Models des Server-Dienstes ist damit eine manuelle Wartung der Datenbank nicht mehr nötig.

Wie oben bereits erwähnt ist die Abstraktion einer Datenbanktabelle auch auf dem Client möglich. Ähnlich einer lokalen Datenstruktur können auf diesem Abstraktionsobjekt Anfragen ausgeführt werden, die dann automatisch an den Server gesendet und dort von einem TableController geprüft werden. Erst diese Kontrollklasse führt nach Interpretation der Daten und eventueller weiterer Funktionalitäten die eigentliche Datenbankanfrage aus und gibt die Ergebnisse, eventuell weiter bearbeitet, zurück.

6. Dynamik und Ablauf

6.1. Aktionen (lokal)

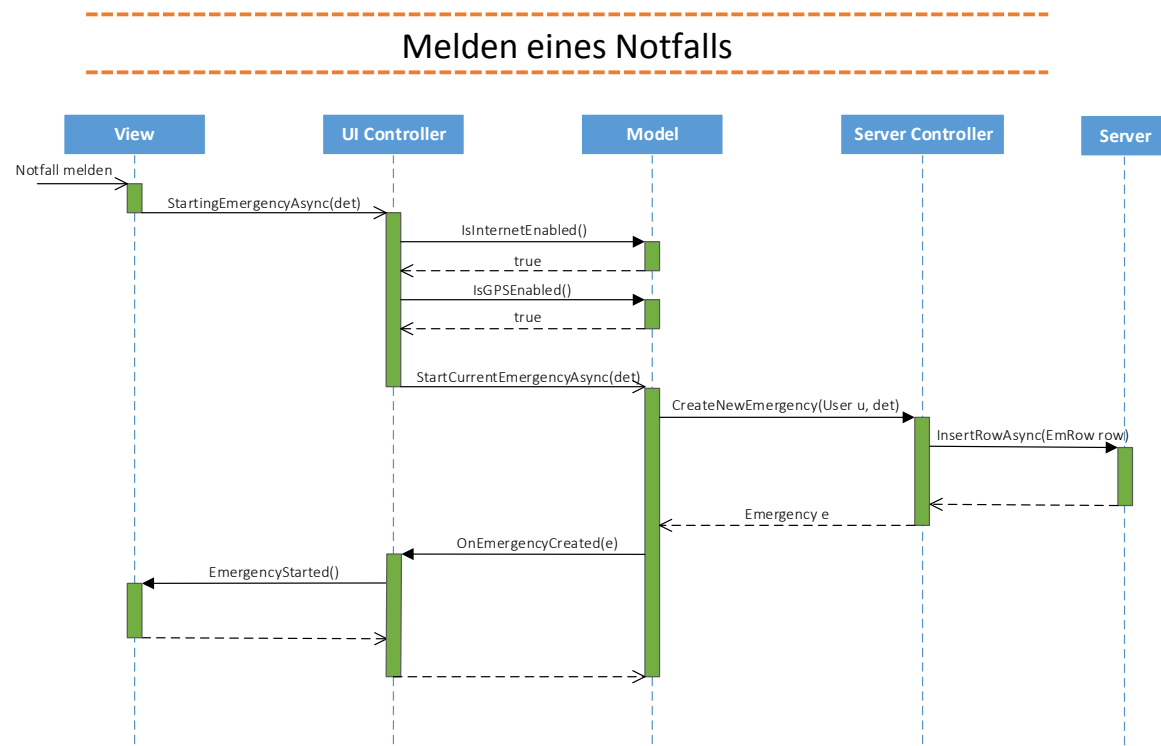


Abbildung 6.1 – Notfall wird gemeldet

Durch den Notfallmeldeknopf der Applikation meldet der Benutzer einen Notfall. Im View wird die zugehörige Animation durchgeführt, während der Ereignishandler des Knopfes den UIController über das Melden eines Notfalls sowie der angegebenen Details benachrichtigt (*StartingEmergencyAsync*). Dieser wiederum überprüft, ob GPS (*IsGPSEnabled*) und Internet (*IsInternetEnabled*) aktiviert sind, bevor er die Daten an das Model weiterreicht (*StartCurrentEmergencyAsync*). Die Weitergabe der Daten bis hierhin erfolgt ohne Rückgaben.

Das Model gibt diese Details zusammen mit dem aktuellen Benutzer an den ServerController weiter (*CreateNewEmergency*). Dieser erstellt aus den gegebenen Daten eine neue Tabellenzeile für die Datenbank und gibt dem Server den Auftrag zum Einfügen dieser Zeile (*InsertRowAsync*). Falls das Einfügen in die Datenbank erfolgreich ist, gibt der ServerController den neuen Notfall nun als passendes Emergency-Objekt an das Model zurück, welches damit seine internen Daten aktualisiert.

Die Aktualisierung im Model wiederum löst die Benachrichtigung der *IEmergencyObserver* – hier dem UIController – aus, denen der neu erstellte Notfall mitgegeben wird (*OnEmergencyCreated*). Dieser kann damit auf dem View die Aktualisierung der Anzeige anfordern (*EmergencyStarted*). Damit weiß der Benutzer, dass sein Notfall erfolgreich gemeldet wurde.

Beenden eines Notfalls

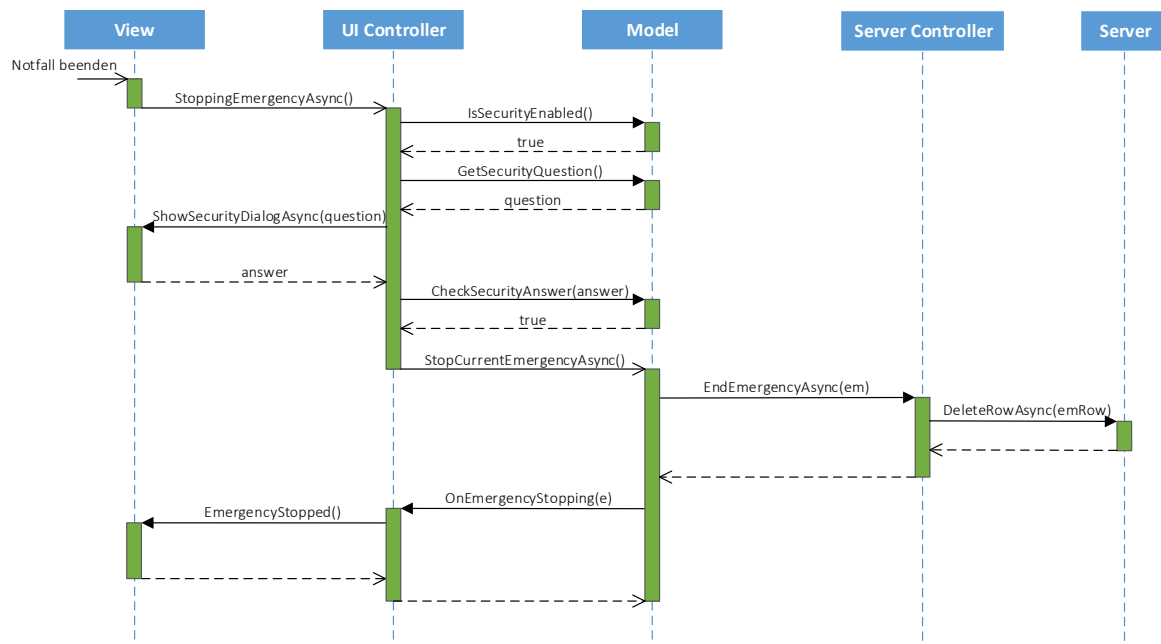


Abbildung 6.2 – Notfall wird beendet

Der Benutzer wählt über einen Knopfdruck aus, dass er den gemeldeten Notfall beenden möchte, was der Ereignishandler an den UIController weitergibt (*StoppingEmergencyAsync*).

Der UIController muss erst nun erst überprüfen, ob das Beenden des Notfalls als privilegierte Aktion durch die Sicherheitsfrage geschützt ist. Dazu fragt er am Model an, ob dieser Schutzmechanismus in den Einstellungen aktiviert wurde (*IsSecurityEnabled*) und falls ja, wie die Sicherheitsfrage lautet (*GetSecurityQuestion*). Der UIController lässt den View nun die Frage anzeigen (*ShowSecurityDialogAsync*), wartet auf die Antwort und überprüft diese mit Hilfe des Models (*CheckSecurityAnswer*).

Nach korrekter Authentifizierung des Benutzers kann nun dem Model die eigentliche Anfrage zum Beenden des Notfalls weitergegeben werden (*StopCurrentEmergency*). Das Model gibt den Aufruf zusammen mit dem aktuellen Notfall an den ServerController weiter, welcher nun in der Datenbank die Löschung der betroffenen Datenzeile veranlassen kann.

Nach dem erfolgreichen Beenden kann das Model nun den Notfall aus den eigenen Daten entfernen, worüber der UIController als *IEmergencyObserver* benachrichtigt wird (*OnEmergencyStopping*) und die Anzeige des Views anpassen kann (*EmergencyStopped*).

Erhalt einer Benachrichtigung

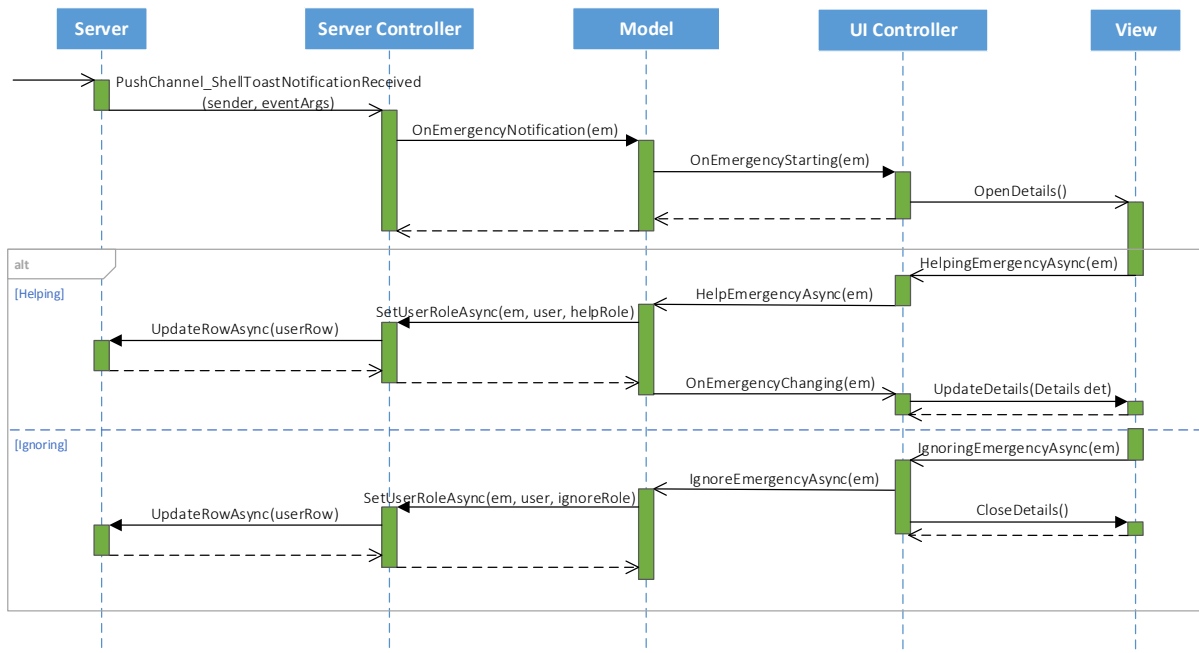


Abbildung 6.3 – Benachrichtigung wird erhalten

Über die Erstellung eines Notfalls im Server werden die Clients in der näheren Umgebung mithilfe einer Push-Benachrichtigung (*PushChannel_ShellToastNotification*) informiert. Der ServerController erstellt aus den erhaltenen Daten nun ein Emergency-Objekt und gibt es an das Model als *IPushDataObserver* weiter (*OnEmergencyNotification*). Das Model speichert die Daten des Notfalls intern und veranlasst damit die Benachrichtigung des UIControllers als *IEmergencyObserver* (*OnEmergencyStarting*), welcher den View auffordert, die Notfalldetails anzuzeigen.

Der Benutzer sieht die Details und hat nun die Wahl, bei diesem Notfall zu Hilfe zu kommen oder diesen zu ignorieren. Nachdem der Benutzer seine Wahl per Knopfdruck getroffen hat, wird diese Wahl über den UIController (*HelpingEmergencyAsync*/*IgnoringEmergencyAsync*) direkt ans Model weitergegeben (*HelpEmergencyAsync*/*IgnoreEmergencyAsync*).

Das Model übernimmt nun die Änderung der Rolle des Benutzers im betroffenen Notfall und übergibt diese Daten dem ServerController (*SetUserRoleAsync*) zur Aktualisierung im Server. Dieser veranlasst dann die eigentliche Änderung in der Datenbank (*UpdateRowAsync*), welche im Falle des Helfens andere Informierte darüber informiert und im Falle des Ignorierens den Benutzer von zukünftigen Benachrichtigungen über den Notfall abmeldet.

Nachdem die Daten im Server erfolgreich gespeichert wurden, werden die Notfalldetails geändert (*OnEmergencyChanging* -> *UpdateDetails*), falls der Benutzer bei dem Notfall hilft, oder geschlossen (*CloseDetails*), falls er den Notfall ignoriert.

Aktualisieren der Notfalldetails

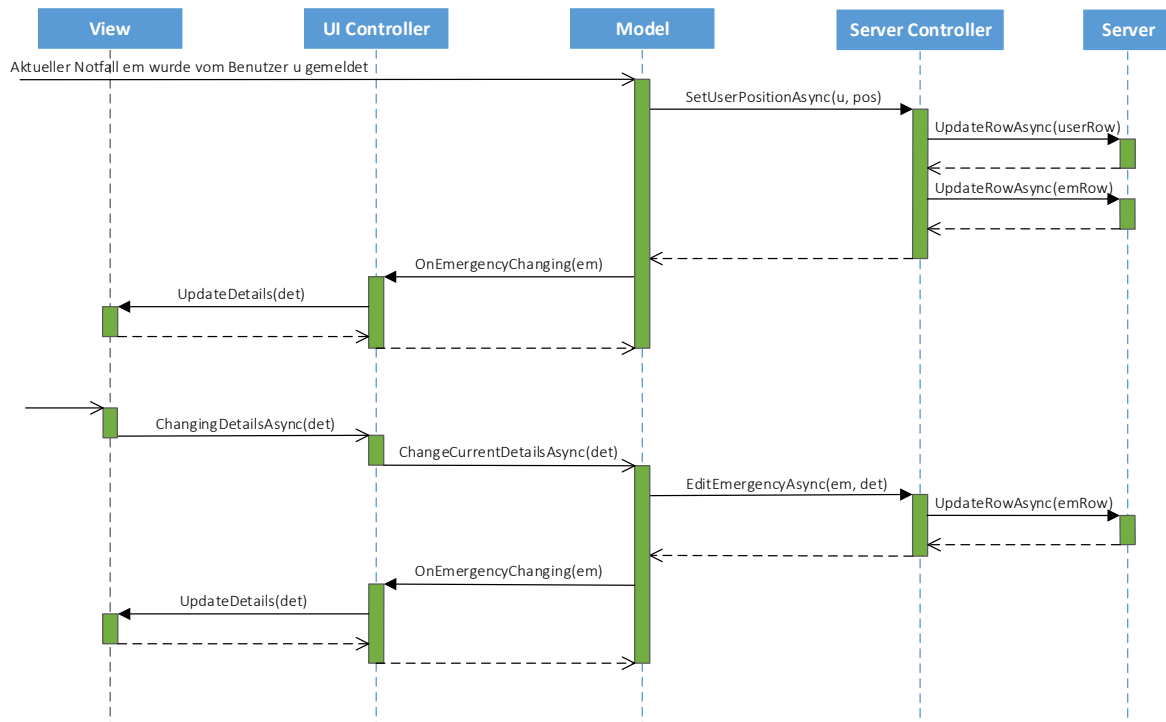


Abbildung 6.4 – Notfalldetails werden aktualisiert

Nach der Meldung eines Notfalls durch einen Benutzer können im Server zwei Dinge zu einer Änderung der Details führen: Die Position des Melders kann sich verändern (falls er die Standortverfolgung aktiviert hat) oder der Melder ändert die Spezifikationen des Notfalls.

Das Model prüft automatisch im Hintergrund, ob sich die Position des Geräts ändert. In einem solchen Fall fordert es den ServerController auf, die Position des Benutzers im Server zu aktualisieren (*SetUserPositionAsync*). Da der Standort des Benutzers auch dem Standort des Notfalls entspricht, müssen im Server sowohl die Position des Benutzers als auch die des Notfalls aktualisiert werden (*UpdateRowAsync*).

Falls der Benutzer nun manuell die Notfalldetails ändert, so erstellt der View aus den eingegebenen Daten nun ein *EmergencyDetails*-Objekt und gibt es an den UIController weiter (*ChangingDetailsAsync*). Dieser informiert das Model, dass die Details des aktuellen Notfalls aktualisiert werden müssen (*ChangeCurrentDetailsAsync*) und das Model fordert den ServerController auf, die neuen Details auf den Server hochzuladen (*EditEmergencyAsync*). Dazu muss erst ein *EmergenciesRow*-Objekt mit den neuen Daten erzeugt und an den Server weitergegeben werden (*UpdateRowAsync*).

In beiden Fällen speichert das Model (bei allen Informierten) die aktualisierten Daten intern und veranlasst damit die Benachrichtigung des UIControllers als *IEmergencyObserver* (*OnEmergencyChanging*), welcher den View auffordert, die angezeigten Notfalldetails zu aktualisieren (*UpdateDetails*).

Aktualisieren des LiveFeeds

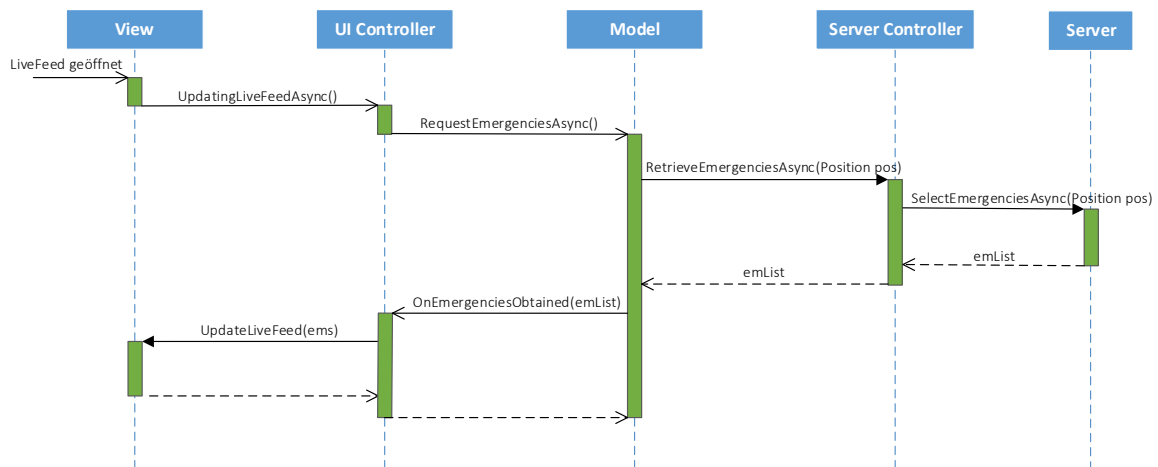


Abbildung 6.5 – LiveFeed wird aktualisiert

Der Benutzer öffnet den LiveFeed. Dies wird dem UIController über *UpdatingLiveFeedAsync* mitgeteilt, welcher die Notfall-Daten beim Model anfragt (*RequestEmergenciesAsync*).

Das Model fordert nun die Daten aus der Datenbank vom ServerController an (*RetrieveEmergenciesAsync*), welcher die Informationen dann direkt vom Server mit Hilfe von *SelectEmergenciesAsync* anfragt. Der Server bearbeitet die Anfrage und gibt eine Liste von *EmergenciesRow*-Objekten zurück, welche vom ServerController empfangen, in *Emergency*-Objekte umgewandelt und an das Model zur Speicherung übergeben werden.

Dem UIController wird nun mitgeteilt, dass neue Daten im Model angekommen sind (*OnEmergenciesObtained*), welche er auch direkt erhält. Er fordert dann schließlich den View auf, die Anzeige des LiveFeed zu aktualisieren (*UpdateLiveFeed*).

Ruhemodus verwenden

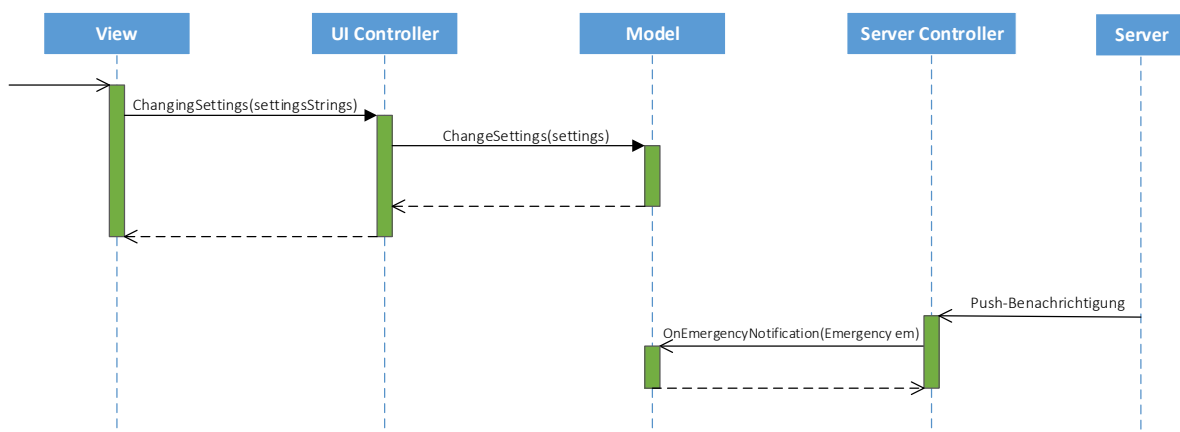


Abbildung 6.6 – Ruhemodus wird eingeschaltet

Der Benutzer aktiviert in den Einstellungen den Ruhemodus, was der Ereignishandler an den UIController weitergibt (*ChangingSettings*). Der UIController muss diese Änderung jetzt an das Model weitergeben (*ChangeSettings*).

Dies führt dazu, dass das Model alle vom ServerController ankommenden Push-Benachrichtigungen (*OnEmergencyNotification*) ignoriert und nicht an den UIController weitergibt.

Ablauf eines Notfallalarms

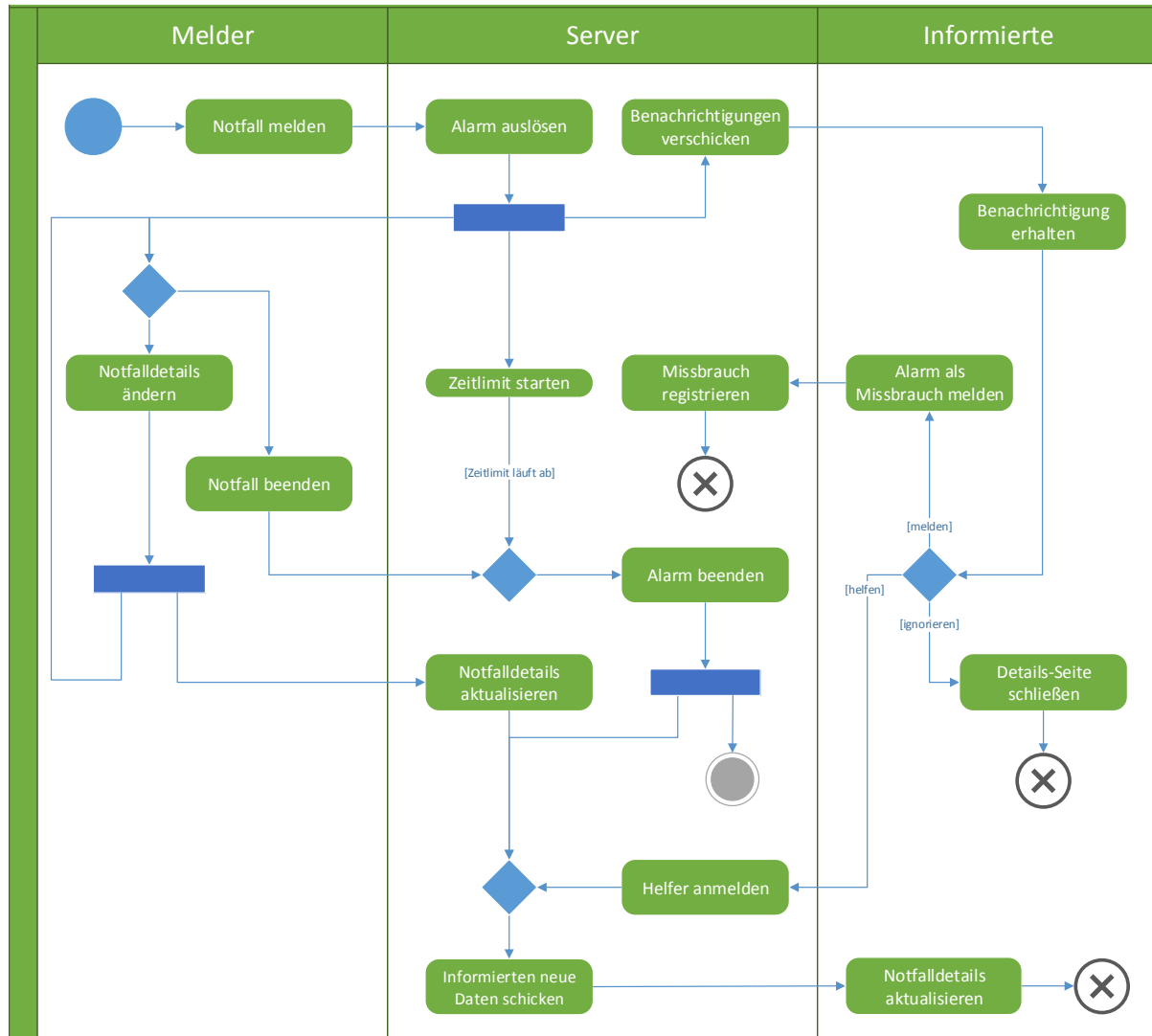


Abbildung 6.7 – Ablauf eines Notfallalarms

Der Ablauf beginnt mit dem Melden eines Notfalls ausgehend vom Gerät eines Nutzers, wodurch im Server der Notfall erstellt und ein Alarm ausgelöst wird. Während nun serverseitig das Zeitlimit des Notfalls gestartet und alle Benachrichtigungen verschickt werden, hat der Melder nun die Wahl, die Notfalldetails zu bearbeiten oder den Notfall wieder zu beenden.

Eine Bearbeitung der Details stößt dabei eine Aktualisierung der Daten im Server an, worüber – wie bei allen serverseitigen Datenänderungen – die Informierten benachrichtigt werden müssen.

Nachdem nun die Benachrichtigten die Meldung über einen neuen Notfall erhalten und die Details angezeigt haben, haben diese die Wahl, den Notfall entweder als Missbrauch zu melden, ihn zu ignorieren oder anzugeben, zur Hilfe zu eilen.

Eine Missbrauchsmeldung muss dabei dem Server übergeben werden, ebenso wie die Anmeldung als Helfer, was später auch den anderen Informierten angezeigt werden muss. Das Ignorieren eines Notfalls löst hingegen nur lokal eine Reaktion aus, bei der die Detail-Seite geschlossen wird.

Sowohl ein abgelaufenes Zeitlimit als auch die manuelle Beendigung eines Notfalls durch den Melder sorgt dabei im Server für die Beendigung des Alarms, wodurch nach der Benachrichtigung aller Informierten auch die ganze Aktivität endet.

6.2. Aktionen (serverseitig)

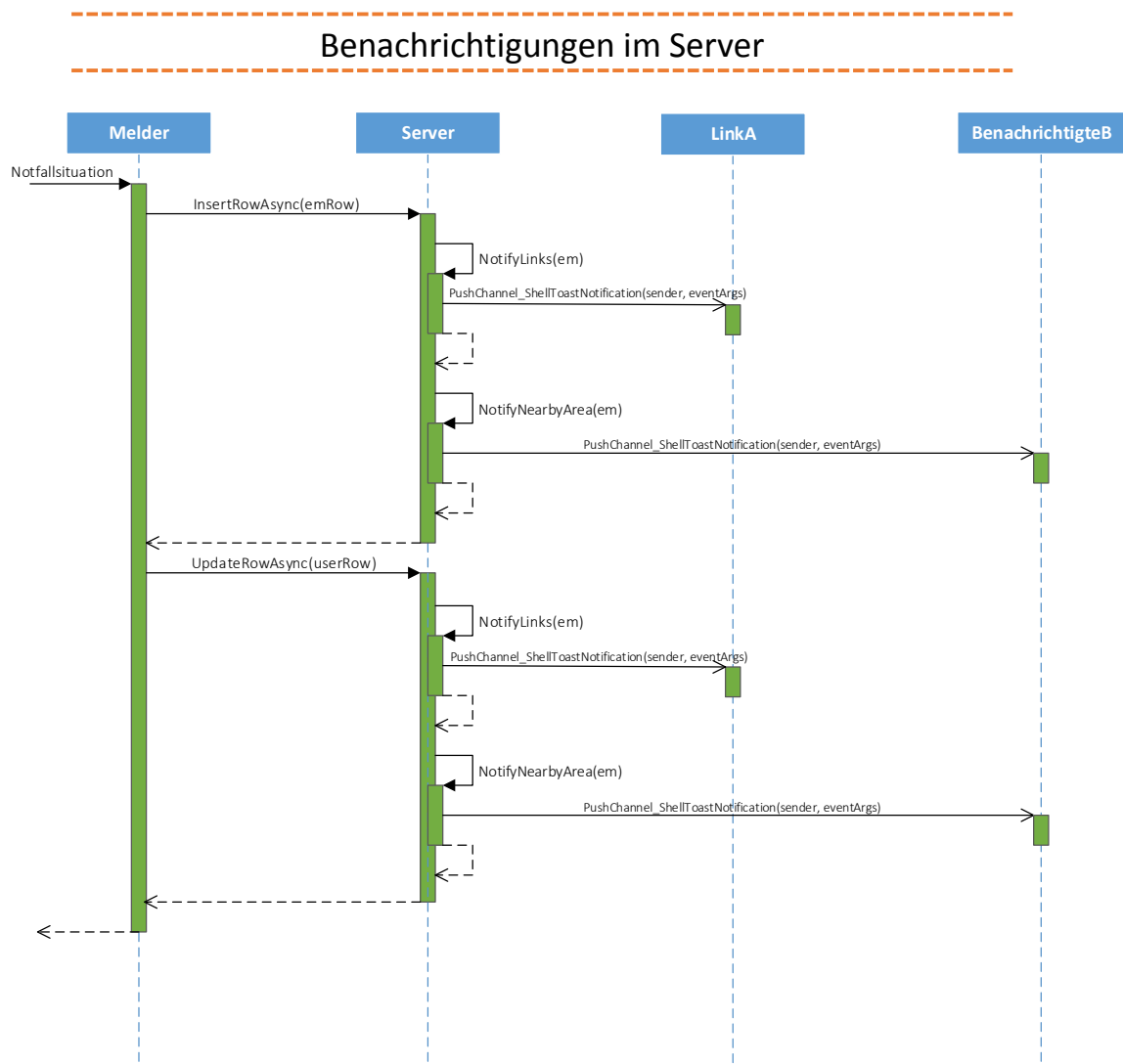


Abbildung 6.8 – Benachrichtigung durch Server

Ein neuer Notfall wird dem Server über *InsertRowAsync* übergeben. Der Server benachrichtigt nun zuerst die verknüpften Personen (*NotifyLinks*), indem er alle Verknüpfungen des Melders aus der Datenbank lädt und allen eine Push-Benachrichtigung per *PushChannel_ShellToastNotification* sendet.

Nach demselben Muster werden nun auch die Personen in der näheren Umgebung benachrichtigt (*NotifyNearbyArea*). Hierbei müssen zuerst alle relevanten Personen in der Datenbank gefunden und dann schließlich ebenso über *PushChannel_ShellToastNotification* benachrichtigt werden.

Analog funktioniert auch die Benachrichtigung der verknüpften und nahen Personen im Falle einer Änderung der Notfalldetails. Hierbei wird allerdings keine Toast Notification für die Statuszeile, sondern eine Raw Notification gesendet.

6.3. UI-Navigation

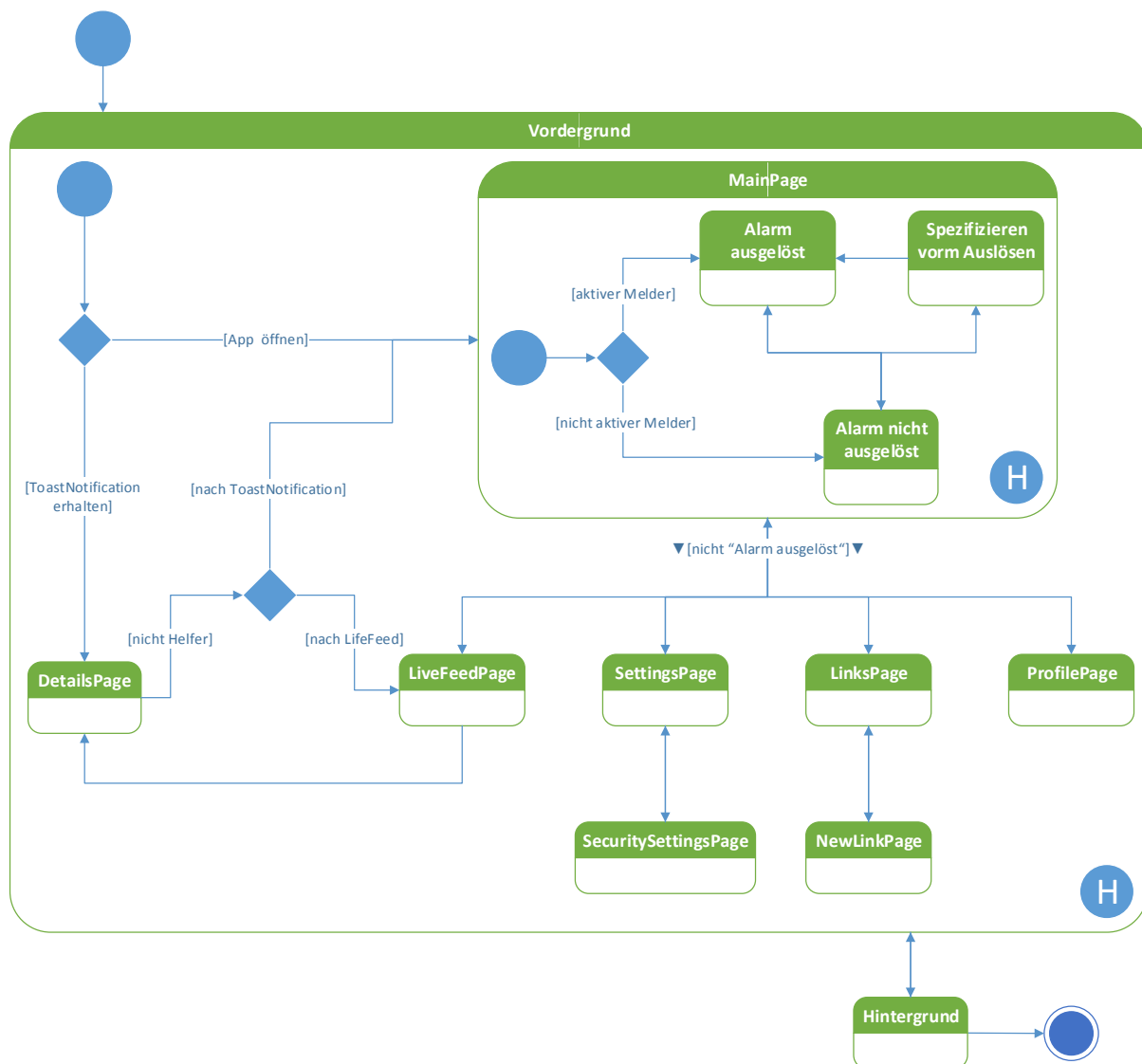


Abbildung 6.9 – Zustandsdiagramm der UI-Seiten

Es gibt zwei verschiedene Möglichkeiten die Applikation zu starten:

Öffnet der Benutzer die Applikation auf dem Startbildschirm seines Geräts, so gibt es zwei verschiedene Zustände, in die der Benutzer geraten kann. Hierbei spielt es eine Rolle, in welchem Zustand sich der Benutzer beim letzten Beenden der Applikation befunden hat. War er selber Melder eines Notfalls, der nicht von ihm selber beendet wurde, so landet er direkt auf der `MainPage(Alarm ausgelöst)`. Befand er sich in einem der anderen Zustände, so öffnet sich direkt die `MainPage (Alarm nicht ausgelöst)`.

Erhält der Nutzer eine ToastNotification über einen neuen Notfall so kann er mit dem Öffnen dieser Benachrichtigung direkt die DetailsPage des Notfalls erreichen.

Wenn der Alarm nicht ausgelöst ist, so kann der Benutzer von der MainPage zu der LiveFeedPage, der SettingsPage, der LinksPage oder der ProfilePage navigieren. Von all diesen Pages kann der Benutzer auch wieder zurück zur MainPage gelangen.

Von der `SettingsPage` kann man zur `SecuritySettingsPage` und auch wieder zurück.

Von der LinksPage lässt sich zur NewLinkPage navigieren.

Die DetailsPage kann nur verlassen werden, wenn man sich in dem zugehörigen Notfall nicht als Helfer registriert hat. Hat man zur DetailsPage zuvor durch eine **ToastNotification** gewechselt, so gelangt man beim Wähle des Zurück-Knopfes auf die MainPage(Alarm nicht ausgelöst). War man zuvor auf der LiveFeedPage, wird der Benutzer dorthin zurückgeführt.

Aus jedem Zustand kann die Applikation beendet werden, wobei gespeichert wird, ob der Benutzer Melder eines aktiven Notfalls war.

6.4. UI-Seiten

Da die Benutzerfreundlichkeit der Applikation an zentraler Stelle steht, wird für die Applikation eine einheitliche, minimalistische und Inhalte hervorhebende Design-Richtlinie verwendet.

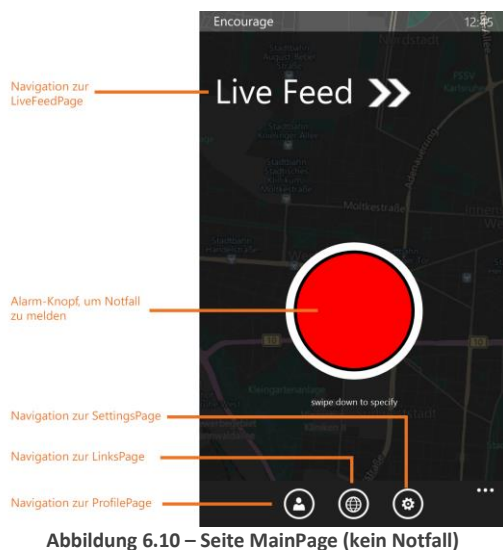


Abbildung 6.10 – Seite MainPage (kein Notfall)

Beim ersten Öffnen der mobilen Anwendung (MainPage) sieht man im Zentrum des Bildschirms einen roten Knopf und die für Windows Phone typische CommandBar am unteren Rand des Bildschirms.

Der rote Knopf dient dem Melden von Notfällen. Um einen Alarm auszulösen, muss der Benutzer den Knopf gedrückt halten, bis sich der äußere Ring gefüllt hat. Wird der Knopf zu früh losgelassen (z.B. kurzes Antippen), dann wird kein Notfall gemeldet (Dead man switch). Entscheidet sich der Benutzer dazu, den Notfall doch nicht zu melden, während sein Finger auf dem Knopf ist, kann er ihn in Richtung des Mülleimer-Symbols im oberen Bereich des Bildschirms ziehen.

Ebenso kann der Melder des Notfalls vor dem Auslösen des Alarms die Details des Notfalls spezifizieren, indem er den Knopf nach unten zieht.

Über die drei Knöpfe in der CommandBar gelangt man zu den folgenden Seiten: **ProfilePage**, **LinksPage** und **SettingsPage**.



Abbildung 6.11 – Seite MainPage (Notfall gemeldet)

Nach dem Melden eines Notfalls ist es dem Benutzer so lange nicht mehr möglich, zu anderen Seiten zu navigieren, bis der Notfall beendet wurde. Zum Beenden muss er auf den „Notfall Beenden“-Knopf drücken, welcher allerdings durch die eingestellte Sicherheitsfrage die Identität des Benutzers überprüft. Wurde die Sicherheitsfrage richtig beantwortet, so kehrt die Applikation auf die MainPage im „Alarm nicht ausgelöst“-Zustand zurück.

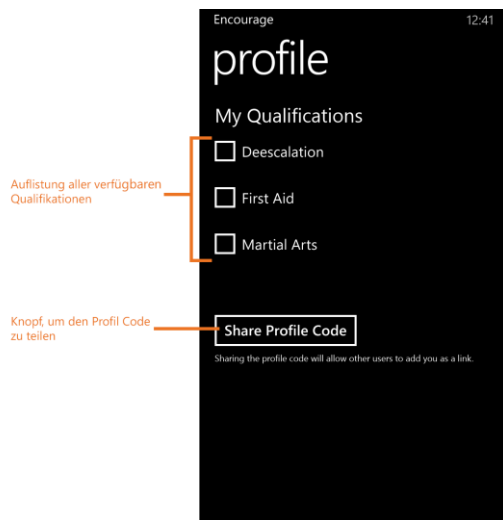


Abbildung 6.12 – Seite ProfilePage

Auf der Seite **ProfilePage** kann der Benutzer seine Qualifikationen angeben und seinen Profilcode (als QR-Code oder Text-Link) teilen, um von Anderen als verknüpfte Person hinzugefügt zu werden.



Abbildung 6.13 – Seite LinksPage

Die Seite **LinksPage** bietet dem Nutzer eine einfache Möglichkeit, die verknüpften Personen zu überblicken, zu bearbeiten oder zu löschen. Über die CommandBar am unteren Rand des Bildschirms ist es möglich, zur **AddLinkPage** zu navigieren, auf der der Benutzer eine neue Person verknüpfen kann.

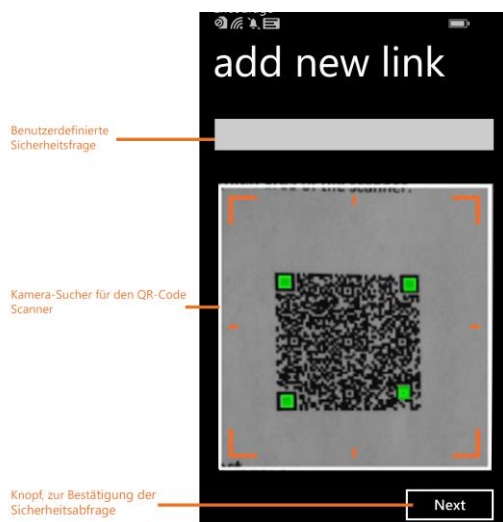


Abbildung 6.14 – Seite NewLinkPage

Auf der Seite **NewLinkPage** sieht der Benutzer eine TextBox und eine Anzeige mit dem Bild der Rückkamera. Den Profilcode kann dann entweder manuell in die TextBox eingeben werden oder mit Hilfe der Kamera als QR-Code gescannt werden. Nachdem der Profilcode eingetragen wurde, kann mit dem Klicken auf den „Weiter“-Knopf das zum Profilcode gehörige Gerät hinzugefügt werden.

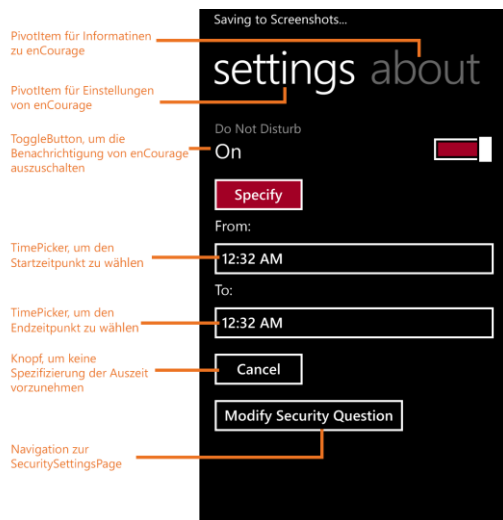


Abbildung 6.15 – Seite SettingsPage

Auf der Seite [SettingsPage](#) kann der Benutzer einstellen, in welchem Zeitraum er die App in einen Ruhemodus schalten will. Diese Einstellung kann entweder dauerhaft (an/aus) oder durch für ein wiederkehrendes Zeitintervall eingestellt werden.

Zusätzlich kann der Benutzer über einen Knopf zur Seite [SecuritySettingsPage](#) navigieren, auf der er die Sicherheitsfrage und -Antwort einstellen kann.



Abbildung 6.16 – Seite SecuritySettingsPage

Wurde bereits eine Sicherheitsfrage mit Antwort festgelegt, muss zum Bearbeiten zunächst die Identität des Benutzers verifiziert werden. Durch das Tippen auf den Weiter-Knopf kann der Benutzer dann eine neue Sicherheitsfrage und -Antwort einstellen. Wurde bisher noch keine Sicherheitsfrage festgelegt, so kann dieser Vorgang ohne Verifizierung durchgeführt werden.

Die Antwort muss beim neuen Festlegen zweimal eingegeben werden, um Tippfehler zu vermeiden. Durch das Tippen auf den Bestätigen-Knopf wird die neue Frage und Antwort gespeichert und eine automatische Rück-Navigation zur Seite [SettingsPage](#) erfolgt.

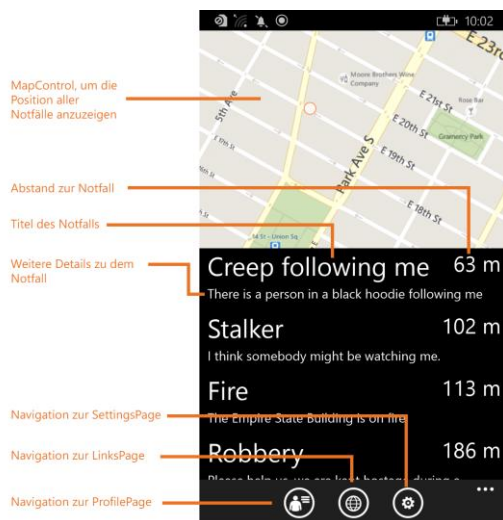


Abbildung 6.17 – Seite LiveFeedPage

Auf der Seite [MainPage](#) kann der Benutzer über einen Knopf zur Seite [LiveFeedPage](#) navigieren. Auf dieser Seite wird auf dem oberen Teil des Bildschirms eine Karte angezeigt, worauf die umliegenden Notfälle markiert sind. Unterhalb der Karte befindet sich außerdem eine Liste all dieser Notfälle und einiger zugehöriger Informationen. Durch ein Klicken auf einen dieser Notfälle gelangt man zur Seite [DetailsPage](#).

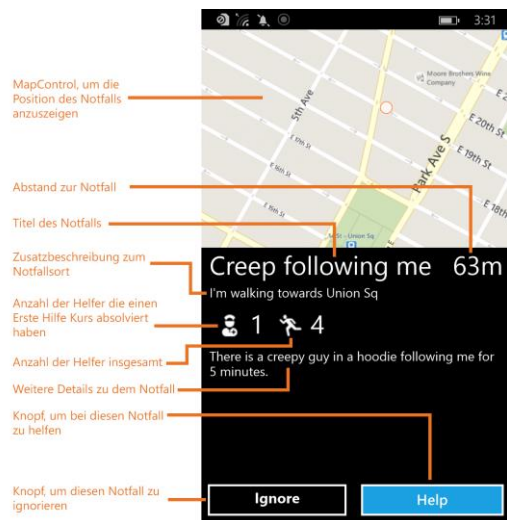


Abbildung 6.18 – Seite DetailsPage

Auf der Seite [DetailsPage](#) wird ebenfalls eine Karte angezeigt, die auf die Position des Notfalls zentriert ist. Unterhalb der Karte werden der Titel, die Anzahl der Helfer, die Anzahl der Qualifizierten, die zusätzliche Umgebungsbeschreibung, und weitere Details des Notfalls dargestellt. Am unteren Rand des Bildschirms sind zwei Knöpfe sichtbar.

Durch das Drücken auf den linken Knopf wird der angezeigte Notfall ignoriert. Durch das Tippen auf den rechten Knopf hingegen erklärt sich der Benutzer bereit, zur Hilfe zu eilen, was anderen Informierten angezeigt wird.

7. Klassendiagramm

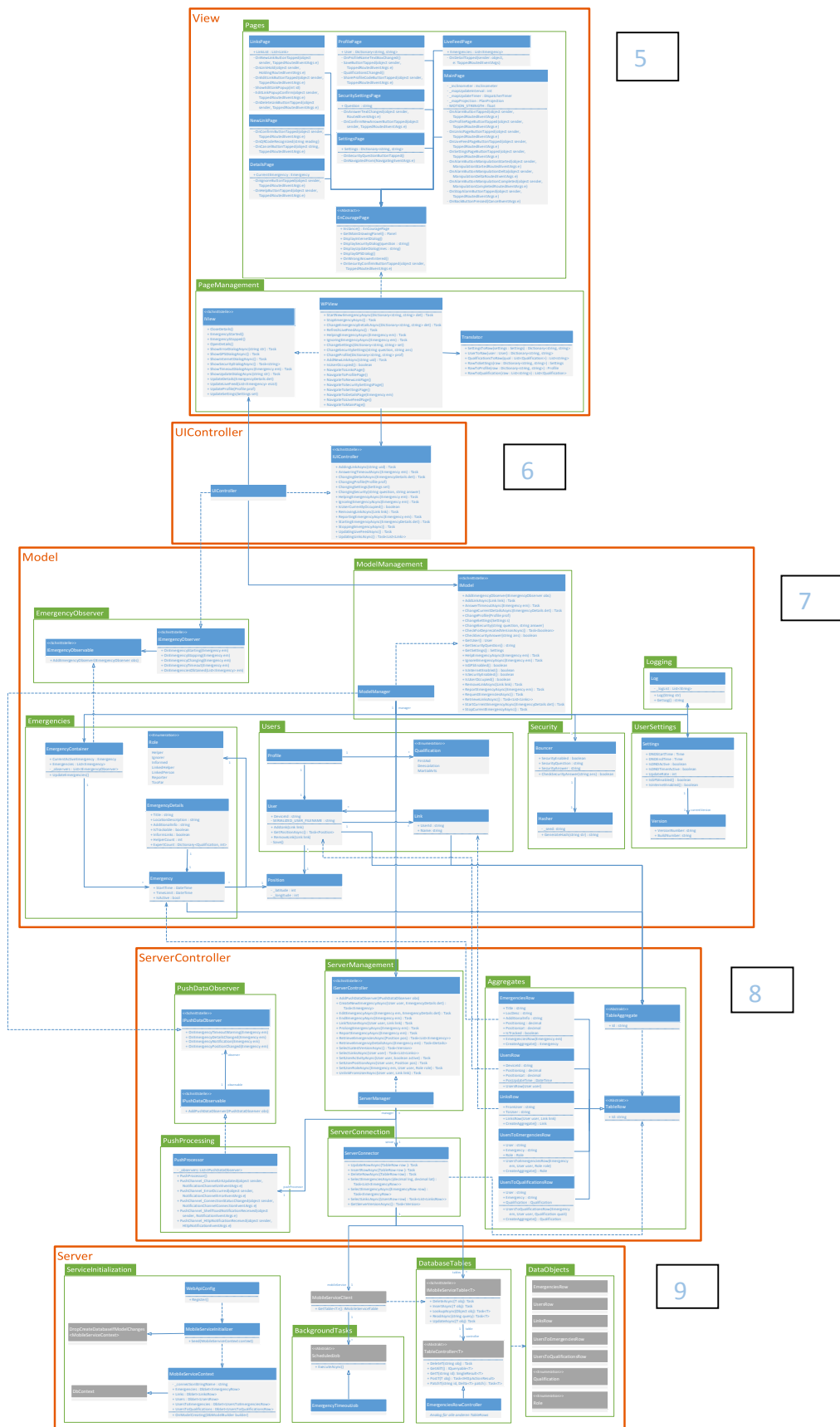


Abbildung 7.1 – System-Klassendiagramm

8. Anhang

8.1. Abbildungsverzeichnis

Abbildung 2.1 – Server-Architektur	4
Abbildung 2.2 – Klassen des Views	5
Abbildung 2.3 – Klassen des UIControllers	6
Abbildung 2.4 – Klassen des Models.....	7
Abbildung 2.5 – Klassen des ServerControllers	8
Abbildung 2.6 – Klassen des Servers.....	9
Abbildung 2.7 – Subsystem-Schnittstelle des Views.....	10
Abbildung 2.8 – Subsystem-Schnittstelle des UIControllers.....	11
Abbildung 2.9 – Subsystem-Schnittstelle des Models	13
Abbildung 2.10 – Subsystem-Schnittstelle des ServerControllers.....	15
Abbildung 3.1 – Pakete des Subsystems View	17
Abbildung 3.2 – Pakete des Subsystems UIController	18
Abbildung 3.3 – Pakete des Subsystems Model	19
Abbildung 3.4 – Pakete des Subsystems ServerController	21
Abbildung 3.5 – Pakete des Subsystems Server	23
Abbildung 3.6 – Benutzrelation des Systems	25
Abbildung 3.7 – Vorlage eines Hintergrundauftrags	26
Abbildung 3.8 – Vorlage einer Klasse zum Überwachen der Position	26
Abbildung 4.1 – Paket PageManagement.....	28
Abbildung 4.2 – Paket Pages.....	30
Abbildung 4.3 – Paket UIController	35
Abbildung 4.4 – Paket ModelManagement.....	37
Abbildung 4.5 – Paket Emergencies.....	39
Abbildung 4.6 – Paket EmergencyObserver	42
Abbildung 4.7 – Paket Users	43
Abbildung 4.8 – Paket Security	45
Abbildung 4.9 – Paket UserSettings.....	46
Abbildung 4.10 – Paket Logging.....	47
Abbildung 4.11 – Paket ServerManagement	48
Abbildung 4.12 – Paket PushDataObserver	50
Abbildung 4.13 – Paket PushProcessing	51
Abbildung 4.14 – Paket ServerConnection	52
Abbildung 4.15 – Paket Aggregates	54
Abbildung 4.16 – Paket ServiceInitialization.....	58
Abbildung 4.17 – Paket BackgroundTasks	60
Abbildung 4.18 – Paket DatabaseTables.....	61
Abbildung 5.1 – Versionen der Aggregate	64
Abbildung 5.2 – Aufbau der Datenbank.....	65
Abbildung 6.1 – Notfall wird gemeldet.....	67
Abbildung 6.2 – Notfall wird beendet.....	68
Abbildung 6.3 – Benachrichtigung wird erhalten	69
Abbildung 6.4 – Notfalldetails werden aktualisiert	70

Abbildung 6.5 – LiveFeed wird aktualisiert.....	71
Abbildung 6.6 – Ruhemodus wird eingeschaltet	71
Abbildung 6.7 – Ablauf eines Notfallalarms	72
Abbildung 6.8 – Benachrichtigung durch Server	73
Abbildung 6.9 – Zustandsdiagramm der UI-Seiten	74
Abbildung 6.10 – Seite MainPage (kein Notfall)	75
Abbildung 6.11 – Seite MainPage (Notfall gemeldet).....	75
Abbildung 6.12 – Seite ProfilePage.....	76
Abbildung 6.13 – Seite LinksPage	76
Abbildung 6.14 – Seite NewLinkPage	76
Abbildung 6.15 – Seite SettingsPage	77
Abbildung 6.16 – Seite SecuritySettingsPage	77
Abbildung 6.17 – Seite LiveFeedPage	77
Abbildung 6.18 – Seite DetailsPage	78
Abbildung 7.1 – System-Klassendiagramm.....	79

8.2. Klassenverzeichnis

Klasse 1 - IView	28
Klasse 2 - WPView	29
Klasse 3 - Translator	29
Klasse 4 - EnCouragePage	31
Klasse 5 - MainPage	32
Klasse 6 - LiveFeedPage	33
Klasse 7 - DetailsPage	33
Klasse 8 - SettingsPage	33
Klasse 9 - SecuritySettingsPage	34
Klasse 10 - LinksPage	34
Klasse 11 - NewLinkPage	34
Klasse 12 - ProfilePage	35
Klasse 13 - IUIController	36
Klasse 14 - UIController	36
Klasse 15 - IModel	37
Klasse 16 - ModelManager	38
Klasse 17 - EmergencyContainer	40
Klasse 18 - Emergency	40
Klasse 19 - EmergencyDetails	41
Klasse 20 - Role	41
Klasse 22 - IEmergencyObservable	42
Klasse 23 - IEmergencyObserver	42
Klasse 24 - User	43
Klasse 25 - Link	44
Klasse 26 - Qualification	44
Klasse 27 - Profile	44
Klasse 28 - Bouncer	45
Klasse 29 - Hasher	46
Klasse 30 - Settings	47
Klasse 31 - Version	47
Klasse 32 - Log	48
Klasse 33 - IServerController	49
Klasse 34 - ServerManager	49
Klasse 35 - IPushDataObserver	50
Klasse 36 - IPushDataObservable	51
Klasse 37 - PushProcessor	51
Klasse 38 - ServerConnector	53
Klasse 39 - TableRow	55
Klasse 40 - EmergenciesRow	55
Klasse 41 - UsersRow	56
Klasse 42 - LinksRow	56
Klasse 43 - UsersToEmergenciesRow	57
Klasse 44 - UsersToQualificationsRow	57
Klasse 45 - TableAggregate	58
Klasse 46 - WebApiConfig	59

Klasse 47 - MobileServiceInitializer	59
Klasse 48 - MobileServiceContext.....	59
Klasse 49 - EmergencyTimeoutJob	60
Klasse 50 – TableController<T>.....	62

8.3. Glossar

AKTIVE PERSON <i>Active User</i>	Person, die die Applikation momentan nicht im Ruhemodus hat.
ALARM <i>Alarm</i>	Funktion eines Notfalls, die Personen benachrichtigt und auf dem LiveFeed erscheint.
BENACHRICHTIGUNG <i>Notification</i>	Statusleistennachricht/Notification durch die Applikation.
BENUTZER <i>User</i>	Person, die die Applikation geöffnet hat und momentan verwendet.
CLIENT <i>Client</i>	Gerät des Benutzers, auf dem die Applikation installiert ist.
DETAILS <i>Details</i>	Alle Eigenschaften eines Notfalls, die ein Informierter einsehen kann. Gebündelt verfügbar in der Detailansicht eines Notfalls.
EXPERTE <i>Expert</i>	Person, die in ihrem Profil besondere Qualifikationen angegeben hat.
GRÖßERE UMGEBUNG <i>Larger Area</i>	Alle räumlichen Punkte in einem festgelegten, größeren Radius.
HELFENDE <i>Responder</i>	Informierte, die in der Applikation ausgewählt haben, dass sie auf dem Weg sind.
INFORMIERTE <i>Informed User</i>	Melder oder benachrichtigte Personen, die den Notfall nicht ignoriert haben.
LIVEFEED <i>LiveFeed</i>	Karte/Liste mit allen aktuellen Notfällen
MELDER <i>Reporter</i>	Person, die den Notfall als Erste gemeldet hat.
NÄHERE UMGEBUNG <i>Nearby Area</i>	Alle räumlichen Punkte in einem festgelegten, kleineren Radius.
NOTFALL <i>Emergency</i>	Vorfall, bei dem Menschen, Tiere oder Eigentum ohne menschliches Eingreifen Schaden nehmen können.
PERSON <i>Person</i>	Mensch, der die Applikation auf seinem Smartphone installiert hat.
POP-UP <i>Pop-Up</i>	Dialog auf der grafischen Benutzeroberfläche, der sich (teilweise) über einen anderen Bildschirm legt und Informationen darstellt. Kann geschlossen werden.
PROFIL <i>Profile</i>	Auswahlbildschirm für gerätespezifische Informationen, die weitere Details über den Benutzer preisgeben.
RUHEMODUS <i>Do Not Disturb Mode</i>	Modus der Applikation, in der alle Benachrichtigungen und Standortdaten deaktiviert sind.
SPEZIFIKATION <i>Specification</i>	Die Angabe und Änderung der Details eines Notfalls durch den Melder.
VERKNÜPFT PERSONEN <i>Linked Person</i>	Personen, die freiwillig mit dem Gerät des Benutzers verknüpft sind, um priorisiert behandelt zu werden.