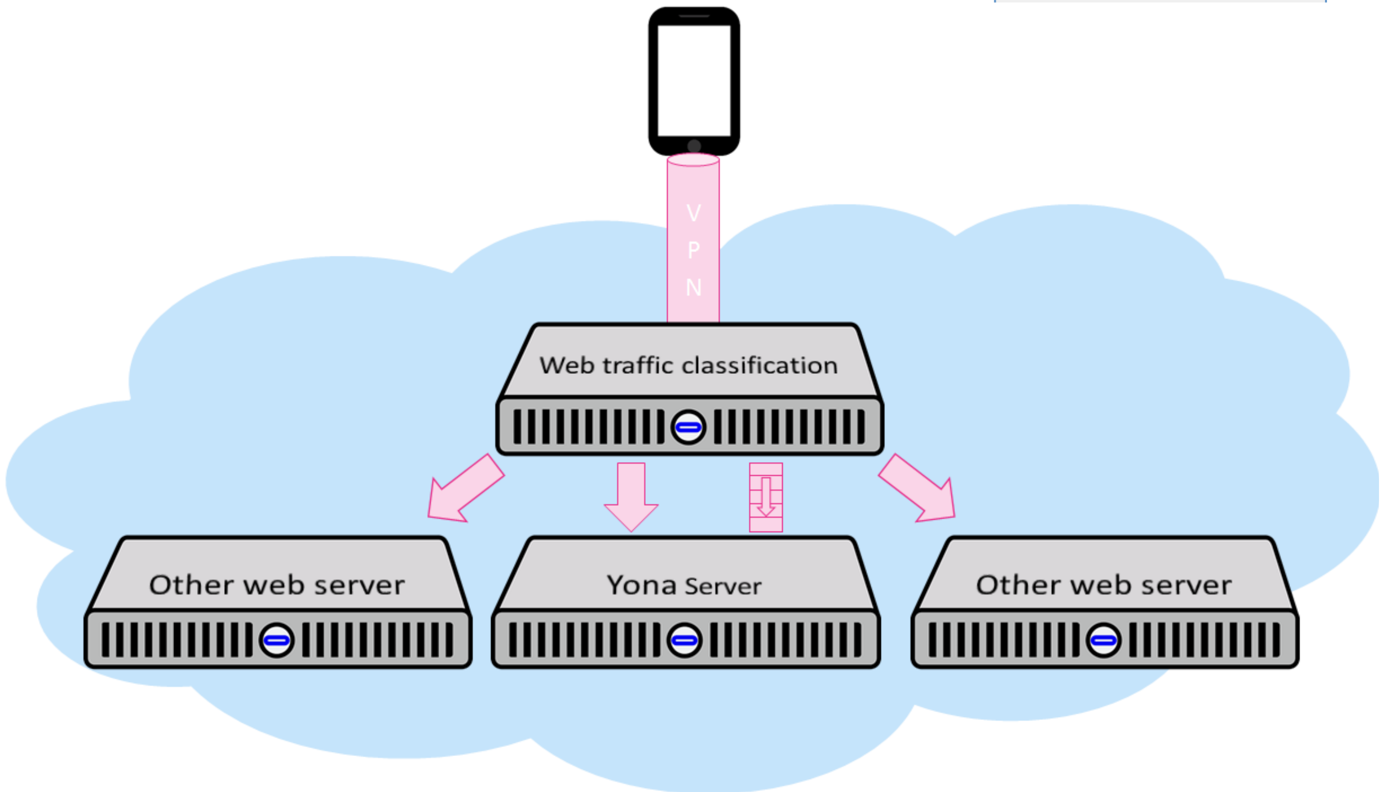
Architecture

This page describes the architecture of Yona.

Table of contents

High level overview

On network level, this is how it looks like:



Initial support is limited to mobile devices. These devices will access the internet through a VPN tunnel that is terminated on the web traffic classification server (SmoothWall). The server will not filter, in the sense of blocking sites, but just classify the sites. This classification will happen for HTTP and HTTPS traffic, see this page. The requests will be passed on to the target web server. For requests that in categories of interest to Yona, the classification server will post a message on a queue to the Yona server. The Yona server is responsible for maintaining the user and buddy administration and to inform users about events that conflict the goals defined by the users. The Yona app on the device interacts with the Yona server to see buddy events and everything supported by Yona.

Subsequent sections zoom in on the app, the web traffic classification server and the Yona server.

Mobile App

The mobile app is the only user interface provided for Yona. There are no plans to provide a web site. The mobile app has the following high level responsibilities:

1. Provide the user interface for all features provided to Yona users
2. Configure the VPN connection
3. Reconnect the VPN in case it disconnects
4. Send a request to the Yona server to inform the buddy in case:
    1. The user uninstalls
    2. The user disables the app
    3. The user disables or otherwise hampers the VPN connection

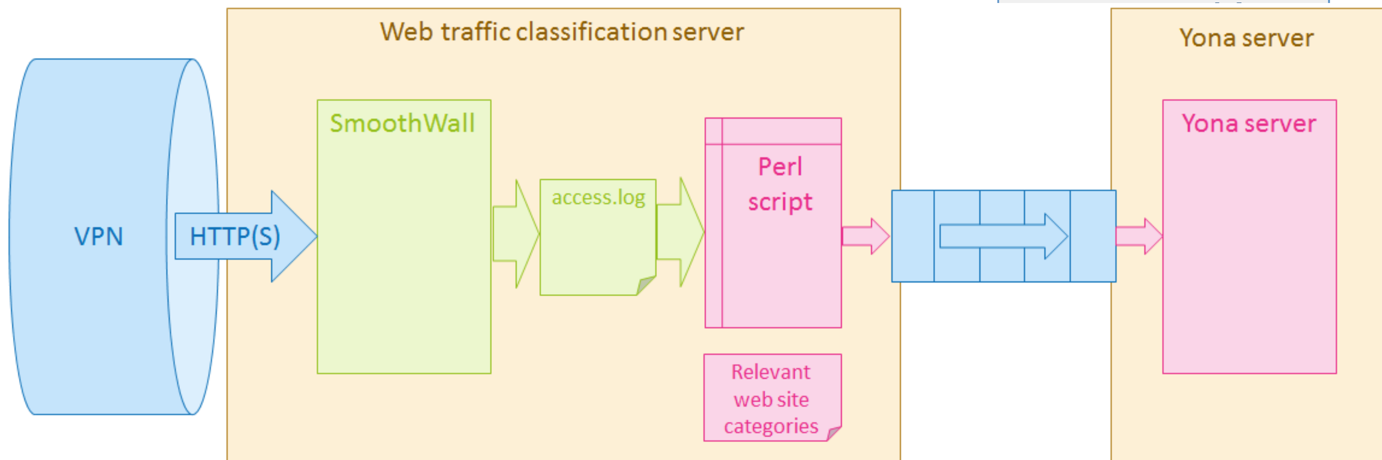
The app generates and stores a key in the secret storage of the app, that is required for all interactions with the server.

Web traffic classification server

The heart of this server is SmoothWall. This product is commonly used for web filtering. For Yona, it will be deployed as a classification server. The SmoothWall server provides us with the following features:

1. VPN server
2. Classification engine
3. Man-in-the-middle proxy for HTTPS servers. See this page.

Every request passed through the filter engine is logged in the DansGuardian log. Given the constraints of the Linux OS of the SmoothWall server, we will use a Perl script to filter the events that are of relevance to Yona and post these on the queue to the Yona server. The log file (`access.log`) is normally written in a file. Instead of the file, we will create a named pipe and have the Perl script read from it. See an example log file parser here. The named pipe approach is chosen for two reasons: it is a very efficient mechanism that does not required disk reads/writes and it prevents from storing sensitive data.



The Perl script will read all SmoothWall log events and match them with a list of Yona-relevant web site categories. If the category is relevant, a message with the relevant data is posted on the queue to the Yona server. Otherwise, the message is discarded.

Yona server

The Yona server is responsible for:

- Providing the web services that back the mobile app
- Administration of users and buddy relationship
- Producing notifications in case users access websites that conflict their objectives

Security model

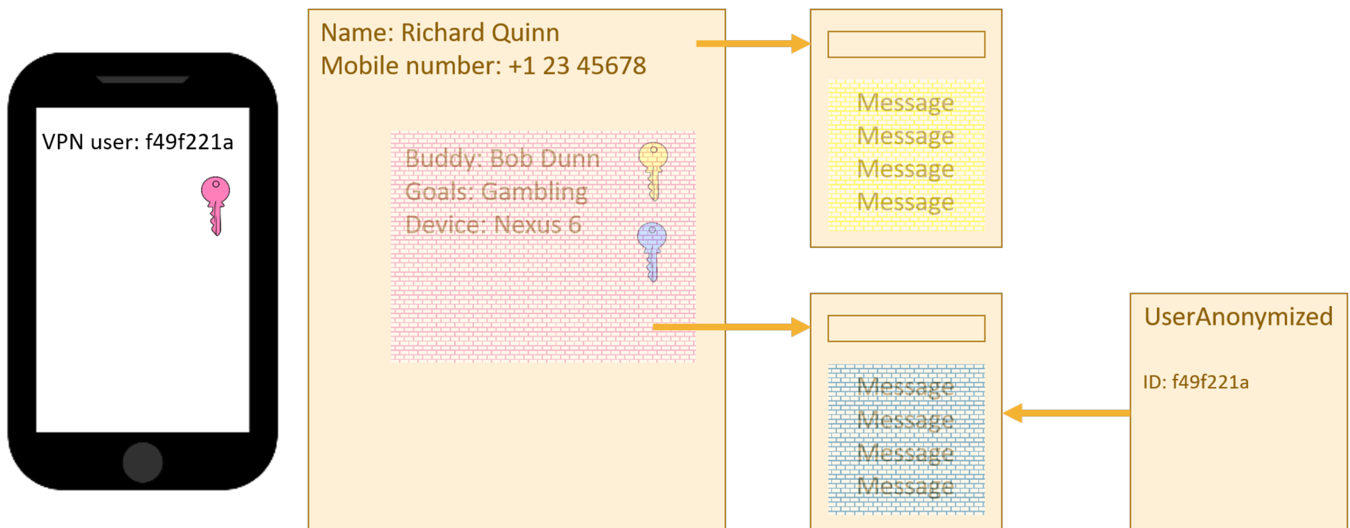
Given the objective of Yona to help people that "do the very thing they hate" (Romans 7:15), the system by nature stores sensitive information, so security is a prime concern. For an overview of the various design alternatives considered, consult the page Flow - Register a goal conflict. These are the design goals:

- Private information about a user (buddies, goals, devices, messages from buddies, messages about goal conflicts, etc.) should be inaccessible to everyone, including those that have administrative access to the systems
- It should be impossible to build a quantitative understanding of how well a user acts against his own goals, even when having administrative access to the systems

- High level overview
- Mobile App
- Web traffic classification server
- Yona server
  - Security model
  - Layered architecture
  - Domain
      and buddy messages
    - User with buddies and activities
    - Time
- Cluster architecture

The server design uses a combination of public-key encryption and symmetric-key encryption to accomplish these goals. Symmetric-key encryption is used to encrypt all private information of the user: buddies, goals, devices, etc. Public-key encryption is used to implement a secure messaging system to deliver messages from buddies and messages from the web traffic classification. Every user has two "message boxes", comparable to a classic mailbox with lock:

Everyone can drop a message in the box, but you need the key to take a message out of it. Every user has two of these. One is publicly linked to the user account, so it's technically possible to send a message to a user if their identity is known. The other message box is linked indirectly and used for messages related to web traffic. When a user is provisioned, a VPN account is created for them. The user name of this VPN account is a UUID (named accessor ID) that cannot be linked to a user, Through their private key, the user knows what their accessor ID. but without that private key, that link cannot be established. This can be depicted as follows:



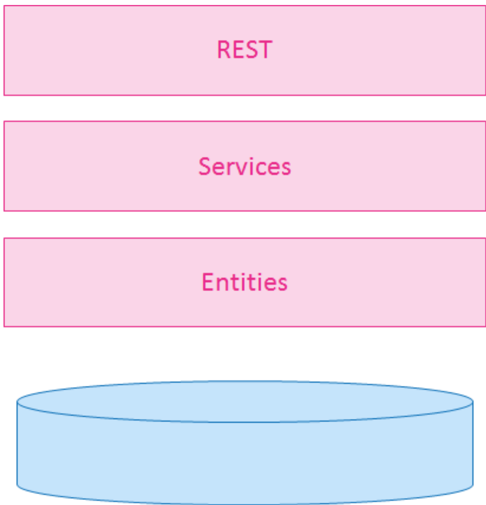
The server maintains a little bit of unencrypted user information: their name, mobile number and a reference to their public message box. The other information is encrypted with the key that is stored on in the secure storage of the device (e.g. Android non-exported content provider). This encrypted data includes the buddy relationships, the defined goals and two private keys to access the two message boxes. Writing into the message boxes can be done through the public key (not depicted, to prevent confusion). For reading, the private key is required, which is part of the encrypted data of the user. The message box for direct messages is linked from the public data of the user, so given a user, it is possible to find the direct message box. The sole purpose of the direct message box is to be able to send an buddy connect request to a user.

The anonymous message box is linked from the private data of the user, so only the user "knows" their anonymous message box. Besides this, the anonymous message box is also linked to the so-called user-anonymized entity, which carries the same ID as the VPN user name. Thus the analysis engine (see below) knows where to post messages when a goal conflict is detected for a given VPN account. Once the buddy relationship is established, all inter-buddy messages go through the anonymous message box.

See Encryption approach for a more indepth description.

Layered architecture

The Yona server has a simple layered architecture:



- **REST**. This layer exposes the RESTful API used by the mobile app. Follows HATEOAS and HAL (see primer here) to provide an efficient and convenient RESTful API. This layer is implemented through Spring HATEOAS.
- **Services**. The services are independent of the access technology (today REST, yesterday SOAP) and enable creating users, finding goals, etc. Services define the transaction boundaries and can depend on one another to fulfill their task. The services layer take Data Transfer Objects (DTOs) as inputs and outputs. These DTOs are annotated with Jackson annotations, so the REST layer can directly return the DTOs. The services layer is implemented with the Spring Framework.
- **Entities**. The entities layer hosts the domain model and the repositories. The implementation technology is JPA and Spring Data.

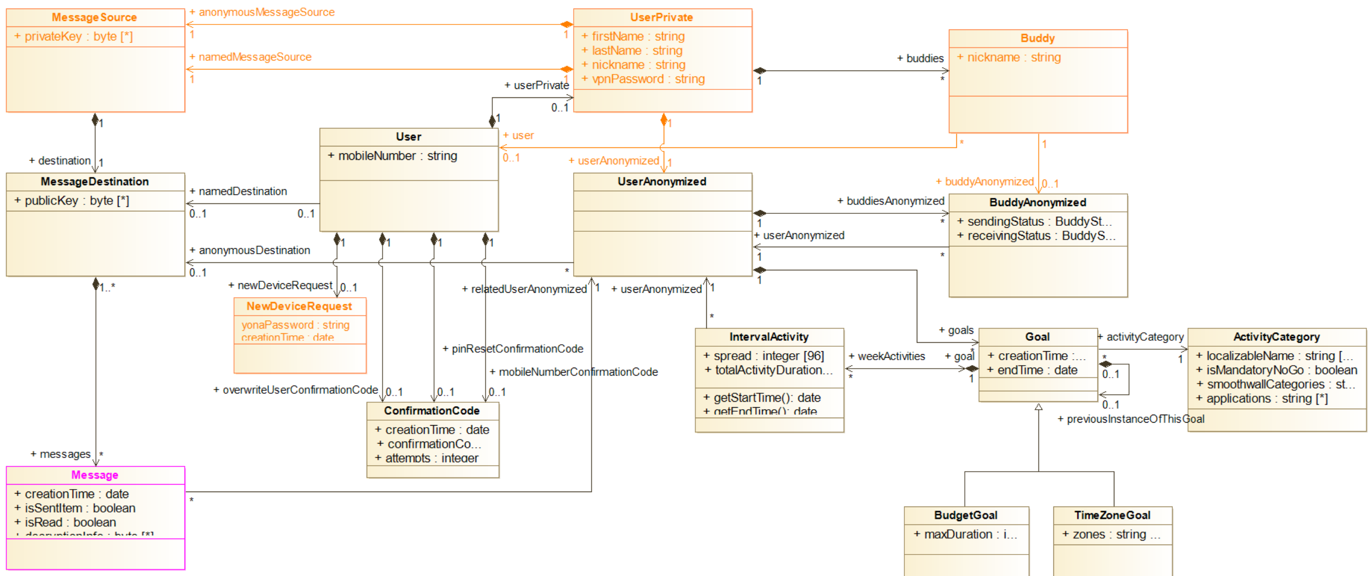
Domain model

To make the domain model somewhat comprehensible, it is described based on multiple class diagrams that cover subsets of the model. All diagrams use the same graphical convention:

- Orange: encrypted with the Yona password stored on the device
- Magenta: encrypted through public/private key encryption
- Black: not encrypted. This holds anonymous information that cannot be related to identified users, but users can find the data that belongs to them, by decrypting the associations. The exception to this is the `User` entity. That is not encrypted but contains identifieable information: the user name and the e-mail address. This is not encrypted because users need to be able to send a buddy connect request to other users. This information is not exposed through the web APIs. If the system gets hacked, an intruder would be able to fetch a list of names and phone numbers. This is not considered sensitive, as that is also available from public directories.

User with buddies and goals

The below diagram shows the heart of the domain model: users with buddies and goals.



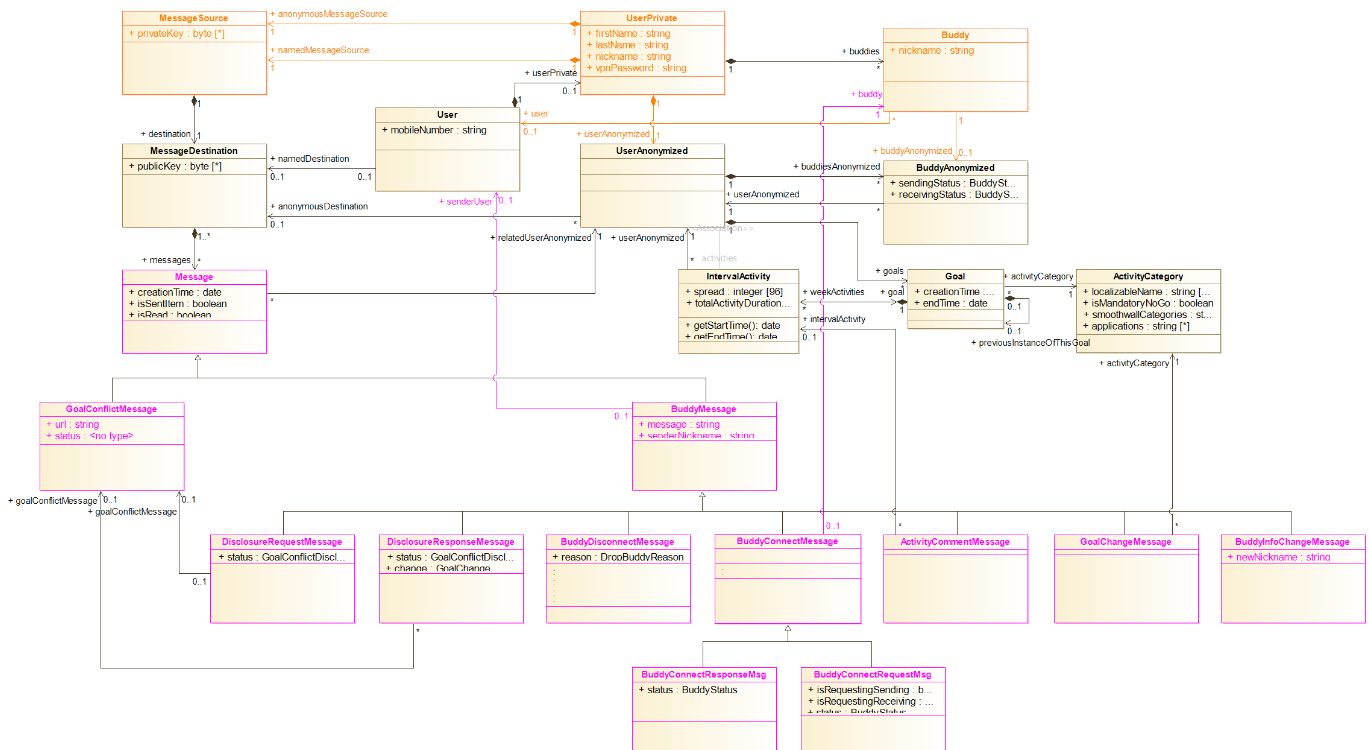
The orange texts and associations represent data that is encrypted with the key stored inside the app. The magenta texts and associations represent message related data that is encrypted through the private key of the message box containing them.

Here is a short description of every class, in a somewhat logical order:

1. **`User`**. This is the heart of the domain model. It contains the so-called public information of the user. It's called public because it can be accessed without the key of that user. The linked `MessageDestination` represents the user's direct message box. The user owns a `UserPrivate` instance containing the private user data. This is optional. In case a user sends a buddy request to a person that is not known in the system yet, then only the `User` object will already be created, so the connect request message can be stored for that user and the buddy-to-be can be linked to the requesting user.
2. **`UserPrivate`**. This is the second piece of user data. Everything in this object is encrypted with the password/key stored in the app. It compositely owns the two message boxes, the buddies of this user and the anonymized user data.
3. **`UserAnonymized`**. This is the third piece of user data. This information is not encrypted and thus accessible to the analysis engine. The information in this object and the ones associated with it cannot be related to a user without the password/key stored in the app. The ID of this object is also the login ID of the user on the VPN. It maintains associations with: the anonymous destination of this user, their anonymized buddy information and their goals.
4. **`Buddy`**. An object of this class represents another user as buddy for the user owning this instance. I.e. if Richard Quinn requests Bob Dunn to become his buddy, then Richard will own a `Buddy` instance representing Bob to him. If Bob accepts that request, then he will own another `Buddy` object representing Richard to him. Richard's `Buddy` instance, actually the `BuddyAnonymized` object owned by it, (representing Bob) will have the `sendingStatus` as `Accepted` and `receivingStatus` as `NotRequested`. On Bob's side, the sending and receiving status are reversed. This implies that Bob will receive messages when Richard does things conflicting with his goals, while Richard does not receive such messages for Bob. In general usage patterns, the buddy relationship is symmetrical, but that that is implemented in the app: that will automatically request the revers relationship when a user accepts a buddy request. The `Buddy` object has associations with: the `User` of the buddy and their goals. Besides this, it compositely owns a `BuddyAnonimized` instance.
5. **`BuddyAnonymized`**. This is the other half of buddy information. This information is not encypted and thus accessible to the analysis engine. The information in this object and the ones associated with it cannot be related to a user without the password/key stored in the app. It contains the status (whether information is to be sent to/received from) this buddy and it maintains an association with the anonymized user.
6. **`ActivityCategory.`** A type of activity (e.g. Multimedia, Gaming, Adult content) for which the user can define a goal.
7. **`Goal`**. A goal defined by the user, specialized as:
    1. **`BudgetGoal`**. the user has assigned themselves a certain amount of time for this activity category.
    2. **`TimeZoneGoal`**. The user has defined one or more blocks of time in which the want to do this category of activities.
8. **`MessageSource`**. The message box mentioned in the text above is implemented in two classes: `MessageSource` and `MessageDestination`. The `MessageSource` allows fetching messages while decrypting them with the embedded private key. It holds a `MessageDestination`.
9. **`MessageDestination`**. A `MessageDestination` allows sending message to that message box, while encrypting them with the embedded public key. It holds a collection of `Message` instances.
10. **`Message`**. This is the base class for all messages. It holds an (encrypted) association with the related anonymized user and provides the encryption API.
11. **`NewDeviceRequest`**. An instance of this type is created when the user wants to add another device to their account. This object temporarily holds the Yona password of the user (which is normally never stored on the server), encrypted with a temporary password.
12. **`ConfirmationCode`**. Several user actions require a confirmation code, sent by SMS.
13. **`IntervalActivity`**. Base type for day and week activities. See below for more information.

Users with buddies and buddy messages

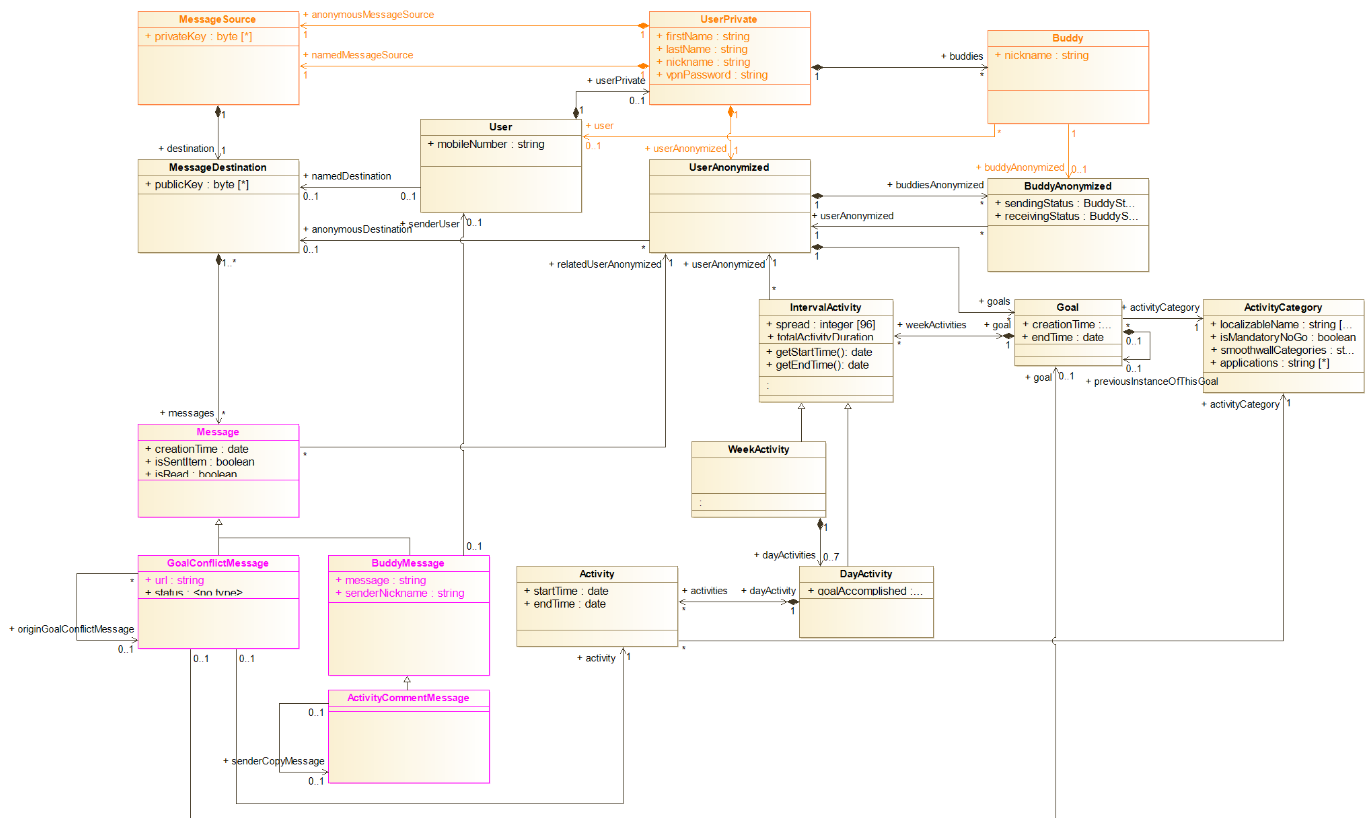
The domain model contains a considerable set of different message types:



1. **`GoalConflictMessage`**. The analysis engine issues a message of this type when it detects that a user visited a web site that conflicts with the goals set for that user. This message has an association with the conflicted goal.
2. **`BuddyMessage`**. Base type for all messages sent by a buddy. It contains an association to the user sending the message, a textual message (currently not used by the app) and the nickname of the user sending the message.
3. **`DisclosureRequestMessage`**. Sent when a buddy requests disclosure of the URL causing a goal conflict message.
4. **`DisclosureResponseMessage`**. Sent as response to the disclosure request message, indicating whether or not the buddy is allowed to see the URL.
5. **`BuddyDisconnectMessage`**. Sent when a user wants to break the buddy relationship. The reason to break the relationship (disconnect or remove user account) is part of the message.
6. **`BuddyConnectMessage`**. This is the base class for the buddy connect request and response messages. It holds an association to the involved buddy object.
7. **`BuddyConnectRequestMessage`**. If a user requests someone to become their buddy, this message is added to the direct messages of that user. It contains a set of data that is private to the requesting user, which is now shared to the buddy-to-be. The data in the base class currently suffices.
8. **`BuddyConnectResponseMessage`**. If the user accepts a buddy request, a message of this type is returned to the requestor. It carries information that is private to the new buddy, as well as information that is required to process the connect response correctly. The data in the base class currently suffices.
9. **`BuddyInfoChangeMessage`**. Sent to all buddies when a user changes their personal data. The new nickname is carried inside the message. The other information is already available to the user.
10. **`GoalChangeMessage`**. Sent to all buddies when a user changes their goals (add new one, or change or delete existing one). It references the activity category of the goal being added/changed/deleted.
11. **`ActivityCommentMessage`**. Users can comment on the activities of their buddies (see next section).

User with buddies and activities

A key objective of the Yona application is to keep track of the time users spend on various activities. The below diagram focuses on that.



1. **`IntervalActivity`**. The base type for activities aggregated by day or week.
2. **`WeekActivity`**. Activities aggregated by week. The values in the base type apply to the entire week. This class acts as composite parent for the day activities.
3. **`DayActivity`**. Activities aggregated by day. The values in the base type apply to this particular day. This class acts as composite parent for the individual activities.
4. **`Activity`**. An instance of this class denotes a particular activity (e.g. YouTube from 14:05 till 14:37).

These classes capture the information that is returned through the activity reports. More on that is available on the page Design change to capture app activity. That page also provides background information on the evolution of the design.

Time

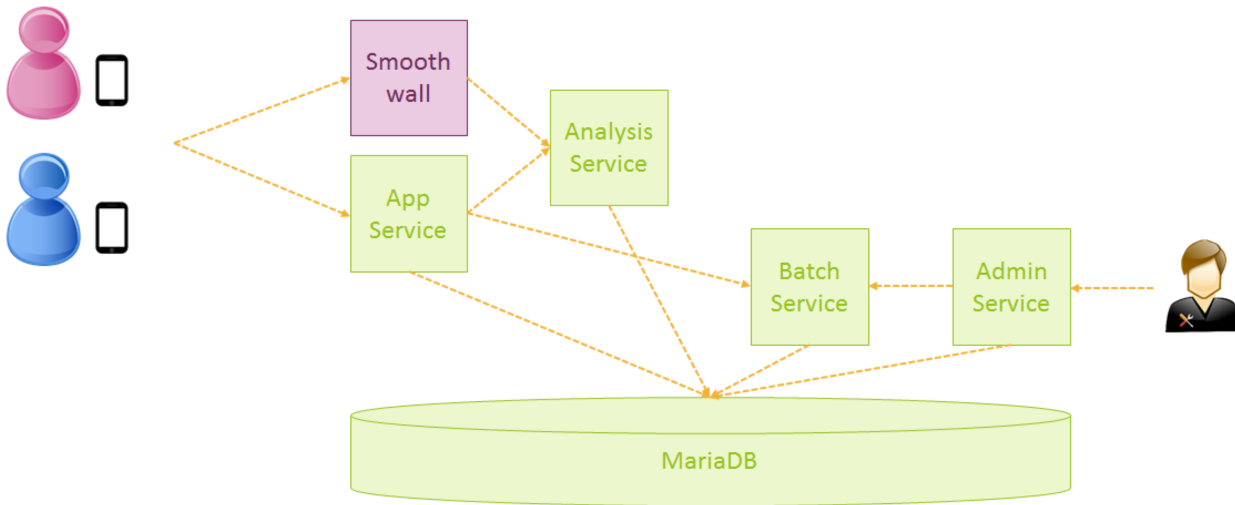
Various entities carry type stamps. A goal for instance has a creation time and an optional end time, an activity has a start and end time, etc. The convention is that all times are in UTC, except for activities. The design aims to support users that travel through different time zones. To enable that, the entities `IntervalActivity` and `Activity` carry a time zone. The start and end of day or week is thus time zone specific.

Cluster architecture

Yona consists of 4 different services with different purposes:

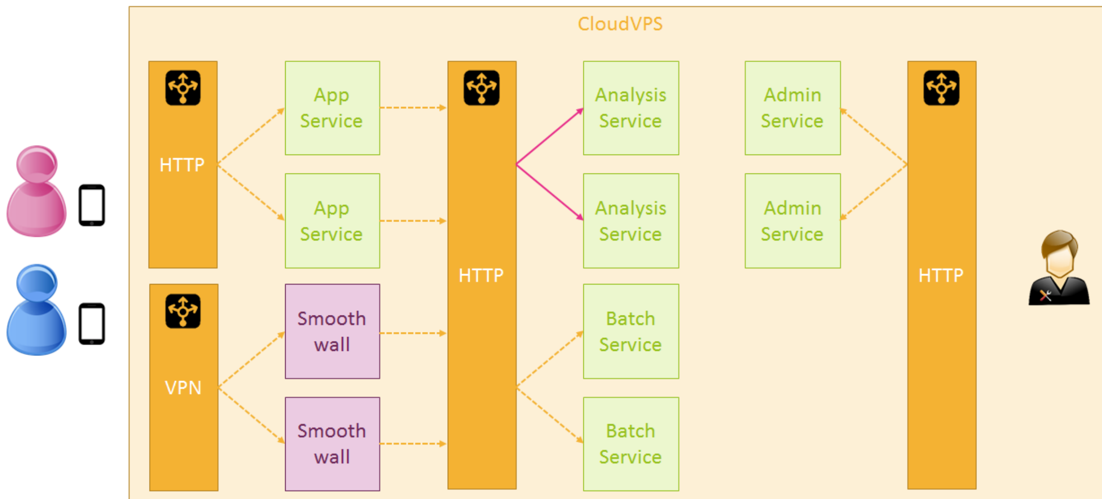
- App service – This is the sole point of contact for the app, providing all backing functions for it.
- Analysis service – This service analyses the activities of the user and depending on the activity and the goals of the user creates entries in the activity log and/or sends messages to the user and their buddies. These activities are passed on to this service by either the Perl script on the Smoothwall server (in case of network activities) or the app service (in case of app activities)
- Batch service – Runs scheduled tasks and longer running tasks. Longer running tasks can be triggered by the app service and the admin service.
- Admin service – This is the point of contact for administrative access to Yona. This service exposes a rudimentary web UI.

The relationships between the services can be depicted as follows:



All Yona services use the same database schema and they share a cache, synchronized through Hazelcast. Communication between the services (e.g. from the app service to the analysis service or the batch service) is always through HTTP.

To prevent concurrency issues, the requests of a single user cannot be spread over multiple instances of the analysis service. In case of a multinode cluster, we will need to have a load balancer in front of all services (to take care of load balancing and fail over) and in case of the analysis service, sticky sessions are needed. This is visualized in the below diagram:



Note that the admin service talks to the batch service, so that should also be linked to the middle load balancer, but that's omitted from the diagram to prevent too much clutter.