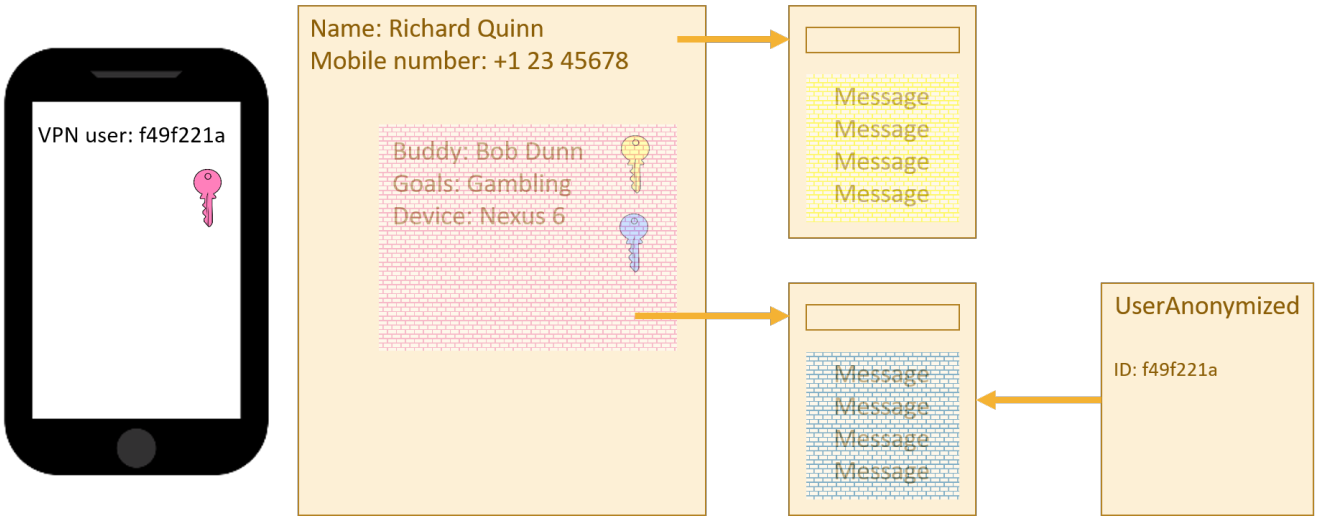
Encryption approach

Introduction

The Architecture page gives a high level understanding of the encryption approach taken for Yona. This page takes it a level deeper and describes the details. Yona employs symmetric (secret-key) and asymmetric (public-key) encryption. The asymmetric encryption approach allows us to implement the mailbox metaphor as given on the architecture page: everyone can put a letter in the box but only the one in possession of the key can read the them. The symmetric approach is used most commonly because of the performance benefits. The below picture provides an overview of the encryption approach:



The app contains a secret key (the Yona password, visualized above with the pink key) that is passed to the server in every request. The server uses this to decrypt the private data of the user, including information about buddies and goals, but also the private key of the two message boxes. The messages in the boxes are secured through the public/private key pair of that message box.

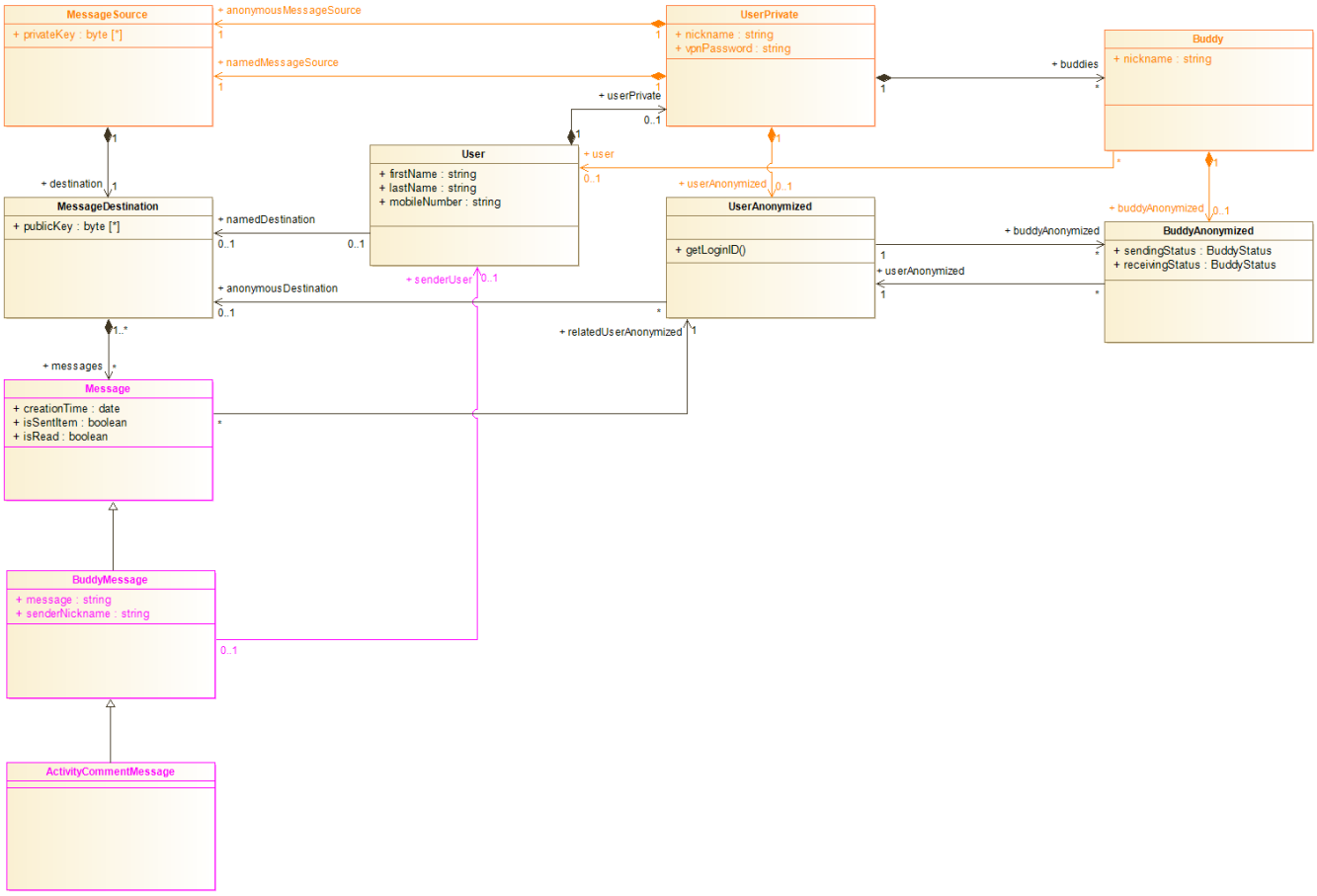
Encryption of private user data

The class diagrams on the Architecture page show orange classes that contain private user information, encrypted with the Yona password. For this encryption, we use AES, with a random key that is generated by the server at the time the user account is created. This key is returned to the app as part of the private user data. With every subsequent request, the key is passed to the server in the `Yona-Password` header. Every such request is executed in the context of a so-called `CryptoSession`. This session is initialized with the key that was received through the `Yona-Password` header. Code that needs to encrypt or decrypt data can fetch the current `CryptoSession` and request it to encrypt or decrypt the data. The encrypted data in JPA entities is encrypted and decrypted through an `AttributeConverter` while storing/loading the entities.

Encryption of message data

The messages contain encrypted data too (shown in magenta on the Architecture page). For this, we use asymmetric encryption (RSA), but we combine that with symmetric encryption for performance reasons. Every message contains a property `decryptionInfo` that contains the symmetric key used to encrypt the encrypted properties of that message. The `decryptionInfo` itself is encrypted with the public key of that message box.

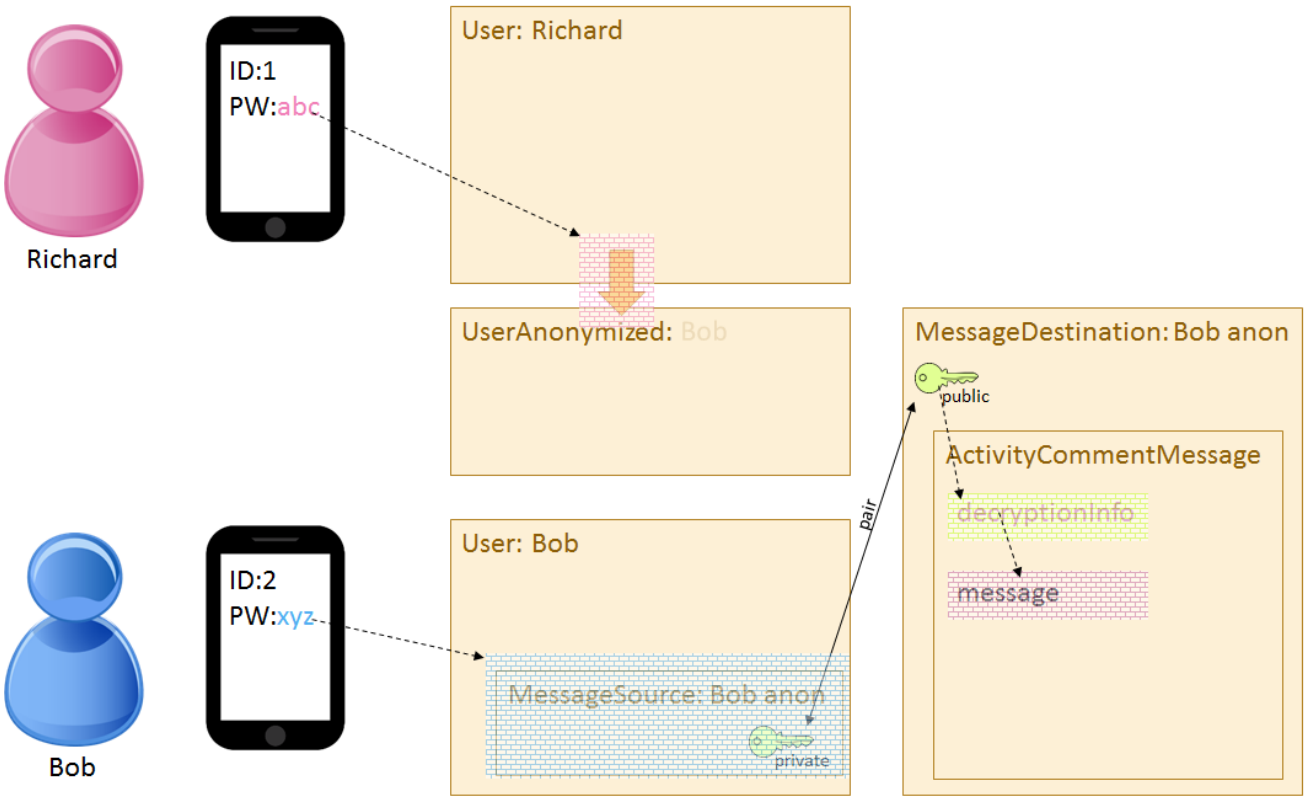
To see this more in depth, we focus on an activity comment message that one user sends to another. This is the relevant class diagram fragment:



To send that message, the we find the `MessageDestination` by decrypting the private data of the sending user and then navigating from `User` to `UserPrivate` to the appropriate `Buddy`. Then to the `BuddyAnonymized`, its `UserAnonymized` and then the related anonymous `MessageDestination`. On that, we call `sendMessage`. That will result in the following steps:

1. Generate a new symmetric encryption key
2. Encrypt that with the public key of the message destination
3. Store it in `decryptionInfo` on the `Message`
4. Encrypt all relevant message properties with the symmetric key

The data can be depicted as follows:



It would have been possible to encrypt the relevant message properties with the public key, but that has three downsides:

- RSA is a block cipher, so
  - The encrypted data is always at least one block (with an 1024-bits key, that is 128 bytes), so that often causes considerable bloat
  - It can only encrypt as much as fits in the block (with an 1024-bits key and OAEP padding, that's only 86 bytes), so longer data (for instance a URL) needs special handling
- Asymmetric algorithms are weak in comparison to symmetric algorithms, so they need a considerably longer key, causing more computational overhead