Messaging Module

Tickets · Browse Source · SVN Repository · Setup (OUTDATED)

Overview

The messaging module allows you to send all kinds of messages, including email and SMS. It also helps you handle various ancillary aspects of messaging, including managing addresses and browsing past conversations. Lastly, it has an easy-to-use API so that other modules can incorporate messaging into their own projects.

Developer Quick Start

Sending a Message

To send an SMS message, use the following code:

```
Context.getService(MessagingService.class).sendMessage("Hello, world!",
"+18007654321", SMSProtocol.class);
```

This sends the message "Hello, world!" to the number 18007654321 via SMS. If the supplied phone number or message are badly formatted, an exception will be thrown.

Sending a Message to Multiple Recipients

To send a message to multiple recipients, you must create a Message object and add messaging addresses as recipients. You should retrieve

Listening for Received Messages

To register a listener that will receive alerts when a message comes in, use this line of code:

```
Context.getService(MessagingService.class).registerListener(IncomingMessageListener
listener);
```

The listener must implement the IncomingMessageListener interface.

Architecture Overview

The MessagingService is the starting point for all messaging related tasks. Inside the MessagingService are a family of methods known as the "sendMessage" methods. They all look mostly like this:

```
public void sendMessage(String message, String address, Class protocolClass) throws
Exception
```

This method's parameters contain the 3 basic things that you need to send a message: a message, a destination, and a protocol. A protocol is a set of standards that govern how addresses and messages can be formatted. In the real world we know that "billybones" is not a phone number and that Twitter messages must be less than 140 characters. Protocol objects contain this formatting information and validate your messages when you send them. In general, you should not interact directly with protocols, as they are only used by the MessagingService to make sure that your messages are well formed.

When you ask the MessagingService to send a message, it is checked for validity and then saved to the database with a status of "Outbox". Every 5 seconds, a scheduler task (called DispatchMessagesTask) grabs the new outbox messages from the database and sends them via

"Gateways". Gateways are the messaging module's connections to the outside world - they are where the actual message sending takes place. For example, the Google Voice gateway maintains a connection to the Google Voice server using credentials provided in the management interface. Then, when it is told to send a message, it hands it off to Google Voice via an HTTP request.

At any given time there will be several gateways running. The messaging Module currently has 3 different gateways for sending SMS, one for sending Email, and one for sending OMail. You can check the status of your gateways and start or stop them in the "Manage Messaging Gateways" page. If there is a type of messaging that you would like to support, please feel free to implement a Gateway and a Protocol yourself.

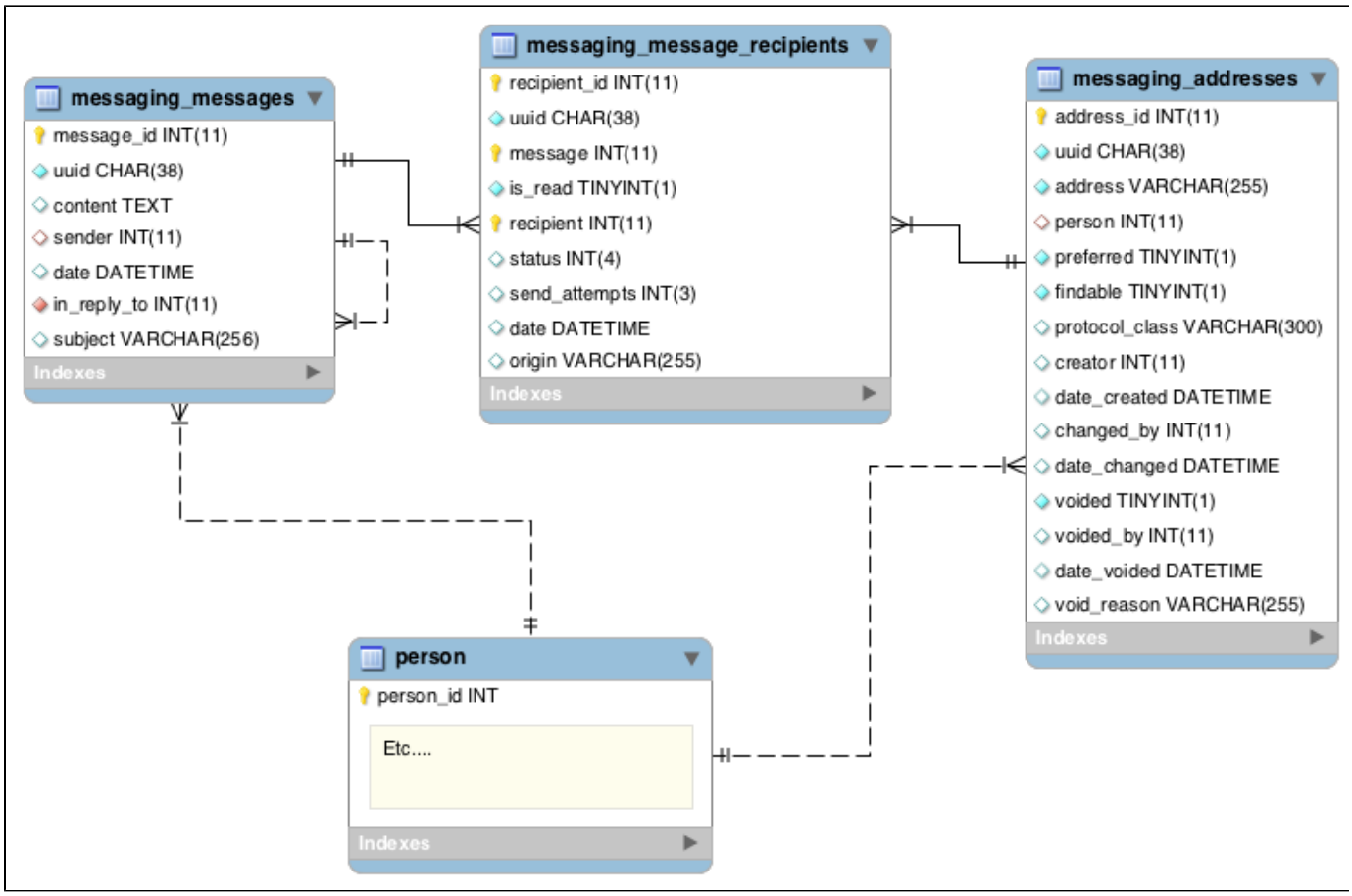
The Lifecycle of a Message

When using or developing on the Messaging module, it's helpful to have an overview of exactly what happens when you send a message, what fields are set where, etc... The 'Architecture Overview' section above details some of this, but this section talks about the lifecycle of a message in more detail.

1. A message is created using a constructor. The fields necessary to successfully send a message are content and recipient, but sender is also desirable. In the recipient, the necessary fields are destination and protocol, but receiver (the actual person that the address belongs to) is desirable. There is currently no 'system sender' unless you have a daemon person (not just a daemon user).
2. MessagingService.sendMessage is called
    a. The message is checked for well formed-ness, i.e. that the message content is sendable using the proposed protocols and that all messaging addresses are valid.
    b. MessagingService checks if there are gateways running that can send to the destination addresses.
    c. The status of each of the MessageRecipient objects is set to 'outbox'.
    d. The message is saved to the database.
3. The message is retrieved from the database by DispatchMessagesTask. Because messages can be sent to multiple destination addresses that use different protocols, each MessageRecipient is treated separately.
    a. DispatchMessagesTask iterates through each MessageRecipient, and passes each one to a Gateway that can carry the MessageRecipient's protocol.
        i. The gateway sends the message
        ii. The gateway will set the MessageRecipient's origin address according to its configuration.
        iii. If something goes wrong, an exception will be thrown, but no message status changes will be made by the gateway. The DispatchMessagesTask handles message status changes.
        iv. If messages would take a prohibitively long time to send, it is best to use an asynchronous method and modify the statuses later. See SmsLibGateway for an example of this.
    b. DispatchMessagesTask sets the MessageRecipient's date to the current time.
    c. DispatchMessagesTask sets the MessageRecipient's status based on whether an exception was thrown.
    d. The Message date field is set by DispatchMessagesTask
4. Each message gateway is told to process incoming messages before outgoing messages are dispatched.
    a. At the end of the receiveMessages method, each message recipient object attached to a new message should have a status of 'received'.

Data Model

Below is a diagram of the data model of the Messaging Module. The MySQL Workbench file is included so you can play around with it and edit it if you want.



messaging_schema.mwb