UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

API User Manual
Mindmap PIM

Client: IMINISYS

Team: A-Cube-N

Grobler, Arno	Lochner, Amy	Maree, Armand
14011396	14038600	12017800

Department of Computer Science, University of Pretoria

Contents

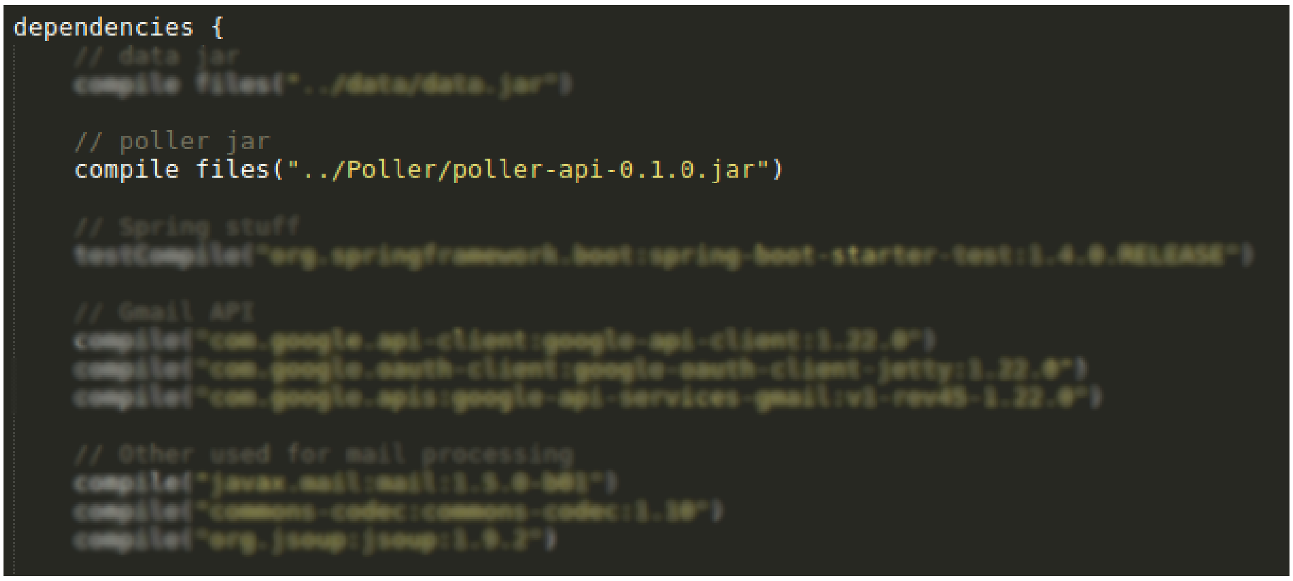
1 Introduction

This document will serve as a manual to external developers to interface with the Unclutter Polling API. This API will allow developers to more easily and more rapidly develop pollers for new platforms while at the same time hiding the technical details of the interaction between the pollers and the rest of the system.

2 Adding the Dependency

It is recommended that the poller is built with Gradle since the rest of the system also makes use of this, although it is not enforced.

Figure 1 shows the *build.gradle* file. It indicates that the API must be added as a dependency.



```
dependencies {
    [redacted]

    // poller jar
    compile files("../Poller/poller-api-0.1.0.jar")

    [redacted]
```

Figure 1: Gradle Dependency

3 Bean Setup

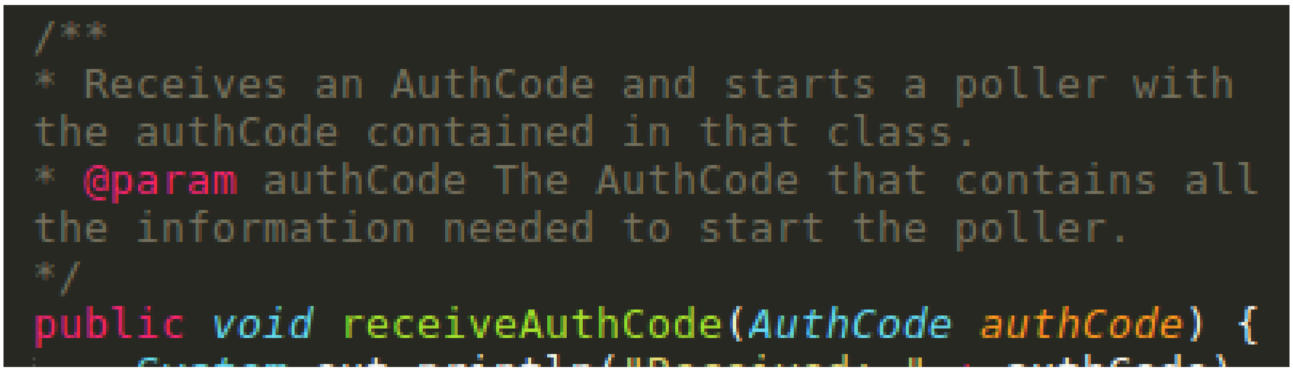
To simplify the processes it is recommended that you use Spring and allow it to manage the beans inside of the API. In order for Spring to do this, only a few simple steps need to be implemented.

Firstly Let's assume your main Spring application class is called *Application* saved in a file named *Application.java*. In this class you need to create 3 beans (2 of which are optional, but recommended). These beans are:

- AuthCodeListener
- ItemRequestListener
- MessageBrokerFactory

3.1 AuthCodeListener

The object contained in this bean should have one method that takes a single AuthCode object in as a parameter (see figure 2). This is the method that should receive an authorization code and start the polling processes for the specified user. This bean is one of the optional ones, but if this object is not placed in a bean, then you will have to manage it manually.

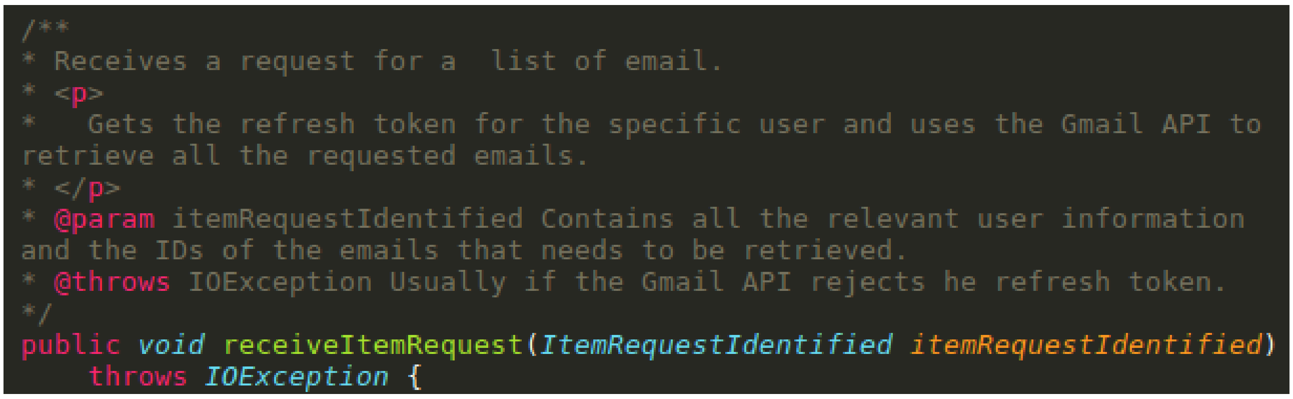


```
/**
* Receives an AuthCode and starts a poller with
the authCode contained in that class.
* @param authCode The AuthCode that contains all
the information needed to start the poller.
*/
public void receiveAuthCode(AuthCode authCode) {
```

Figure 2: AuthCodeListener Receiving Method

3.2 ItemRequestListener

The object contained in this bean should have one method that takes a single ItemRequest object in as a parameter (see figure 3). This is the method that should receive a request for an item, retrieve the specified item and send the item back (we will discuss this in section 5) This bean is one of the optional ones, but if this object is not placed in a bean, then you will have to manage it manually.



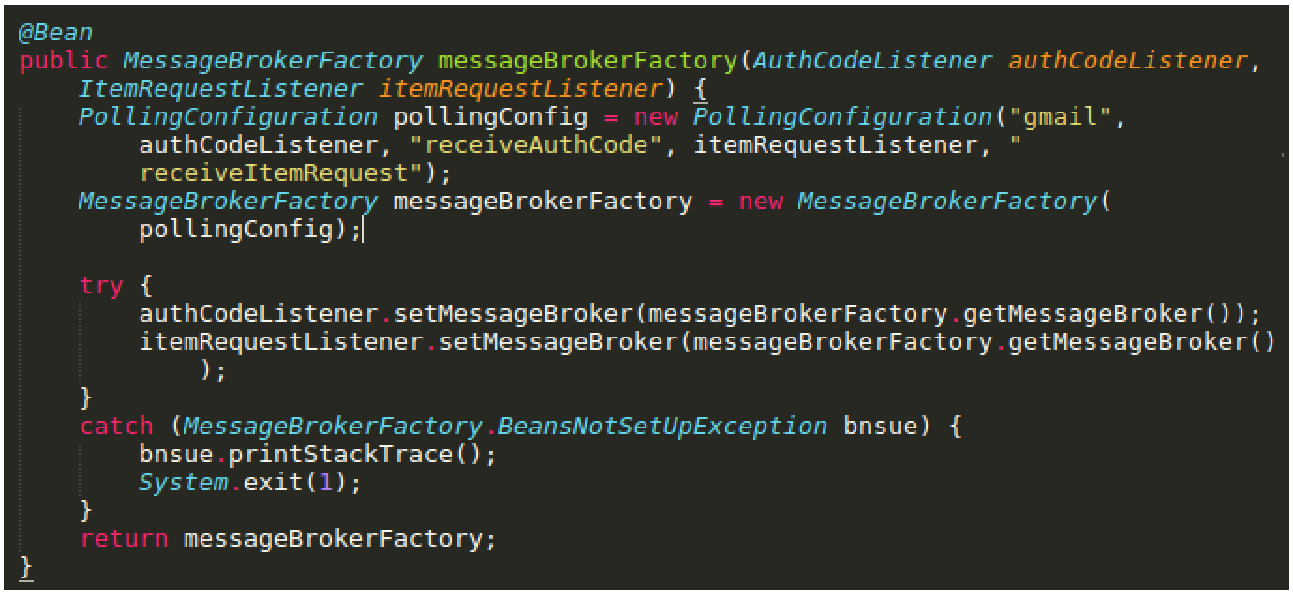
```
/**
* Receives a request for a  list of email.
* <p>
*   Gets the refresh token for the specific user and uses the Gmail API to
retrieve all the requested emails.
* </p>
* @param itemRequestIdentified Contains all the relevant user information
and the IDs of the emails that needs to be retrieved.
* @throws IOException Usually if the Gmail API rejects he refresh token.
*/
public void receiveItemRequest(ItemRequestIdentified itemRequestIdentified)
    throws IOException {
```

Figure 3: ItemRequestListener Receiving Method

3.3 MessageBrokerFactory

The MessageBrokerFactory is the core of the API. It contains all the complicated configuration for the Spring beans that interact with the rest of the system so that you do not have do it manually. Creating this bean requires the following steps (code is provided in figure 4):

- Create a *PollingConfiguration* object and in the constructor provide the following in this order:
  - Name of platform (like "gmail" or "facebook"). A string that contains lowercase letters.
  - The AuthCodeListener object created in the bean in the previous section.
  - A string of the name of the method in AuthCodeListener that will be receiving the AuthCode object.
  - The ItemRequestListener object created in the bean in the previous section.
  - A string of the name of the method in ItemRequestListener that will be receiving the ItemRequest object.
- Create a new *MessageBrokerFactory* and in the constructor parameter pass in the *PollingConfiguration* object you just created.
- Since the factory is now ready to produce *MessageBroker*s you can pass a instance of this class (created by the factory) to each listener.



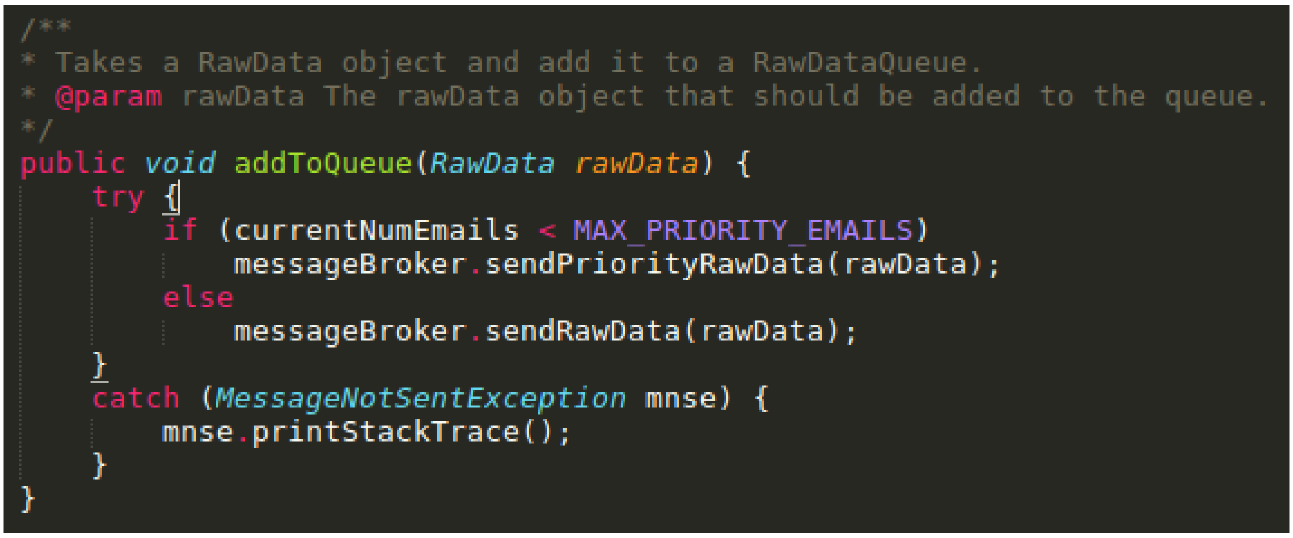
```java
@Bean
public MessageBrokerFactory messageBrokerFactory(AuthCodeListener authCodeListener,
    ItemRequestListener itemRequestListener) {
    PollingConfiguration pollingConfig = new PollingConfiguration("gmail",
        authCodeListener, "receiveAuthCode", itemRequestListener, "
        receiveItemRequest");
    MessageBrokerFactory messageBrokerFactory = new MessageBrokerFactory(
        pollingConfig);

    try {
        authCodeListener.setMessageBroker(messageBrokerFactory.getMessageBroker());
        itemRequestListener.setMessageBroker(messageBrokerFactory.getMessageBroker()
            );
    }
    catch (MessageBrokerFactory.BeansNotSetUpException bnsue) {
        bnsue.printStackTrace();
        System.exit(1);
    }
    return messageBrokerFactory;
}
```

Figure 4: MessageBrokerFactory Bean Example

4 Sending RawData Objects

The poller will send *RawData* objects via a *MessageBroker* object that is created by the *MessageBroker-Factory* that is set up in the previous section. There are two kinds of RawData. The first is called priority RawData, i.e. it needs to be processed rapidly by the rest of the system, and secondly standard non-priority *RawData*, i.e. it does not have to be processed immediately.

When a poller has received a new item (like an email or post), the poller can send this item (in the form of a *RawData* object) to the rest of the system for processing and persistence via two methods contained in the *MessageBroker* class. Priority *RawData* objects can be sent with the *sendPriorityRawData* method and non-priority *RawData* objects can be sent with the *sendRawData* method. See figure 5.

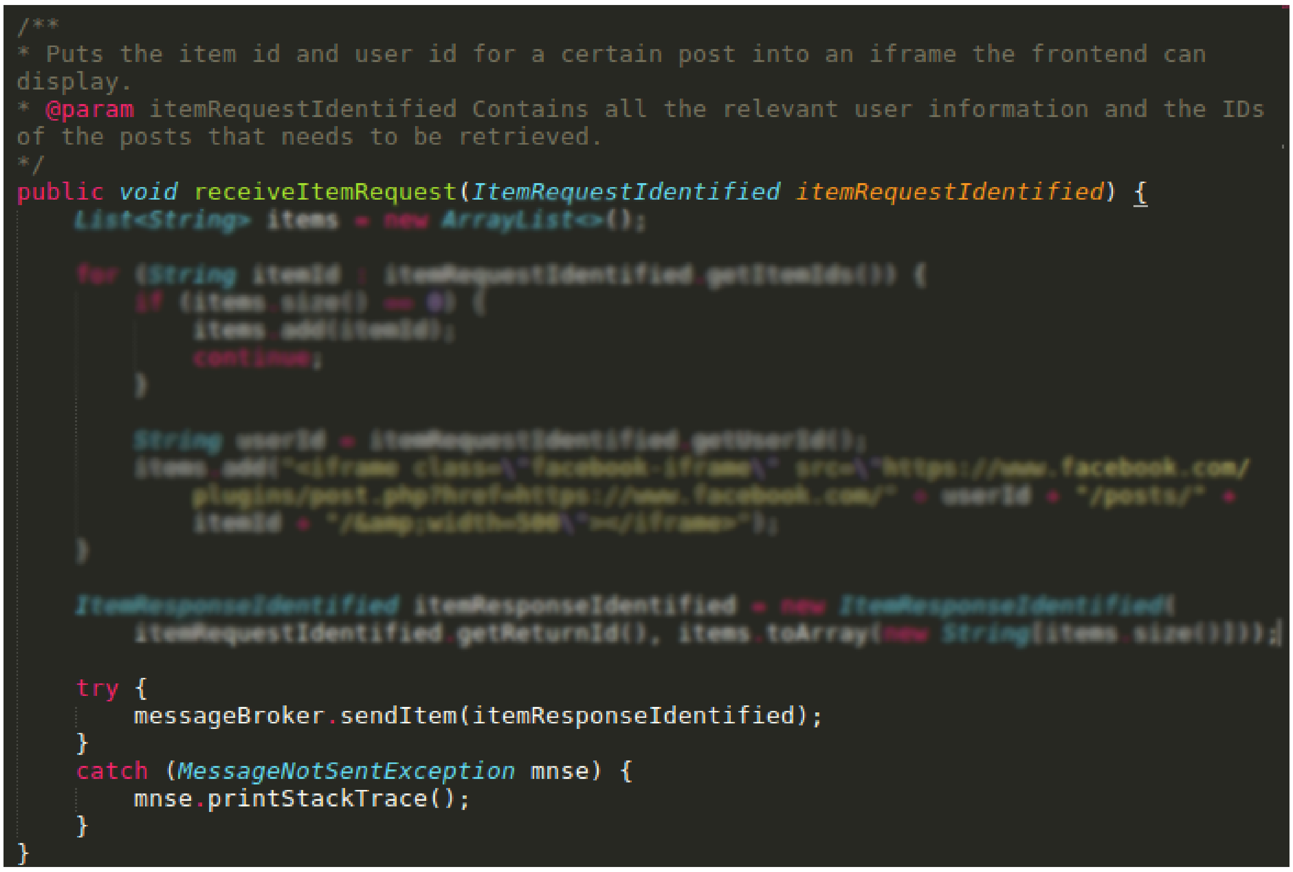


```java
/**
* Takes a RawData object and add it to a RawDataQueue.
* @param rawData The rawData object that should be added to the queue.
*/
public void addToQueue(RawData rawData) {
    try {
        if (currentNumEmails < MAX_PRIORITY_EMAILS)
            messageBroker.sendPriorityRawData(rawData);
        else
            messageBroker.sendRawData(rawData);
    }
    catch (MessageNotSentException mnse) {
        mnse.printStackTrace();
    }
}
```

Figure 5: Send RawData

5 Sending ItemResponse Objects

So once you have received an item request and retrieved the item from the specific external platform, it is time to send it back to the requesting service. This is as simple as calling the *sendItemResponse* method of the *MessageBroker* class. See figure 6.



```
/**
* Puts the item id and user id for a certain post into an iframe the frontend can
display.
* @param itemRequestIdentified Contains all the relevant user information and the IDs
of the posts that needs to be retrieved.
*/
public void receiveItemRequest(ItemRequestIdentified itemRequestIdentified) {
    [redacted]

    try {
        messageBroker.sendItem(itemResponseIdentified);
    }
    catch (MessageNotSentException mnse) {
        mnse.printStackTrace();
    }
}
```

Figure 6: Send RawData

6 Conclusion

As seen here, the API hides a lot of the complicated internal working from the poller developer which should speed up and simplify the development of new pollers. Should this guide not be sufficient for you, you are more than welcome to look at the JavaDoc documentation provided. The JavaDoc explains in great detail the purpose of each method, parameter and class.