Just the Right Information

Mobile Platform v1.0.0
Developer's Guide

17 November 2009

Tel: +27 21 913 6519 | Fax: +27 86 671 2354 | info@korwe.com | www.korwe.com
PO Box 6513, Welgemoed 7538, Cape Town, South Africa
Directors: Dr DJ Hislop, N Govender | Company Reg. No. 2007/009575/07

1 Introduction

This document forms part of The Core software development kit.

1.1 Document structure

1.2 Audience

This document is written for technical people who want to use or understand the Korwe integration platform

1.3 Comments and feedback

Please contact Korwe at admin@korwe.com if with any comments or questions.

1.4 Licensing

TBD

2 Architecture

The platform provides and application layer integration platform for mobile as shown in Figure 1. The intention of the platform is to consolidate and control of information flow to one place, the platform. At this central location, application logic can be applied for personalisation and customisation.

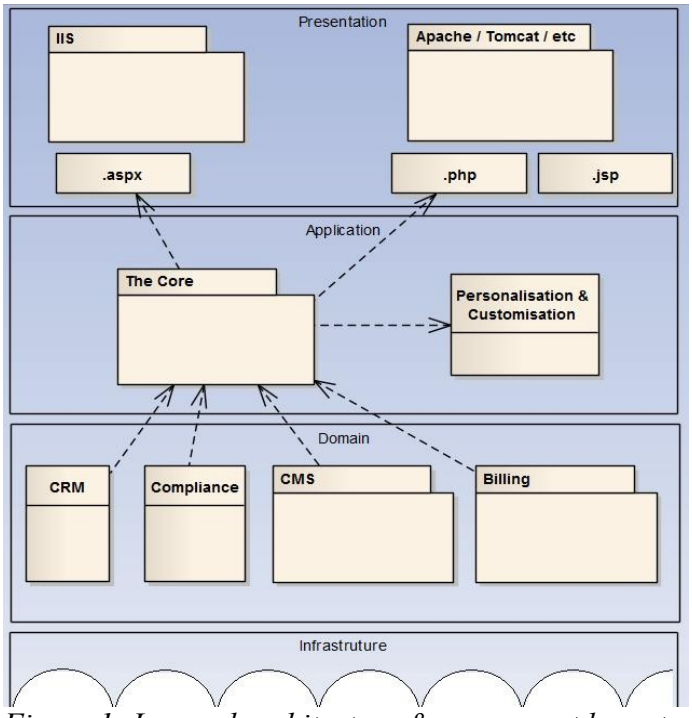


Figure 1: Layered architecture & component layout

3 Getting Started

3.1 Sending messages

Read through TestClient.cpp code to see how messages are sent. The program is functional and can be run under CentOS. It will write success and failure messages to the console.

3.2 Receiving messages

Read through TestServer.cpp code to see how messages are received. The program is functional and can be run under CentOS. It will write success and failure messages to the console.

Figure 2 is a simple illustration of sending and receiving Core messages.

Figure 2: Send and Receive Code examples

C++:

```
MessageSender sender(Q_SERVER, Queues::CLIENT_CORE);
MessageReceiver receiver(Q_SERVER, Queues::CORE_CLIENT, SESSION_ID);
sender.sendCoreMessage(InitiateSessionRequest(SESSION_ID), SESSION_ID);
InitiateSessionResponse* response =
   dynamic_cast<InitiateSessionResponse*>(receiver.getNextCoreMessage(500));
if (response == 0) {
    std::cout << "No response to initiate session" << std::endl;
    return 1;
}
else {
    std::cout << "Received initiate session response" << std::endl;
    std::cout << "Session initiation " << (response->isSuccessful() ?
       "was " : "was not ") << "successful" << std::endl;
}
delete response;
```

C#:

```
var sender = new CoreSender(queueServer, Queues.Queue.ClientToCore);
var receiver = new ReceiveBuffer(queueServer, Queues.Queue.CoreToClient,
                                 sessionId);
receiver.start();
var init = new InitiateSessionRequest(sessionId);
sender.sendMessage(init);
Thread.Sleep(500);
if (receiver.ContainsKey(init.Guid)) {
    CoreMessage msg = receiver.getFirstMessage(init.Guid);
    var initResponse = msg as InitiateSessionResponse;
    Console.Out.WriteLine("initResponse = {0}", initResponse);
    bool success = initResponse.Successful;
}
```

4 The Core API

4.1 API Overview

The Core provides a message based API. It uses AMQP as its underlying messaging standard, on which it builds a specialized XML message protocol that is optimised for application level integration for mobile technologies and standards.

Figure 3 indicates the conceptual location of The Core APIs.

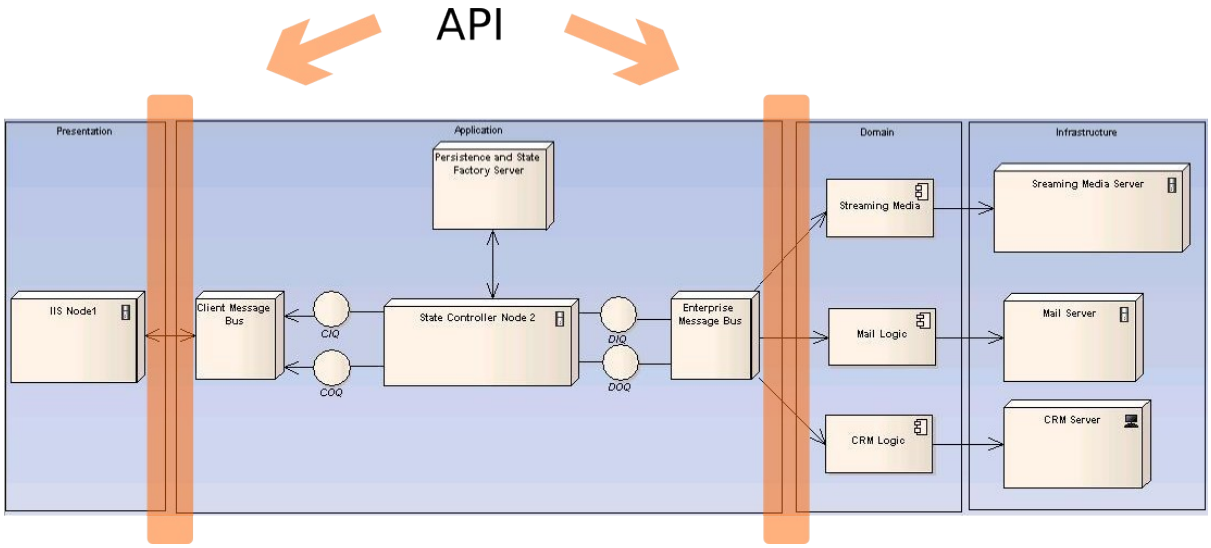


Figure 3: API Location

4.2 Messaging API

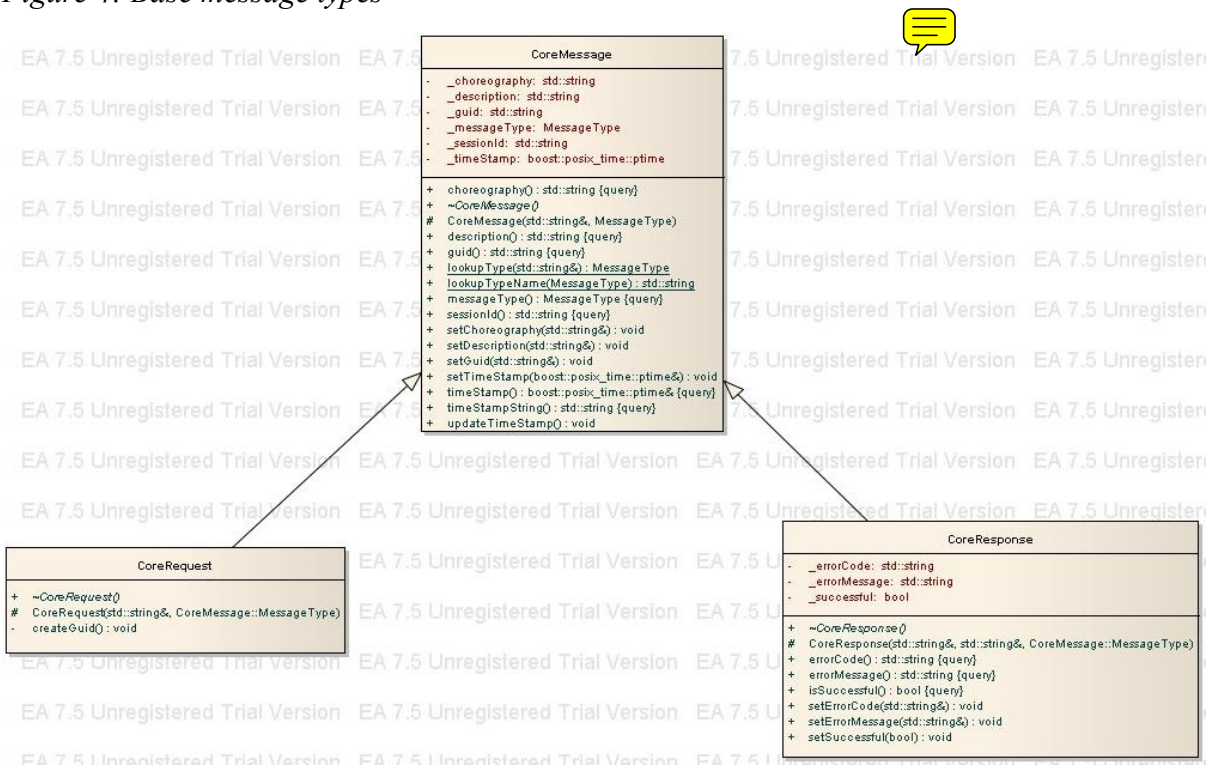
There are currently C++, C# and Java libraries for interfacing with The Core, as well as a REST-style web service API.

All messages inherit from CoreMessage and at the highest level are divided between request and response messages.

Client and Service Applications

The Core distinguishes between client and service applications. Client applications are applications that request services and data from service applications and services respond to requests from clients. In other words client applications send ServiceRequest messages and receive ServiceResponse and DataResponse messages, while service applications receive ServiceRequest messages and send ServiceResponse and DataResponse messages. Clients are required to create a Core session before sending any service requests and to end it when they are done. There is no reason why a single application cannot act as both client and service, e.g., one service may require information from another.

Figure 4: Base message types



Request Messages

Request messages are usually sent by client applications. Requests are processed by The Core and may be forwarded to service applications for further processing. Each request contains a client-specified SessionId, which identifies the client session and a system-generated GUID, which identifies the request and can be used to correlate with matching responses.

There are 3 types of requests: InitiateSessionRequest, KillSessionRequest and ServiceRequest. InitiateSessionRequest establishes a client session within The Core while

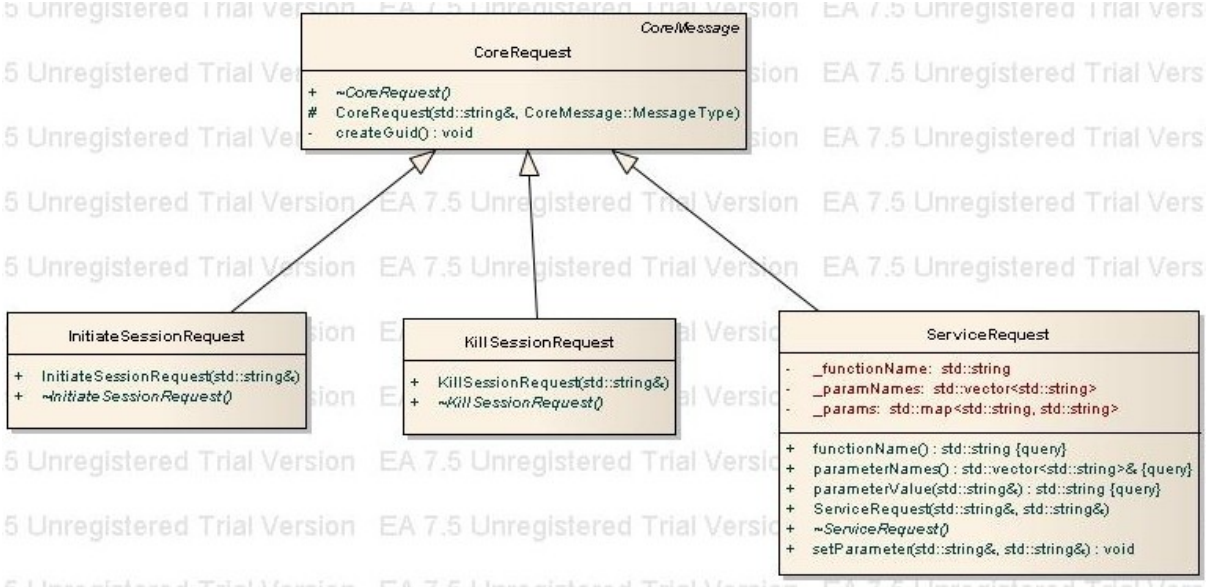


Figure 5: Core Request message types

KillSessionRequest ends the session. While a session is active, a client may send any number of ServiceRequests, which will ultimately be processed by a service application. Every request should result in a response message being sent back to the client.

Response Messages

Response messages are sent either by The Core or by a service application. Each response corresponds to a request message and contains the same SessionId and GUID.

Figure 6: Core Response Message Types

InitiateSessionResponse and KillSessionResponse messages are sent by The Core itself, while ServiceResponse and DataResponse are sent by service applications. DataResponse messages contain the actual data resulting from a service request. The ServiceResponse itself contains only an indication of success or failure and whether or not there is a DataResponse. This allows the Core to handle only control messages and allows data to be sent on an independent channel.

Sending and Receiving Messages

The API contains a few classes to deal with sending and receiving Core messages. The sender class is instantiated with a queue server name and a queue name (see Figure 2) and sends all messages to that queue on that server. Serialization of the message object into XML is handled by the system.

The receiver classes are instantiated with a queue server and queue name as well as an identifier which identifies the client or service application for which you want to receive messages. In the case of a client application, this is the session id, and for a service application, it should be the service name.

The following table describes the queue conventions:

Client Applications		Service Applications	
Send	**Receive**	**Send**	**Receive**
ClientToCore CLIENT_CORE	CoreToClient CORE_CLIENT	ServiceToCore SERVICE_CORE	CoreToService CORE_SERVICE
	Data CORE_DATA	Data CORE_DATA	

4.3 XML Schema

The following is a basic schema for Core XML messages. The XML is usually abstracted away by the message types, but can be useful for message tracing and debugging.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="coreMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sessionId" minOccurs="1"/>
        <xs:element ref="guid" minOccurs="1"/>
        <xs:element ref="messageType" minOccurs="1"/>
        <xs:element ref="timeStamp" minOccurs="1"/>
        <xs:element ref="choreography" minOccurs="0"/>
        <xs:element ref="function" minOccurs="0"/>
        <xs:element ref="parameters" minOccurs="0"/>
        <xs:element ref="successful" minOccurs="0"/>
        <xs:element ref="errorCode" minOccurs="0"/>
        <xs:element ref="errorMessage" minOccurs="0"/>
        <xs:element ref="hasData" minOccurs="0"/>
        <xs:element ref="data" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="sessionId" type="xs:string"/>
  <xs:element name="guid" type="xs:string"/>
  <xs:element name="messageType">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="InitiateSessionRequest"/>
        <xs:enumeration value="InitiateSessionResponse"/>
        <xs:enumeration value="KillSessionRequest"/>
        <xs:enumeration value="KillSessionResponse"/>
        <xs:enumeration value="ServiceRequest"/>
        <xs:enumeration value="ServiceResponse"/>
        <xs:enumeration value="DataResponse"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="timeStamp" type="xs:dateTime"/>
  <xs:element name="choreography" type="xs:string"/>
  <xs:element name="function" type="xs:string"/>
  <xs:element name="parameters">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="parameter" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="successful" type="xs:boolean"/>
  <xs:element name="errorCode" type="xs:string"/>
  <xs:element name="errorMessage" type="xs:string"/>
  <xs:element name="hasData" type="xs:boolean"/>
  <xs:element name="data" type="xs:string"/>
  <xs:element name="parameter">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="value"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="value" type="xs:string"/>
</xs:schema>
```

5 Installation

5.1 System Requirements

Currently The Core can run on both CentOS and Windows machines.

TODO: Add storage space requirements, operating system versions, RAM, network connectivity etc.

5.2 Environment

5.3 Installing

See README in installation directory for applicable Operating System.

6 Appendix A – Acronyms

7 Appendix B – Resources