Kinetics Group Limited

Activiser for Android Specification

# Introduction

Activiser is the name of the current work force mobility solution used by Kinetics. It was originally written for Kinetics in 2004 by a team of university students, as part of an under-graduate information systems program. At the time the only viable platform was Windows Mobile (or Pocket PC as it was then).

This document describes our requirements for the Android version, by describing the existing Activiser screens as seen on a Windows Mobile device.

# Data Structure Overview

The screen structure is naturally related to the data structure the program uses. The data structure is dictated by the web service which cannot be substantially modified (although minor changes may be possible if required to properly support a new platform).

There are just four ‘entity’ tables and a few ‘lookup’ tables in the database. The names are historical and don’t always make sense to people seeing them for the first time, so here’s an overview. The four entities are:

1. Client Site: A physical or virtual customer location
2. Request: A request for work. Also known as a Case or Ticket in other systems.
3. Job: A piece of work done. Also known as a Time Sheet or Activity in other systems.
4. Consultant: someone who does jobs. Consultants can see each other’s time sheets for a request and/or client that they’re doing work for.

The business process is: Client’s **Request** work to be done on their **Client Site**. A **Consultant** completes one or more **Job**s against that request, and they may be completed on or off site (depending on the client’s requirements).

There are a lookup tables for Client Site Status, Request Status and Job Status.

# Our expectations of the replacement program

We don’t want a look-a-like program; we want a do-a-like program. In other words, we want something that allows our engineers to enter time sheets and get them signed. To support this, they need to be able to access client information and details about the requests being made of them. We expect the new program to operate using the relevant paradigm for its target platform, and so actually expect individual screens to have a somewhat different ‘look and feel’ from those shown in this document.

# JSON in the Web Service

To simplify development on modern platforms, the web service will be augmented with ‘JSON’ versions of all web methods required by the mobile client. These will work by making simple conversions between XML (the native serialization format for .NET DataSets) and JSON. Since we have no way of testing these new methods at Kinetics, we will need to coordinate their development and testing.

# General

## Device ID

Every device that connects to an Activiser database needs to have a unique ‘device ID’. This is for security. The device ID should not be stored anywhere on the device, nor be human-readable on the device except within the Activiser program, and should be derived from the device itself (perhaps a hash of the IMEI number and a MAC address, or some such). For consistency with the existing client, it would be a 20-byte field base32 encoded to 30 characters, but can be any human-readable string up to 128 Unicode characters in length. The device is looked up in a case-insensitive database, so the id must be unique independent of case.

This device ID is part of the first parameter of every web service method call. The other part is a version number. The version number is used by the web service to confirm compatibility with the device. For this version, we will change from 4.0.x.y to 4.1.x.y (where x and y are build and revision numbers).

## Security

The users of the Activiser system all work for the same team, they do not compete with each other. There are therefore no ‘secrets’ between users of the system. Any user of the system is able to see any other users’ data.

Because of this, the web service can take a simple approach to security: as long as the user can connect and provide a valid device id to the server, then it is ‘authenticated’.

However, a user may only record data as themselves, so the application uses the user’s login details to determine the appropriate ConsultantUid to use for all new records.

There are two levels of user security in Activiser. In the current implementation at Kinetics, we are using ‘integrated’ security, meaning we link a user’s Windows account to their Activiser account. Initial development of the Android version can assume we will continue with this.

However, to remove the need to store windows username and password information for specific users, we may consider changing to ‘activiser’ security, in which case all users will use a generic username/password to make the connection to the web service, and provide an ‘activiser’ username and password to identify which consultant they are. This is a fairly minor change to the program (see the next section), so we can make that decision later.

## Password security

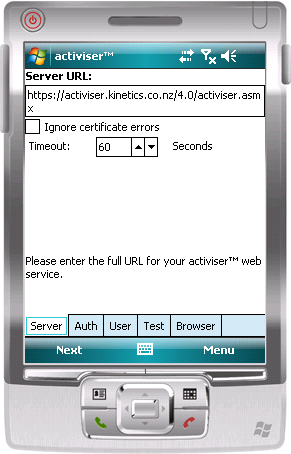
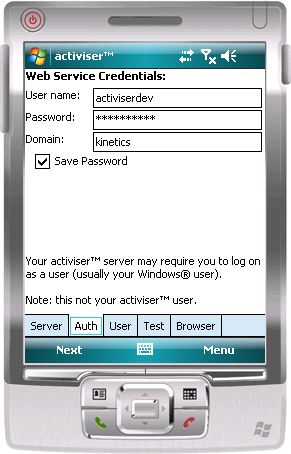
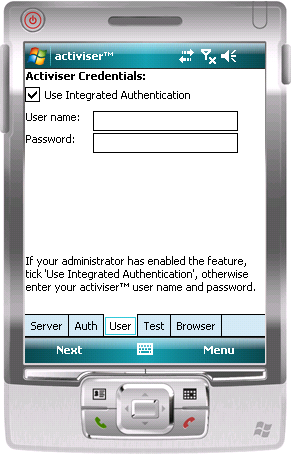
Whichever security model is used, username/password information will need to be stored which should be considered security-sensitive and treated as such and only stored securely and encrypted.

## Web Service Gotchas

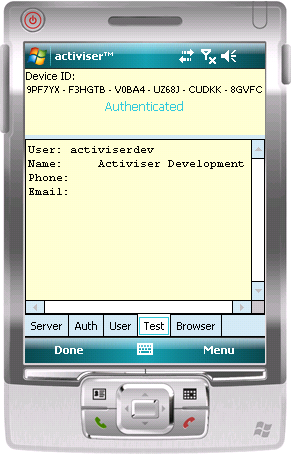
In order to improve performance of synchronizing with mobile devices over poor quality network connections, some compromises were made in the web service design. The most significant being that the Activiser web service is not a ‘stateless’ web service. The web service maintains a record of what it believes is on each device that is registered in its database. This makes it simple to ‘recreate’ the state of an Activiser database for a user should their device be replaced. It also impacts the synchronization process: some aspects are simplified, but others are made are a bit more complex.

# Service setup screen

When starting the program for the first time, or whenever a change is needed, there should be a dialog for configuring the web service, where URL and appropriate credentials are entered. The web server hosting Kinetics’ implementation of Activiser employs ‘Windows Integrated’ authentication mechanism, which links a valid Windows username and password to an Activiser account. This may change to ‘Activiser’ authentication, in which case it would be necessary to also add the Activiser credentials page:

The ‘Test’ page, has a ‘Test’ button (surprise!) and displays the device id (so that it can be entered into the activiser database at the server end), and some results of the test. The ‘Test’ button becomes a ‘Done’ button if the test is successful.

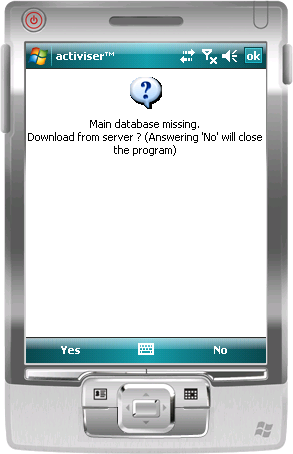
The web service method to use for authenticating a user is ClientGetUserDetails(AsJSON). If using integrated authentication, the user and password fields for this method are left blank, and the web service will look up the Activiser user based on their Windows username/password (note: the ‘checkbox’ for integrated authentication is a user-interface aid only, the decision to use integrated authentication is made by passing a null string to the web service).

If using ‘activiser’ authentication, the password needs to be hashed prior to calling the web service. The hash is simply a 16-byte MD5 hash of the password, expressed as a Hex string.

The ‘browser’ page is for debugging – it merely opens a standard browser window using the web service URL; this allows us to confirm whether a failure is with the connection to the web service, or some other problem.

# Initial Download

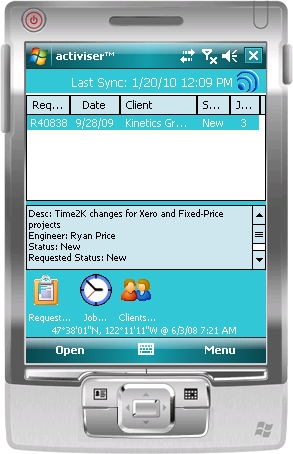
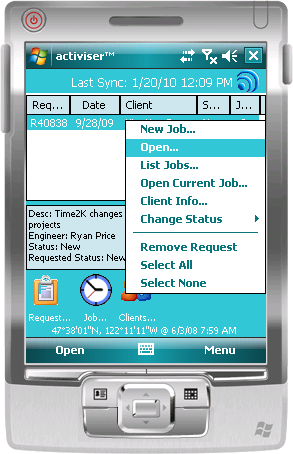
Once the program has been setup for the first time, there is an ‘initial’ download of the database. Given that the program was written for an environment that pre-dated mobile broadband, the database download could take a while. We expect this to be less of an issue with our modern network…

All downloads after this initial download will only include changes to the data since last sync’d.

# Main Screen

The main screen is the ‘dashboard’ from where the field engineer will see what work they have to do. This is a simple list of requests, which they should be able to sort and filter. When they select a request, they can choose to view it, view the client details for that request, or create a new job against the request; the default behaviour should be to open the request. The ‘List Jobs’ menu option opens the request, but with the ‘Jobs’ tab having focus.

## Removing a Request

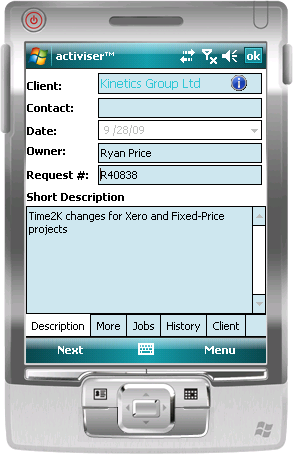
Removing a request removes the request from the device. It does not delete it on the web service; however the web service needs to be advised that the request is no longer on the device. So any record being deleted on the device needs to be recorded in a ‘deletion queue’ so that its deletion can be reported to the web service during a sync.

## Changing Status

A request can go through several status changes in its lifetime, and a subset of these changes can be \*requested\* by the user. The requested status change is specified by changing the ‘ConsultantStatusId’ field. Some changes may also require explanation (e.g. cancellation); this explanation is recorded in a special ‘Job’ record for the request, the specific details can be found in the ‘ChangeRequestStatus’ method in ‘RequestUtilities’ class of the existing client (source code to be provided).

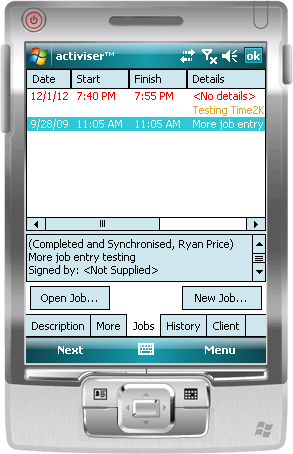
# Request Screen

The request screen has several tabs for different information. The user can also create a new job from this screen. The ‘Client’ tab just displays the site information in a text box – there wasn’t really a need to have a full ‘Client Info’ screen from the ‘Request’ screen – and any form we didn’t have to load made the program faster.

The main tab shows the main information an engineer will need to progress with work: Who the client is, who to contact, and the ‘Short description’ of the problem. The ‘More’ tab contains the ‘Long Description’ of the request.

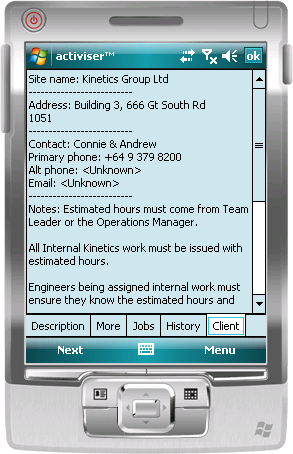
For a new request created on the device, the client, contact, description and date fields will be editable (client is a lookup, owner is the user, request # is an auto-number)

For the Android client, the ‘Description’ and ‘More’ tabs could probably be merged and the user expected to scroll up to see the additional data.

The ‘Jobs’ tab shows a list of job records previously recorded against this specific request. Incomplete jobs are displayed in red, complete jobs in green, synchronised or other people’s jobs in yellow/orange. From here, an engineer can open an existing, incomplete job for completion, or view completed jobs.

The ‘History’ tab shows recent job records \*for this client\* that were not for this request.

For the Android client, these tabs can probably be merged, provided that it is obvious whether a job is for the same or different request.

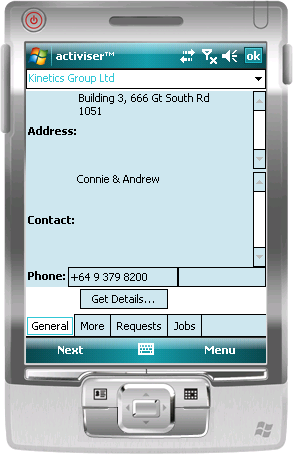
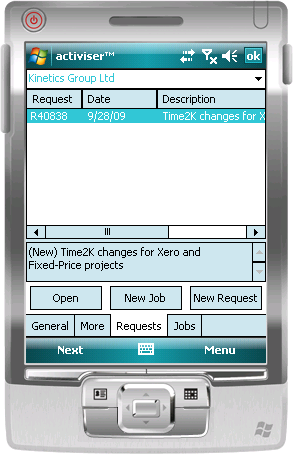


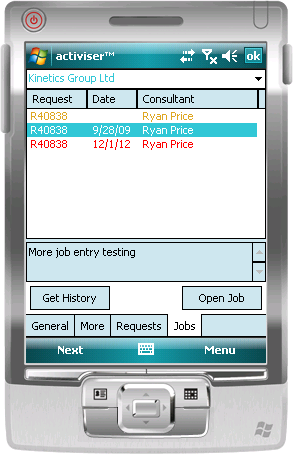
The ‘Client’ tab just shows a simple overview of the client in a text box (rather than crafting a whole screen for it). This was done to a) avoid loops (where the user opens a client screen from a request, and then opens a request screen from the client screen) b) to improve performance (a text box is simple and requires much less resource than a whole form or multiple labels and text boxes) and c) because it’s easy to read and copy/paste if necessary.

For the Android version, I expect this tab will not be necessary – a popup from the first tab’s client field would provide a better UX.

# Client Info Screen

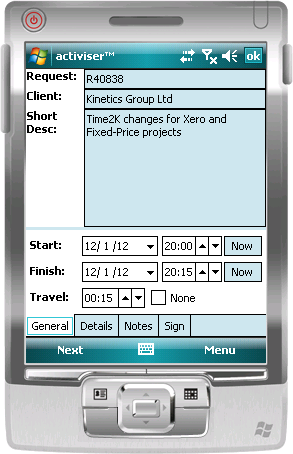
This screen should be fairly self-explanatory. But it is probably more complex than it needs to be. This information is read-only on devices, so a single flow-style UI element will probably work better than the dedicated label/textbox fields. There should be menu options to open/edit requests and jobs.

The exception to the self-explanatory bit. The ‘Get History’ button will retrieve the most recent job records for this client \*that are not already on the device\*. It will also retrieve request records if necessary. Each time the button is clicked, older jobs will be retrieved. Remember: the web method behind this button is not ‘stateless’.

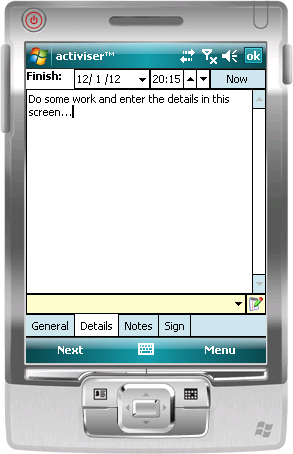
# Job Screen

This is the main data entry screen in the program (all other screens are mostly information screens)

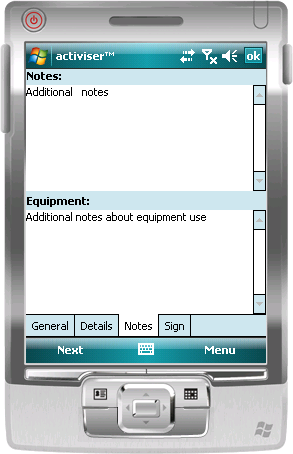


On the first tab, the user records the start and stop date and time. Internally, these are stored in universal time in the StartTime and FinishTime fields. The Local-Time Start Time is stored in the ‘JobDate’ field. (This is an example of a field whose semantics changed, but didn’t get renamed). The date fields must be ‘culturally sensitive’ – In the U.S. you normally do MM/DD/YY (as shown), but in NZ, most people prefer ‘DD/MM/YY’. So it must either respect the device’s locale setting, or provide a setting for the user to change. The ‘Now’ buttons round the current time to the nearest 5 minutes and populate the relevant field.

Travel time is recorded as minutes in the ‘MinutesTravelled’ field.

The ‘Details’ screen is where the actual job details are recorded. The ‘Finish Time’ field is repeated so that the user doesn’t have to go back to the first tab to change it. This field is limited(?) to 3500 characters.

Note the drop box and associated edit button at the bottom of the screen. This feature allows engineers to add personalized text snippets that they can insert into the detail text (for commonly used phrases). This may not be necessary (very few people are using it

On the ‘Notes’ tab, are two additional text boxes. These are just free-text fields. The first, ‘Notes’ is for notes about the job intended for support staff back in the office, and not intended for the client invoice. This field is the only one that can be edited \*after\* a signature has been captured.

The ‘Equipment’ field is just that: a place for field engineers to note any equipment used on the site. This is not a common thing for Kinetics, which is why there isn’t anything more formal than this.

On the signature tab is a space for a client signature. Note the bottom of the screen also reports the current GPS coordinates and time. Additionally, the client’s email address can be recorded (So that they can receive a copy of the job record via email). The ‘return’ date is an estimate by the engineer on when a return visit will be required. The engineer can also request a change to the request’s status. The ‘Job Complete’ check box is for when a job does not get a signature (e.g. if for remote work). The engineer then just ticks the box. Otherwise, the action of signing the job automatically flags the job as complete. A ‘complete’ job is considered ready to invoice.

# Job Status.

A job’s status is from a hard-coded list, and have the following meanings:

0: Draft; any changes permissible, including cancellation/deletion.

1: Complete but not signed; changes permissible, including change back to ‘draft’.

2: Signed; changes permissible, but any change will invalidate the signature.

3: Complete, not signed, synchronized; changes on device no longer permitted.

4: Signed and synchronized; changes on device no longer permitted.

5: History; not used on device

6: Request status change explanation record; not visible on device (see source code for the ‘RequestStatusChange’ method for more information)

# Request Status

A request’s status comes from the ‘RequestStatus’ table in the database. Besides an ID and description, it has the following key fields:

DisplayOrder: this is the order in which the statuses should be displayed on a menu.

Colour: The text colour for requests with this status. This is a 32-bit value (ARGB, with the alpha channel always ‘opaque’)

BackColour: the background colour for requests with this status. This is a 32-bit value (ARGB, with the alpha channel always ‘opaque’)

IsClientStatus: whether or not requests with this status are visible on a client device

IsClientMenuItem: whether or not a client device can request a change to this status

IsReasonRequired: whether or not a reason is required to change to this status

IsNewStatus, IsInProgressStatus, IsCompleteStatus, IsCancelledStatus: largely self-explanatory. A request is ‘New’ until it is ‘Cancelled’, work is started (it becomes ‘InProgress’) or work is ‘Completed’. (Strictly speaking, a request can’t be completed without work being done on it, but it’s permitted because we don’t necessarily sync enough to notice the progression).

# GPS

We would like to record location information for the devices. Activiser has the data structures to allow us to record GPS strings ($GPRMC records), and use this information to track the whereabouts of a user from the Activiser console. We only need position recorded every few minutes.

# Synchronization

Synchronization is probably the most complicated part of the Activiser device. This is because it was designed to be failure-tolerant on unpredictable and slow networks.

Under Windows Mobile, it was further complicated by limitations of the hardware and framework.

Currently, Kinetics engineers synchronize manually, on demand. Support for background synching is built into the Windows Mobile client, but it is unreliable because of the way that merging of changes is handled by the dataset structures in the .NET Framework. It is our hope that the Android environment will improve this, and allow for quicker and more reliable sync’ing, both manual and background.

The ‘pseudo-code’ for the sync process goes something like this (from the mobile client):

1. Authenticate. Call ‘CheckDeviceID’. Result of ‘False’ means we fail.
2. Upload changes that we’ve made.
   1. Create a dataset of changes.
   2. Call UploadClientDataSetUpdates(AsJSON). This method will return a set of checksums.
   3. (optional sanity check) For each checksum returned, check the corresponding request or job; if there is a ‘checksum’ error, then the server and client have different data. The checksum algorithm is in the source code in ‘RequestCheckSum’ and ‘JobCheckSum’.
3. Download changes that have been made on the server.
   1. Call GetClientDataSetUpdates(AsJSON)
   2. Merge the results with the current data on the device.
4. Upload the GPS log
   1. Call UploadDeviceTrackingInfo(AsJSON)\*.
5. Upload the Event log (events on the device that warrant further investigation).
   1. Call UploadEventLog(AsJSON)\*.

\*Note: web methods not implemented for JSON yet.