SYBASE®
An SAP Company

WHITE PAPER

# Building Mobile Enterprise Applications with Android and SQL Anywhere®

**TABLE OF CONTENTS**

## ANDROID AND THE MOBILE ENTERPRISE

### Android and the Smartphone Market

You probably do not need to be told just how spectacular the rise of Android has been in 2010 and 2011, but just in case you do, this chart should make it clear. With Nokia's recent decision to move away from Symbian for its smartphones, Android is now clearly set to be the largest selling smartphone operating system in the world in 2011 and 2012.

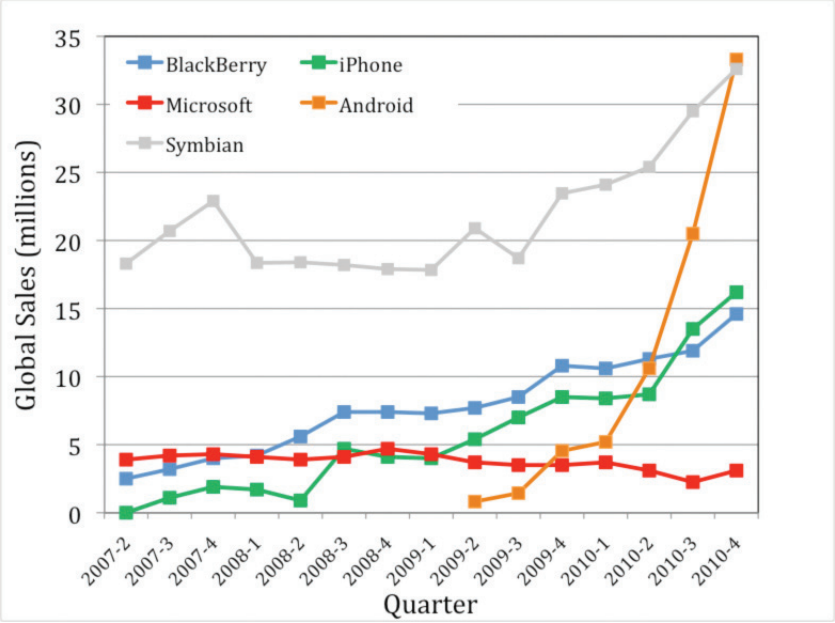



**Figure 1.** The Global Smartphone Market

### Smartphones in the Enterprise

The chart in Figure 1 is dominated by consumer sales, but smartphones are also entering the enterprise at high speed. Compared to traditional enterprise handhelds they have a number of benefits but also some drawbacks.

- User adoption is not a problem for consumer-oriented smartphones, particularly compared to the boxy and heavy enterprise handhelds that have been the mainstay for line-of-business applications. On the other hand, for the heavy duty use that some line-of-business applications demand, smartphones are a lot easier to break than ruggedized devices, and battery life is shorter.
- A drawback for consumer devices is that for a large number of high-value applications such as direct store delivery (where you need signature capture), or warehouse applications that need scanners, you can buy enterprise handhelds with these peripherals built in. If you want to use a consumer device to carry out such tasks you often have to use third party Bluetooth peripherals, which adds to the price.
- Compared to enterprise handhelds, smartphones are cheap, but it's important to remember that you only get the subsidized price for the first device — if you break it and need a replacement, the cost might be higher. Of course, from an enterprise point of view the cheapest devices are those that they don't have to pay for, so there is a big trend to bringing employee owned devices into the enterprise.

So while there's no doubt that consumer devices are working their way in and rapidly, they aren't the right choice in every circumstance. Among the applications best suited to smartphones are those that can be expressed as surveys — manual data entry with a structured, and potentially very complex, form. Field sales is another obvious category, and time and expense management is another application that is important for many mobile workers.

Tablets are also moving quickly into the enterprise. Their larger form factor makes data entry easier, and their long battery life compared to laptops makes them useful for delivery applications, for retail environments where there is a need for customer-facing applications, and for healthcare applications. Mobile inspections, where there is a large amount of structured data entry on the road, are also good candidates for tablet applications.

**Mobile Data Management**

If there is a distinguishing feature to many enterprise applications, it is that they rely on and work with data that is hosted at an enterprise site. If you are doing time and expense reporting, you need those numbers to go back to the enterprise expense reporting system. If you are taking orders, or updating notes about customers, they belong in the enterprise system.

As a developer, you have two top-level options for how to manage data in your application. You can fetch it over the network when you need it and send updates back when the work is done, perhaps by HTTP posts to a web service that updates the enterprise system. This works for some applications, but the approach does run into two issues: *availability* and *latency*.

The other choice is to store the data locally, as email and calendar applications typically do. This gives you quick response times and an always-available capability because your application can always work with data stored on the device. The catch, because nothing comes for free, is that you have to worry about device *horsepower* and about *consistency*. The point is not whether one model is better than the other, but to make sure you choose the right one for your application. SQL Anywhere helps people develop applications that use a local data model.

Let's look at the issues for network storage.

*Availability* is often called the "airplane problem" because for many of us being in an airplane is the longest time we go without network availability. This name both understates and overstates the case. It overstates because the airplane problem itself is on its way to being solved as WiFi starts to become available on flights, and understates because there are a lot of terrestrial places where networks are spotty or completely unavailable. Sybase was involved in the 2010 US Census address canvassing operation and the application there had to be used all across the USA, including some very remote areas. Another customer we work with sends delivery trucks to grocery stores and supermarkets. The drivers need to carry out business tasks within the supermarket, such as taking orders, and they have found repeatedly that they cannot rely on network coverage inside the building. It varies from store to store, but some of the bigger and cheaper big box stores make excellent Faraday cages and block out networks very effectively. And of course, many of us have been in conferences in the basements of hotels and found our phone battery used up at the end of the day as the phone continually loses and then tries to re-establish network connectivity as we prowl the basement in search of the next talk.

So availability is a big problem for some applications, and there is not much that you, as an application developer, can do about it except to decide whether you can live with it or not, and whether to wait until coverage reaches the places you care about. But availability is not the only problem for network-based storage.

*Latency* for internet connections from desktop and laptop computers these days is remarkable, and yet it is still a big issue that major Internet companies spend money and time improving. Major companies invest hugely in reducing latency because they have found through watching customer behavior that every tenth of a second matters to customer satisfaction. Google tried delivering more results per page in its web search and found usage dropping off because those extra results meant the request was a bit slower: half a second delay caused a 20% drop in traffic. Amazon tried experiments from its web site delaying page loading and found that "even very small delays would result in substantial and costly drops in revenue".

Mobile networks are, of course, much higher latency than Ethernet-based Internet connections. Try a search over a 3G network and you'll find the simplest and fastest-responding web page — Google for example — can take more than a second to fetch and load. And that's one round-trip. If a page has several separate requests to make then this time is multiplied. So if you have a data-rich application that is going to make several requests to fill a page for your application, then you have your users waiting multiple seconds every time they interact with your application, and that makes for frustrated users.

If you're thinking of local storage, then you have a couple of issues: horsepower and data consistency. Of these, let's look at horsepower first. Can the device handle the data processing tasks you give it?

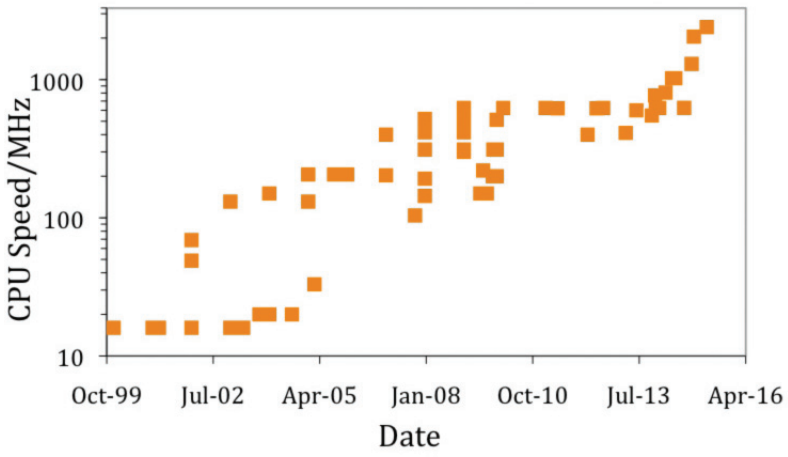



**Figure 2.** Mobile Device CPU Speed

There's no one-size-fits all answer here, but it is worth reminding ourselves how far the mobile world has come. Figure 2 shows a chart with a selection of devices over the years. At the bottom left is the Palm Pilot, with its 12 MHz chips. There is a steady growth and then a plateau from 2005 to 2009, at 625MHz. During this period the highest performing mobile devices were the enterprise handhelds made by Symbol (now Motorola), Intermec and others. They had big batteries and were able to sustain significant CPU speeds, but there wasn't a big enough market to drive CPU performance beyond a certain point, and progress stalled. The lower values in this period are from early smartphones — early BlackBerrys and Nokia phones. More recently smartphones approached, caught up to, and finally overtook the enterprise handhelds. Around the beginning of this year the first Gigahertz chips for smartphones started appearing and now there are several Android phones with Gigahertz chips, some with dual cores, and we can expect many more during 2011. The end result is that there has been a hundred-fold increase in processing power over the last 15 years.

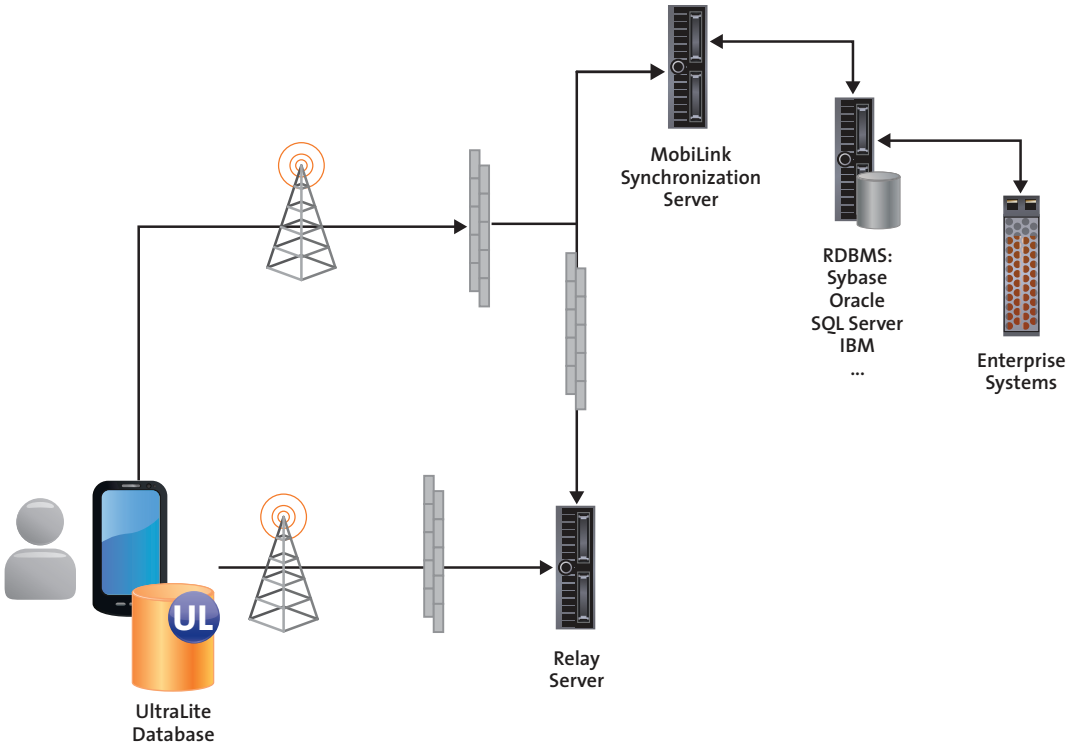
The other issue you have to deal with if you choose local data storage is consistency. Having a database on device is only the beginning of the mobile data management solution. Data is not just kept on device but must be shared with some other store — perhaps a web-based service, or an enterprise system, or a big database. So if data is being moved around and people are making changes to it, and you can't lock all the other databases every time a change is made, then you have to worry about keeping data up to date and consistent throughout the whole widely-dispersed system.

Unlike latency and availability, consistency is an issue that you can do something about: as a mobile data management product, SQL Anywhere solves the consistency problem.

## SQL ANYWHERE: SOLVING MOBILE DATA MANAGEMENT PROBLEMS

SQL Anywhere is database management software built for enterprises, and for ISVs to embed in their applications, and for use as high-performance mobile data management software.

As you can see here, SQL Anywhere provides a mobile database (bottom left), a relay server for managing traffic, and a scalable synchronization server called MobiLink™ that handles business logic, security, and keeping mobile databases in sync with enterprise data stores.



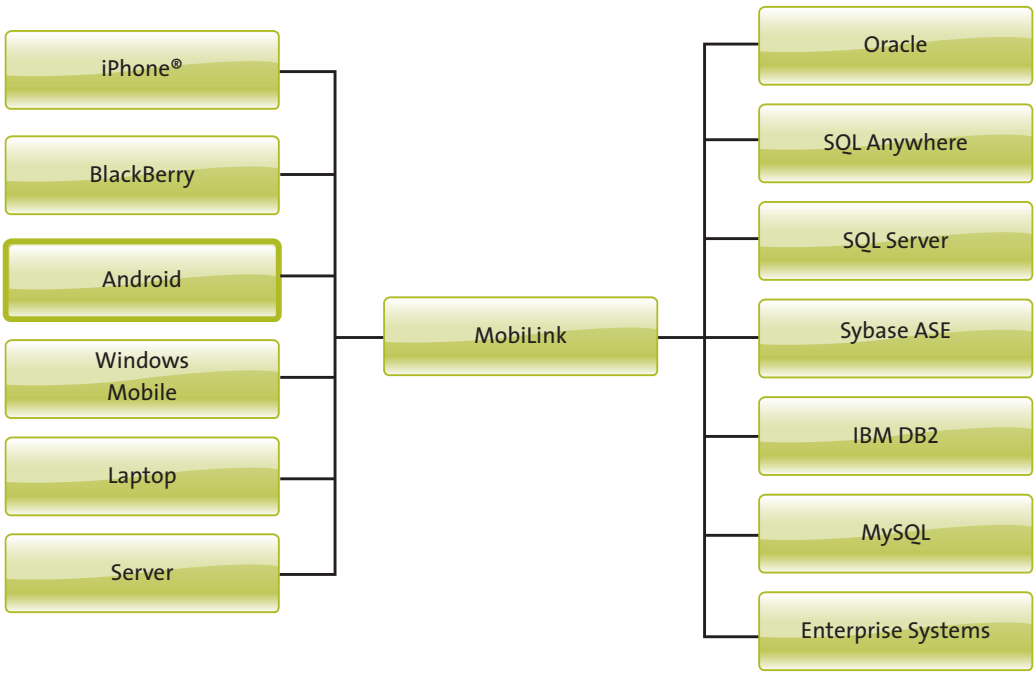

SQL Anywhere solves the problem of local data storage and of consistency: it provides the data infrastructure to keep large deployments of mobile applications running. Some installations have all mobile devices talking to a single central server, and others have regional servers that form a mid-tier layer between the devices and the central database.

Mobile applications can be just as complex as any desktop application. Some require hundreds of tables, and the number of screens in a field service application, for example, can easily be in three figures. To support these applications we have had to build a lot of flexibility into SQL Anywhere, with conflict resolution, flexible data partitioning being just the beginning. We provide for denormalization on the client, high-priority data subsets, and more. The system is end-to-end encrypted, and we provide tools not just for developers, but also for IT professionals who need to estimate hardware needs, test performance under heavy load, and monitor live installations.

The focus of SQL Anywhere's mobile technology is to provide optimized, high performance database management and data movement software that ISVs can embed in the most demanding of applications.

SQL Anywhere is a mature product that has built a substantial list of sound customers and successful deployments, some with tens of thousands of devices. It has been used by direct store delivery drivers, by letter carriers, and by census workers in large deployments that must meet stringent security and availability metrics.

Our job is to link mobile devices to enterprise systems, and particularly to back end databases. With that in mind, here are the platforms we support.




On the left are the mobile devices, with Android highlighted as a new platform. At the right are the supported enterprise systems, each of which is tested with every release of SQL Anywhere.

**DEVELOPING APPLICATIONS**

The remainder of this paper is taken up with device-side programming. How do you develop applications that access data on device, and that exchange data with the enterprise?

SQL Anywhere provides UltraLite, its mobile database and synchronization client, as a Java library and an associated shared object (C++ library) that you can access from the Android SDK.

Here are the main classes you will be working with as you develop your application. If you've done much database development, many of these will look familiar.

- The *Configuration* object sets up the properties of your database (if you are creating a new one) or your connection. You use this object to specify the file name, a cache size if you want, a password, and so on.
- The *Connection* object is where everything starts for any operations you wish to carry out on the database. You can have multiple open connections at once, but for many applications you just need one.
- To execute a command, you create a *PreparedStatement* and execute it. If it's a query you get a *ResultSet* back and you can loop over the results.
- The synchronization parameters may be a bit less familiar. *SyncParms* sets up the basic data, like where the host is, some authentication credentials, and so on. During a synchronization you can use a *SyncObserver* to show progress, and after a synchronization you can look at the *SyncResult* to see what happened.

There are other classes as well, but these are the ones you will see most often.

### A Database Connection

Here is a singleton class to handle making a database connection, so that you can ask for a Connection wherever you are in the application without using up additional resources. The Android Configuration object takes an application context as an argument, and this is discussed below.

```
import com.ianywhere.ultralitejni12.*;
class Database{
    private static Database _instance = null;
    private static Connection _dbconn;
    Context _context;

    private Database(Context context){
        _context = context;
        if( _dbconn == null ){
            ConfigFileAndroid config =
            DatabaseManager.createConfigurationFileAndroid (
            "mydb.udb", _context);
           config.setPageSize(8192);
           try {
               _dbconn = DatabaseManager.connect(config);
            }
           catch {
               _dbconn = DatabaseManager.createDatabase(config);
            }
       }
    }
    protected static Database getInstance(Context context){
        if(_instance == null){
            _instance = new Database(context);
        }
        return _instance;
    }
}
```

The application calls *Database.getInstance(this)* from a class that extends Activity to get a database connection.

### Data Access

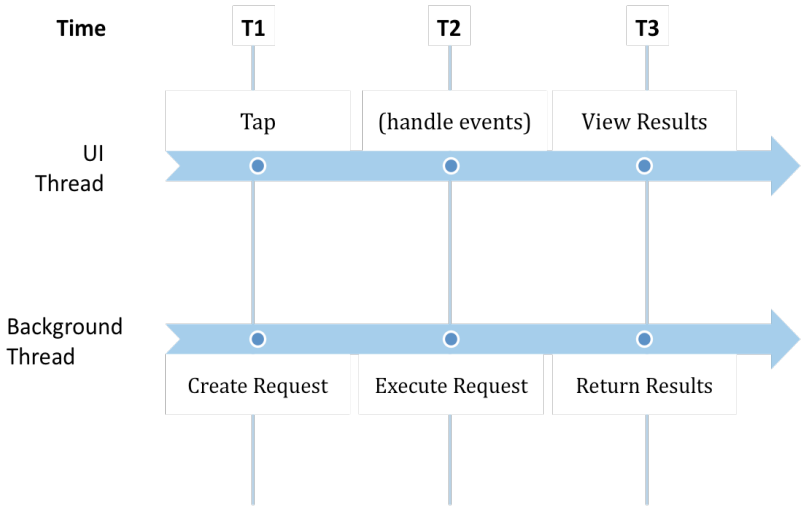
Data access will look very familiar to anyone who has done database programming. Here is a simple query:

```
String sql = "SELECT test_id FROM test";
PreparedStatement ps = _dbconn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
List<Integer> testChoicesList = new ArrayList<Integer>();
while (rs.next()) {
    testChoicesList.add(rs.getInt(1));
}
rs.close();
ps.close();
```

This example puts the values from the result set into an ArrayList, suitable for binding to an Android ListView or other visual control.

**Database Tasks and Threading**

Many mobile applications use a common threading model: a UI thread handles user interaction, and a background worker thread does potentially time-consuming tasks. The background thread means you don't lock up the device, in case a phone call comes in for example. Database access is a good candidate for background thread operation.




The user taps the screen, and this creates a request on the background thread, the UI is freed up to handle other user actions, and when the request is done the UI gets notified so it can display the results.

That sounds tricky because it sounds like you have to manage your own threads, which can be challenging. Fortunately, Android comes to the rescue and you don't need to do thread management yourself. Instead, you use a special class called an *AsyncTask*, which does it for you.

*AsyncTask* provides a few methods that you must implement, and Android takes care of where they get executed.

There is an *onPreExecute* method, which is executed on the UI thread. This means that if you want to manipulate the UI before starting your background task (disabling some controls, perhaps, or showing a progress bar) you can do so here.

The *doInBackground* method gets called on a separate thread. Here is where you will access the database and get your results, or perhaps synchronize data with a server. You can use *publishProgress* and *onProgessUpdate* to display progress status if it is a really long-running task.

Finally, the *onPostExecute* method runs on the UI thread, and this lets you do whatever you need to prepare the UI, like display the results or re-enable controls.

Information must be handed between these methods, and there are three separate places for putting this data. The first is a set of parameters (may be the query, for example). The second is a Progress value, which may be an integer or a set of integers showing how far the operation has got, and the final is the return value of the *doInBackground* method, which may be an array or collection of results.

Here is a simple database task.

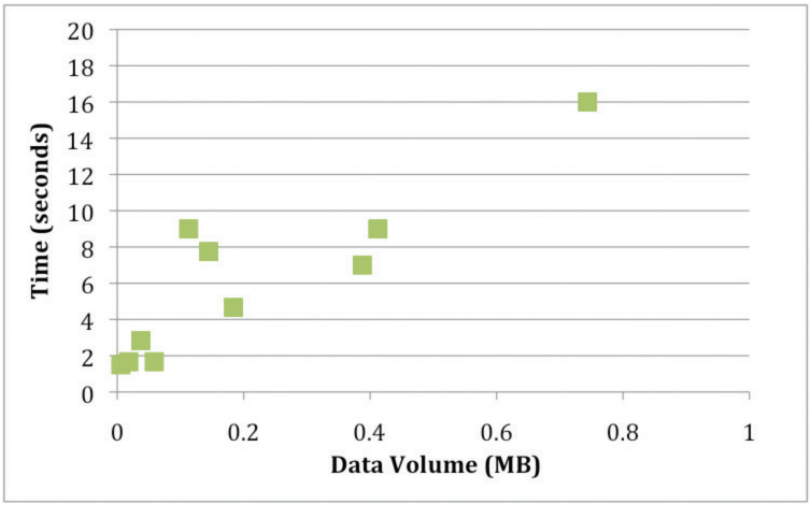
```
static class MyDatabaseTask extends AsyncTask<MyApp, Void, String> {
  MyApp _myapp;
  protected void onPreExecute() {
       Log.d(MyApp.TAG , “onPreExecute”);
  }
  protected String doInBackground(MyApp... myapp) {
       _myapp = myapp[0];
       boolean success = false;
       Database db = Database.getInstance(_myapp);
       Log.d(MyApp.TAG , “doInBackground” );
       try { db.executeRequest(); }
       catch {
        Log.e(DLTest.TAG, e.toString());
        return “Database creation failed”;
       }
       return “Database created”;
  }
  protected void onPostExecute(String displayText) {
       Toast.makeText(_myapp, displayText, Toast.LENGTH_SHORT).show();
  }
}
```

### Data Synchronization

Data synchronization is a bit more unusual, but here is what you need.

```
SyncParms sp = _dbconn.createSyncParms(
       SyncParms.HTTP_STREAM,
       username, version);
StreamHTTPParms httpParms = sp.getStreamParms();
httpParms.setHost(“relayserver.sybase.com”);
httpParms.setPort(80);
httpParms.setURLSuffix(suffix);
sp.setPublications(testName);
sp.setSyncObserver(new DLTestSyncObserver());
_dbconn.synchronize(sp);
```

Data synchronization on modern phones is a speedy operation for incremental changes to data. The following chart shows that small volumes (100KB or less) can be synchronized across 3G networks in just a second or two, while larger volumes are also comfortably handled.




**DELIVERING DATA-RICH APPLICATIONS**

What this means is that using SQL Anywhere, developers can build and deliver effective data-rich mobile applications for the enterprise. Large-scale deployments of high value applications that need to access enterprise data, modify it, and integrate with other systems, that deliver a responsive user experience and work anywhere, whether there is data coverage or not, are practical to build and deploy using SQL Anywhere's field-tested technology and its new Android support.

Sybase, Inc.
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902
U.S.A
1 800 8 sybase

www.sybase.com



SYBASE®
An SAP Company