1. Introduction to Mobile Computing

Device Categories:

- Mobile Phones;
- Smartphones;
- PDAs;
- Wearables (aka privacy leeches);
- Handhelds;
- Tablets;

Apps can:

- Access Storage;
- Use Internet, Bluetooth, NFC;
- Be interacted using touch input, multitouch or gestures;
- Access sensors such as GPS, compass, and acceleration;

Can be used in enterprises, or for learning/entertainment/social interaction;

Platforms:

- Android;
- iOS (Apple);
- Blackberry;
- Symbian;
- Chrome OS;
- Others;

Frameworks:

- Java
- PhoneGap, Ionic (PWAs);
- Zámárin
- React Native

App types:

- Web Apps: sites disguised as apps;
- Hybrid Apps: web technologies but encapsulated to native container;
- Native Apps: uses the OS specific framework;

Has a three tier has a presentation layer, but the screen outputs images and serves as input to the "Business Layer", which alters the data on the Data Layer.

On a connected application, the app connects with the server on a wireless connection and syncs up all the data required. The server can send notifications to the client this way.

Apps can be suspended from other apps, need to know how to recover and handle gracefully this situation.

Apps store locally information in several formats, including relational databases.

Connectivity is not 100% reliable, and is used to synchronize data or other functionalities that require being online.

Mobile design patterns:

- Interaction:
  - Save information when user leaves the screen;
  - Auto save input periodically;
  - Avoid user input especially on text;
  - User should know all actions and options available
  - Operations longer than 1s should be async
  - Credentials should be asked only once
- Presentation:
  - No hard-coding layouts;
  - Use recommended look-and-feel of the native platform;
  - Use lists and vertical scrolling, no horizontal;
- Behavioral:
  - Fetch data before hand, in case connectivity fails later;
  - Save state when app goes into background;

There are lots of devices with different layouts, and there are several cross platform approaches:

- Abstraction layer (generic API that calls native platform specific functions, like Xamarin or Titanium Mobile);
- Web shell (a web app in disguise, like PhoneGap);

PhoneGap apps are web apps, that have plugins for different functions (sensors, storage access, contacts, etc), that call the native OS APIs.

Titanium compiles different builds to create "native" apps.

Xamarin:

- .NET APIs call Android Binding, that call Android files. It also calls the Linux Kernel;
- Android files call Dalvik, a Java-based virtual machine that runs .apk files;
- Dalvik connects with the Linux Kernel and vice-versa;

React Native:

- Has a Reactive Core that saves the current states, and updates the Web app or Mobile app when state changes;
- Mobile and Web apps can trigger an action that prompts a state change in the Reactive Core;

2. Android Basics

Android:

- Created by Google and Open Handset Alliance in 2008
- Based on the Linux kernel at a lower level, and on a higher level on a Java virtual machine;
- Versions: 1.0, 1.1, 1.5 (cupcake), 1.6 (donut), 2.0, 2.1 (éclair), 2.2 (froyo), 2.3 (gingerbread), 3.0-2 (honeycomb) (tablets), 4.0 (ice cream sandwich), 4.1-3 (jelly beans), 4.4 (kitkat), 5.0-1 (lollipop), 6.0 (marshmallow), 7.0-1 (nougat), 8.0-1 (oreo), 9.0 (pie), 10.0 (Q) (sei lá se isto sai)
- Supports a lot of hardware with different sensors, and different screen resolutions;

Android Features:

- Framework based in reuse and extension patterns;
- Optimized virtual machine named Android Runtime (ART);
- Has a browser named Blink;
- Supports 2D and 3D Graphics from OpenGL 1.1 to 3.2;
- Local databases use SQLite;
- Supports standard multimedia formats;
- Has 3G/4G, Wifi, Bluetooth and NFC;
- Has Camera and sensors;
- Applications are developed using Java or Kotlin;

Software Architecture (aplicashõesh têm camadash, chebolash têm camadash):

- *App Layer:* a camada mesmo da aplicação em si, onde o utilizador interage;
- *Application Framework Layer:* o código de Java que compõe a aplicação, e faz uma gestão de mais alto nível dos serviços de Android;
- *Library Layer:* o código compilado de C++ para o dispositivo, incluindo o ART;
- *Linux Kernel Layer:* o sistema operativo em si, faz o management de processos, memória e aceder ao hardware como sensores;

Build Process:

- .java files are compiled into .class files;
- .class files are compiled into .dex files to be read by the Dalvik VM or ART;
- .dex files are converted into an .aapk, a compiled version of the source code;
- .xml resources and other assets are compiled with the .aapk file to generate an .apk file, ready to be executed;

Android Components:

- *Activities:*
    - Contain the Interface and execute a well defined task;
    - There can be many Activities, and they may reference each other;
    - One of them must be the starting Activity;
- *Services:*
    - Don't have interface, are executed in the background;
    - Application can communicate with a service through an API;

- *Broadcast Receivers:*
  - Applications can send *broadcast* notifications, receivers intercept and react to them;
- *Content Providers:*
  - Allow access to data collection to other apps, with an access interface;

Application planning can follow the flow of the app.

Activities are launched by calling Intents, Back button goes back to the previous Activity in the stack (Android maintains stack of activities);

Component Activation:

- *Activities and Services:*
  - Activities and Services are launched using an Intent;
  - An Intent can reference a specific class (explicit, M-Rated), or can specify an action, like opening contacts application (implicit);
  - Activities and services can specify intent-filters, outlining which intents they can catch;
- *Broadcast Receivers:*
  - Broadcast intents contain a message to be delivered to the matching receivers;
  - Sent via *sendBroadcast(Intent)*, the message is specified inside the Intent object, and the receiver will catch it using *onReceive(Context, Intent)*;
- *Content Providers:*
  - Must provide an *authority* in the manifest;
  - Must have a name for its data collection;
  - Support CRUD operations;
  - Activated through a ContentResolver object, obtained with *getContentResolver()*

Intents:

- Have a name, and can have more data with *setData()* (URI, category, messages), or *putExtra()* for a bundle of information (putExtra(String name, ... value));
- A class in Android API;
- Many predefined intents in the Android API, like Intent.ACTION_DIAL;
- Invoked activities can return results, with *startActivityForResult()*, like asking the camera to take a picture and return it to the main application;

Example code:

```
@Override
public void onActivityResult(intrequestCode, intresultCode, Intentdata) {
    super.onActivityResult(requestCode, resultCode, data);
    switch(requestCode) {
        case (MY_CHILD_ACTIVITY) : {
            if (resultCode == Activity.RESULT_OK)
                StringreturnValue= data.getStringExtra("some_key");
            break;
        }
    }
}
```

3. Android Application Architecture and Development

Applications usually run on a single process, and that object is always in memory.

Processes have priorities depending on their states:

- Active process (in interaction) has critical priority;
- Visible and started service processes have high priority;
- Background or empty processes have low priority;

Android state:

- Active: running and in interaction;
- Paused: visible;
- Stopped: non visible;
- Inactive or destroyed: without an instance object;

Activity lifecycle (arrows represent possible transitions):

- ```
  Starting -> Running -> Paused -> Running
                                -> Destroyed
                                -> Stopped -> Running
                                           -> Destroyed
  ```

Starting an app:

- Summoned with *onCreate()*;
- Override the method to support what we want, like setting the view that we want with *setContentView()*;
- To find the View we use *findViewById()*;

Activity Navigation:

- Back button goes to the previous activity in the stack;
- Defined hierarchy of parents;

Resources organized in /res folder:

- *drawable*: bitmaps and other graphics;
- *anim*: XML animations between configurations;
- *color*: colors for interface;
- *layout*: Views and ViewGroups defining an interface screen;
- *menu*: options and context menus;
- *values*: values 😃 strings, arrays, colors and whatnot;
- *xml*: other XML files;

ViewGroups:

- Specify how to organize contained Views;
- Examples are:
  ```

    - *LinearLayout*: Organizes in a single row or column;
    - *RelativeLayout*: Organizes views relative to others;
    - *TableLayout*: Rows and Columns similar to rows and tables in HTML;
    - *FrameLayout*: All descendants positioned relatively to the top left corner of the parent;

match_parent: size matches ancestor size; **wrap_content:** size matches content;

A style is a resource for views and activities. There are preset styles available for Android called themes.

More resources:

- Language;
- Screen size, aspect, orientation;
- Versions;
- *Hour*: night, notnight for whatever reason;

Icons should be multiple resolutions.

Events and Listeners:

- User performs action (input);
- This action can call callback on activity or event listener on handler, that calls callback on activity;

To register events there need to be listeners, that handle:

- *onClick()*;
- *onLongClick()*;
- *onFocusChange()*;
- *onKey()*;
- *onTouch()*;
- *onCreateContextMenu()*;

To define a listener, the Activity needs to do *implements onClickListener* for example, and have a function inside titled *public void onClick(View v)*.

There can also be anonymous listeners, by setting listener to a *new onClickListener()*.

Tabs in Android are done combining TabHost, TabWidget and FrameLayout.

Menus:

- Option Menu (like the three dots on a corner);
- Context Menu (initiated by a long tap, like replying on a Messenger conversation);
- Are defined as a resource in /menu, and contain menu items and submenus;

Dialog boxes:

- Are prompts that show up, like "Do you want to give this app permission X? No Yes" options;
- Name of the class is android.app.Dialog;
- Can change title using *setTitle()*, and change contentView using *setContentView()*;
- Appears calling *show()*, and terminates using *dismiss()*;
- Presets are AlertDialog, ProgressDialog, DatePickerDialog, TimePickerDialog

Dialogs should be reused.

Application preferences are stored in an .xml file. Shared Preferences are ways to store state even after shutting down application.

Before shutting down, application will call *onSaveInstanceState(Bundle)*, and on restoring will call *onRestoreInstanceState(Bundle)*. Used to handle shutting down application or switching contexts (like changing orientation).

On device rotation, Android destroys and then recreates activities.

Features and permissions to use need to be written in AndroidManifest.xml.

4. Android Fragments

Activities define full screen interface, but with a larger screen real estate (like tablets), it becomes useful to have fragments, which are basically sub-activities.

Fragments:

- Have their own layout (have a `<fragment>` tag);
- Have their own lifecycle;
- Are used inside an activity, using the same thread and context;
- But can respond to the back button;

Lifecycle:

- ```
  Starting -> Created -> Active -> Inactive -> Destroyed
                                            -> Created
  ```

Lifecycle callbacks:

- *onInflate(Activity, AttributeSet, Bundle)*: Called when the activity sets its content layout and a fragment is in it. AttributeSet contains the attributes for the activity layout. Fragment is not yet attached to activity;
- *onAttach(Activity)*: Activity with fragment is attached, access it using *getActivity()*. Set arguments with *setArguments()* and retrieve them using *getArguments()*;
- *onCreate(Bundle)*: Called on parent Activity *onCreate()*;
- *onCreateView(LayoutInflater, ViewGroup, Bundle)*: Returns the inflated view, if no container (ViewGroup) in layout then return null.
- *onActivityCreated(Bundle)*: Everything is finished here;
- *onStart()* and *onResume()*: self-explanatory;
- *onPause()*: First to be called, should stop playing sounds;
- *onDestroyView()*: Fragment is being killed;
- *onDestroy()*: Fragment is no longer in use, but still in memory;
- *onDetach()*: Fragment does not belong to the activity, resources are freed;
- *onSaveInstanceState(Bundle)*: same as the other

Fragment Manager:

- Manipulates Activities and Fragments (including back stack);
- Accessible through with *getFragmentManager()*;

Dialogs don't respond to lifecycle events automatically, so use a DialogFragment.
  ```

5. NFC Communications in Android

NFC (Near Field Communications):

- Standard for short-range wireless communications;
- Simple and safe 2-way interactions in close proximity (< 1cm);
- NFC tags can contain data;
- Card emulation with secure element;

NFC Modes:

- R/W tags;
- P2P communication between devices;
- Card Emulation, remote reader talks directly to a secure element applet using NFC;

Core Classes:

- NFCManager;
- NFCAdapter;
- Tag Technology Classes;
- HostApduService;

Hardware includes a secure element, may be the phone SIM.

Messages follow NDEF (NFC Data Exchange Format), called *NdefMessage* class in Android. Constructor accepts array of *NdefRecord* objects. Access records using *getRecords()*.

NdefRecord constructor accepts a TNF field, type, id and payload data (in byte[]). TNF field can be `TNF_ABSOLUTE_URI`, `TNF_WELL_KNOWN` or `TNF_MIME_MEDIA`.

Requirements:

- Hardware (duh);
- API version 10 at least;
- Manifest must explicitly state that it requires NFC permission;

Intent Filters must be declared in the Manifest, and they can be:

- *android.nfc.action.NDEF_DISCOVERED*;
- *android.nfc.action.TECH_DISCOVERED*;
- *android.nfc.action.TAG_DISCOVERED*;

In order to write a tag, the device needs to be discovered first by sending a message first. Code below:

```
if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
    Tag detectedTag = getIntent().getParcelableExtra(NfcAdapter.EXTRA_TAG);
    // PREPARE THE NDEF MESSAGE TO WRITE  ....
    writeNdefMessageToTag(newMessage, detectedTag);
}
```

writeNdefMessageToTag basically gets the Ndef from the tag, connects, writes a message with *writeNdefMessage(message)*, and then closes the connection.

Receiving NFC messages generate an intent. This will create a new activity instance matching that filter, even if an activity already exists.

To avoid creating new instances, use *singleTop* in launchMode property in the manifest. This will call the *onNewInstance()* method instead of creating an entire new instance.

To send P2P messages, two implementations need to be defined:

- *onNdefPushComplete()*: called when NFC Message is delivered;
- *createNdefMessage()*: called before message is sent, to create it;

Another way to create a message is to call *setNdefPushMessage()* usually in the *onCreate()* callback.

In order to get the *NfcAdapterfor* the device we can go through the *NfcManager*;

Card Emulation:

- Card reader talks directly to an Android Service emulating a secure environment in the host;
- Communication based on AID and APDU packets;
- The service derives from HostApduService;

Needs to set intent-filter in Manifest *android.nfc.cardemulation.host_apdu_service*.

6. HTTP Web Services and Asynchronous Execution

Web services use HTTP (*HttpURLConnection* class).

Needs a separate thread and manifest permission *android.permission.INTERNET*

After API 27 only HTTPS traffic is allowed by default, use `usesCleartextTraffic="true"` to bypass restriction.

Create a new thread (implements *Runnable*) Requires Configuration, Payload and Response;

The Activity is in the main thread, it communicates with another thread by attaching a handler to the thread. This handler will deal with all communications.

It is possible to create an Async task class that takes care of everything.

7. Graphical and Media Android Interface Elements

Soft Keyboards:

- Most devices don't have keyboards;
- Soft keyboards are controlled by the input method editor;

Many input types on EditText determine the keyboard (numeric, date, email...);

Images are drawn to the screen using the *Activity View*, with the *onDraw(Canvas)* method.

Canvas defines primitives, with an instance of *Paint*. Geometric instances are defined using a Path (use them with *canvas.drawPath()*).

Full custom views can be created, but need to override several methods from the View class, like the *onMeasure()* or the *onDraw()*.

MediaPlayer class can play audio, asynchronously, and supports a lot of media formats.

VideoView encapsulates a video to be played in an activity, supports MP4, 3GP and AVC. Set video path with, well, *setVideoPath()*. Start with the aptly named *start()* method.

Preview Camera in Activity:

- Add *SurfaceView* to Activity layout;
- Get *SurfaceHolder* from SurfaceView;
- Add *SurfaceHolder.callback* object to SurfaceHolder;
- In the *surfaceChanged()* callback, *setPreviewDisplay()* and configure camera;
- *startPreview()* and when stopped, *stopPreview()* and *release()*;

OpenGL:

- Library for 3D graphics, OpenGL ES used for mobile;
- To use OpenGL, use with *setContentView()* a GLSurfaceView, that creates a GLRenderer;

Many types of touch events, so they are abstracted into click, long click, etc.

To intercept all input use with *setOnTouchListener()* a OnTouchListener. It intercepts MotionEvent objects, with a specific action like and properties of the touch (like position);

GestureDetector detects and trigger one finger gestures, whereas ScaleGestureDetector detects the pinch two finger gesture.

8. Sensors in Android

Sensors:

- *Location*:
  - Lat and long, altitude, accuracy;
  - Obtained from GPS (accurate, but expensive) or derived from Wi-fi access points or mobile communication towers (less accurate, but less expensive);
  - Accessible through `GPS_PROVIDER`, `NETWORK_PROVIDER` or `PASSIVE_PROVIDER` (another app);
  - App must declare `ACCESS_COURSE_LOCATION` and/or `ACCESS_FINE_LOCATION` in the Manifest;
- *Movement (speed)*: `TYPE_ACCELEROMETER`, `TYPE_GYROSCOPE`;
- *Orientation*: `TYPE_MAGNETIC_FIELD`;
- *Environment*: `TYPE_AMBIENT_TEMPERATURE`, `TYPE_LIGHT`, `...PRESSURE`, `...PROXIMITY`, `...RELATIVE_HUMIDITY`;

Location Classes:

- *LocationListener*: listens to location events, like location changing, provider being disabled or enabled...
- *LocationManager*: a service for requesting locations and selecting a provider, accessible through *Context.LOCATION_SERVICE*. Can request updates periodically based on distance or time, but should only request them if the activity is in the foreground;
- *Locations*: delivered to *LocationListener*, with latitude, longitude, time and provider, and more information;

Sensors can be physical (give actual measurements) or synthesized (process measurements to calculate another quantity). Synthesized sensors are `TYPE_GRAVITY`, `...LINEAR_ACCELERATION`, `...ORIENTATION`, `...ROTATION_VECTOR`, `...SIGNIFICANT_MOTION`.

To get all sensors, use `TYPE_ALL`.

Sensors have a range and resolution, rate of measurement and power consumption.

Sensor Classes:

- *SensorManager*: service accessible through Context.SENSOR_SERVICE, knows all sensors, registers SensorEventListener;
- *SensorEventListener* or *TriggerEventListener*: Interfaces that respond to sensor measurements;
- *SensorEvent:* a measurement;

Noise Processing:

- Some sensors produce noise, use filters (low pass, high pass, band pass, Kalman filters).

9. Android Services and Other App Components

Broadcast receivers were mentioned in summary #2, can be declared by the `<receiver>` tag.

Services, extra information:

- If service not in memory started with *onCreate()*;
- Any client can start service asynchronously, with *startService()* which calls *onStartCommand()*, and terminate it with *stopService()*, which calls *onDestroy()*, or the service kills itself with *stopSelf()*. Tragic, really;
- Services are freed when stopped, or Android needs the memory. In this case, services can be brought to memory again;
- Can also be created with by a connection (bind);

IntentService is a special Service subclass that creates a worker thread;

Remote call services:

- Invoked using RPC (Remote Procedure Call);
- Standalone in their own processes;
- Activated (brought to memory and invoked) through *bindService()* and freed when last client calls *unbindService()*;
- *onServiceConnected()* tells when a service is ready to be called, and *onServiceDisconnected()* tells that it's not available;

Notifications:

- Appear on the status bar;
- Extended status drawer for more information;
- Notifications are a system service, called *Context.NOTIFICATIONSERVICE*;
- Access that service with a NotificationManager;
- Customize notifications by using a *Notification.Builder()*, and then specifying the characteristics like icons, text, things like that;

Alarms are another service, accessible through *Context.ALARMSERVICE*, and then modifying it using a AlarmManager. Can be repeating with *setRepeating()*;

10. Introduction to Apple iOS

Iphone:

- Appeared in 2007;
- Multi-touch display with gesture recognition;
- Sensors, camera, wireless communications;

iOS derives from UNIX;

iOS provides the APIs for the developer to use, serving as a bridge between the hardware and software;

iOS API Layer:

- *Cocoa Touch*: used to handle input events, access tp sensors, camera, etc;
- *Media*: everything graphics, audio and video;
- *Core Services*: collections, data and time, application bundle, and other things, such as SQLite, security certificates, and XML stuff;
- *Core OS*: threads, networking, files, etc.

Apps use MVC (Model-View-Controller):

- Model has objects and their state, Controller takes care of event loop, and feeds information to the View.

ViewController:

- *IBOutlet* represents an object in the interface builder;
- *IBAction* are handlers of events generated in interface objects;

11. Introduction to Xamarin

Best performance and experience (API coverage and performance) on a native app, but worst portability across platforms.

Best portability in a web app, but worst performance and experience.

Hybrid app is a wrapper that contains a "web app" and transforms it into a native app, middle ground in performance, portability and experience.

`Xamarin.Forms` simplifies UI for all platforms, sharing the same app logic and most of the UI code.

Can produce native apps to iOS and Android using Ahead Of Time and Just In Time compilation respectively.

Supports all native APIs.

Uses Visual Studio and Windows (bleugh).

Pages: `ContentPage`, `MasterDetailPage`, `NavigationPage`, `TabbedPage`, `TemplatePage`, `CarouselPage`.
Layouts: `StackLayout`, `AbsoluteLayout`, `RelativeLayout`, `GridLayout`.

Probably the same as in Android.

Views: `ActivityIndicator`, `BoxView`, `Button`, `DatePicker`, `Editor`, `Entry`, `Image`, `Label`, `ListView`, `Picker`, `ProgressBar`, `SearchBar`, `Slider`, `Stepper`, `Switch`, `TableView`, `TimePicker`, `WebView`.

Has Documentation (wow)

More in-depth information about every single page in summary #12.

12. Xamarin Interface and Navigation

More Zámárin:

To configure a VS Project, install Xamarin.Android for Android, Microsoft.NETCore.UniversalWindowsPlatform for UWP and Xamarin.Forms for everything.

Xamarin.Forms interface is in common project and is used by all platforms. Each interface is composed of Pages and a singleton App.

First page is the MainPage property of the App (similar to launchPage in Android Manifest);

Pages:

- Contain a Layout or View;
- Layout can contain multiple Views or Layouts (nested);
- Some pages (like TabbedPage or CarouselPage) can contain sub-pages;
- Only one ContentPage;
- If there needs to be multiple pages, the MainPage should be a NavigationPage that contains a ContentPage. The other pages to navigate are ContentPage as well.

Pages:

- *TemplatePage:* allows to define ControlTemplate property, that appears in all ContentPage objects;
- *ContentPage:* the page that fills the screen;
- *NavigationPage:* defines navigation methods;
- *MasterDetailPage:* two pages that can go back and forth;
- *TabbedPage:* multi-page collection with selectable pages;
- *CarouselPage:* multi-page collection with swiping to cycle pages;

Layouts work just as you expect, they can contain views or other layouts inside, serve to organize how things will look.

Pages can be created by code or using XAML, similar to XML. For every XAML there is a code file that handles listeners and other events.

Views can do linear geometric transformations of translation, scale and rotation. These are all done according to an Anchor, by default in the middle of the view.

Modeless navigation: when you go to another page, put it on a stack and when you go back, pop the stack;
Modal navigation: ignores stack:

When switching pages, you can pass values to the constructor of the other page. When switching back to a parent page, if we want to pass information we need to define an interface for that transaction. We could also using the *MessagingCenter* class.

Saving and restoring data is done by accessing the Properties hash table, available from every page. Put the values to store there.

Changing from Portrait to Landscape is done by defining grid properties.

Device class has properties about the device, like idiom (includes DPI and whether it is for a phone, table or desktop), OS, etc.

To adapt the common code to device specific code, a high level interface that maps to the target device must be implemented.

13. Xamarin Data Binding and MVVM

Data binding:

- In simple apps, setting variables can be done directly by defining them like usual. Can be error prone and doesn't scale;
- Can be done using XAML, by linking control properties to data in the application. The source class is the ViewModel, and that is the only class that updates the data, others simply access it;

ViewModel class (source object) has some BindableObject with a BindableProperty. This is what binds the View to the ViewModel.

Bindings allow automatic update between source and target, and these transfers can be OneWay, OneWayOrAnother (jk), OneWayToSource and TwoWay.

Para dar bind no XAML é preciso fazer algo do género `BindingContext="{x:ReferenceName=slider}"`, e para referenciar é tipo `Opacity="{Binding Path=Value}"`.

Binding constructor takes as parameters for constructor:

- *path*: source property;
- *mode*: see OneWayOrAnother joke above;
- *converter*: object implementing IValueConverter, to *Convert()* and *ConvertBack()* values;
- *convPar*: optional parameter for converter;
- *strFormat*: if target is a string, specify format;
- *source*: source object, usually a ViewModel, different from path. Can also be specified as the *BindingContext*, as mentioned above;

Data binding works by making the binding property on the XAML subscribe to the *PropertyChanged* callback on the ViewModel.

ViewModels must implement INotifyPropertyChanged because of the above. Otherwise the target does not know when the values change. If the implementation uses *CallerMemberName*, then the triggering of the event handler can be done inside the method.

MVVM (Model-View-ViewModel) Pattern:

- *Model*: class that stores persistent data and does external service calls;
- *ViewModel*: class that has properties and methods that can be bound to a View;
- *View*: calls that displays things and takes user input;

If a View receives an input that requires data update, the ViewModel is responsible to update the Model with a *ICommand*. When the View event is triggered, the *Execute()* method of the ICommand object.

It's possible to create an Interface for this Command thingy, and bind it to the View.

ListViews support data binding, templates for each item, and data binding for each display item. The source of the collection is the *ItemSource* property.

Data binding updates whenever a new item is added or removed. Pickers are not bindable. TableViews are bindable, and can be used for data items and other stuff.

14. Push Notifications

Push notifications:

- Could be done by polling, but inefficient and battery draining;
- Server informs device of notification, but channel between client and server must be always open and it can change. For this, the client informs server in case of address changed and has associated channel for notifications;

Basically device registers, application service notifies servers, servers deliver message to device.

Cross-platform push notifications are done by having a notification hub that sends a message to each server, that notifies respective devices.

Firebase has this, in Xamarin there must be code written for each platform.