

Mash

An Android Motr.io Dashboard

By Craig Hesling

What is Mortr.io?

- Mortr.io is a distributed platform that manages communication between applications and devices.
- Carnegie Mellon University has developed it's schema and runs an implementation on the `sensor.andrew.cmu.edu` machine.

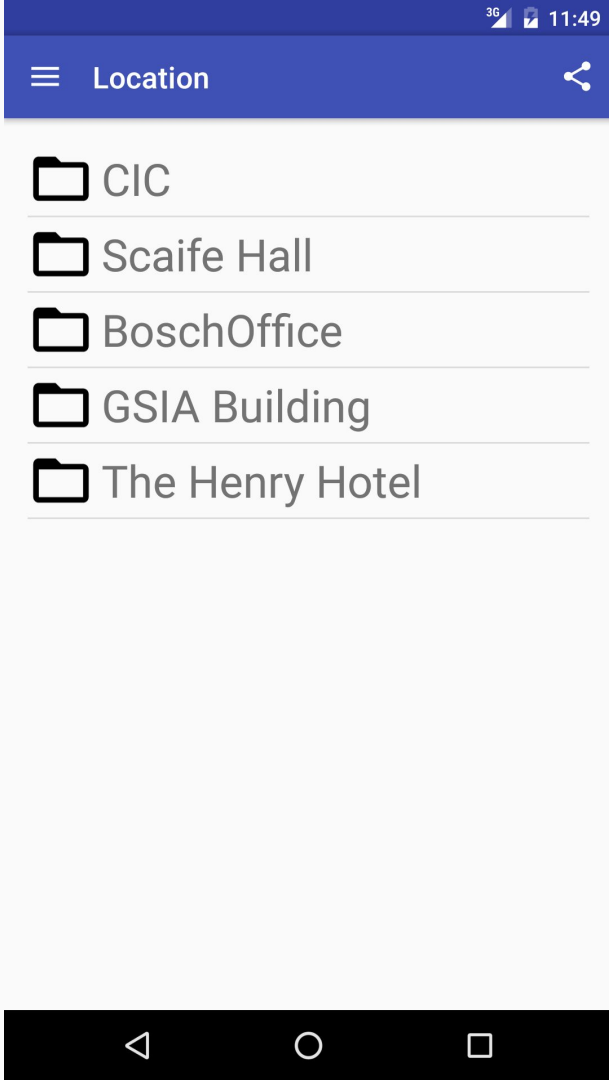
Motivation

- Sensor/control networks are good for two things
 - Ease daily tasks
 - Study sensor data and phenomena
 - The common denominator is that both require human interaction
- What good is a sensor/control network if you must write code to even see a light turn on and off?
- A sensor/control network needs to be tangible and ease to discover
- It needs an interface that can make daily routines easy and accessible
- It needs a basic interface that can test and manipulate the end points

Project Description

I have created an Android Dashboard app for the Mortr.io framework (Mash). Simple, effective, and easy to use was the goal. Here are some core features:

- Multiple ways to discover devices in a Mortr.io network
 - Manually browse tree
 - NFC discovery
 - QR code discovery
 - Nearby BLE tag discovery
- See last data published
- Send control commands
- Share devices with others



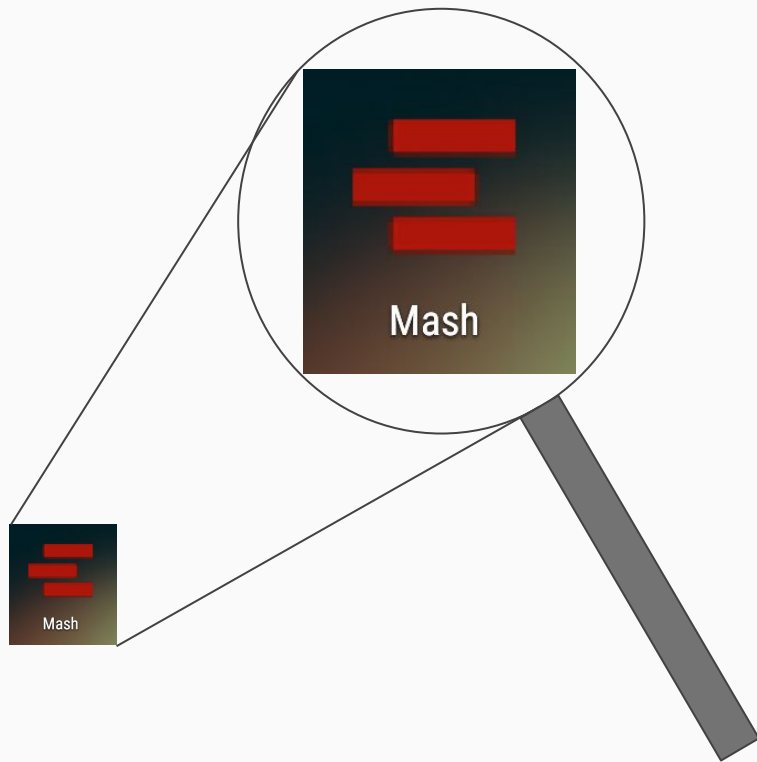
Outline

- App Walk Through
- Review of Design
 - UI
 - Backend
- Lessons Learned
- App Demo & Questions

App Walk Through

Beautiful Mash Launcher Icon

Stole it from the [Mortr.io](https://mortr.io) project with
permission



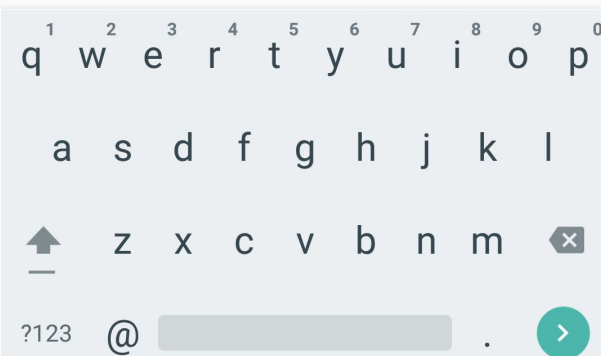
Login

JID

craig@sensor.andrew.cmu.edu

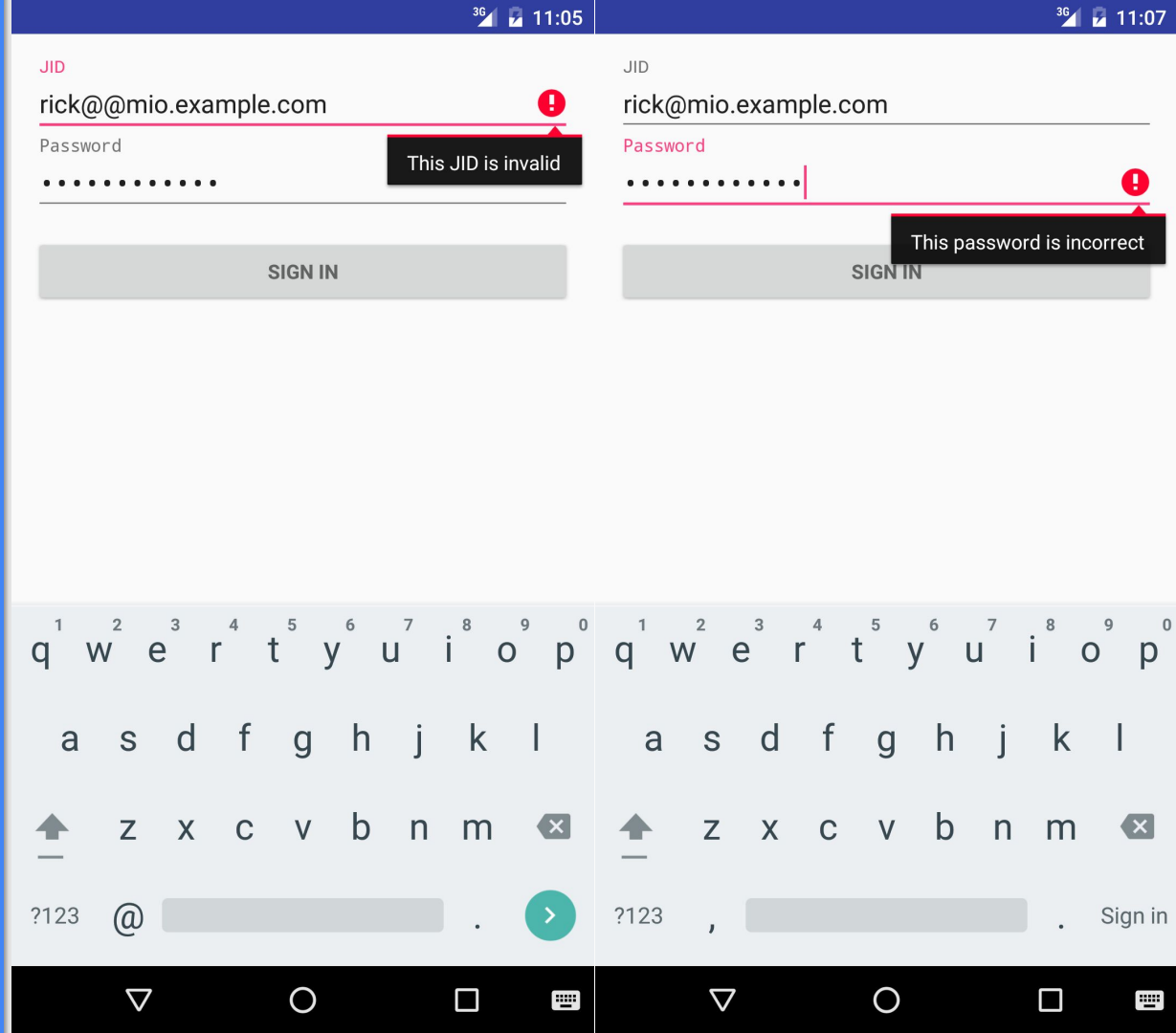
Password

SIGN IN



Login Features

- JID validity/sanity checking
- Ensure credential verification



Main Location

We open the MIO root directory by default.

The MIO root is the highest level of the MIO directory tree. All directories and devices are accessible through the root directory.

In it's basic form, the MIO tree is organized based on common physical location names. As in, Building → Floor → Room → Device.



Location



Gateways



templates



3G 11:49

☰ Location

🔗

📁 CIC

📁 Scaife Hall

📁 BoschOffice

📁 GSIA Building

📁 The Henry Hotel

3G 11:49

☰ CIC

🔗

📁 Lab 1101

3G 11:49

☰ Lab 1101

🔗

🔧 LabDoor

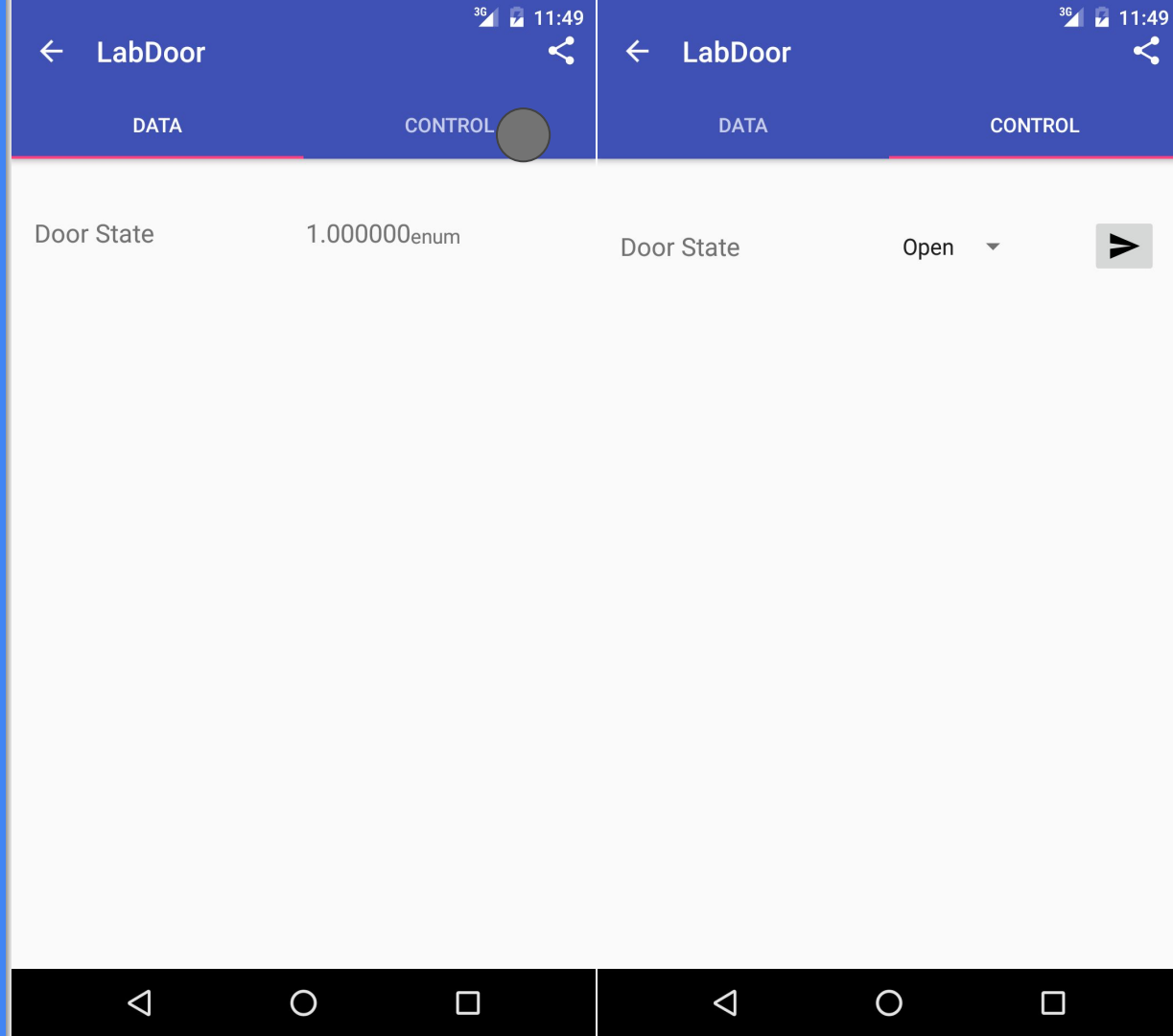
📁 Lighting Board

📁 Thermostat Board

📁 Sensing Board

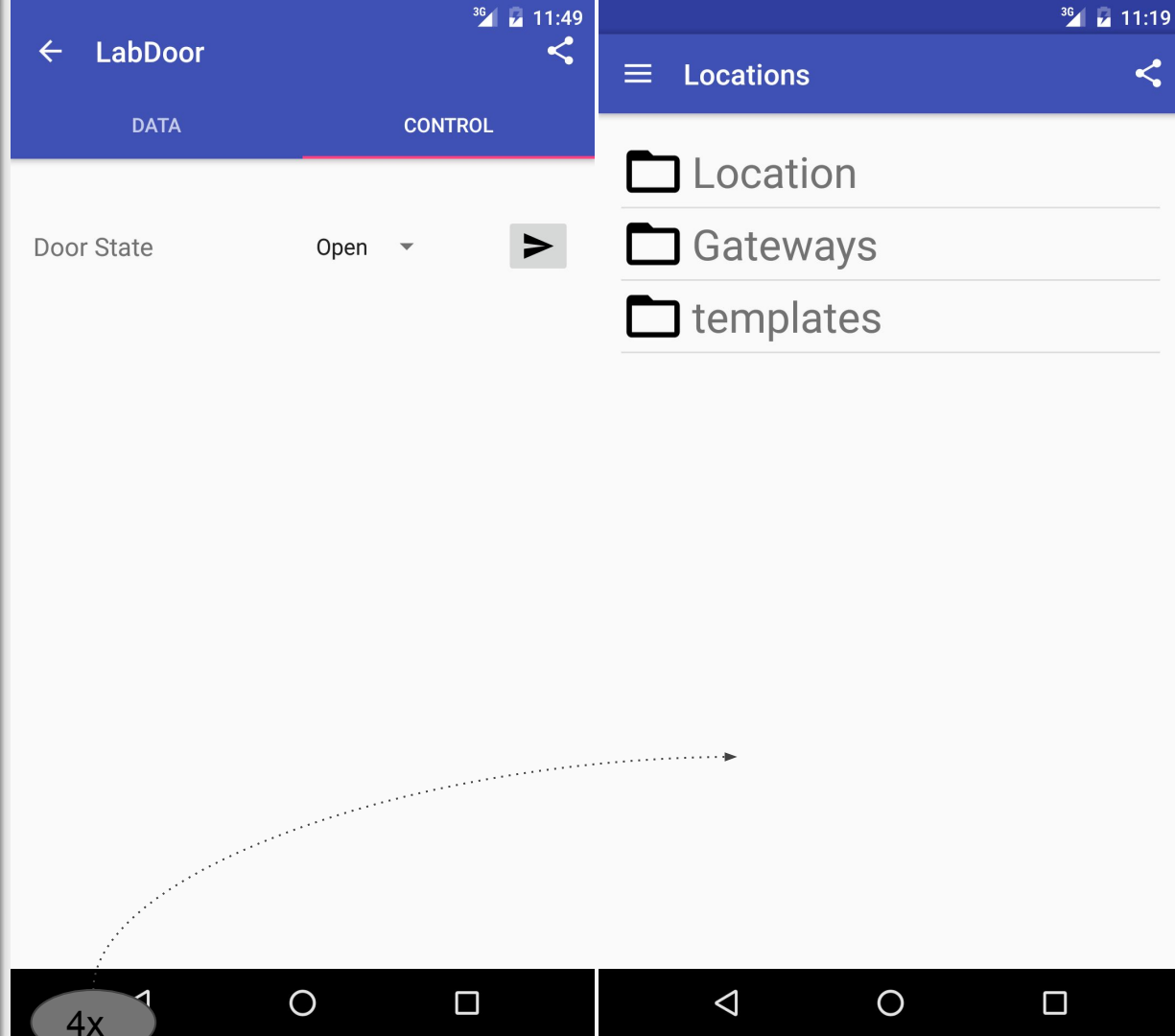
Lab Door device

We now have the option to check the Lab Door state Data or Control the Lab Door.



Going Back ←

We move back up the tree by tapping the Android back button.

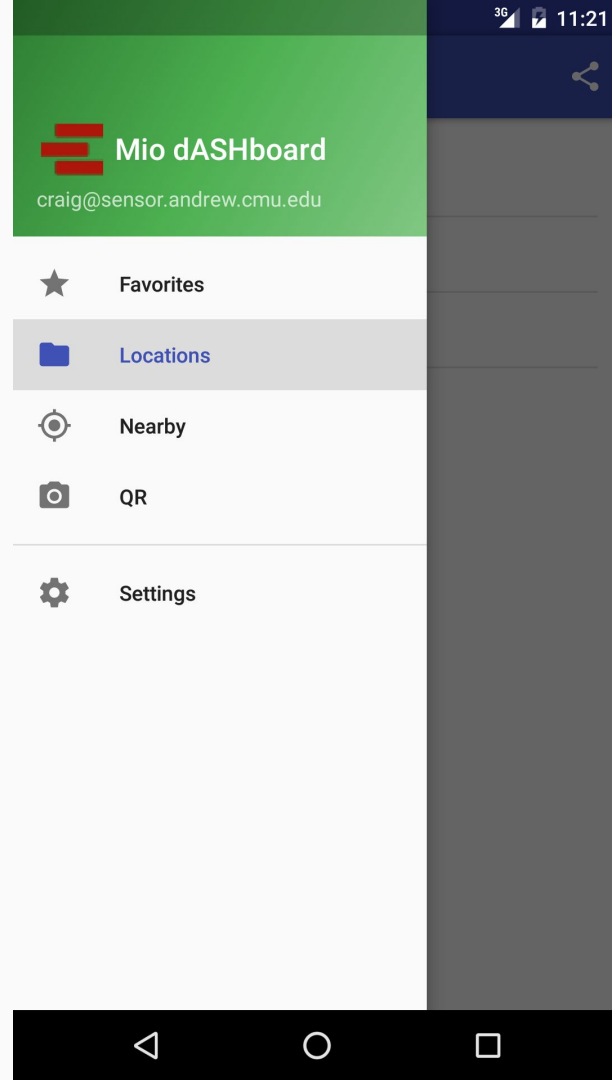


Navigation Drawer

Although the root directory can reach all nodes, it is not always the most effective way to reach your device. There are other way to categorize and lookup the devices you are interested.

The **Navigation Drawer** is where you select which method you would like to use to lookup a device/location.

- Favorites
- Nearby
- QR



Favorites

The MIO structure contains a private favorites directory for each user. It houses nodes that you deem are important and would like a shortcut to.



 FORK012884551147

 Twist 3968

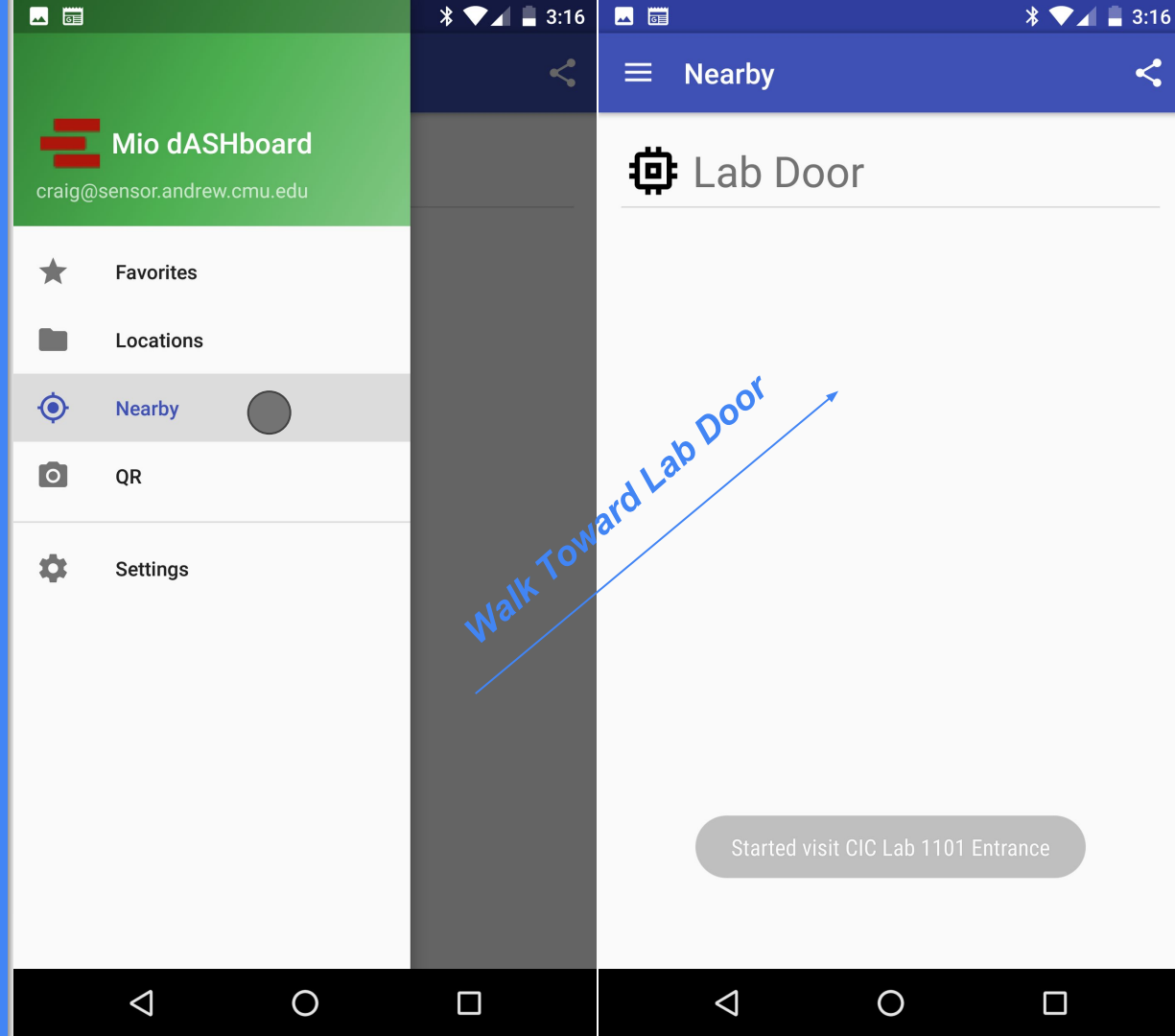
 Lab Door



Nearby

Shows MIO devices and locations that are being advertised by nearby Gimbal Bluetooth Low Energy(BLE) beacons.

This is another way we try to shortcut the lookup and discovery of devices you are interested in.

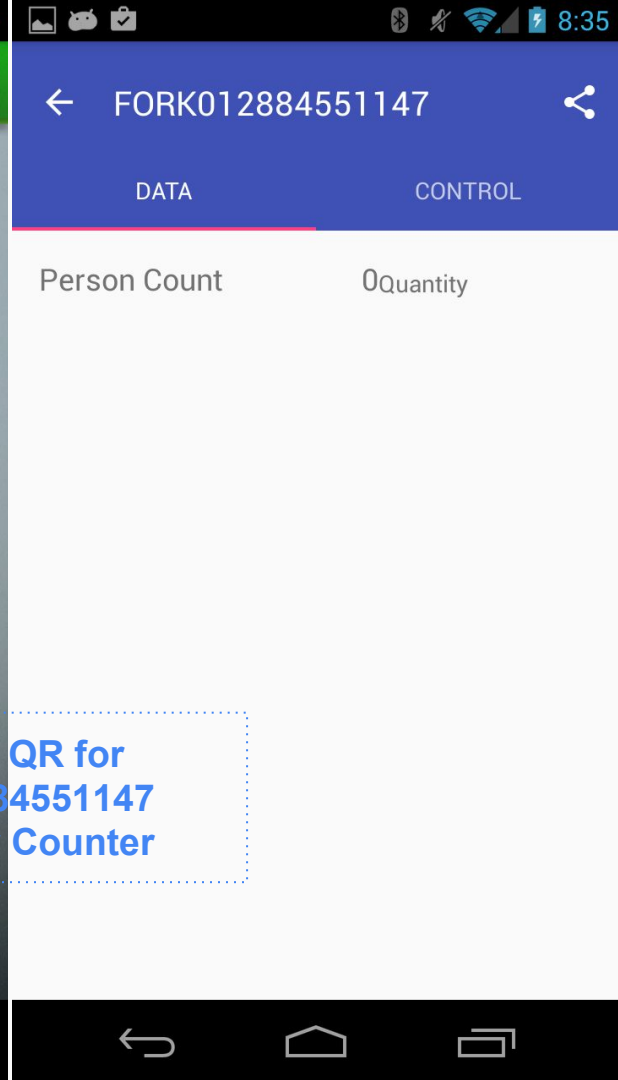
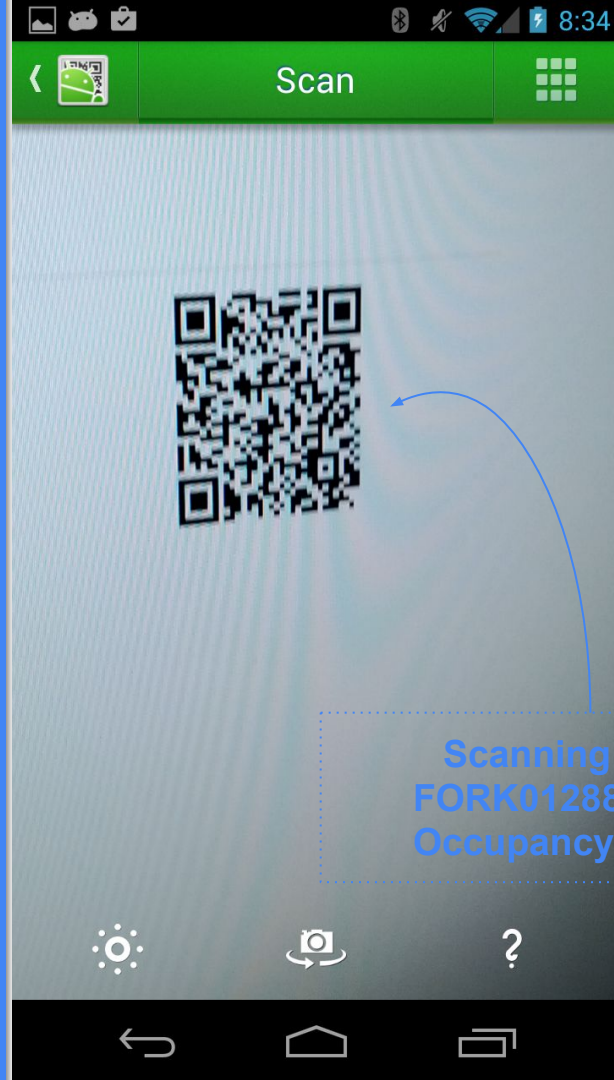


QR

Quickly lookup a device/location by scanning its QR code!

Side Note:

Notice that the screenshots were taken on an older Android device. The Mash app has been tested and supports older Android 4.2 and later.

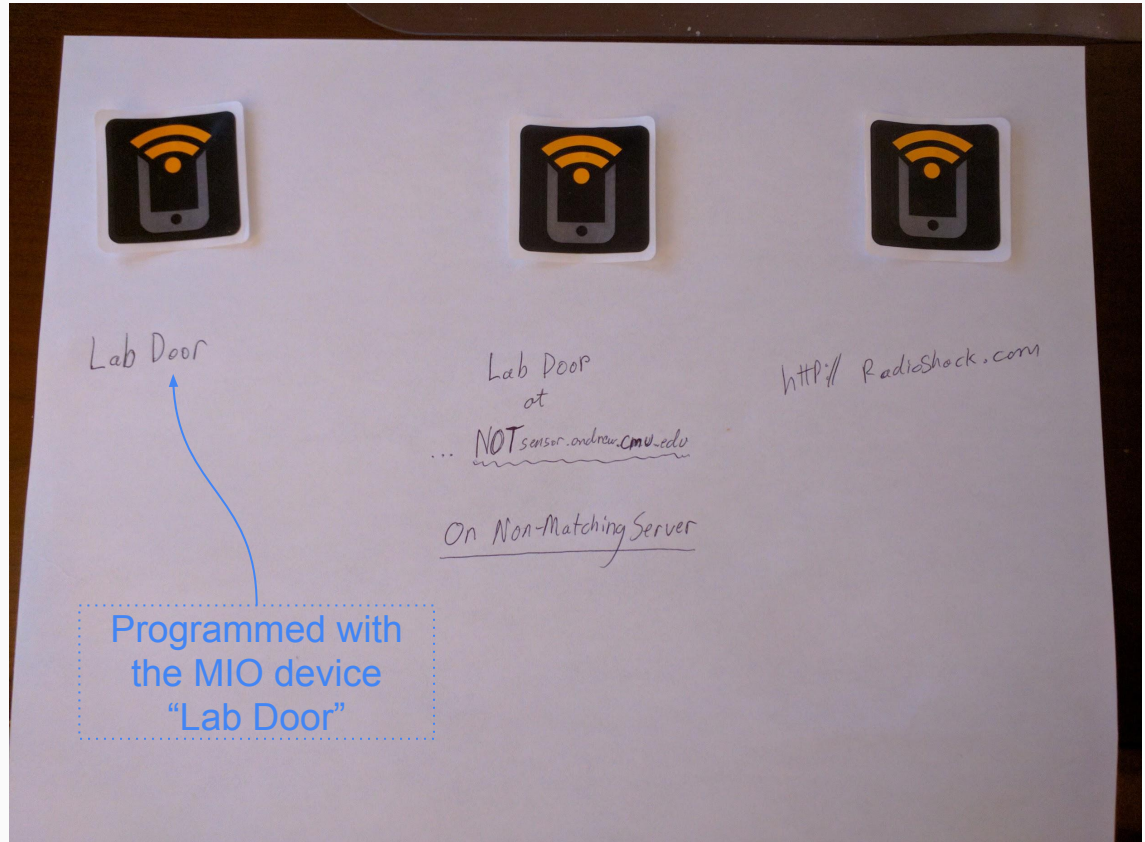


NFC

Yet another way to quickly jump to a location or device page is to tap a MIO NFC tag.

Side Note:

NFC tags can be programmed to hold lots of different information and URIs



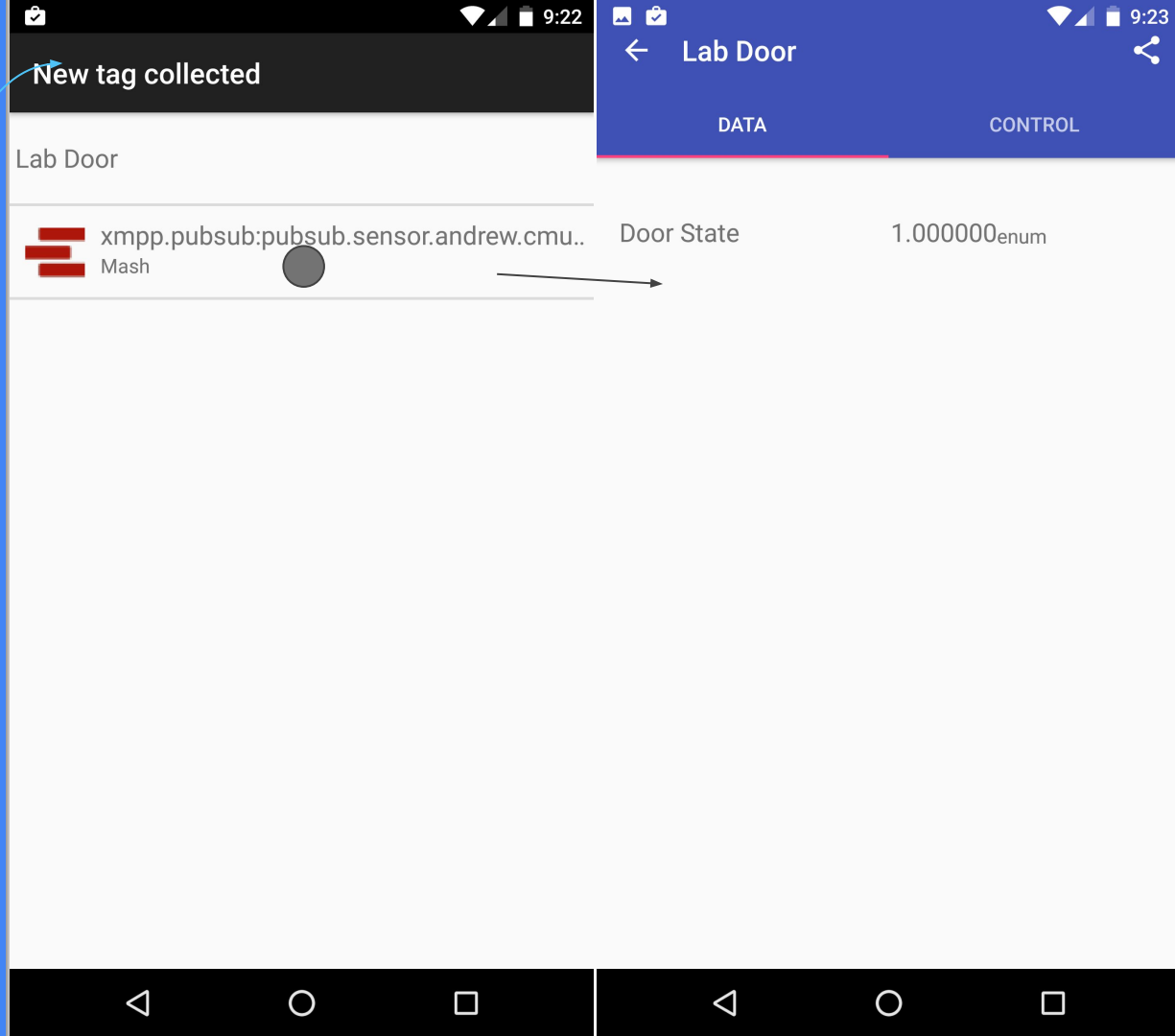
NFC

Tap
Tag!

Yet another way to quickly jump to a location or device page is to tap a MIO NFC tag.

The beauty behind this method is that NFC supporting Android devices are always ready to read an NFC tag, when the screen is on and unlocked. This means you can **tap a MIO NFC tag on any screen** of Mash or some other Android app. It doesn't matter! It will prompt you to open the MIO node.

This is my favorite lookup method.



UI Feedback

Now is a great time to point out the nice **loading circles** that appear between screen loadings.



Review of Design

Major Design Split

User Interface

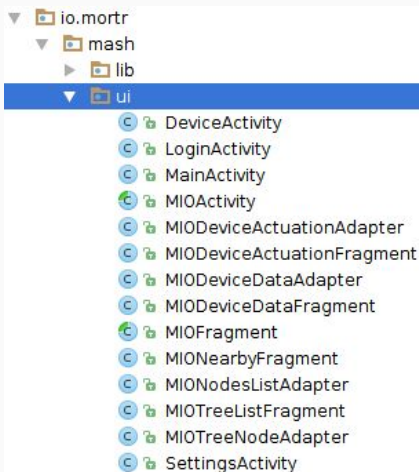
- io.mortr.mash.ui package
 - Contains all classes that interact with the Android context in order to manipulate graphical elements
 - Activities, Fragments, and Adapters

Backend

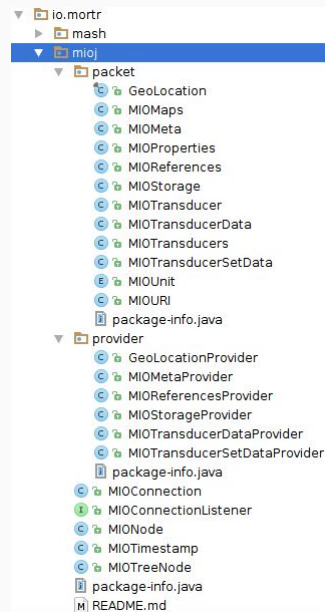
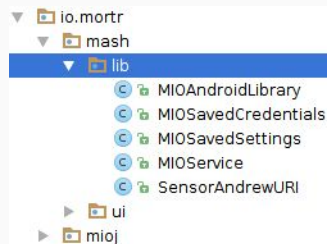
- io.mortr.mash.lib package
 - Contains classes whose primary mission is to manage some state relating to the Android app.
 - Settings managers, service managers, special app URIs, and interface libraries
- io.mortr.mioj package
 - A Java non-specific library that connects with the Mortr.io framework
- Gimbal Framework

Major Design Split

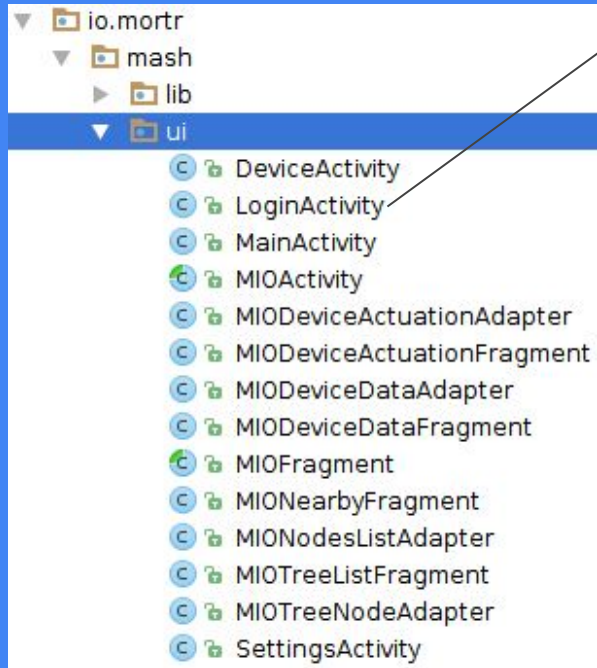
User Interface



Backend



User Interface

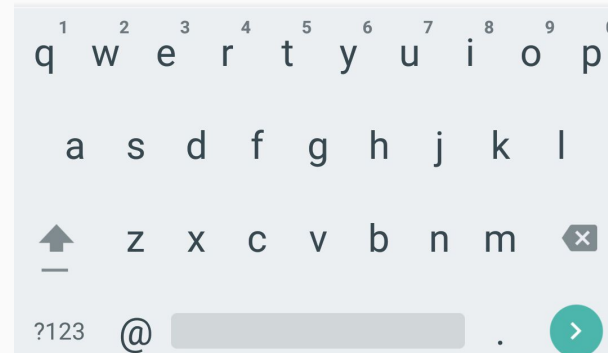


JID

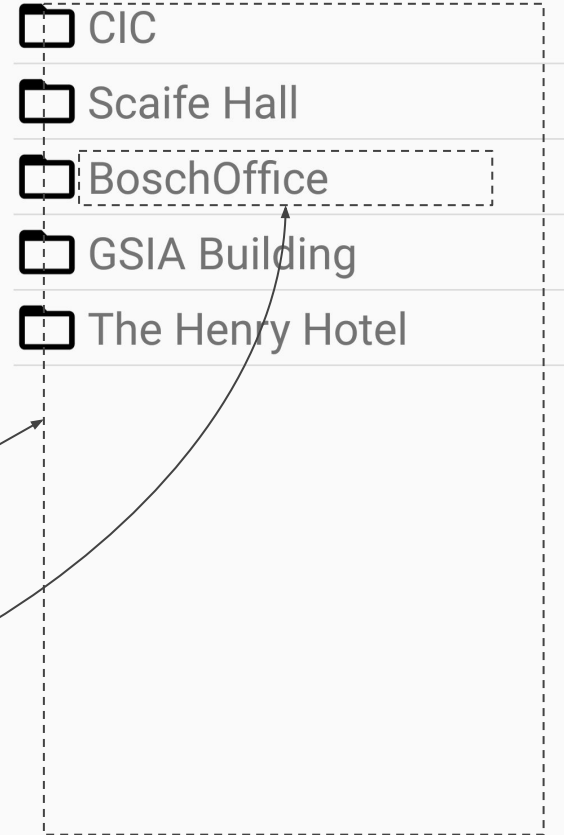
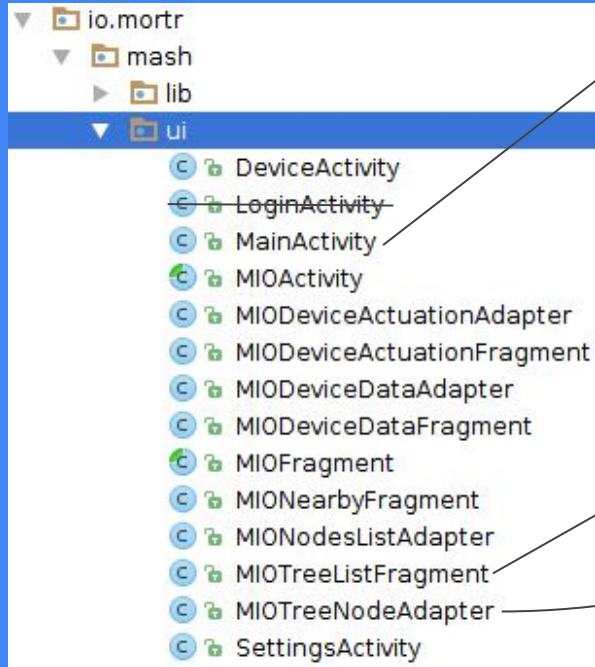
craig@sensor.andrew.cmu.edu

Password

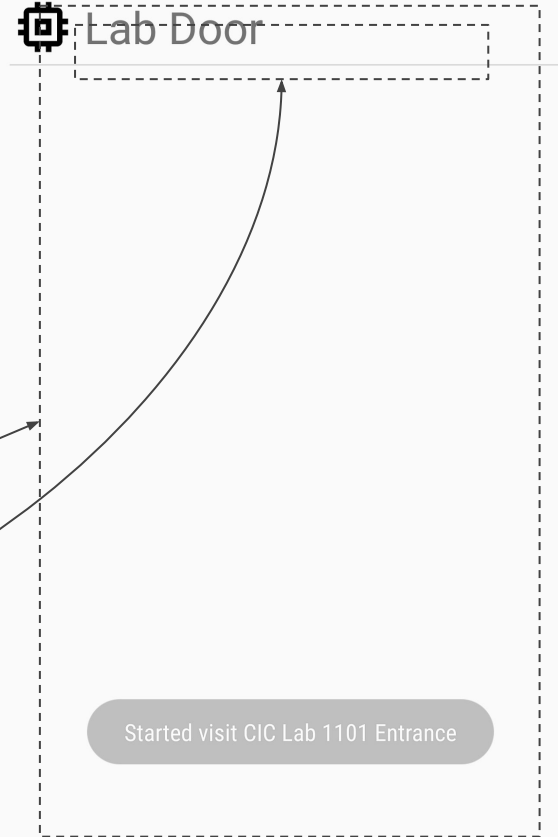
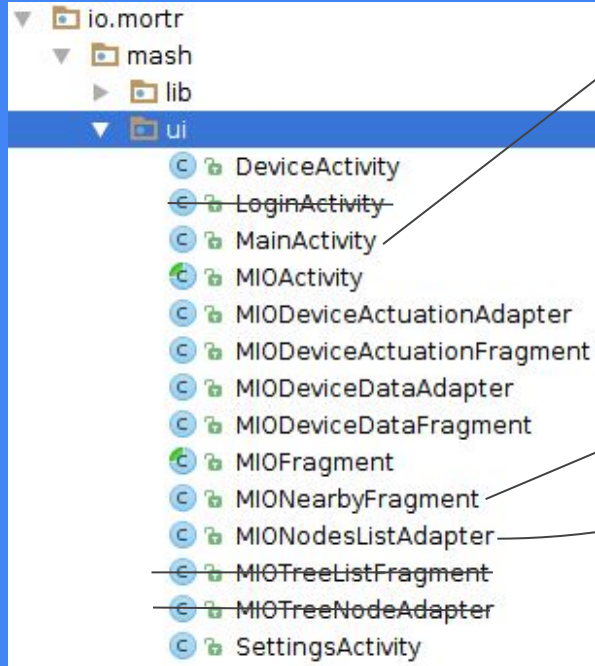
SIGN IN



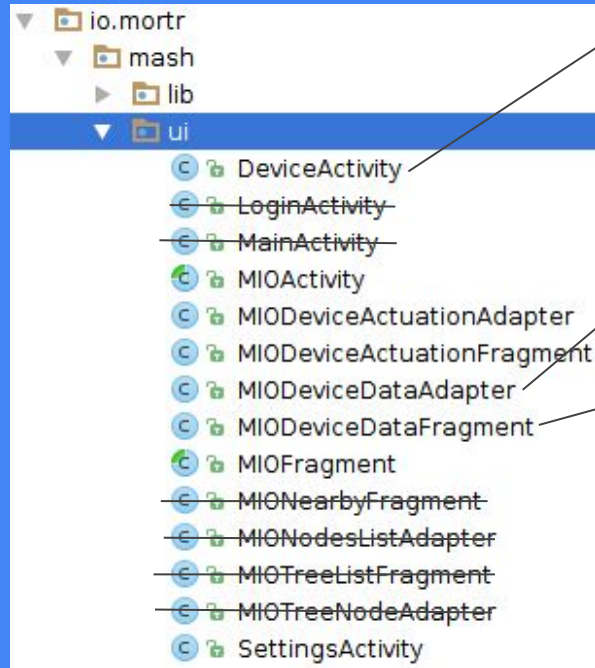
User Interface



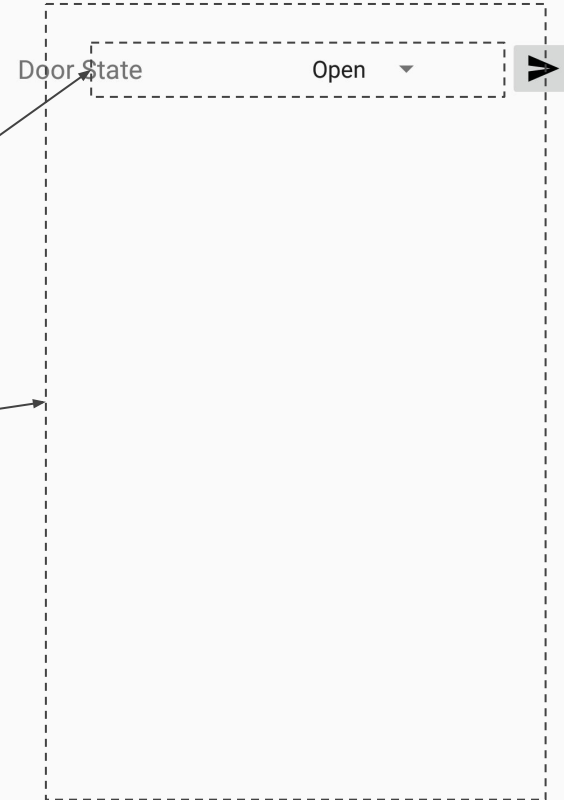
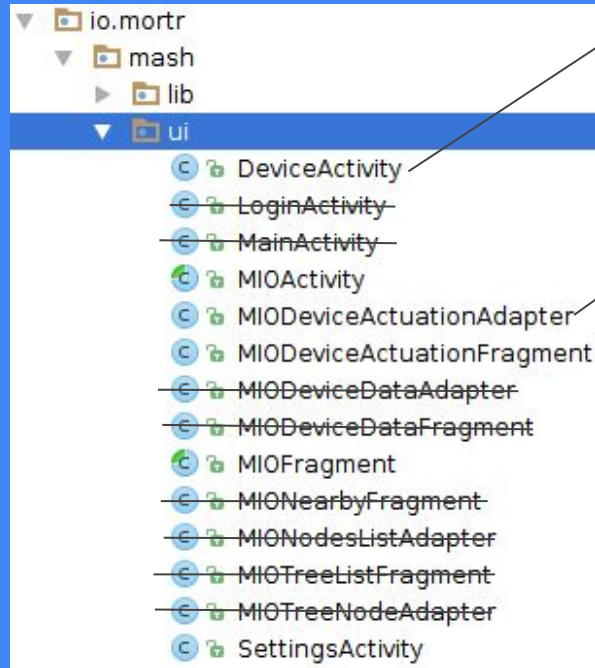
User Interface



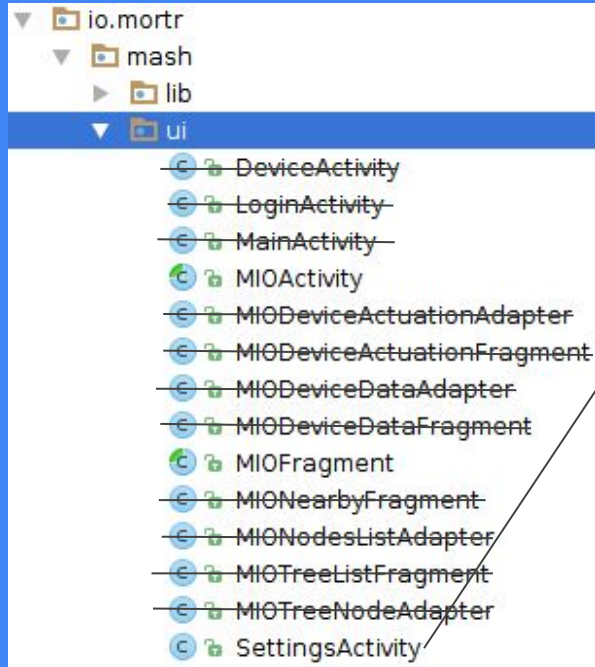
User Interface



User Interface



User Interface



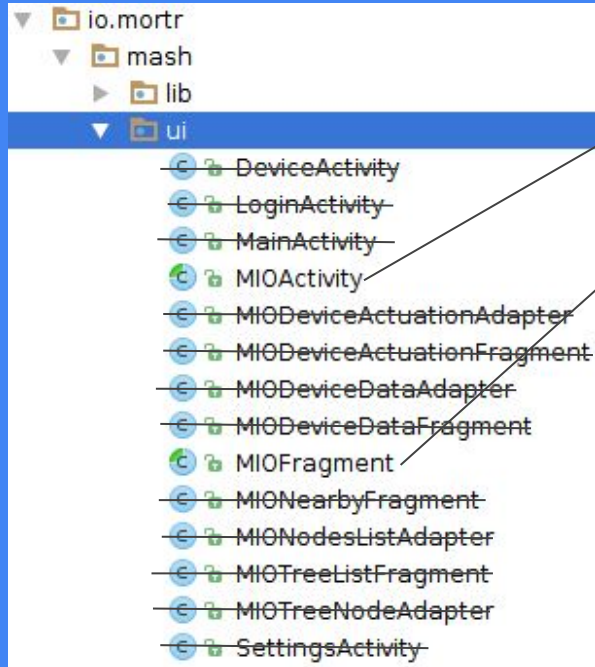
XMPP Connection

- ☒ Compression
- ☒ Disable Security

RESET CREDENTIALS



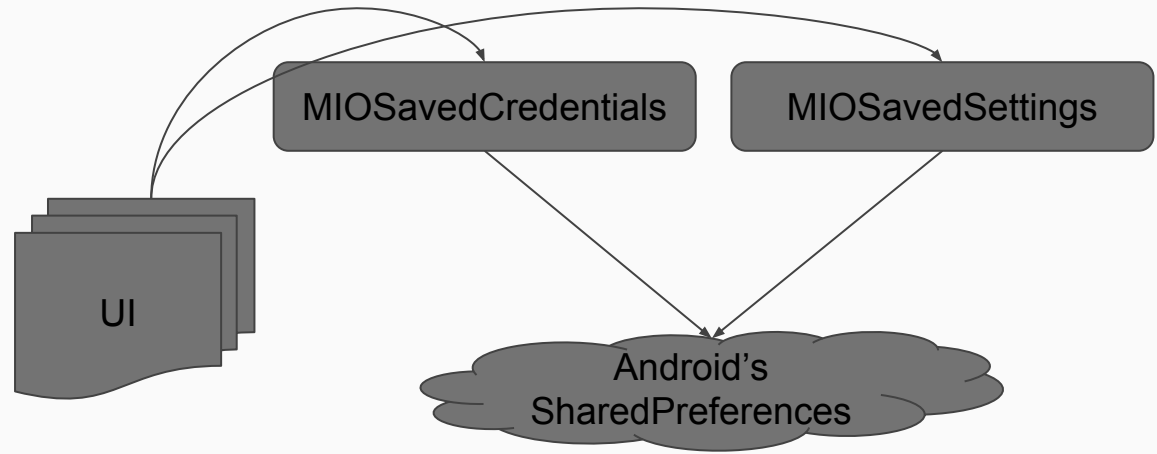
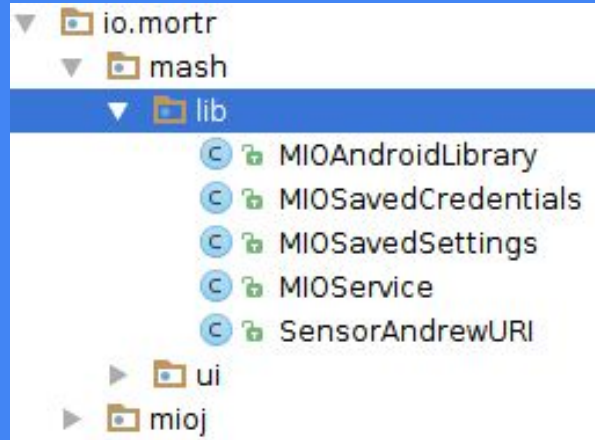
User Interface



MIOActivity MIOFragment

- These are abstract classes that extend the standard Android Activity and Fragment classes
- Extended by all local UI Activities and Fragments
- Defines a child-class usable instance of the MIO Library, which is used to communicate with the MIO Service and MIO framework
- Installs the login trigger for all Activities

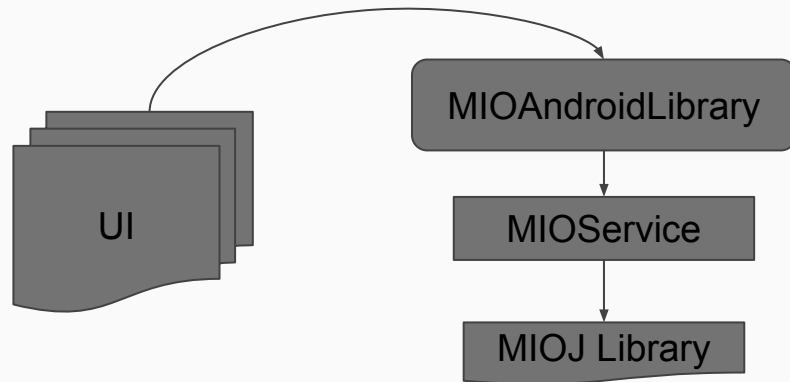
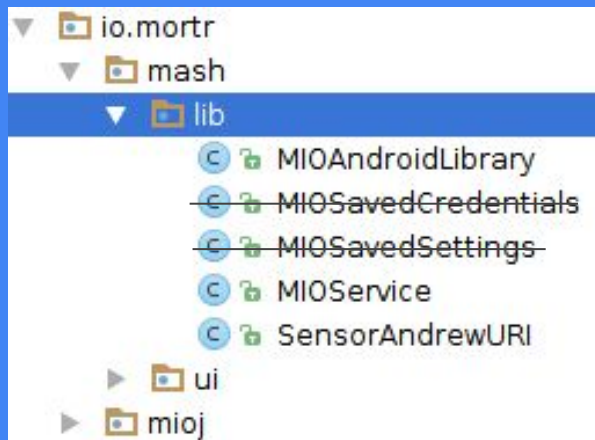
Backend



`MIOSavedCredentials` `MIOSavedSettings`

- Simple managers/connectors for getting and setting the app login credentials and app settings respectively.
- Due to the simplicity of the interface, they are used throughout the app

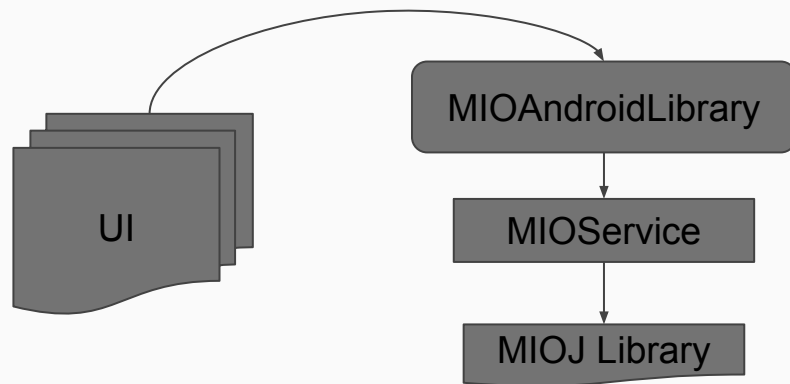
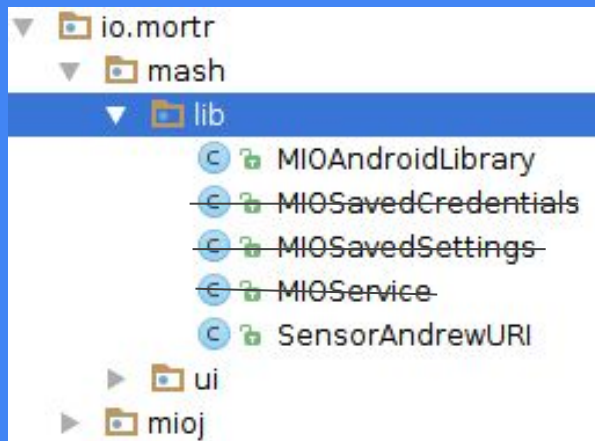
Backend



MIOService

- A Bindable Android Service that manages the app's sole MIOJ connection and interface
- Handles spawning and joining background threads that call explicit network operations in the MIOJ library
- Coalesces spurious/redundant reconnect/connect actions and enforces thread synchronization
- Does NOT auto retry requests
- Does NOT auto reconnect
- Note: No UI elements were harmed the making

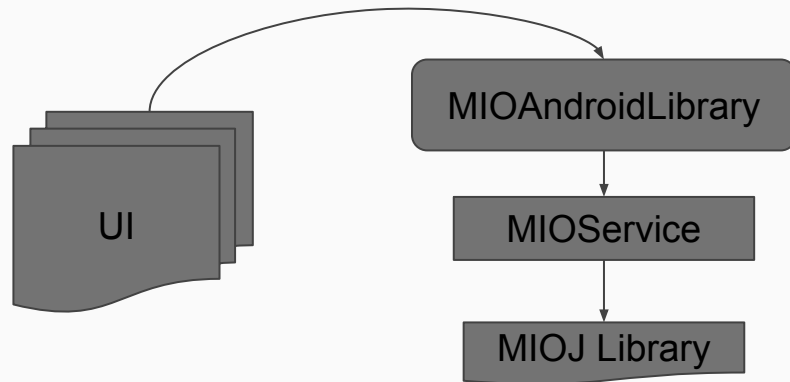
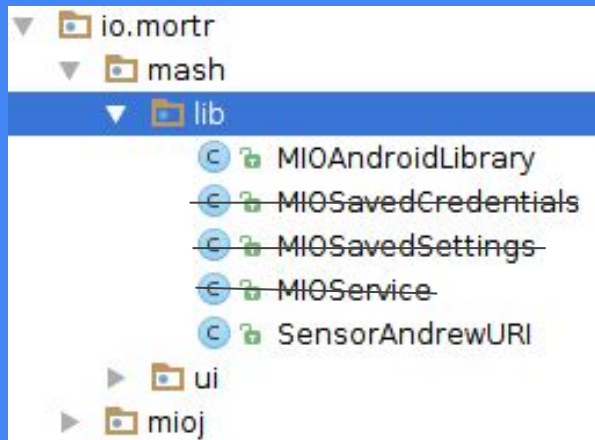
Backend



MIOAndroidLibrary

- This is the UI friendly interface for the MIOService and MIOJ library
- All Fragments and Activities grab an instance
 - Binds to the local MIOService
 - Auto connect MIOService OR launch Login
- Tries to auto fix a broken connection and reconnect
- **Auto retries requests** that fail due to connection **[big feature]**
- Can interact with UI for Login Activity and Connection status/retry Snackbars

Backend

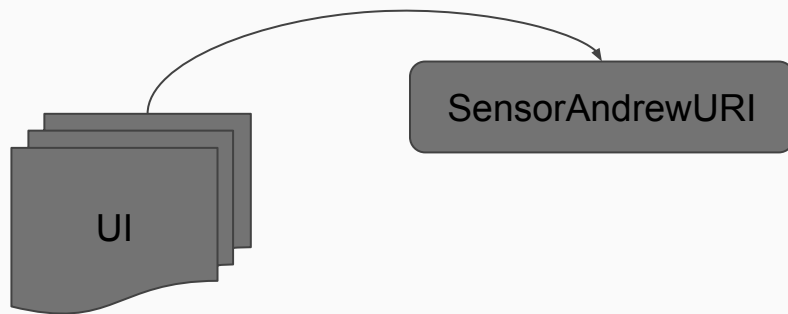
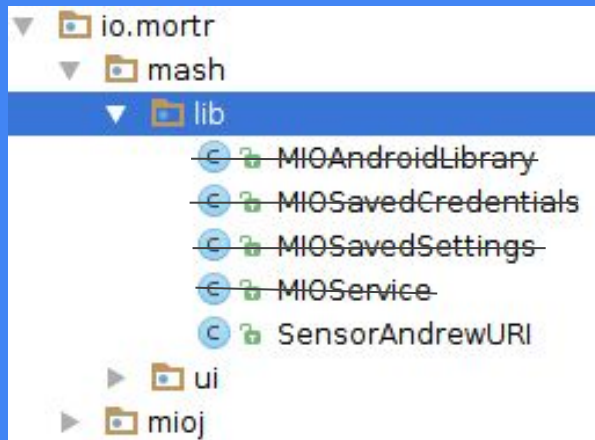


MIOAndroidLibrary

→ Auto Retry Requests

1. This is so cool that I must elaborate
2. The UI requests data from MIOAndroidLibrary and gives it a callback that can populate the UI with the results asynchronously. This pair is a "Job".
3. MIOAndroidLibrary will hold onto this "Job" while the network is disconnected or if the Job fails for a reason that may be fixable(network failure).
4. When the library fixes the connection, it reruns all Jobs until they are considered done.
5. So, UI data to be strongly/dynamically tied to MIO

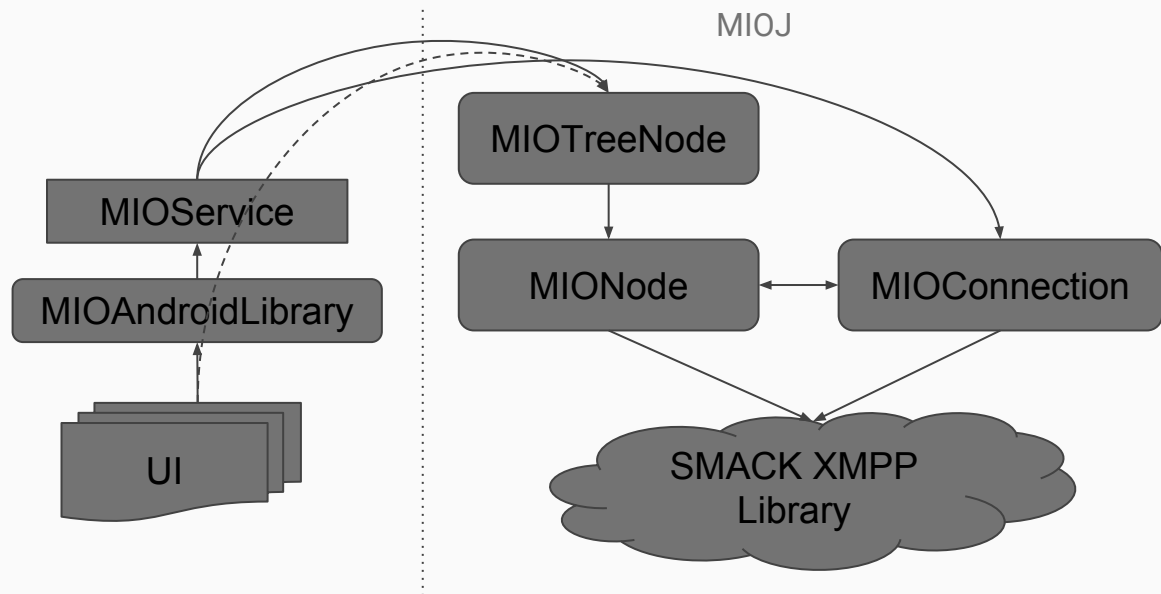
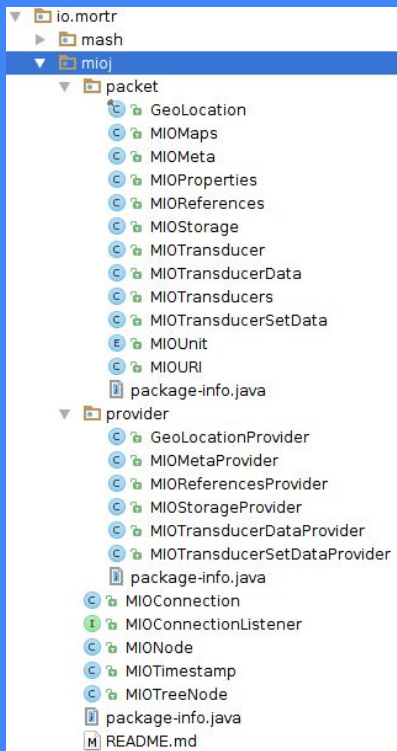
Backend



SensorAndrewURI

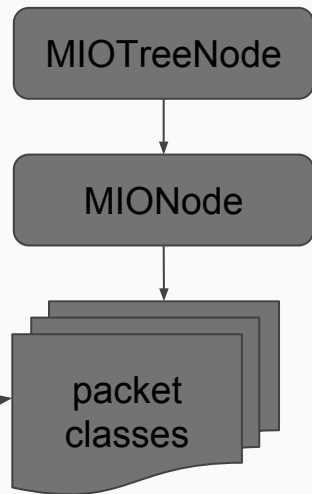
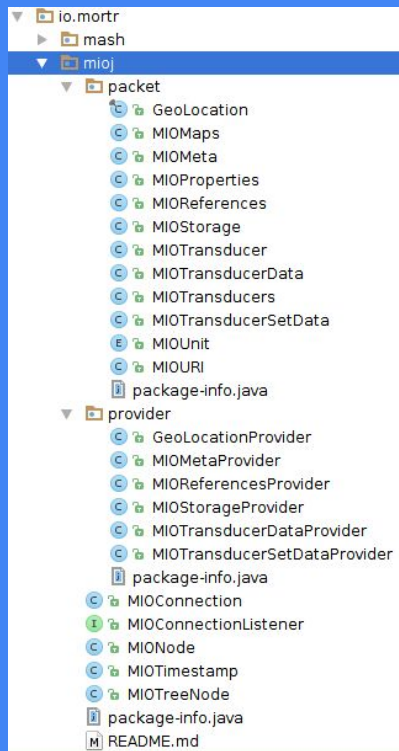
- This is simply an abstracted URI parser for the specific URI used for QR codes.
- Unfortunately, this format must differ from a MIOURI by specification. (no control over this)
- Extra Info
Parses something like the following:
`http://sensor.andrew.cmu.edu/#/device/03a4fdc0-dce5-11e4-ba47-d7d35fb6b294`

Backend



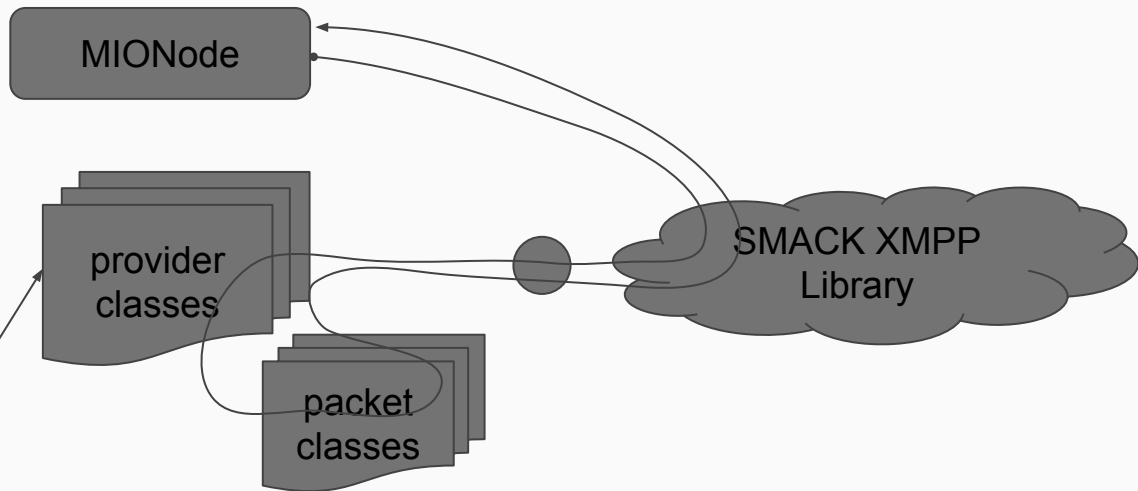
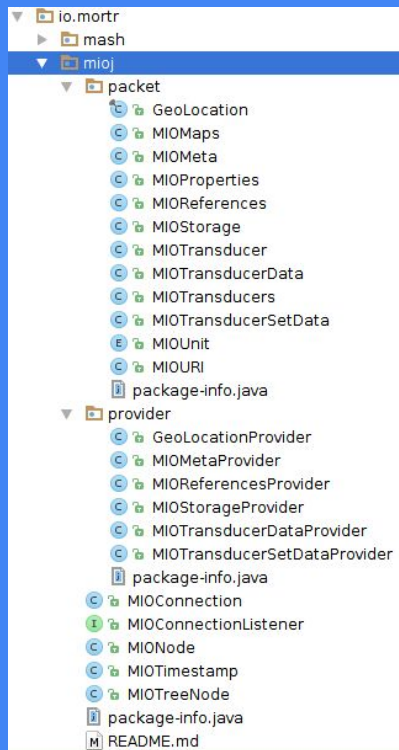
- As previously mentioned, MIOService handles the underlying MIOConnection and processes data requests.
- The UI gets its requested data through a MIOTreeNode object on callback

Backend



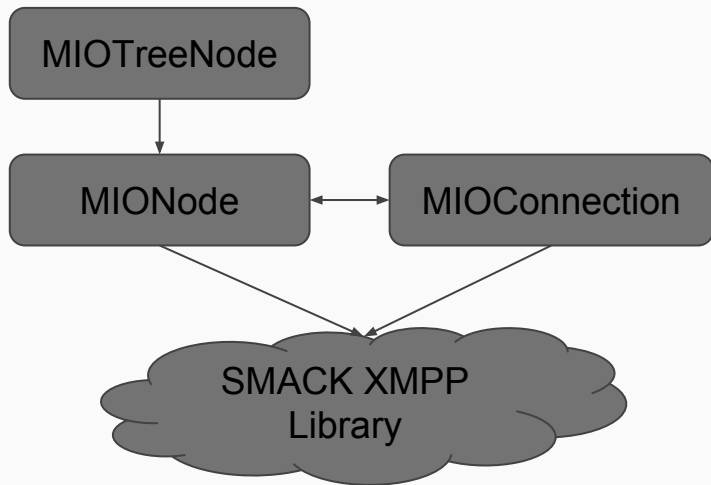
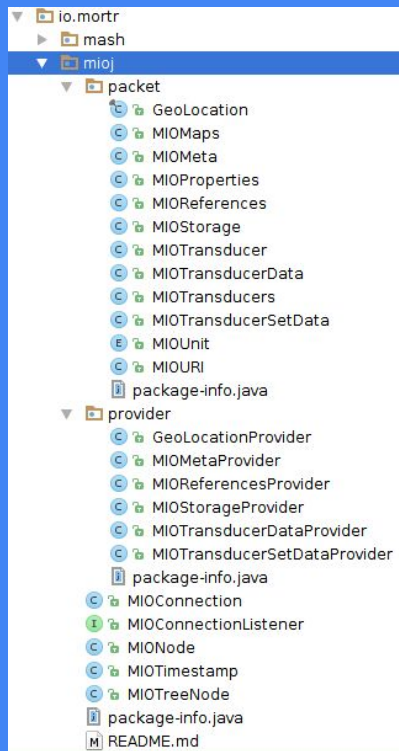
- The packet classes represent abstractly the raw data entities in the Mortr.io framework.
- A MIONode is composed of the packet types combined in a way to mimic the Mortr.io scheme
- A MIONode is the lowest level representation of a Mortr.io event node. (Event node == directory or device)
- A MIOTreeNode extends MIONode to be a more functional model

Backend



- The provider classes are used by the SMACK library to make sense of incoming and outgoing Mortr.io schemed data
- They provide parsers to build packet types from incoming MIO data
- They provide generators to build outgoing MIO data from packet types
- Providers are registered with SMACK on init
- SMACK can then interface with MIONode using native packet types

Backend



- The rest of the library helps abstract the XMPP connection and MIO Event Nodes
- The most important take away is that the `MIONode`'s procedures that trigger synchronous network activity from SMACK are segregated and explicit to allow users(ANDROID!) to explicitly control when network operations will take place. These methods are labeled `fetch*` and `setActuation`.

Backend

The Gimbal BLE Beacon API

- The core reason I use the Gimbal API is because it facilitates the delivery of the extra MIOURI information when a location is visited. This means that the app is dynamically able to discover any number of new locations and MIO Nodes. (adding new location-nodes is done via Gimbal web manager)
- Pros
 - The Gimbal framework is quite sophisticated.
 - You can setup virtual locations with lots of configuration parameters.
 - Web interface gives you tons of statistics, including battery level
- Cons
 - It can be slow to discover BLE beacons
 - It is mainly useful only for starting and ending visit events, although you can get realtime beacon siting data
- The MIOURI is encoded into the extra “properties” of a “visit” event.

Lessons Learned

Android Consideration

- The interface laid out by Android doesn't give a basic Java procedural option
- You are forced to use some “non-basic” Java features.
 - Inheritance(extends) and implements
 - asynchronous callbacks
- Must learn lots all Android quirks
 - The Activity instance object isn't saved only the bundle
 - No synchronous network actions on UI thread
 - Android Services simplify and modularize the task(but are far from perfect vs. static)

Encapsulation vs. Containment

- Containment
 - Easier to implement, but can expose a complex interface
 - Can expose more advanced control of contained instances
- Encapsulation
 - Gives solid separation of underlying interfaces and variables
 - Can expose simpler interface to user
 - More time consuming to implement and can seem redundant at times

Inner Class vs. Outer Class

- Use inner class when the class functionality is associated with it's encapsulating class
- Use a new file with an outer class when the functionality can be used by other classes or is too complex to hide in file shared by another class.
- To be honest, it really depends and that is why I titled this something vs. something.

Inheritance Examples

All activities extend the MIOActivity

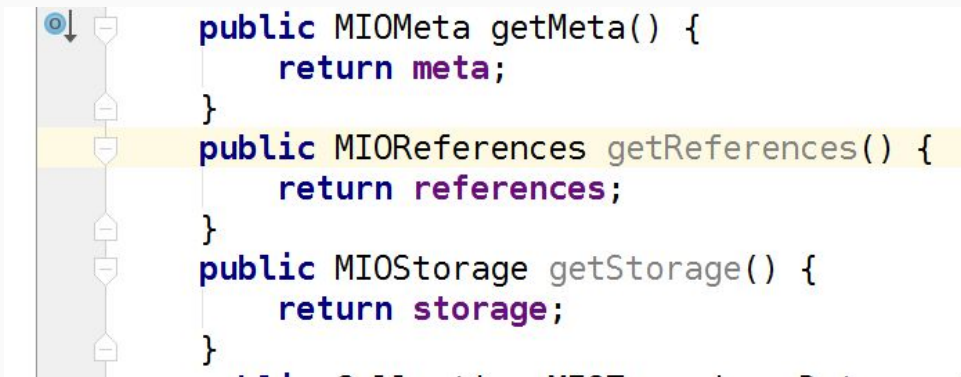
```
... which has different elements than the normal.  
💡  
public class MainActivity extends MIOActivity  
    implements NavigationView.OnNavigationItemSelectedListener {  
    private final String SAVE_NAV_RES_CHECKED = "navResChecked";
```

MIOTreeNode actually “extends” the functionality of MIONode. It adds more encapsulation type features and hides the lower level details.

```
*/  
public class MIOTreeNode extends MIONode {  
  
    /* higher level interfaces */  
    protected MIOTreeNode parent = null;  
    protected ConcurrentHashMap<String, MIOTreeNode> children = null;
```

Containment Example

MIONode simply provides the container for the hierarchical data and the methods to fetch it. The getters simply grab the contained instances.



```
public MIOMeta getMeta() {  
    return meta;  
}  
public MIOReferences getReferences() {  
    return references;  
}  
public MIOStorage getStorage() {  
    return storage;  
}
```

Encapsulation Example

MIOTreeNode provides high level methods to access data inside it's encapsulated instance variables

```
/**
 * Get MIO node name from meta information.
 * @return The MIO Node Name or null if meta not fetched for name is missing
 */
public String getName() {
    if(meta == null) {
        return null;
    }
    return meta.getName();
}
```

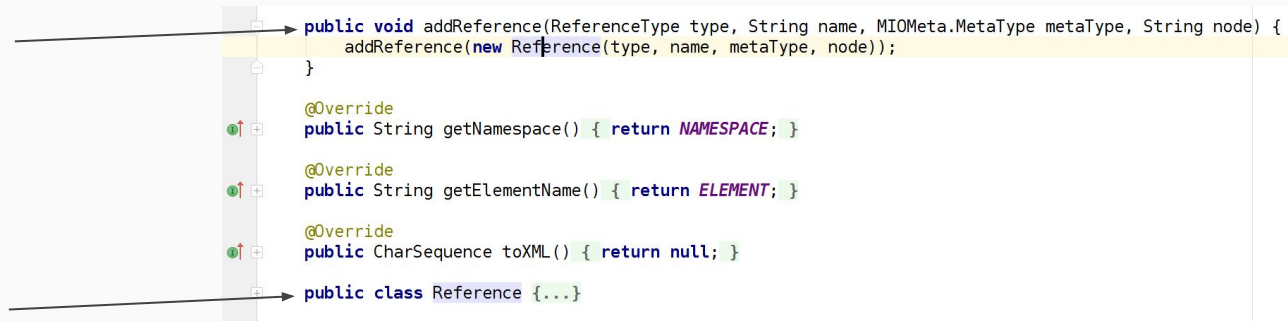
Abstract Example

MIOActivity is an abstract class in order to denote that it cannot be instantiated. It must be extended.

```
/**
 * This class handles the installation of the MIOAndroidLibrary for
 * use by child classes.
 */
public abstract class MIOActivity extends AppCompatActivity {
    private boolean connect = true;
    private boolean login = true;
```


Inner Class Example

MIOMReferences uses an inner class to help organize references. References are a product of the encapsulating MIOMReferences class.



```
public void addReference(ReferenceType type, String name, MIOMeta.MetaType metaType, String node) {  
    addReference(new Reference(type, name, metaType, node));  
}  
  
@Override  
public String getNamespace() { return NAMESPACE; }  
  
@Override  
public String getElementName() { return ELEMENT; }  
  
@Override  
public CharSequence toXML() { return null; }  
  
public class Reference { ... }
```

Synchronization Example

Synchronization is use in MIOService to ensure the async background threads are not conflicting with one another.

```
/**
 * Connect a new connection. Cannot be used to reconnect.
 * @param callback
 * @return
 */
public boolean connect(@Nullable ConnectionStateChangeListener callback) {
    synchronized (stateRequestChangeLock) {
        Log.i(LOG_TAG, "Connecting Request");

        // check that no one else is changing connection state
        if(stateChangeLock.tryAcquire() == false) {
            return false;
        }
        state = ServiceState.CONNECTING;
        new Connect().execute(info);
        return true;
    }
}
```

Async task starts and then
unlocks stateChangeLock when
finished

Interface Example

Many interfaces are used and defined in the MIOJ library and in the Mash app.

The simplest to understand is the `FetchNodeResultCallback` in `MIOService`. It simply asks the user to provide the callback method for when the node data is available.

```
public boolean fetchNodeAll(String nodeId, FetchNodeResultCallback result) {  
    synchronized (stateRequestChangeLock) {  
        FetchNodeInfo info = new FetchNodeInfo(nodeId, result);  
        new FetchNodeAll().execute(info);  
        return true;  
    }  
}
```

```
public interface FetchNodeResultCallback {  
    void onFetchNode(MIOTreeNode node, Exception e);  
}
```

Additional Remarks

- The lesson learned on using separate Java packages should have been evident when we walked through the “Review of Design”.
- The lessons learned on using separate files vs. combined/anonymous classes should be evident from “Review of Design” as well. I have clearly separated Activity-Fragment-Adapters.
- I feel I have implement this app and supporting libraries using the most accepted object oriented practices. If you see room for improvement, please leave me feedback in any form.

Product Demo & Questions