Stanislav Simovski

Centrally managed launcher application for Android-based MDM solution with re-branding capabilities

Helsinki Metropolia University of Applied Sciences

Bachelor

Media Engineering

Thesis

7.11.2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Stanislav Simovski Centrally managed launcher application for Android-based MDM solution with re-branding capabilities 47 pages + 2 appendices 5 May 2010
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Digital Media
Instructor(s)	Kari Salo, Principal Lecturer Juha Ranta, Chief Architect

Mobile Device Management (MDM) is gaining popularity among businesses and large organizations as it allows for improved integration of that organization's IT environment, while preserving user's or employee's ability to personalize the device within the organization's guidelines. However, despite there being demand for it, there is no solution available that provides mobile devices to end users with a customized User Interface (UI), branded after the organization owning it. That is the problem this solution aims to solve - to provide a fully cloud-managed service with a customized user interface on the Android platform. Customization of the UI is achieved through a Launcher Application, while mobile management itself is utilizing Samsung KNOX technology, available on most Samsung smart devices.

This thesis focuses on designing and implementing a set of components to enable robust remote configuration of the launcher's UI. Configuration is transported in JSON format, and a cloud-based platform can be used to generate this configuration file, which is then delivered to the MDM platform. Organization IT admins are, therefore, given full control to enforce not only security policies, preventing unauthorized usage of devices, but they can also customize the end user experience and to ensure that organization's visual identity is appropriately represented during usage. This platform has shown to be of interest to hotel chains, restaurants, schools, libraries and other organizations that wish to rent mobile devices for a limited time to the public, allowing short term use in public places, but preserving user's privacy, while promoting company's identity at the same time.

Keywords	MDM JSON Android Re-branding UI Autonomous Remote

Contents

1 Introduction

Mobile Device Management (MDM) is in relatively high-demand now, due to continued growth of the mobile devices industry. As hardware continues to improve, enterprises see more ways to incorporate mobile solutions into their business, whether by issuing them for the duration of the employment contract or allowing employees to use their own devices, supplying integration solutions. However, until now, most MDM solutions have focused on continuous use by, for example, employees. Although this is a product idea to market – as it increases employee efficiency at a relatively low cost [1], nevertheless, I have observed at work that there is also high demand for managed mobile devices intended for public use. This new demand is most likely a result of continuous reductions to smartphone and tablet prices, as well as a popular trend of digitalization. When renting devices for short to medium periods of time (from a few hours to a few weeks) to end customers, there are several important problems that need to be solved:

- Data privacy: an organization must ensure that user's private data will be securely erased once the loan time is over.
- Device security: device's security policies must not be vulnerable to trivial circumvention.
- Deployment: same or similar software setup must be deployed to many devices.
- Management: devices must be centrally managed, with the ability to control key functions, such as factory reset, data wipe or push notifications on one or more of them from the cloud-based control panel.
- Integration: management platform and devices themselves must allow for integration with the organization's own IT solutions, such as hotel ERB systems, custom applications or organization's own cloud services.

MDM solutions such as VMware AirWatch, IBM Maas360 or Soti provide some of these functionalities, such as data privacy, automatic enrolment or centralized management. However, I was not able to find any solutions on the market that provide even some level of UI customization without resorting to "Kiosk mode"(running a single purpose-built application on the device). [2] [3] [4]

At Hublet Oy, we focus on a different type of MDM, providing a hardware and software solution for public use. Short-term loans for visitors in places such as libraries, hotels,

restaurants of simply lounges in different places is the focus of our business model. While working there, I had the opportunity to observe that niche of the market, and saw rather high demand for our solution, and could also get customer feedback first-hand about the types of features, which customers desired for the intended application of such a solution. Specifically, during the summer of 2016, there has been a lot of interest in an MDM solution that allows for transmission of brand identity throughout use. Among those who were interested were several hotel and restaurant chains, as well as an insurance company. Given the interest demonstrated by our customers, it was clear that a solution, which is targeted for public use of mobile devices is needed, and providing such a solution is the goal of this project. A scalable and fully-featured MDM solution is a very large topic, therefore for the purposes of this thesis I will focus on our solution's most innovative aspect: cloud-managed user interface. Since Android is our target platform, customization of the interface was achieved by developing a highly configurable Android launcher application. Development of this launcher application, its system architecture, deployment and challenges faced along the way constitute the scope of this thesis.

2 Theoretical background

2.1 Section preface

This section describes technologies relevant to this project, why those technologies are used and how they are relevant. It also outlines solutions the market provides as alternatives, what those solutions possess, and what they lack, which makes this project relevant.

2.2 Information availability and aspects of NDA

Due to the highly secure nature of this industry, most of the material and research is not available to the public. MDM solutions are for the most part focused on B2B contracts, and as such contain a fair amount of confidential information, preventing publications. In addition, cloud-managed user interface on mobile devices is something that has never been done before, which limits the amount and quality of relevant materials. It could be speculated that the reason for such low availability is that until recently there was no demand for such solutions due to high prices of smart mobile devices. Regardless of the reason, this solution is highly innovative, which limits the amount of available information that could be applied to its development.

In addition to the above, I am also required to maintain a certain level of non-disclosure in order not to compromise platform security and preserve customer confidentiality. Therefore, I may be forced to use ambiguous names when describing certain components: for example, exposing certain class names can compromise security and as such I will refer to them by their intended purpose instead of a real class name. Also, some custom Android Intents, permissions and Content Provider schemas are used, which I am also not able to disclose. However, all that information is simply implementation details, which are not essential to, or representative of the project's overall idea, and thus should not compromise this paper's academic value.

2.3 Android launcher

In Android, a launcher application is the user's first level of interaction with the Android OS. A launcher typically consists of a set of home screens (desktops), which host several different UI components such as Widgets – miniature application views, that can be imbedded in other applications [5] or static components such as application shortcuts. A user is typically starting other applications from the launcher and there are many applications which do not allow any other UI-based start-up method, other than a launcher shortcut. Most launchers, except those intended for use as "Kiosk mode" applications (an application designed to be the only one users can interact with, and prevents any other applications from being run), also have a so-called "application drawer", which contains icons allowing the user to graphically launch any application installed and launchable on the device. It could be speculated that this is the reason why launchers got their name, however the true reason for this fascinating phenomenon is not within the scope of this thesis. There are several different launcher applications available on Android, besides the default one, developed by Google (see Appendix 1).

2.4 Mobile Device Management overview

2.4.1 Definition

> Mobile device management (MDM) is a type of security software used by an IT department to monitor, manage and secure employees' mobile devices that are deployed across multiple mobile service providers and across multiple mobile operating systems being used in the organization.
> Mobile device management software is often combined with additional security services and tools such as Mobile Application Management to create a complete mobile device and security Enterprise Mobility Management solution. [6]

2.4.2 Typical features

A typical MDM solution usually includes at least the following set of features:

- One or more software platforms supported

- Ability for users to enrol their device to the MDM solution
- Remotely locking devices
- Remotely wiping device data/performing factory reset
- Wiping certain pre-defined data directories on the device
- Role-based authentication

Other features may include location tracking, hardware control (e.g. disabling device camera, or forcing a vibration signal), single sign on integration (e.g. signing into the device with the organization's account), etc. [4]

2.4.3 Relevancy

As the topic of this thesis itself is not specifically about the MDM solution, it is important to explain the relevancy of this component to the topic, on which I am focusing: custom cloud-controlled Android Launcher application. Because this launcher application is targeted at organizations deploying many devices at once to one or more of their facilities, it must have a robust and scalable distribution method as well as be configured autonomously, without end-user interaction. As there were several reasons to make this application capable of being independent (more information on that will be in the next chapter), it's primary method of integrating with the customer's IT environment will be the MDM system that installs it. And while this launcher application can function without a custom settings provider, its primary feature set would be lost. As such I required an MDM system that bundled integration with the cloud API with a specific component that would supply this launcher application with custom setting. It makes sense to offload customer-specific information handling into the MDM component for security purposes and to make sure that the launcher application itself is not extremely environment-dependent.

2.4.4 Existing solutions

There are many MDM solution available today. Among the biggest are SITI MobiControl, VMWare AirWatch and IBM MaaS360. Samsung also provides an MDM solution for its devices, using Knox API. In Finland one of the biggest MDM providers is Miradore. However, all these systems are targeted towards employees, rather than public, and as such do not provide the level of visual customization required for brand communication throughout the end user experience.

Android system itself does also provide a very simple API, called Device Administrator, aimed at use by an MDM platform, but it's feature set is very limited. Most MDM sys-

tems on Android are built using this API, which allows basic functionality such as erasing device data (factory reset), locking the device, password strength checks and enforcement and a few others, such as location tracking. MDM solutions often extend upon these functionalities, although there is not a lot of flexibility one can achieve without building a custom image of the Android system. Samsung KNOX API is based on Android Device Administrator; however, KNOX extends Device Administrator so much that, its base functionality is negligible compared to what KNOX allows developers to do.

The exact details of device enrolment to any of these systems varies on customer per customer basis, therefore they are not openly available, however in general several methods are supported, including out of the box setup, where some generic enrolment software is pre-installed on all manufactured devices, and managed devices simply need to be registered in the MDM cloud using information from the purchase. Most of the existing MDM solutions, however, are targeted towards employees and Bring Your Own Device (BYOD) concept, and as such provide tools for enrolment of any device, usually via manual installation of the MDM administrator agent application.

Since the launcher application, providing the UI, is not dependent on the environment under which it runs, it theoretically is capable of being installed under any existing MDM system, that provides application control capabilities, such as installing applications silently and modifying default application settings.

2.5 Centralized Desktop Management

As this solution's UI is primarily delivered through a launcher application, which is responsible for the visuals and interaction with device's desktop, it is useful to explore centralized desktop configuration solutions that exist on other platforms.

2.5.1 Image-based management

Traditionally centralized management has been done through system images (ROMs). The benefits of using a standardized pre-configured image date back 20 years ago [7], and now cover all the major operating platforms. Metropolia, for example, uses network-distributed images to configure computers inside the campus area, and most system providers have a remote desktop management solution as well, such as Microsoft Remote Desktop or Apple Remote Desktop [8] [9]. While not specifically aimed at it, pre-configured ROMs can include custom desktop configuration. The benefits of this approach are increased predictability – since the system is configured the same on

every machine, the only varying factor during deployment is the configuration of machine's hardware, and a high level if control achieved through manually constructing a precise snapshot of an operating system with all of its settings.

At Hublet, our current solution uses a customized Android ROM image to enable remote management of devices. All of the devices have to be flashed with this custom image, in order to ensure that our software stays on it, gets the required level of access and is secure. This approach has proven to work, with some difficulties, however, there are many disadvantages to this approach:

- From my personal experience pre-set images significantly increase loading time on boot, especially if the configuration is tied to the user profile. In Metropolia campus I have measured the start time of the computer to be on average 112(+/-2) seconds (measured on 6 different computers using the timer on my phone), compared to 12 seconds that my home desktop PC takes to boot, from pressing the power on button, to being in a work-ready state. Similar experience was observed at Aalto university, which uses the same technique to pre-configure their on-premises devices.
- Global changes to the configuration are slow, as they require the computer to restart. While management software usually provides a way to enforce a restart to all managed devices, it doesn't change the fact that due to having to download and update the image, or re-download a new image entirely, any global change to the system configuration will take a long time to deploy, significantly disrupting workflow, and potentially having a significant cost to the organization, if it employs many managed devices.
- Not compatible with BYOD concept. When using a custom ROM device's operating system is replaced with the organization's custom image. If the end user is not willing to replace the system running on their own device with a managed image, then organization's IT department cannot effectively manage their computer. As BYOD gains popularity, ROM-based management solutions will most likely lose value.
- On mobile, as I have mentioned, every device using a custom ROM must first be flashed with it before being shipped to the customer. This adds to the cost of deployment, as flashing large quantities of devices is inefficient outside of the production factory. Furthermore, updating the system is extremely cumbersome, due to a large risk of making device unusable during the process, and a large amount of precautions that need to be taken to enable a robust update

process. Of course, the above issues of long loading times for each new user are true for the mobile devices as well, and we had to work around that by modifying the lifecycle of the device, so that it performs the boot and applies configuration changes during downtime.

2.5.2 Management using administrator agents

There are commercial solutions which attempt to solve the issues caused by Image-based management by enabling end users to enrol any device to management system using administrator agent applications. Administrator agents are a type of applications which can perform device management by accessing various system settings on the system for which they are built. This is not an official term, therefore it's definition is subjective, but the purposes of this text, it shall be sufficient. One of the most feature-complete solutions for device management across both major desktop and major mobile operating systems is ManageEngine. It uses built-in administrator functions in the systems it supports, which allow the admin agent application to perform device management over the network. A centralized management UI can be used by IT administrators to enforce security policies on managed devices. Because there are differences between operating systems as to which settings are and are not available for the admin agent to control, the feature set of any management solution will vary on a system by system basis.

For Windows ManageEngine does provide a certain level of UI management through desktop configuration such as shortcuts, wallpapers, text DPI, resolutions, tray icons, as well as modify the admin agent logo to adjust the organization's brand. Through application policies it is also possible to modify the visuals of the operating system by installing applications such as Stardock Fences, which allow for changes to how desktop behaves. It should be noted, however, that while management solutions provide a way to enforce application policies, they do not necessarily provide a way to control configurations of these applications. For example, we frequently get requests from customers to enable an application to run with certain settings on all their devices, and while the increased level of access does allow our admin agent to access application's internal persistent storage directory, it requires specific application knowledge to be configured each application to be run with pre-set settings. [10]

2.5.3 Mention of relevancy

As a final note in this section, I would like to mention, that most of the existing solutions are aimed at companies, which wish to manage devices used by their employees. As such they are focused on separating private data from corporate data, as well as long-term usage and long-term accounts and integration. The solution described in this work is aimed mostly at public use, in schools, libraries, hotels or restaurants, as well as any other organization wishing digitalize their customer experience. As such it must accommodate short-term usage, and ensure privacy between loan periods. All data and settings are treated as temporary and persist for the duration of the loan, which simplifies data management, however settings changes must happen very quickly as long loading times for every loan would be very inconvenient considering that usage sessions are much shorter.

2.6 Technologies used

2.6.1 Android mobile operating system

Android is the largest mobile operating system in the world. It is founded and maintained by Google and is an open source technology.

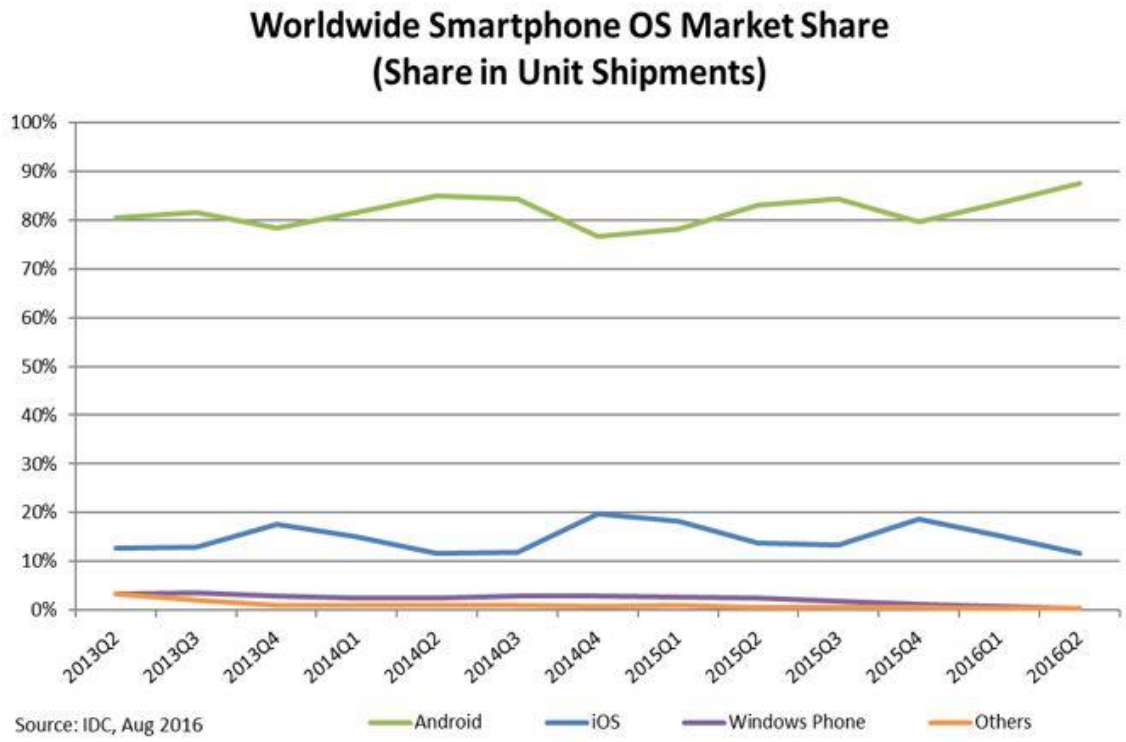


Figure 1 Source: IDC, Smartphone OS Market share 2016, 2015

As can be seen from the figure above, Android had 87.6% market share in the second quarter of 2016 [11]. This dominance was reached through a non-restrictive distribution

system and open source licensing model. Being open source Android significantly lowers the barrier to entry for hardware manufacturers, as they have a fully featured mobile operating system that they can modify to suit their needs or release devices with the stock version provided by Google. In addition, Android does not enforce the usage of Google services such as Google maps and Gmail, even though the system comes with those services built in. As such all of end-user components, including the app store can, in theory, be replaced by vendor's own versions, if they so choose. Together these features lower the barrier to entry and facilitate user-centric competition, which, as many believe, has led to Android's dominance in the mobile OS market. [12] [13]

There were multiple reasons why Android had to be chosen for this MDM platform. The biggest reason, however, was the fact that both I and my colleagues were already familiar with the system, and the first iteration of our product uses Android with a custom ROM image. As such we had a lot of experience with the Android platform, and transitioning to a different one would incur too high a cost. Furthermore, it was decided to use Samsung KNOX technology to provide our MDM agent with increased level of control and security. As Samsung uses their custom version of Android for all their devices, and KNOX is only available on Samsung devices, Android would have to be the platform for which we need to develop. And finally, among the two major operating systems for mobile devices (iOS and Android), Android is the one which allows for modifications to the user's home screen and application drawer, making it the best candidate for our solution's aim to provide customers with extensible rebranding options.

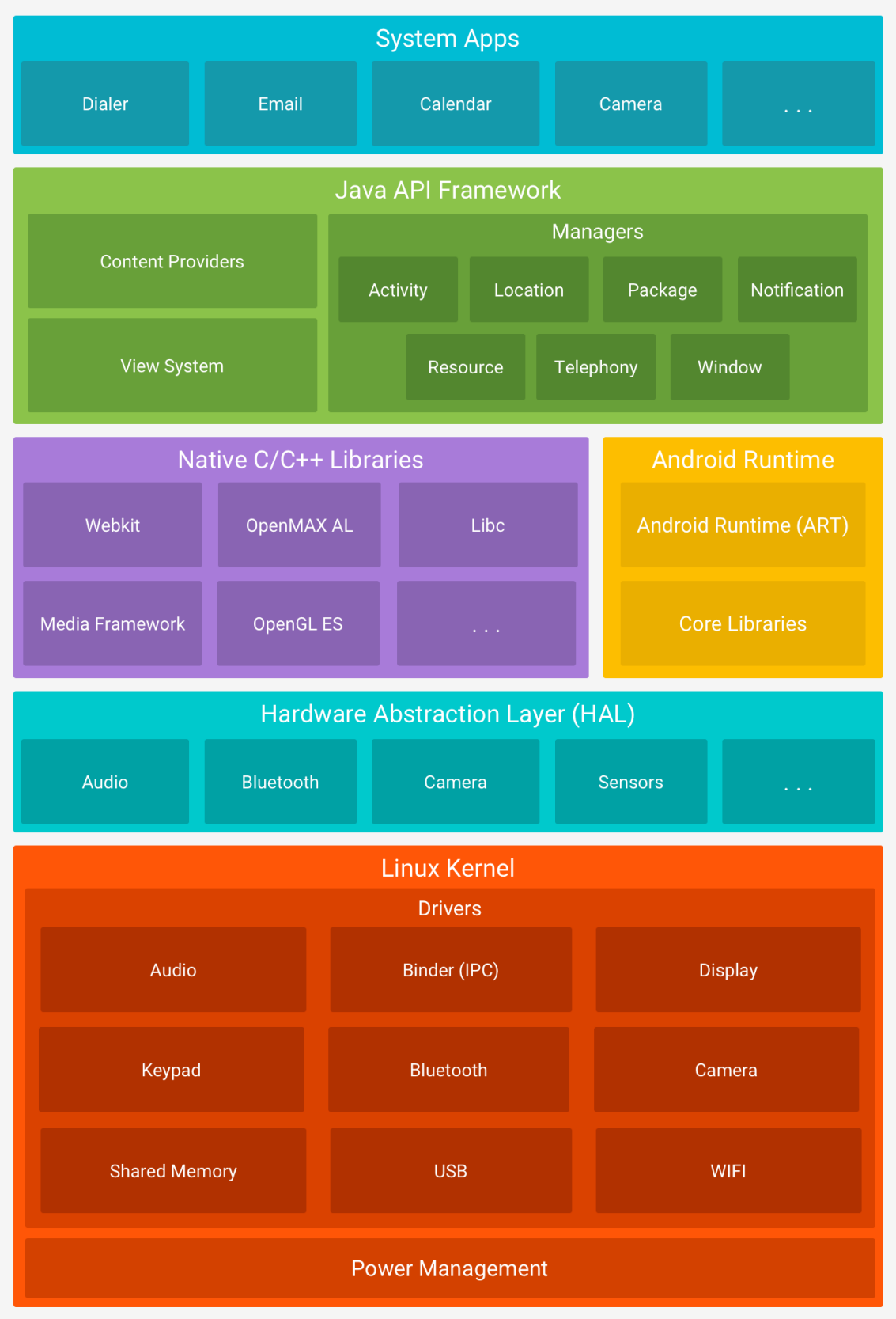


Figure 2 Android System Architecture. Source: [14]

As Figure 2 shows, Android runs on Linux Kernel, which provides the lowest level of interaction with device's hardware components, threading, power and memory management. On top of the Linux Kernel, Android uses Hardware Abstraction Layer (HAL), which provides higher level access to device's hardware such as Bluetooth or Wi-Fi to application developers. Prior to version 21 of the android API (Lollipop), Android applications ran on Dalvik runtime. On API level 21 and higher, Android Runtime (ART) is used as the default runtime, and applications run in separate processes, each with their own instance of the runtime. Applications may even declare to use multiple processes per application, and we utilize this ability in our solution, to enable smoother operation and lower the risk of reaching memory limits while running RAM-heavy tasks. Both ART and Dalvik execute DEX bytecode, which is a bytecode format designed for Android, and is what Android applications compile to. Again, referring to Figure 2, it can be observed that native C and C++ libraries are also used together with the runtime (though they are not run using it). Android uses many open source C++ such as OpenGL and, in modern versions, Vulkan for graphics, as well as Libc for native C code, which are exposed to app developers through Native Development Kit (NDK). The core of most applications is build using Java API, through which the entire Android OS feature set is exposed to application developers. [14] [15]

2.6.2 JSON

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. [16]

JSON is used in this project, as a format for the launcher configuration. I decided to use Google's Gson open source library for processing of this format because it is highly tested and maintained by Google, which provides a certain level of guarantee that it will not be abandoned. JSON is the de-facto data format for back-end web APIs, which, combined with its high readability and the fact that providing a web-based centralized management was one of the primary goals of this project, made it an obvious choice as the data format. In addition, JSON has low network footprint, due to its compact size, which reduces latency and can be especially beneficial in areas with weak or inconsistent connectivity.

There are certain difficulties which must be taken into consideration, however, when using this format:

- JSON is not strict. It does not have a wide variety of types. For example, it does not distinguish between floating point numbers (e.g. 1.563) and full integers (e.g. 2). Gson provides parsing capabilities to transfer JSON numbers into the correct Java type, but it requires knowledge of which type to expect from the source.
- JSON is not a native Android configuration format. App configuration in Android is stored in XML files, and the same format is used for view configuration, therefore JSON files need to be parsed before settings defined in them are applied. This introduces extra complexity into the application design, as it adds a rather significant amount of complexity to the code.
- JSON schema cannot be enforced. Enforcing compliance with a custom JSON schema is outside the scope of this project, and would also introduce too much overhead, as all configuration files would need to be read twice: first to verify validity and then to parse the actual data. Since smooth and responsive user experience was one of the goals for this project, such approach could not be taken. It is possible to verify validity of the file upon creation, however that is outside the scope of this thesis.

2.6.3 Samsung KNOX

Samsung KNOX is one of the most secure mobile management platforms on the market. It consists of multiple features, that operate on different layers of the device on which it runs: from extensive OS control, to verifying OS integrity and detecting whether the device has been tampered with. KNOX operates on very low level, going as far as being embedded into hardware of KNOX-enabled devices. It received certifications of technology security requirements from various government organizations throughout the world, including Finnish KATAKRI II certification (approved 09.2015). [17]

The primary 3 reasons for choosing to tie our MDM platform with Samsung KNOX were

1. It is extremely thorough and secure, which means that using it would allow us to guarantee data security to virtually any organization, including military.
2. It is built into every modern Samsung device. This was a critical feature, as manual enrolment of large quantities of devices into our MDM system would significantly slow down deployment and incur heavy costs. Samsung Knox includes a feature for enrolling any device into an MDM system over the air out of

the box, without any manual installation, which would allow our customers to easily set up their Hublet devices when installing them on site.

3. Our current iteration already utilizes Samsung devices and we have strong contacts and relations with the company.

One of the biggest issues with Samsung Knox is that this technology is still relatively new. Samsung KNOX was first announced at Mobile World Congress 2013 on 26th of Feburary. [18]. This would make it almost 4 years old. While for most IT sectors this is a long time, usually IT security takes longer to mature. KNOX API architecture is still being actively changed, and due to its low level integration, there can be rather large gaps in available features between different devices, as KNOX cannot be fully updated due to it being embedded in the hardware.

2.7 Usage of Open Source technologies

Open Source is getting more and more popular. The quality of open source libraries and solutions can, in my experience, easily compete with commercial alternatives, and as such various open source technologies, besides Android OS itself, have been utilized in this project. A downside to open source code is that there is no warranty, and no guarantee that a library will be maintained in the future. In addition, open source code often does not go through as rigorous testing as commercial code does, and as such can contain bugs and other issues, which can affect the product in unexpected ways. As of this moment the following open source technologies are used in this project:

- Android and it's support libraries
- Apache Commons (utility methods)
- Gson (JSON processing)
- Mockito (for tests)
- EventBus (for help with synchronization)
- Picasso (for image processing)

I attempted to avoid using too many dependencies to minimize security risks and lower the application package size.

3 Practical work

3.1 Launcher system architecture

3.1.1 Preface

This section describes system architecture of the configurable launcher application, which is a focus of this thesis. It will contain description of context in which the application needed to be developed, important considerations for the overall application design, and details of the application's architecture and the components it requires. More logic-oriented description of the core concepts of this application will be presented in the implementation section.

3.1.2 "Chokepoint-based design pattern"

One of the ways to prevent so-called "crash loops", where the launcher crashes, and upon being restarted by the system immediately crashes again, was to follow the so-called chokepoint-based design pattern. I coined the term myself for lack of a better alternative, because the idea is that all the possible code paths must go through this common sequence, a "chokepoint", of sorts, hence the name.

This software design pattern is based around "chokepoints" – pieces of code which are very simple, and unlikely to crash, that start off a complex sequence of code, which may crash. A requirement for a chokepoint is that it must always run first, and that all chokepoints must be in a "normal" user flow. That is: they must never be created to cover edge cases, such as, for example, when the program is run on a device for which it is not intended. The assumption is that when appropriately used, more generic chokepoints will cover those edge cases anyway. Another requirement of a chokepoint is that it must only contain two code paths: normal and failure. A normal code path kicks off more complex code, while a failure code path must attempt to circumvent the issue and restart the complex code. In Android it was usually a good idea to treat `onCreate` methods as chokepoints, because the system runs these methods before most of the app's code.

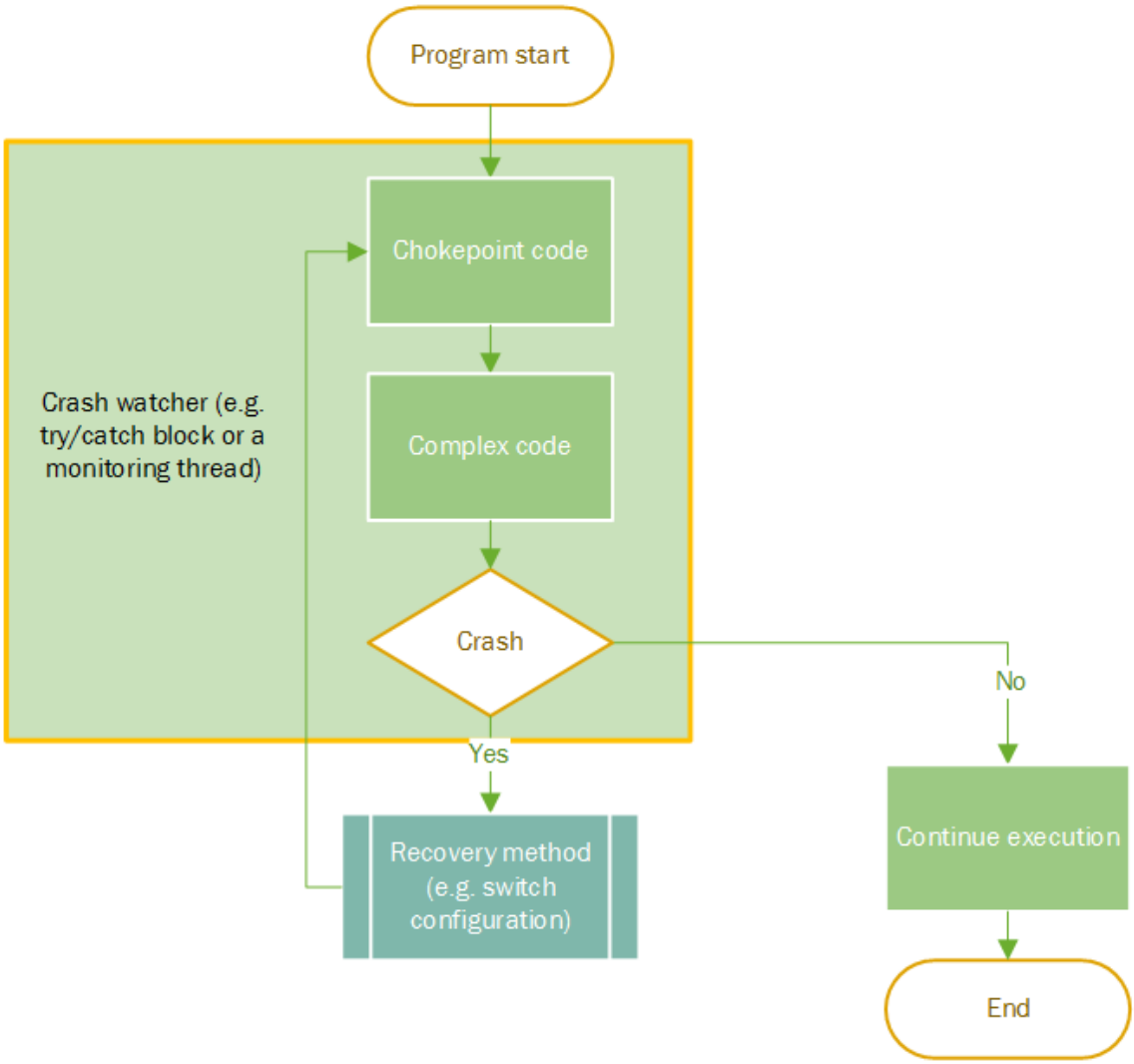


Figure 3 Chokepoint-based code design

From the diagram, above it can be seen that there are a mostly two ways that this approach can fail: when the recovery method crashes and if the crash watcher itself crashes, in case a more complex monitoring method is used (such as a monitoring thread). For that purpose, it is usually a good idea to keep the crash watcher and the recovery methods as simple as possible, so that they do not increase the chances of the application crashing, instead of lowering them. In my code, I tried to avoid complex monitoring logic and used generic try/catch blocks when writing chokepoints.

```
try {
    launcherSettings = new LauncherSettings(HubletLauncherAppli-
cation.this, launcherJson.getAsJsonObject());
    updateLauncherSettingsListeners();
} catch (ClassCastException | NullPointerException e){
    Log.e(LOG_TAG, "CRITICAL ERROR! Json settings null or wrong
format. Settings will not be applied. Using built in de-
faults.");
    Log.e(LOG_TAG, "Launcher settings: \n"+ HubletUtil.jsonTo-
PrettyString(launcherJson));
    e.printStackTrace();
    if(launcherSettings == null) launcherSettings = new Launch-
erSettings(HubletLauncherApplication.this);
}
if (allSettingsReady()) unlock();
```

Figure 4 Example of a chokepoint: handling errors in fetching launcher configuration data

In Figure 4, `launcherJson` could be null or malformed, which would crash the application, as it requires `launcherSettings` variable. If an error occurs the recovery method is to instantiate the variable using baked in verified settings with the default constructor – `new LauncherSettings(HubletLauncherApplication.this)`. This code runs in the Application class and is invoked as one of the call-backs set in the `onCreate` method. When all settings have been correctly parsed, and verified, this method will set a flag that unlocks application's main UI, allowing for further interaction. While the application is locked, further interaction is prohibited and a pre-set image is displayed. It should be noted, that in the code snippet in Figure 4, the "Complex code" from Figure 3 is also instantiation of the `LauncherSettings` class. This code launcher parsing logic for the launcher settings JSON, and while it is repeated in the recovery block, the general assumption is that baked in settings used in the classes "recovery constructor" are well-tested and will not crash. Furthermore, the "Continue execution" node in Figure 3 does not prohibit further chokepoints in that code path, assuming they are each covering a complex code path responsible for one isolated set of functionality (e.g. parsing, displaying UI elements, animating or storing data). Chokepoints may also be located inside the recovery code path, but that is not a good practice.

3.1.3 UI interaction

Typically, Android launchers enable user navigation between launcher screens by having two sets of swiping views – home screens and application drawer (from now on: just drawer, for short). Users switch to the drawer by pressing a button that is fixed on the home screen views and they can switch back to the home screens by pressing Android's pre-defined home button (on many devices it is a physical button, instead of an

active area on the touch screen). This method of interaction stems from Android's default launcher, which many of the existing launchers are based on. Drawer, as well as home screens, in all publicly available launcher applications, are based on a fixed grid. Users can move items between grid cells, however they are not allowed to modify the grid size and only widgets can usually be scaled in size, in a very limited way. Furthermore, other Android launchers are not autonomous and require user interaction to change any settings, and default launcher application requires user interaction to be set in system preferences. Due to the requirements of this project to have a launcher that can be deployed, configured and run without any user interaction, forking Android's default launcher and basing this solution on that was not an option. A lot more freedom in UI customization was required than a fixed grid can provide, and refactoring the code to use a different layout would have been less efficient than starting from scratch. Also, entire MDM solution that the launcher belongs to, must be fully autonomous from the end-used perspective, and not require any interaction to configure. Therefore, this launcher was designed and implemented completely from scratch, without any existing codebase, and UI interactions for it were designed to meet project's requirements.

Primary means of interaction within this launcher application was designed to be swiping. There are multiple possible desktop and drawer screens that the launcher can be configured to have, however since re-branding and completely customizable UI was one of the primary goals of this project, I designed the UI interactions in a way that does not depend on any buttons. By swiping between desktop and drawer screens vertically and within desktop and drawer horizontally, users can navigate launcher's UI components without the need for extra pre-set buttons, allowing customers more control over launcher visuals. In addition, since Android's visual convention is to animate between drawer desktop views by shifting them vertically, while allowing users to swipe between individual screens horizontally, visual convention is also not broken. By using a customized `ViewPager`(`VerticalViewPager`) for vertical swiping between drawer and desktop sections, while having standard horizontal `ViewPager` for navigating between screens it was possible to achieve seamless navigation between all application components without any extra fixed UI components that customers can't modify, and a button to switch to drawer using the convention that most android users are accustomed to is also available as an option, should the customer choose to use it.

3.1.4 Launcher components

The launcher application contains a set of higher-level components that are each responsible for a subset of the overall functionality. One of the primary reasons for splitting application code into components was to facilitate chokepoint-based design requirement of having high level chokepoints for each set of complex functionality. The following diagram shows launcher component structure as well as primary responsibilities of each component:

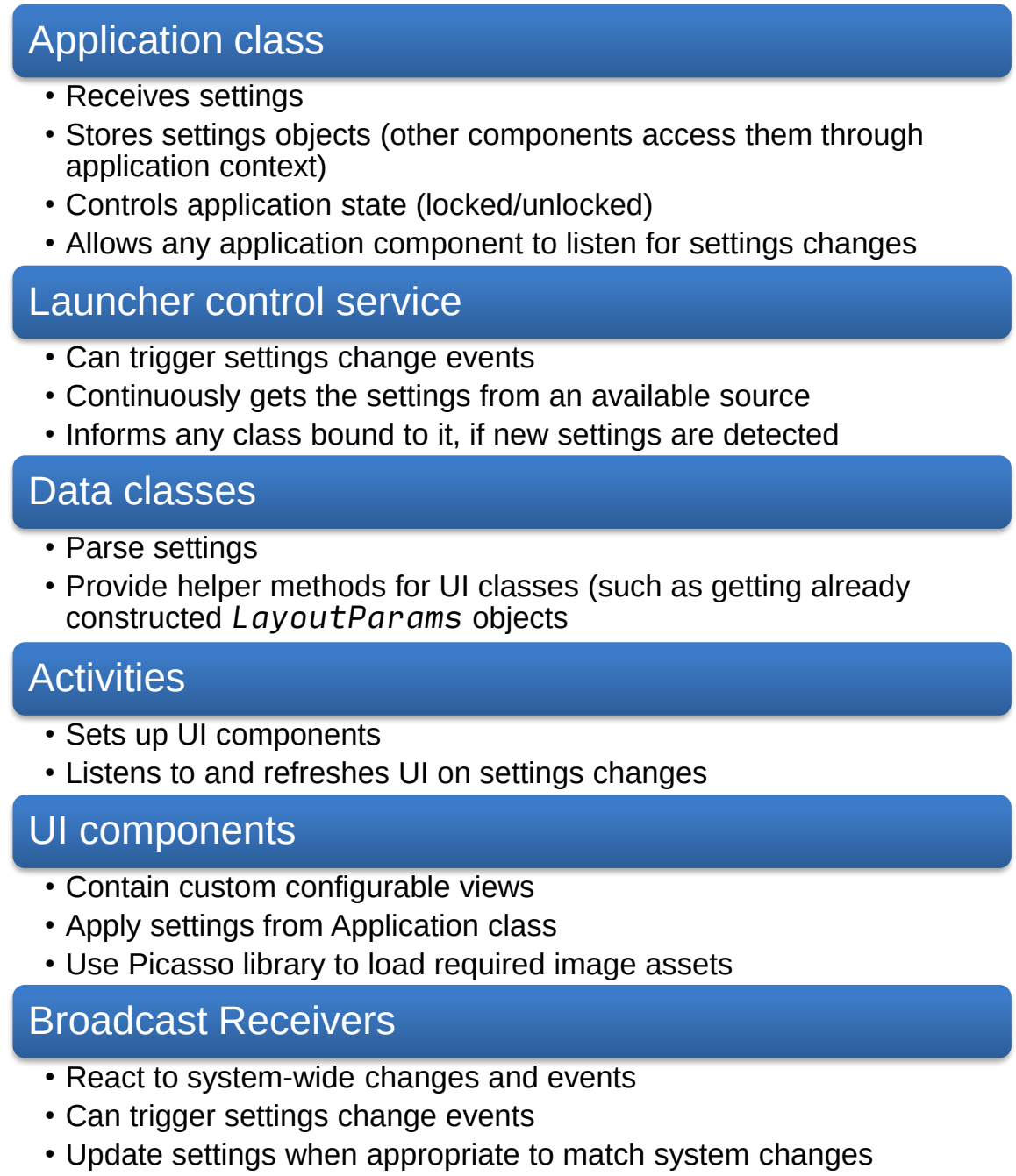


Figure 5 High level component overview

To maximize code flexibility, components have been designed to have as little dependency on each other as possible. For example, while in the current implementation Activity and Fragment classes are the ones listening to settings change events and reloading higher level components when any are detected, the implementation of the listener interface can be made in any class, including individual UI components, which may wish to listen to changes specifically pertaining to that component. Also, while custom UI components can be configured using the data (settings) classes, they are not required for those views to display. In addition, none of the settings classes require other application components, and can be easily copied and used in another project, if desired.

For platform compatibility, I attempted to avoid creating too much custom view logic and layouts, and instead used combinations of Android provided views, configuring them through mapping JSON settings to Android `LayoutParams` objects and other Android-provided configuration methods. The reasoning behind this approach is to avoid having too much custom, unconventional logic, which may introduce severe problems as the Android platform develops. The assumption in this case was that further versions of Android will attempt to avoid breaking changes to components using Android system APIs, however the same is not guaranteed for components implementing custom logic.

3.1.5 Application lifecycle

A default launcher application in Android starts at boot time automatically. By design of our solution, the MDM system is responsible for configuring devices to set this custom launcher as a default one. Once the launcher starts it is responsible for obtaining settings from a content provider, if one is available, and applying them. Application class is used to start the launcher control service, which begins obtaining settings. As that operation, can take a non-negligible amount of time, the application class continues execution, by starting an activity that provides the user with a non-interactive basic activity to communicate that the launcher is starting. When the launcher is ready to display its home screen, this basic activity is automatically removed and the primary activity is started. This primary activity is the highest-level container for all the launcher's UI components. As of this moment the launcher only contains these two activities. It may be beneficial in the future to add more activities, for example: for configuration from a device running in "administrator" mode, however that is outside the scope for the current implementation target.

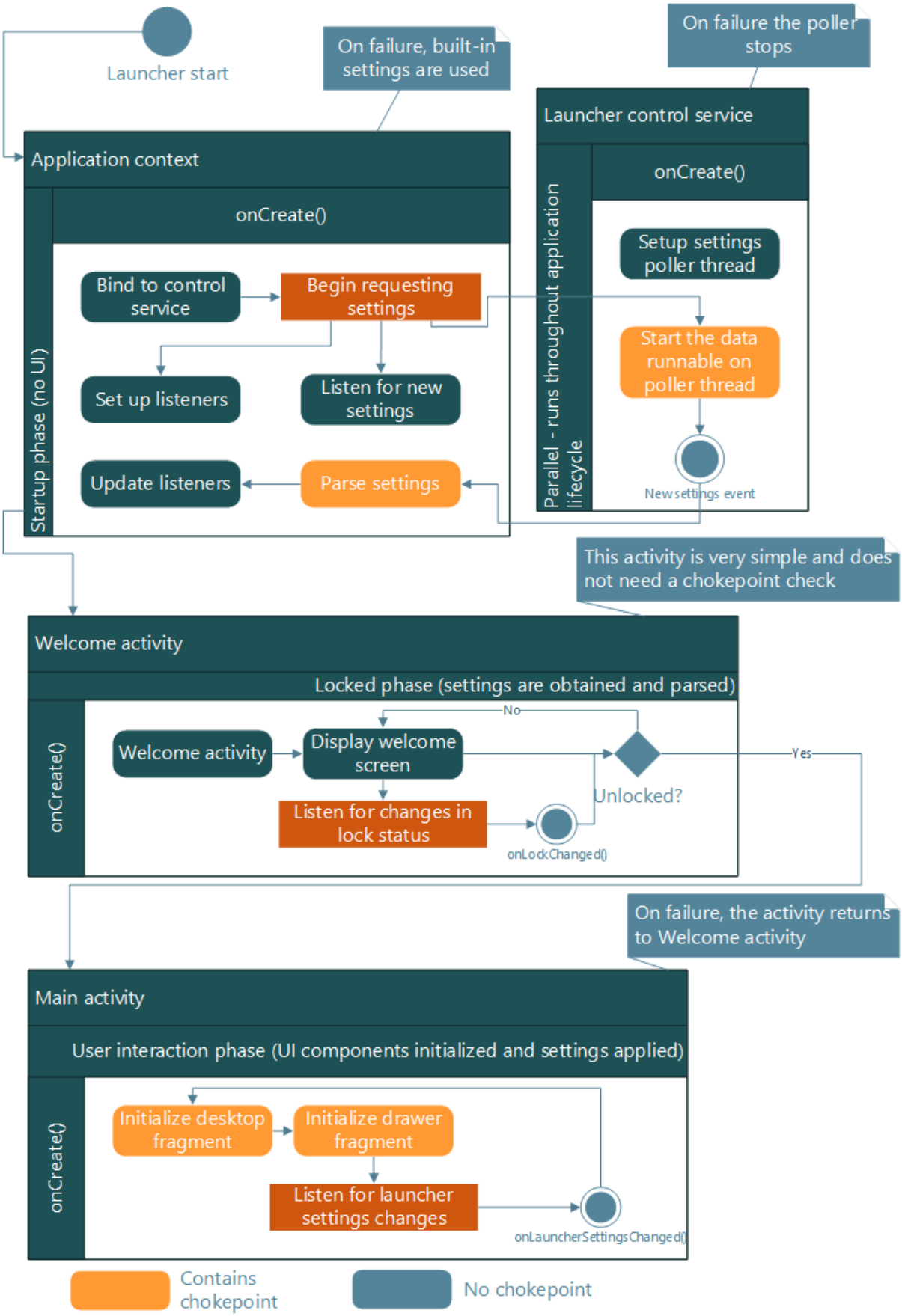


Figure 6 High level application start-up sequence with chokepoints

Figure 6 details individual high-level tasks of each component in the start-up sequence, as well as chokepoint placement in each of them. As can be seen from Figure 6, there is no direct connection to the launcher control service, due it being started by the Android system. While it can often be assumed that the service starts almost immediately upon being bound to, it is not guaranteed, meaning that data exchange between the service context binding to it, must be properly synchronized to avoid null pointers and race conditions. Because the service is local (i.e. it runs in the same process as the main application), there is no need to implement interposes communication, and using a simple interface call-backs is sufficient. The application class adds itself to the service's listeners for the `OnNewSettingsListener` interface and, when the service obtains new settings, it simply calls the appropriate interface method on all listeners.

Figure 6 also shows that the only way to launch main activity is through the lock status listener. When Application class creates all settings objects (either with settings from the service or built-in fallback settings), it changes the lock flag and informs all listeners for the lock status of this change. Welcome activity implements the lock listener interface. If, and only if, it detects that the application is unlocked, it will dismiss itself and start the main activity. This was done for two main reasons: to prevent the event of UI changing due to new settings while the user is interacting with the application (UI refresh is forced the first time, while further updates to settings will apply when the affected view is not in focus), and to prevent the user from seeing potentially unauthorized content.

3.1.6 Performance and reliability considerations

As a primary way of interacting with the user, a launcher application requires more reliability than most other types of applications. If a launcher crashes and does not recover, the user has no way of launching any other applications, and in the case, that this launcher is the only one installed on the device, it can be extremely difficult to recover, as the user would be unable to access the MDM UI to reset the launcher or open application store to downloading and install a different launcher. For that purpose, reliability had to be one of the primary concerns during design of this application.

Following chokepoint-based design is one of the ways to keep the code base relatively crash-resistant, however that design pattern is focused on preventing crashes, at any cost, rather than attempting to recover and preserve application functionality. While it is extremely important that the application does not crash, several other precautions have been taken to account for less critical errors, and preserve most of the application functionality, while disabling that, which has failed:

- Settings classes were designed to use fields, each having a default value. If a setting fails to be parsed, the field's default value will be used for that parameter.
    - For example:
      ```
      public AnimationSettings touchDownAnim = new AnimationSettings();
      ```
    - The above field contains animation settings for a UI element settings object. `AnimationSettings` have a default constructor which simply does not have any animation defined.
    - A UI element using such settings will not have any animation, but it's other functionality will still be intact.
- All items that require media assets (e.g. images) use a default image that comes bundled in the application package. When an asset is requested from the network, it replaces the default asset, however if that operation fails, default assets will be used for the requesting UI element, preserving remaining functionality that may not have been broken.

In addition to reliability; performance has also been considered when designing the application's overall architecture. Since the launcher's main UI consists of two sets of views, each of which using it's own layout and view hierarchy, Android's `ViewPager` classes have been chosen as a primary UI elements to display launcher screens. Per Android convention, `FragmentPagerAdapter` is responsible for binding data to `ViewPager` pages. For this launcher two adapter classes could be used: `FragmentPagerAdapter` and `FragmentStatePagerAdapter`. `FragmentPagerAdapter` is optimized to work with Android Fragments (UI components that can be used to display relevant data in a standardized layout) [19]. Because launchers typically do not have many pages, and the ones this launcher uses can also contain media obtained from the network, `FragmentPagerAdapter` is a better choice than `FragmentStatePagerAdapter`. While `FragmentStatePagerAdapter` conserves more memory by destroying the fragments when they are not in view, it also means that those fragments need to be fully recreated, their settings re-applied and assets either downloaded or fetched from the cache, every time a user swipes them into view. As such for small number of complex fragments such as those used by this application, `FragmentPagerAdapter` works better, by allowing Android system to recycle existing fragments instead of destroying them.

To minimize network footprint and provide users with smooth experience, image assets are loaded using Picasso open source library. Picasso automatically downloads, caches and scales images and can set them into target views [20]. Since it is rather popular, developed and maintained by a well-established enterprise and contains all the required functionality for this project's requirements, I decided to use it for image processing and caching. Picasso maintains an automatic cache in the app's private directory on the device, which allows users to see media assets even if there is a network interruption, as well as significantly reduce loading time (varies depending on the amount and size of assets set by the MDM system administrator).

3.1.7 Reading and parsing configuration data

Before data is read, it must be acquired from a content provider. By design, Hublet MDM solution consists of a set of components that interact with each other, allowing the customer to configure a set of devices that run this solution per their requirements. Settings for each component, including the launcher are therefore customer-specific and can have varied origins, as well as customer-specific integration (e.g. they may be modified based on end-user authentication method – if the user authenticated with Shibboleth that might result in different device settings than if they did not authenticate at all, and just picked the device up to watch a video). To allow greater flexibility without inflating launcher code too much, it was decided that the best approach would be to offload the data acquisition to a separate Android application – the MDM agent app, which connects to backend network services, handles authentication and obtains session-specific settings both for the device itself and any other Hublet components, such as the launcher.

To allow for even more flexibility of deployment, it was decided to use a standard Android mechanism for providing application-specific data – Content providers. A content provider is a class that can be implemented in any Android application. It reads data from files and data structures (e.g. an SQLite database) and can provide pointers to that data to any application that is authorized to query it. The pointers are `Cursor` objects, which in Android are used to retrieve data from SQLite databases. A cursor is usually pointed by the Content provider at a specific row that matches caller's query criterias. Caller application, in this case: the launcher is then responsible for using the provided cursor to retrieve the data from the table. Because this is a native Android mechanism, it can be safely assumed, the breaking changes to this

One of the primary requirements for configuration data was to account for malformed or invalid input at a very high level. Since configuration data is delivered in JSON format, there are no strict rules for it's structure or conformance of data within to the specific scheme expected by the launcher's parsing logic. For that reason it was decided to offload all parsing to a set of specialized settings classes, each of which takes a Gson library's JSON object and attempts to read and map it's data to the respective Java fields.

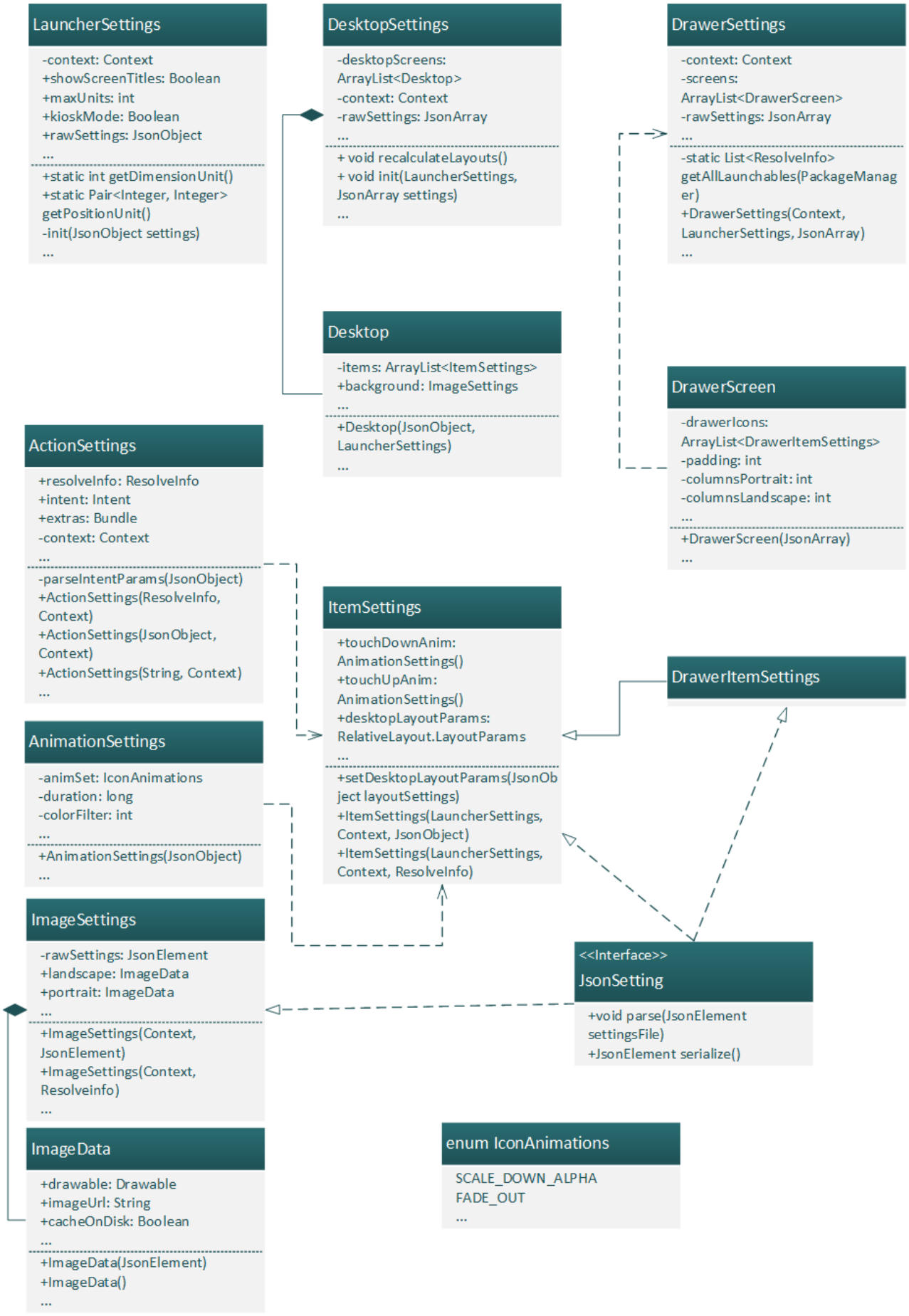


Figure 7 Settings classes

Figure 7 shows a set of classes that parse JSON objects. In interest of space and for the sake of being concise, insignificant class fields and methods are not displayed in the diagram. These classes can all be instantiated using JSON configuration. As could be seen from Figure 6, launcher control service is the component responsible for acquiring new settings. It retrieves JSON in String format from a provider, if any are available in the system, compares it to the currently available settings (if none are active, it will compare to null), and if the newly obtained settings are different, it attempts to parse the string into a Gson library's object representing the appropriate JSON structure (e.g. `JsonObject, JsonArray` or `JsonPrimitive`) and if that succeeds, the service calls listeners for those settings groups that have been changed. Since it is a locally bound service, any application class that runs in the same process can technically bind to it and register itself as a listener. However, in the current design, as illustrated in Figure 7, the application class is listening for those changes and propagates them through its own listener interfaces, that any application component can listen to by obtaining application context and adding that component to the listener list, as well as implementing the appropriate interface, which can look something like this:

```
HubletLauncherApplication application =
((HubletLauncherApplication) context.getApplicationContext());
application.addDesktopSettingsListener(this);
```

Figure 8 Example of making the current object listen to changes in desktop settings

3.1.8 Media asset loading

Launcher may be configured to retrieve assets from the network. In clear majority of cases, it is expected for those assets to be images, though it may also be icon labels, sounds or any other form of media. However, in the current implementation only custom images for icons, as well as static graphics on the home screen are supported. Loading of images is handled using Picasso library for network images. This is typically done when creating a view that holds the image. Thus, until an android view requiring the image is not created, the settings object contains that image's url. When Picasso loads an image using the same settings object, it will use a locally cached version of this image, stored using library's automatic caching.

Icons may also be created using Android's `ResolveInfo` objects, as can be seen from constructor of `ImageSettings` class in Figure 7. In this case icon drawable is being retrieved from the `ResolveInfo` object and stored in the settings object's instance. This may be improved by using a memory mapped file to store icons instead of keeping

them in memory all the time, however since the number of screens in a launcher application is not usually large, and the user is expected to navigate between them relatively often, it may still be beneficial to keep the icon image data in memory with the settings object, to ensure that there are no slowdowns when switching between launcher screens.

3.1.9 Localization

Localization was one of the requirements for this project. When designing the system architecture for this entire project, I tried to avoid complexity as much as possible, to make the code more readable and each part of the system easy to understand within its context. Localization was one of requirements, that could introduce a lot of complexity into design of the application, so it was important to implement it such that it did not introduce any more complexity than absolutely required. As such a decision was made to allow any configuration JSON object to contain a `localizations` attribute. Within this attribute must be a set of language code and each language points to another JSON object. That JSON object is essentially a copy of the parent configuration JSON, and can contain attributes matching any in the parent object. After the parsing logic reads the parent, if it encounters `localizations` attribute, it will read the object under the language code matching the system's current language, overwriting any attributes in the settings with the ones provided by the localized version. This way the complexity of the code is reduced, and customers have more control over localization: they can swap any attribute, including images and dimensions of items for certain languages, but don't have to provide full translations for every item, if they do not wish to.

3.2 Launcher codebase implementation

3.2.1 Preface

This section is an overview of code. It contains little information on the design philosophy or justification for why a decision was made – that is discussed in the previous section. Instead this section details specifics of application's code as well as certain data structures. It may provide justification for why particular logic was chosen for certain parts of code. Due to high amount of details involved in the implementation, the coverage is not full – only the functions most relevant to the scope of this thesis are covered.

3.2.2 Start-up sequence

Application's custom code being is the Application class's `onCreate` method. When the launcher starts, it binds its local control service in auto-create mode, meaning if the service is not running, Android system will automatically create it. After that it initializes

data structures containing references to listeners. Other application components can then listen to changes in settings, as well as lock status. Finally, the application checks the status of the lock flag – if the launcher was restarted, and already has valid settings available, the lock flag would have been saved in the persistent settings, and can indicate that there is no need to wait for settings to process and verify. Then the Welcome activity is started.

Welcome activity first checks whether the application is locked by checking the lock flag from the application context (Application class). If the launcher is unlocked, it will immediately start the main activity without displaying anything. If the launcher is still locked (i.e. settings have not finished processing yet), welcome activity displays a basic full screen view without any interaction, to inform the user that the launcher is being started. There is capability for the welcome activity to display custom content defined by the customer, however that functionality is currently disabled before the application can be tested enough to ensure that this functionality would not interfere or crash other, more important functions. Welcome activity registers itself as a listener to the lock status of the application, and once the Application class informs it is unlocked, the Welcome activity is automatically dismissed with an animation and the application's main activity is started.

To prevent malicious use, the first thing that Main activity does is re-verify application's lock status, and if it is locked (i.e. Main activity was started explicitly, bypassing the Welcome activity, while the application was still locked), it will return to the Welcome screen (activity) and stop further execution. If the application is unlocked normal flow continues. Main activity places all its further UI components inside a custom `RootLayout` class extended from Android `LinearLayout`. This is done to allow for global overlays to be easily displayed over the rest of application's UI. `RootLayout` contains a view that is invisible by default, which overlays all other views in the application. Using a static method, any UI component can obtain a reference to the overlay to

display relevant information, such as the number of pages, the user can swipe between.

```
mainViews = new VerticalViewPager(this);
LauncherSettings launcherSettings = ((HubletLauncherApplication)
getApplication()).getLauncherSettings();
DesktopSettings desktopSettings = ((HubletLauncherApplication)
getApplication()).getDesktopSettings();
DrawerSettings drawerSettings = ((HubletLauncherApplication)
getApplication()).getDrawerSettings();
mainViewsAdapter = new MainViewsAdapter(getSupportFragmentMan-
ager(), launcherSettings, desktopSettings, drawerSettings);
mainViews.setAdapter(mainViewsAdapter);
```

Figure 9 Setting main views (desktops, i.e. home screens, and application drawer screens)

After Root layout is initialized, the *VerticalViewPager* is instantiated to display launcher's primary sets of views, as demonstrated in Figure9. In the current implementation, this vertical view pager will have 2 pages: Desktop and Drawer, allowing the user to swipe vertically between them. As per the android standard, mapping data to views when using a recycler view such as *ViewPager*, is done by an adapter class (*MainViewsAdapter* in Figure 9), thus all application settings are passed to the adapter and it is set to manage data for the vertical view pager. Finally, main activity registers a broadcast receiver for the home button press, which returns the user to the home screen set of views, to follow Android's convention.

3.2.3 Data acquisition and processing

The idea behind a "push UI" is that once the administrator makes changes, they should quickly and automatically reflect in devices they control, be "pushed" to those devices. To obtain new settings as fast as possible, I created a service, which runs a thread with Android Looper attached to it, that continuously asks the content provider for updated settings. Application that implements this content provider is responsible for obtaining correct settings in time using whichever mechanism it prefers. On the launcher control service, a runnable is posted to that Looper thread on the service using Android's *Handler.post()* method. This runnable obtains cursor objects for each set of configurations (desktop, drawer and global launcher configuration), converts them to Gson JSON objects and compares them to existing settings. If the settings do not match (i.e. they are new), and the parsing succeeded, it will trigger it's call back methods on the listener. Only one listener can be set per runnable instance. At the end of one cycle the runnable will use Android's *Handler.postDelayed()* method to post itself again to the same thread after a few seconds. This cycle repeats until the error count of the run-

nable reaches a certain threshold or the application shuts down. A method in the service must be explicitly called by an object that binds to it to start this settings acquisition. Currently it is started in the Application class's `onCreate` method.

3.2.4 Mapping JSON configuration to Android view configuration

Most of the challenges in mapping JSON configuration to Android view parameters lay in creating Android `LayoutParams` object from the respective JSON object. Since the project required to allow control over precise position of items on the home screen, no grid is used there. Instead I use a custom `ManagedRelativeView` class, extending from Android's `RelativeLayout`, and each item placed on the home screen is added to that view. As such, each desktop item configuration must be parsed into extended `RelativeLayout.LayoutParams` object. `ItemSettings` class is responsible for this parsing. For most of the JSON parsing I used a loop and switch statement method for plucking attribute keys that program currently supports, and ignoring other values. This makes the code more readable and allows for much easier addition of extra parameters as the application develops to support them.

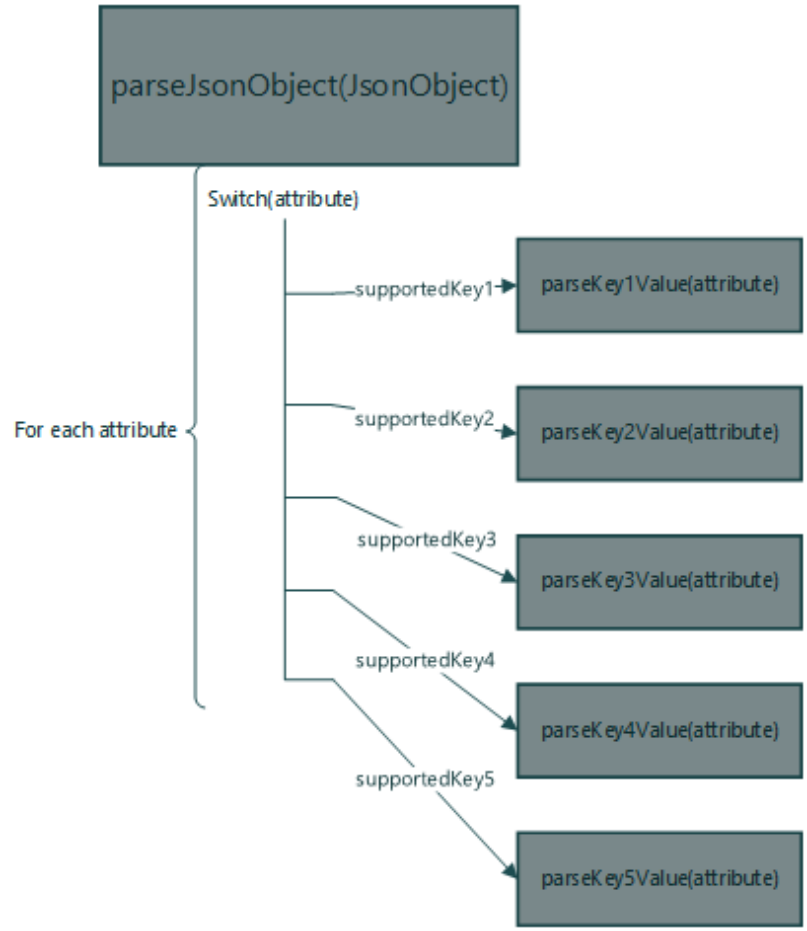


Figure 10 JSON object parsing logic

```
for (Map.Entry<String, JsonElement> entry : layoutSett.entrySet()) {
    switch (entry.getKey()) {
        case "alignment":
            JsonObject alignment = entry.getValue().getAsJsonObject();
            String[] rules = {"right", "left", "below", "above"};
            SparseIntArray mappings = new SparseIntArray(4);
            int targetId = 0;
            for (String rule : rules) {
                if (alignment.get(rule) != null) {
                    targetId = alignment.get(rule).getAsString().hashCode();
                    switch (rule) {
                        case "right":
                            mappings.put(RelativeLayout.RIGHT_OF, targetId);
                            break;
                        case "left":
                            mappings.put(RelativeLayout.LEFT_OF, targetId);
                            break;
                        case "below":
                            mappings.put(RelativeLayout.BELOW, targetId);
                            break;
                        case "above":
                            mappings.put(RelativeLayout.ABOVE, targetId);
                            break;
                    }
                }
            }...
```

Figure 11 Code example of JSON parsing logic (snipped)

Figures 10 and 11 show the implementation of the logic used for almost all the parsing in the application. As can be seen from Figure 11, the action taken when a supported attribute is encounters, is not necessarily a function. In fact, most of the time the attribute value is simply checked, where appropriate and then added to the field, which is later read by the UI element using this setting. The JSON for the part of parsing logic in Figure 11 could look like this:

```
"alignment": {
  "left": "internet_shortcut",
  "right": "youtube_icon",
  "below": "company_logo",
...
```

Figure 12 Example of JSON for alignment rules

The item is therefore set to be positioned in relation to items with the indicated names. In case there are conflicts in positioning order or the items don't exist, Android handles those cases without any errors. Result may ben unexpected, but the application will not crash.

When instantiating items from the configuration, a view is created for each desktop item, and its ID is set to the hash code of the item's name:

```
itemView.setId(itemSetting.name.hashCode())
```

Such logic was chosen to allow for assigning readable names for each item, while still being able to set unique Android IDs (which must be integers) using those names. Other settings include offsets (how far the item is offset in either direction after being aligned per the above rules, padding, size, etc.).

In the drawer, this logic could be omitted, since it is based on a grid, and therefore each item has the same layout parameters, set by the `StaggeredGridLayoutManager` that operates Drawer screen grid. Instead customers are given control over which applications or items are displayed in each drawer screen, the size of the grid (number of columns, which controls the size of individual icons), margins and padding as well as parameters common to each set of views, such as background images, screen titles or item animations. These parameters are parsed using the same logic as desktop items.

To support multiple devices, it was decided to use relative units for all position and dimension values. `LauncherSettings` is responsible for dividing current device's

screen both vertically and horizontally by an increment value provided in the configuration. Other settings compute size based on these increments. For example, if the number of increments is set to 100, and a desktop item width is set to 30, it will occupy 30% of the screen's current width (based on orientation). This allows the application to support many different screens and pixel densities, without restricting customer's control over precision of position and dimension settings (as if they want higher precision, they can simply increase the number of increments, reducing the unit size).

3.2.5 Fault prevention mechanisms

Most of the fault prevention mechanisms are contained inside chokepoint code, as well as in having default values for settings. Each settings class is comprised of public fields, which UI elements read. As such all those fields are given a default value that does not cause the UI element to crash if read. When a JSON configuration is read successfully, those default values are replaced with the ones supplied in that configuration. Nevertheless, an error can still occur if a JSON configuration is read, but the values supplied are not expected by the element reading this configuration field. I tried to design the code of this application in such a way that those cases are handled by chokepoints, however that is not always possible to achieve. To cover most likely crash scenarios, I have identified several key conditions, to which application is most vulnerable, and added chokepoints to catch each, with a fallback mechanism, in case a crash or an invalid state is detected:

1. Content provider is absent or crashes:
    - Runnable that is responsible for obtaining configurations uses a try/catch block an and internal failure count. If an exception is caught, or the obtained data is not in expected format (A table row containing a json string and a timestamp), the failure count is incremented. Failure count is decremented when a successful operation is performed. If the failure count reaches a pre-set threshold, the runnable will stop re-posting itself.
2. Parsing new settings fails
    - Only Application class holds all the settings objects; all other components can access them through the application context reference. This way I could place a chokepoint for when settings fail to parse into a single place – Application's callback methods for the configuration runnable's interface. When a configuration (desktop, drawer or launcher)

fails to parse, it will be set to the built-in generic configuration that is bundled with the launcher package.

3. UI elements get created in an invalid state
    - A common problem in Android, that is not unique to this implementation, is the state of UI elements. Most often crashes can occur due to invalid context or unexpected view hierarchy settings. Unfortunately, there is no way to catch multiple occurrences of these crashes within a chokepoint, as the system API creates views itself. Therefore, I simply did my best to verify the state of UI elements such as fragments before populating them with views. In addition, I used fixed container layouts for each of the programmatically instantiated custom views, to prevent view hierarchy issues that can result from an invalid configuration.

3.3 Other components of the solution

3.3.1 Preface

This section will briefly outline other components in the developed system, which are important to the intended functionality of this launcher application. While not directly a focus of this thesis, those components are still part of the whole solution and were designed to work together with the launcher, affecting its implementation and codebase design.

3.3.2 Obtaining and delivering JSON configuration

To modify UI of the launcher application in the way that customers desire, JSON configuration had to be obtained from the cloud-based management platform, which contained integration with customer's IT environment, as well as a management interface. Having the launcher interact with this cloud platform made little sense, as it would vastly increase the codebase, with code that is not relevant to the purpose of a launcher – to provide users with UI for interacting with other applications. Therefore, a separate application is handling data procurement, and may also require user input such as credentials to obtain launcher configuration specific to that user, the tablet device itself, its location or any other malleable input.

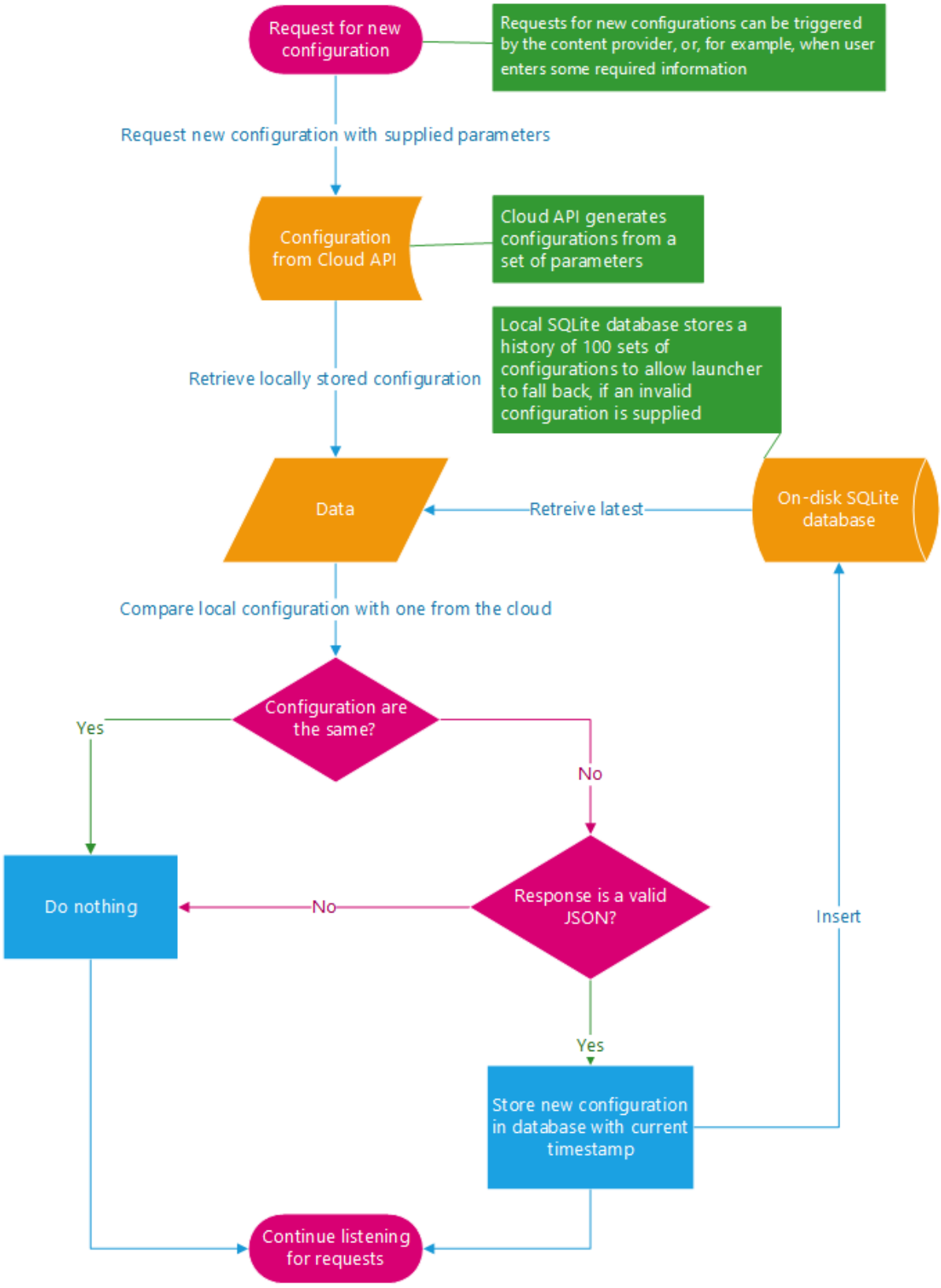


Figure 13 Obtaining JSON configuration

Figure 13 shows, that there are two data sources with which Network service interacts: cloud API and local SQLite database. Network service connects and authenticates with the cloud API. Once that is complete, the cloud knows which customer this device belongs to, and can serve a generic customer-specific configuration. That configuration is stored in the SQLite database table, and will be served as a fallback, when no more specific valid configuration could be obtained. Afterwards the service inquires whether the backend API can provide a more tailored configuration for a set of parameters configured individually for each customer (e.g. the time of day, location of device, current user, etc.). If such configuration is available, the network service obtains it and stores in the same SQLite table. As Figure 13 details, a timestamp is used when storing this data. This timestamp is taken from the system current time in UNIX format, and is later used for fallback logic to be able to revert to or swap settings based on the time they were acquired. The timestamp is also used for debugging purposes. When the launcher queries settings from the content provider that is implemented in the same application as the network service, it will be given the most appropriate valid set of configurations that are currently in the SQLite table. If no configurations are available, the content provider will inform the service that configuration is required, and will wait for a few seconds to see if the database gets any updates. If there have been no updates, it will return nothing, otherwise it will re-run the query and return the most recent available set of configurations.

You may notice from Figure 13, that new configuration is simply stored in the local database, and no messages are sent to inform any listeners that new configuration is available. While it may change in the future, currently this is by design – Network service updates configurations when appropriate, but it is up to the launcher to request new configurations. After updating configurations, the launcher will receive the newest set upon its next request. At the moment, this is accomplished using the launcher control service, the specifics of which were outlined in the previous section.

3.3.3 Network services

Currently the MDM solution has one network service. This service is responsible for all networking operations, and is roughly equivalent in functionality to Android's Google Play Service. The network service runs in a separate process and handles interactions with backend API, downloading APKs and configuration JSONs and storing downloaded data on the device, when appropriate. The decision to separate this service to a separate process is based on two major factors:

1. Network and file operations can be expensive and require a lot of memory. In Android, background services that take too much memory are killed in case there is a memory shortage (which could happen if the user, for example, launches a demanding game). By separating this service into a separate process, the chances of it being killed by the system become very small.
2. Direct interaction with the service can increase component inter-dependency, which was undesirable for this project. By having the service in a separate process each component had to be designed to be able to operate without getting data from the network service, which lessened the negative impact of network interruptions or other network and file handling-related issues.

3.3.4 MDM services

MDM is performed by a separate administrator agent application. This application utilizes Samsung KNOX API to be able to control most aspects of the Android operating system, as well as certain device hardware settings. Like the launcher, MDM component obtains configuration settings from the network services, parses and applies them to devices it runs on. MDM service is critical to autonomous operation of other components, because Android is built around the idea of user interaction and control, which makes managing devices remotely without user interaction very difficult. The MDM service makes sure that all applications and operations required for autonomous operation of the solution have required permissions and that there are no settings in the system, which block operation of these components. In addition to that it, of course, allows customers to apply restriction policies on all the managed devices, as well as remotely configure them as they see fit. Furthermore, it also provides a user interface for “Admin mode” devices to be able to configure administrative settings on the device itself. Customer organization’s IT staff must authenticate themselves on the device, before they can access this administrator interface.

3.3.5 Samsung KNOX

Samsung KNOX is proprietary technology, developed by Samsung corporation, which we utilize to extend our level of control within the device. It allows to programmatically change almost any system setting on an Android device, and is available on all modern Samsung devices. Being able to programmatically configure the system is critical to the operation of the MDM services, as by default Android will require user interaction to edit device settings. KNOX is based on Android’s native “Device Administrator” feature, which gives applications with administrator privileges control over certain device func-

tions, such as controlling the camera, wiping device data or requiring password to unlock the device, as well as remotely locking it. This feature is, however, very limited, and most of the MDM functionality required for this solution is not provided by it. KNOX extends this feature substantially, allowing the device to be fully managed after a few simple one-time setup steps. In addition, KNOX includes an enrolment feature, which to enable scalable deployment of the MDM admin agent application on as many devices as customers decide to purchase. Furthermore, an enrolled device stays enrolled even after a factory reset, which is also very important for managing a large quantity of devices. This feature was critical to our requirements as some of our potential contracts using this solution can reach thousands of devices, and the cost of manually installing our solution on each of them would be too high.

3.3.6 Inter-process communication

As there are multiple components in this solution, that run in different processes, a robust inter-process communication (IPC) method was required. Android provides several IPC methods, each tailored to specific use cases. In the context of this solution, a two-way communication method was required for securely issuing requests to different components. For that I decided to use Android's `Messenger` objects. Messengers are wrappers around Android's Binders, which are used in Android for lightweight remote procedure calls [21] [22]. Designing our own AIDL (Android Interface Definition Language, used in Android for custom IPC interfaces) interface seemed unnecessary, as Messenger contained all functionality required for our components to communicate. Main limitation of Messenger is that it can only be used to communicate with Services (for example activities cannot use it to communicate to activities in another process), however in our solution, only communication with Services was needed. Therefore, I created a set of classes that wrapped request-response logic using our company's own protocol, for safe communications between solution's components. These classes provide interfaces to services and activities that use them, simplifying communications by abstracting synchronization logic and making code more readable.

3.3.7 Backend API (brief mention)

Lastly I wish to briefly mention the backend API. While the focus of the project is the solution that runs on Android, almost all customer-specific integration as well as customization of the launcher and device settings happens on this cloud API. I cannot write any details about the structure and logic of the cloud services for security reasons, however it is important to note that this component is the one responsible for the correct construction of configuration data, and providing them in an expected and valid

format. The system has been designed in a way that lets it obtain data from any network location (including local network), if it has a server that can provide data in the correct format, however it does not include any logic for “fixing” the data, if it is partially malformed or invalid, and will simply discard all the data if it encounters any issues.

4 Results

4.1 Deployment strategy

Using Samsung KNOX Enrolment has proven to be very difficult due to lack of information available on the usage of the system, however a seamless deployment was achieved in the end. Once the MDM system’s administrator agent is installed and started by KNOX Enrolment, it downloads other components, including the launcher, and sets their relevant roles and permissions. When the launcher is set as default for a device, next press of the home button starts it. At that point the MDM agent would have already acquired the relevant launcher settings, so when the launcher requests settings, it obtains appropriate settings for the current user and applies them. The welcome screen is then removed and the user can begin interacting with the launcher. Since MDM agent is responsible for acquiring launcher settings, any customer-specific authentication methods would need to be handled by the agent, before launcher is installed. However, if something goes wrong during authentication, or if the device loses internet connection while the launcher settings are obtained, the launcher can still run using its bundled settings.

4.2 Performance and usability

As a launcher is primarily a UI application, performance was heavily monitored during development, and has proven to be surprisingly good, even without many specific optimizations (other than the ones implied by design, such as using `ViewPager` and holder patterns). There are no on-screen frames per second counter in Android, however android debug bridge displays skipped frames during debug runs, if there are noticeable slowdowns.

Test device	Highest reported skipped frames
Samsung Galaxy Tab E	9, on a configuration with 10 high-resolution custom images on the desktop screen
Samsung Galaxy Tab A	0
Samsung Galaxy Note 10.1	0
Sony Xperia Z3	0

Table 1 Performance tests

Table 1 shows the number of skipped frames reported by Android system over ADB bridge. As can be observed only Galaxy Tab E reported any skipped frames, to which I can also add that this happened only once, during the first swipe to the drawer screen. Presumably the system later cached images used in the views, increasing performance.

Since running applications in debug mode is always slower than running release versions, it is safe to assume that launcher would not lag on any of the tested devices. During performance testing, I did every possible UI interaction with 3 different sets of configurations. It should be noted, however, that of all tested devices only Galaxy Tab E was the only low-end one, while the rest of test devices are considered medium to high end. Note, also, that this test was for standalone launcher application, and not the entire solution, because Samsing KNOX only works on Samsung devices, which have never been flashed with any non-Samsung OS. Since Sony Xperia Z3 is a Sony device, and we have flashed the Galaxy Tab E with our custom image, no KNOX-enabled applications could run on those devices.

During testing, from an end user perspective there were no crashes when supplied configuration and system settings were both valid and compatible, however UI reloads in response to settings change have not been very consistent. The launcher is programmed to apply new visuals upon configuration change, when the view affected by changes is not currently on screen. This was done to avoid jarring experiences such as UI changing while end-users are interacting with it, however during testing new configurations sometimes didn’t apply even when view was reloaded. Most likely cause of this behaviour is Android’s fragment handling logic, where it preserves the fragment instances between configuration changes, and as such the chokepoint that starts recreation of the view may not be triggered. Furthermore, it was revealed that even after numerous improvements and safety checks, certain combinations of JSON settings can

cause a crash even after bypassing all chokepoints. This most likely indicates that some of the settings classes do not have strict enough validity checks.

Overall the launcher has proven to work rather smoothly and well when the configuration is the same, however crashes and inconsistent behaviour still occur when the configuration is being changed, as well as for certain combinations of settings. Another possible usability issue caused by certain configurations is the size of application's icons. Launchers typically display applications in fixed grids and each application's icon size is therefore fixed. In this case, however, customers can modify sizes of icons in both the drawer and the home screen, which can result in pixelated images if the application does not supply an icon of big enough size, of if the custom image from the network is not optimized for the target device's DPI (dots per inch, often used to measure screen pixel density).

4.3 Data handling

Three different types of data have been required in this solution so far:

- Media assets
- APK files
- JSON configurations

Media assets are mostly images, required by the launcher application. I offloaded their network handling to Picasso library, which has proven to work very reliably. Even when multiple relatively large (500x500 pixels) images were used for icons on the home screens, Picasso has loaded and scaled them very quickly. On a network with a download speed of 24Mbit/s, using 5 images of size 300KB-1.2MB, the delay before all images appeared was approximately 200 milliseconds on the first run. After the first run, the delay was completely unnoticeable, because Picasso library retrieved those items from cache.

APK files are required for the MDM admin agent to install other components of the system, such as the launcher, and in some cases – install third-party applications from outside Google Play Store. While downloading the APK itself has proven to be rather simple, an issue was encountered with Android versions 6 (Marshmallow) and higher, where default system behaviour is to disallow installation of non-Google Play applications on the device without explicit end-user approval. Due to the requirement that regular users must be prohibited to modify such settings, the MDM agent had to be used

to set correct system settings and allow installation of downloaded APKs. This was achieved through the use Samsung KNOX API.

JSON configuration is required for both the MDM agent, as well as the launcher. MDM agent receives JSON configuration that is used to manage the device, while the launcher needed to be provided with UI settings that would correspond to the current environment, state of the device and customer's own configuration. Since most of the integration with customer's own IT environment is done on the backend API, the backend was responsible for delivering the correct data. The MDM agent, which contained network service used for procurement of these JSON configurations, then was responsible for supplying the backend with relevant data. While some basic testing was done for these interactions, and JSON configurations were successfully fetched and applied, as of the time of righting this text, there have not been production implementation of this system, so it is difficult to say how it would perform when the scale increases.

4.4 Faults

There have been multiple failures during development of this system, most of which could have been avoided by having access to more information about the APIs that are being used. Since the focus of this thesis is the launcher, I will focus on the problems with the launcher application.

One of the most common and difficult to solve issues, has proved to be dealing with erroneous JSON configurations. Even with multiple checks for validity of the JSON string, default parameters when JSON does not have them, and type checking during parsing, there were still ways to create a configuration which would pass all checks and then cause a crash when being applied to a UI component. One possible solution to this problem would be, of course, to make sure that JSON strings are being correctly generated, with the valid attributes and values on the backend service that provides them, however that is still not a complete solution, as customers may wish to have their own JSON service providing the configurations.

A severe problem was encountered when certain system settings were changed. Modifying those system-wide preferences caused the launcher to enter a crash loop. This meant that the launcher crashed before it could allow user any interaction, which could in some cases, render the device completely unusable due to inability to access the

MDM agent application and disable the launcher, or revert the problematic system setting (as system preferences could be disabled with a security policy that only an MDM agent could override). While the cause for this problem has already been identified, the solution has not been tested at the moment of writing this thesis, since this problem only occurs outside of development environment, and as such testing takes a relatively long time.

A usability issue was encountered during testing related to the navigation between different launcher views. At first use, users that were accustomed to standard launcher behaviour were confused at the absence of the drawer button. That confusion, however, passed rather quickly, once they realized they can swipe between views. A more prominent issue was absence of an indicator about which view, relative to others, the user is currently viewing. A solution to that is already planned, using `RootLayout`'s overlay view, however it has not been implemented at the time of writing.

A significant codebase design flaw has been identified in the structure of the item view UI component. This UI component serves as a basis for all item views displayed on the desktop and the drawer. Being a basis, it's structure should have been a lot more generic. Currently the view is a compound view – an `ImageButton` and `TextView` contained within a `LinearLayout`. While this structure works well for displaying icons, it is not ideal for extensions, as the methods it provides are all focused on modifying properties of it's text view and image button. This has proven problematic when adjusting descending view objects to operate in different scenarios (such as grid view, list view, being non-interactive or displaying them with different sets of layout parameters).

5 Discussion

5.1 Obstacles in development

5.1.1 Launcher application

One difficulty that has been impeding development since it began is small amount of information on building this type of application. Launchers are not a typical application, which is why there are relatively few of them compared to other application types. Thus, development details specific to launcher application were often very difficult to debug.

For example, a launcher must have a way to display all installed applications in the drawer. Most materials I went through illustrated a method of obtaining a list of installed application packages, and then creating a shortcut for each. However, this approach does not work for a launcher, because some application packages may have multiple launchable activities and a launcher must create a shortcut for each of them. Therefore, a solution was to use `ResolveInfo` objects for this problem, which was not an easy solution to obtain. That, however, also introduced a problem, as some applications indicate several launchable components under the same name (presumably for different system versions). In this case the results had to be filtered by name, and an icon created for each launchable component with a unique name.

That was just one example of an issue specific to developing launcher applications. There have been numerous other problems related to this application unique configuration method and the MDM environment it had to run in. Majority of those issues are not common enough to be researched, and had to be solved in-house, which has caused significant slowdowns in development times.

5.1.2 MDM system and deployment

As mentioned in section 2, Samsung KNOX is still relatively new, and there have been several issues using the API due to missing or vague documentation and lack of information available for problem resolutions. One of the main problems when developing the MDM component was handling different permissions and license activations. To call KNOX API, an application must be declared as a “Device Administrator” in Android system settings and activate all required KNOX licenses. Only then it can proceed to install and set the launcher. Synchronizing all these settings and licenses to achieve the correct state has proven to be a challenge, although the issues were eventually overcome.

Linked to the above, Samsung KNOX Enrolment was a poorly documented technology and has led to a lot of confusion during initial deployment and automation tests. The goal of the project is to achieve a state of autonomous deployment, where the only thing required from the customer is to connect the device to a Wi-Fi network, before the system gets installed and starts working. That has proven to be quite a challenge, due to missing information on how Samsung KNOX Enrolment affects the code of MDM agents it installs.

5.2 Lessons learned

One of the main lessons learned from developing a management platform is the importance of providing fail-safes at almost every step. Due to a highly secure environment, there are many scenarios where there is no way to recover from a crash loop or another program failure, because the program in question is the only component that has any control over the system. As such it is a good idea to always have a very simple error-prone separate monitoring mechanism, which checks that everything else works well, and can provide a way to do an emergency reset if an unrecoverable issue is encountered.

5.3 Limitations and opportunities

As mentioned in section 2.4.4, due to the way this launcher was designed, it can be deployed under any existing MDM, which allows altering application policies, and even be deployed standalone, directly for end users. However, significant parts of the whole solution are its autonomous deployment, security and configuration, which would be lost, if the launcher is used standalone or with another MDM solution. To preserve those features, the solution must be run on a Samsung Android device, due to its extensive use of Samsung KNOX API.

Another limitation of this project is that it can only be used on Android devices and is highly dependent on internet connection. Without a connection to our backend API, many of the management and monitoring functionalities, and all autonomous functionality is lost.

On the other hand, one of this solution’s big strengths comes for the modularity of its codebase design. Due to all components being designed to work independently, and all interaction being done using standardized Android methodology, it is possible to individually develop, update and swap each component, without disrupting overall system integrity. This makes the solution easier to maintain and expand, as well provides relatively strong safeguards against changes in future versions of Android affecting this MDM platform’s and launcher’s integrity.

6 Conclusions

6.1 Restatement of project goal

The goal for this project was to provide organizations with a solution for managing a scalable number of mobile devices running Android mobile operating system and being

able to centrally configure not only their settings, but also appearance of the interface. This was to be achieved using a set of components including an MDM administrator agent for managing device settings, network services for interacting with cloud API and a custom launcher application that could be configured using supplied JSON settings.

6.2 Summary of results

When the system was deployed through Samsung KNOX Enrolment technology, it could download and install the custom launcher application, setting it as default and retrieving custom settings for it. The launcher application could successfully query the settings and apply them, customizing its visuals as defined in the configuration.

The system has shown to work well, when settings have been properly formed and delivered, however it has shown to be vulnerable to malformed or invalid settings, as well certain system configurations. Supplying certain combinations of settings has shown to crash the launcher, bypassing all the safety checks, and changing certain system preferences could cause the launcher to enter a crash loop, where it would crash shortly after starting, prohibiting any action from being taken. In addition, certain usability problems have been found with the user interface, which need to be resolved.

6.3 Implications of results

Considering the scale on which this system can be deployed, I am reluctant to call it ready for production. Initial results are very promising and they show that the underlying system design and logic work and can achieve the desired goal, however there are still several critical areas that need to be improved to increase the safety of usage of this system. More testing is clearly required for the configuration processing logic, as well as applying the configuration to UI. Furthermore, the launcher needs to be better equipped to deal with unexpected system preference changes.

If this solution is deployed on hundreds or thousands of devices, and a configuration changes in either device settings, or launcher settings causes a crash loop or another serious issue, the consequences of such failure could be catastrophic. It cannot be shipped to production until such a failure can be completely prevented or at the very least, functionality included that allows all affected devices to be remotely reset back to working settings through a centralized management console.

6.4 Future work

As a highest priority now, the system must be able to provide a way to remotely reset devices, even if none of the components work. Such functionality could be achieved by

having a separate very simple MDM agent application that is installed alongside the main one, which consists of a service that monitors status of all other components of the solution and is also connected to the cloud API and capable of receiving one command. If the device is rendered unusable, the IT staff can issue a command to this “heartbeat” service to perform a factory reset on the device. Since the service would have the same permissions as any other MDM agent, it would be able to perform such an operation. In addition, Samsung KNOX allows for different MDM agent applications to override each-other’s settings, if they run within the same context and have the same permissions, so if reset is disabled on the affected device, the heartbeat service would be able to re-enable it and perform the reset.

In addition, UI improvements also need to be done on the launcher itself. Most of the UI issues have planned improvements that are in development right now, however could not be completed at the time of writing. Adding a visual indicator to inform users of how many screens they can swipe between will be one addition. Also, response of the launcher to configuration changes could be improved to be more reactive without disrupting user experience. Furthermore, the launcher needs an ability to host application widgets, which all other launchers have, as well as react to application installation and uninstallation. Both these features are currently in development, but have not been finished yet. Several smaller improvements and bug fixes also need to be made to the current implementation to improve class structure and flexibility of the configuration.

As a bottom line, I would conclude that the system and the launcher are developed enough to not be considered prototypes or proofs of concept anymore, however they are not yet ready to be beta tested. Before piloting can be conducted, a significant amount of polish and bug fixing needs to be done and an emergency failsafe functionality needs be added.

7 References

[1] N. D. Bui, A. G. Kravets, L. T. T. Nguyen and T. A. Nguyen, “Tracking events in mobile device management system,” in *Information, Intelligence, Systems and Applications (IISA)*, 2015.

[2] IBM, “IBM - Maas460 - Finland,” 2016. [Online]. Available: https://www.ibm.com/marketplace/cloud/mobile-device-management/details/fi/en-fi. [Accessed 01 November 2016].

[3] VMware, “Enterprise Mobility Management - AirWatch,” 2016. [Online]. Available: https://www.vmware.com/products/enterprise-mobility-management.html. [Accessed 01 November 2016].

[4] P. Ferrill, “The Best Mobile Device Management (MDM) Solutions of 2016,” 2016. [Online]. Available: http://uk.pcmag.com/cloud-services/76018/guide/the-best-mobile-device-management-mdm-solutions-of-2016. [Accessed 01 November 2016].

[5] Google Inc., “App Widgets | Android Developers,” 2016. [Online]. Available: https://developer.android.com/guide/topics/appwidgets/index.html. [Accessed 05 November 2016].

[6] V. Beal, “What is Mobile Device Management? - Webopedia Definition,” 2016. [Online]. Available: http://www.webopedia.com/TERM/M/mobile_device_management.html. [Accessed 1 November 2016].

[7] L. Liebmann, “Boot ROMs ease desktop management through centralized booting,” *Computer Technology Review,* p. 24, 1996.

[8] Apple Inc., “Apple Remote Desktop,” [Online]. Available: http://www.apple.com/remotedesktop/. [Accessed 3 November 2016].

[9] L. Poggemeyer, “Remote Desktop clients,” 2016. [Online]. Available: https://technet.microsoft.com/en-us/windows-server-docs/compute/remote-desktop-services/clients/remote-desktop-clients. [Accessed 03 November 2016].

[10] Zoho Corporation Pvt. Ltd., “Remote Windows Desktop Management and Administration Software - Documents,” 2016. [Online]. Available: https://www.manageengine.com/products/desktop-central/help.html. [Accessed 03 November 2016].

[11] IDC, “Smartphone OS Market Share, 2016 Q2,” 2016. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. [Accessed 03 November 2016].

[12] T. Koerber, “Let’s Talk About Android – Observations on Competition in the Field of Mobile Operating Systems,” SSRN, New York, 2014.

[13] Google Inc., “Android is for everyone,” Google Inc., 2016. [Online]. Available: https://www.android.com/everyone/. [Accessed 07 November 2016].

[14] Google Inc., “Platform Architecture,” 2016. [Online]. Available: https://developer.android.com/guide/platform/index.html#native-libs. [Accessed 03 November 2016].

[15] Google Inc., “ART and Dalvik,” 2016. [Online]. Available: http://source.android.com/devices/tech/dalvik/index.html. [Accessed 03 November 2016].

[16] ECMA International, *The JSON Data Interchange Format ECMA-404 1st Edition,* Geneva: ECMA International, 2013.

[17] SAMSUNG, “Knox Technology | Samsung Knox,” 2016. [Online]. Available: https://www.samsungknox.com/en/knox-technology. [Accessed 03 November 2016].

[18] GB Media, “Live from the Mobile World Congress: Samsung announces technology to assist in the security of BYOD,” 26 Feburary 2013. [Online]. Available: http://www.gbmediastudios.com/live-from-the-mobile-world-congress-samsung-announces-technology-to-assist-in-the-security-of-byod. [Accessed 03 November 2016].

[19] Google Inc., “Android Developers,” 2016. [Online]. Available: https://developer.android.com/index.html. [Accessed 06 November 2016].

[20] Square, Inc., “Picasso,” 2016. [Online]. Available: http://square.github.io/picasso/. [Accessed 06 November 2016].

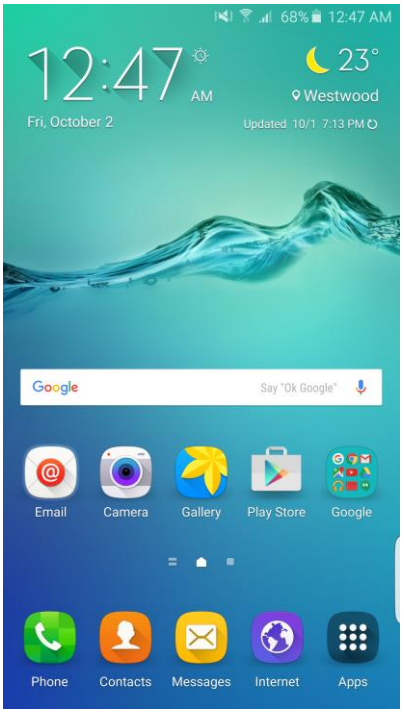
[21] Google Inc, “IBinder | Android Developers,” 2016. [Online]. Available: https://developer.android.com/reference/android/os/IBinder.html. [Accessed 07 November 2016].

[22] Google Inc., “Messenger | Android Developers,” 2016. [Online]. Available: https://developer.android.com/reference/android/os/Messenger.html. [Accessed 07 November 2016].

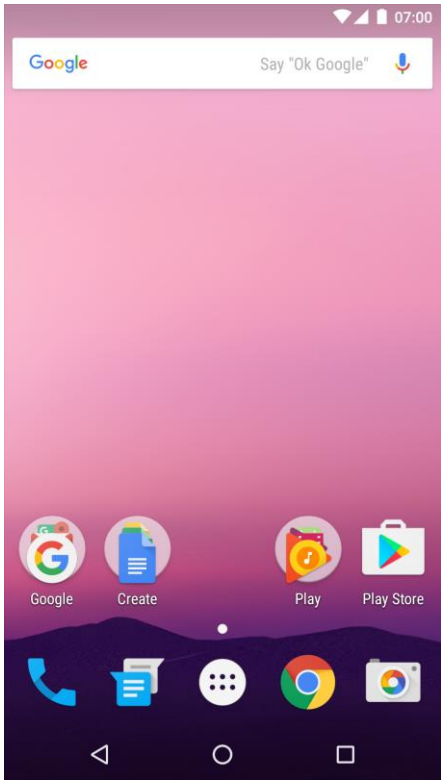
Appendix 1

Home screens of different Launcher applications.

16 ZenUI launcher homescreen (Source: http://droidxhacks.blog-spot.fi/2015/09/ported-asus-zenfone-2-launcher-themes.html)



14 Samsung TouchWiz launcher homescreen (Source: http://im-ages.anandtech.com/doci/9558/Screenshot_2015-10-02-00-47-30.png)



15 Android Nougat default launcher homescreen (Source: https://upload.wiki-media.org/wikipedia/en/7/78/An-droid_7.0_Home_Screen.png)

Appendix 2

Different homescreen configurations of this project's launcher (Taken on Samsung Galaxy Tab A)

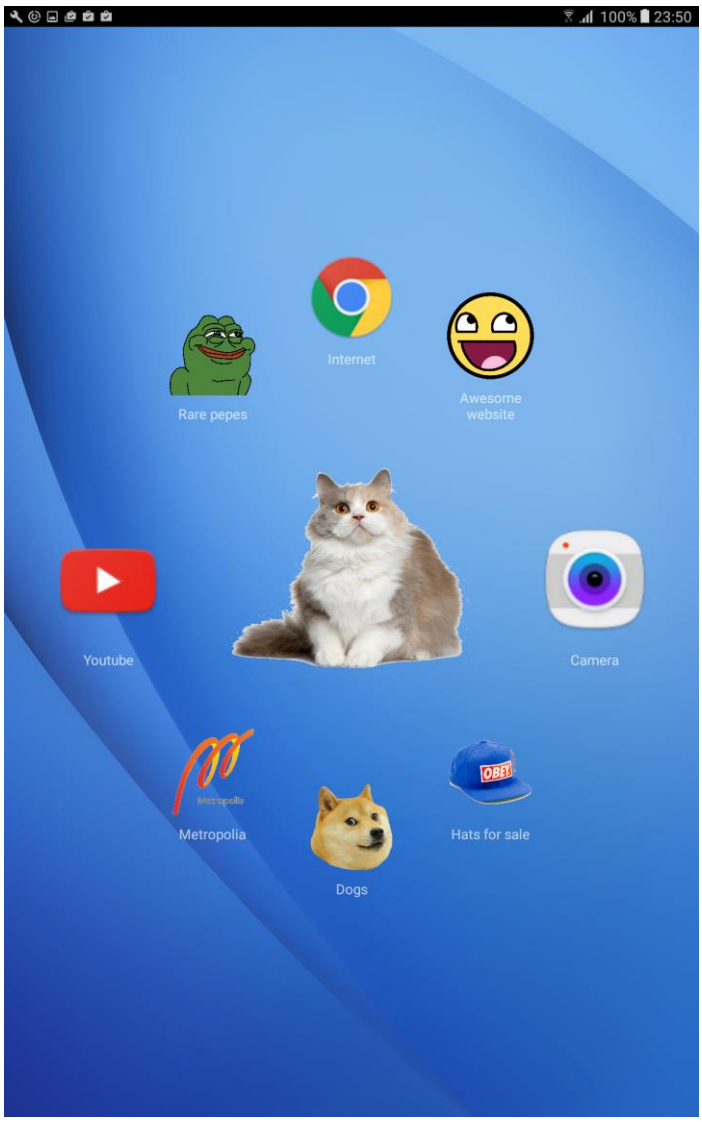


Figure 17 Modified icons for web shortcuts and applications. Custom non-interactive graphics can also be used as an anchor for other icons. All images are obtained from the cloud and cached.

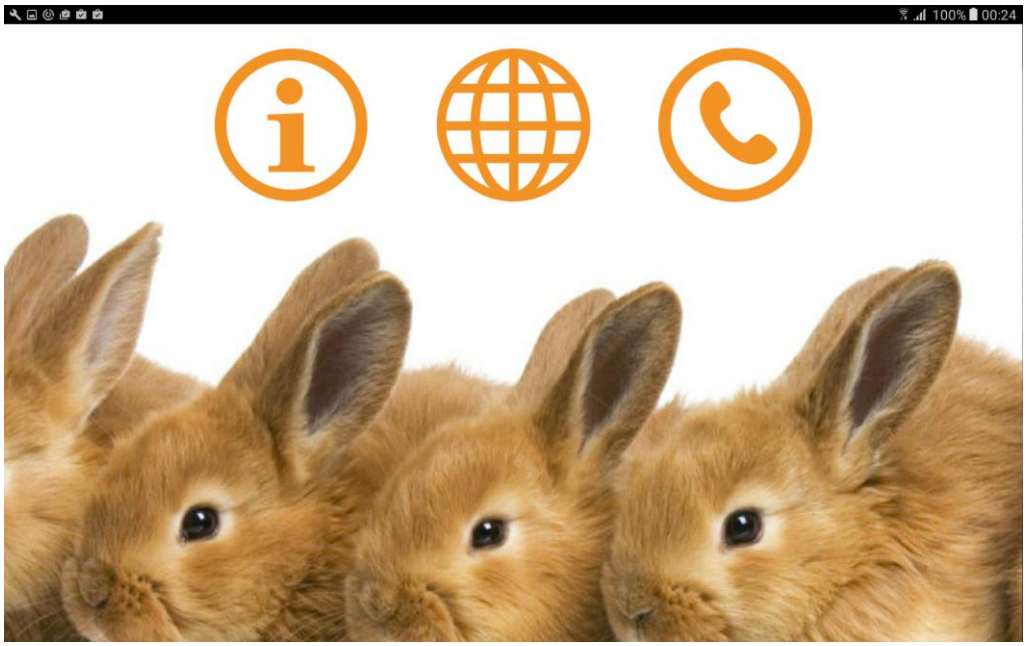


Figure 18 All icons can be positioned relative to each other, and relative to sides of the screen, or centred. All units are relative - no hardcoded pixels. Separate background images are supported on each home screen and each orientation.

Figure 19 Default desktop screen configuration bundled with the launcher package (used as fallback)