

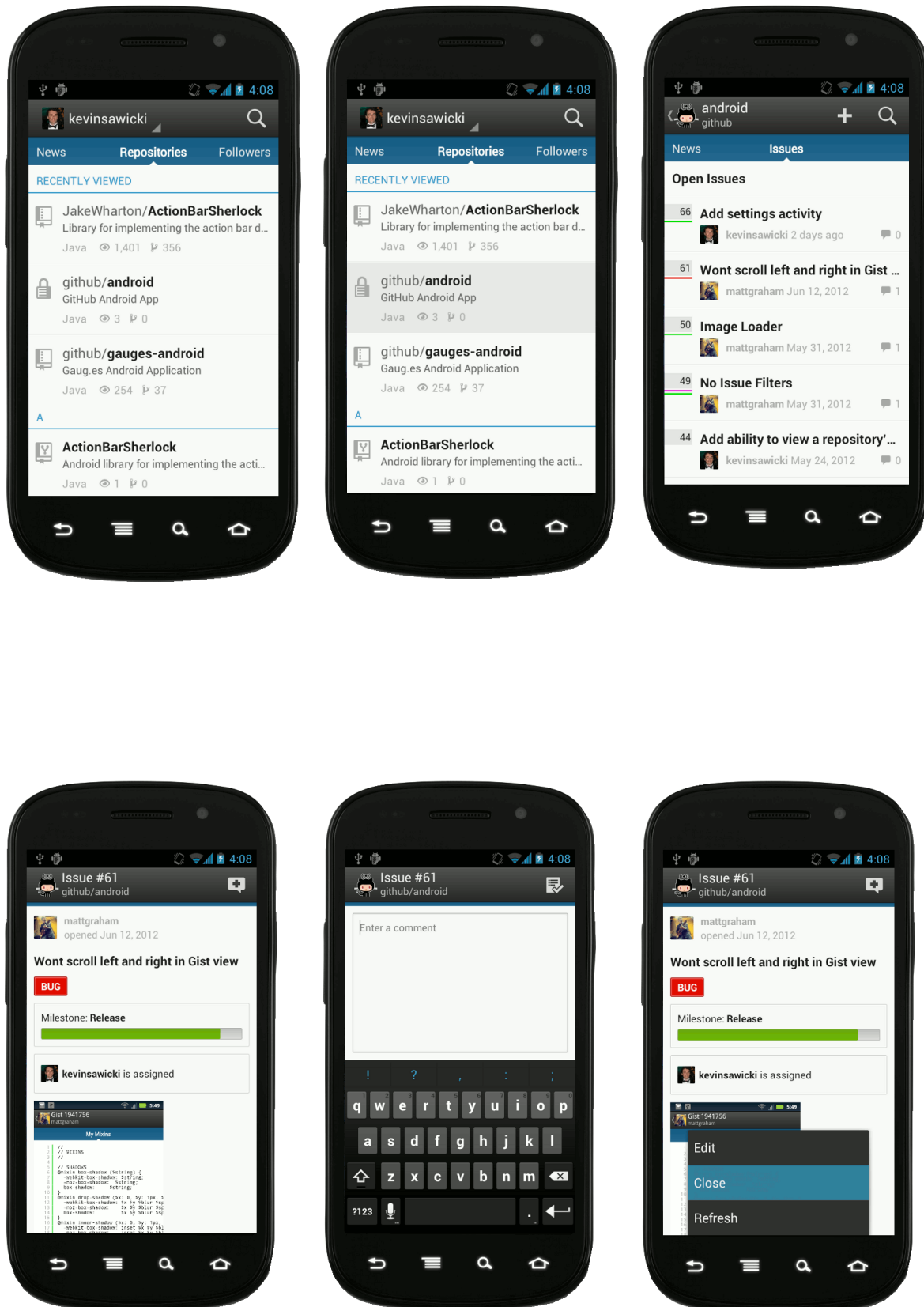
Contents

Identification of the problem	3
Users	4
Proposition	4
Background	4
Git	4
Storage	4
Repositories	5
Commits	5
Heads	5
Branches	5
Merging	6
GitHub	6
Issues	6
Pull requests	8
Projects	9
Integrations	10
Markdown	11
Heading level 1	11
Sub-heading	11
Another deeper heading	11
Lists	12
Images	13
Blockquotes	13
Strikethroughs	13
References	13
Emoji	14
Code	14
HTML	14
Android Application Structure	15
Activities	15
Intents	16

Intent handling	17
Layouts	17
Layout inflation	18
Dimensions	20
View binding	20
Fragments	22
RecyclerViews	24
Basic storage	25
Build system	25
Debugging	27
Proposed design	29
User information Activity	29
Repository information Activity	32
Limitations	45
Processing power	45
Screen size	46
API	48
Rate limiting	48
Lack of endpoints	48
Lack of push API	48

Identification of the problem

In July 2012, GitHub released an official app for Android.



GitHub later dropped support for their app in favour of a mobile website with severely limited features.

GitHub's mobile website allows viewing most important information about a repository, as well as creating comments on issues and commits, however it has no editing functionality for content which has already been created.

Some areas, such as the projects section, are entirely missing from the mobile website.

The source of the original app has been used to continue support for a number of similar apps, however they have the same limitations as the original app as they are built on a 5 year old codebase.

Further, the maintainers of some of the most used GitHub apps have recently dropped their support.

Users

According to GitHub's 2016 statistics there are 5.8 million active GitHub users with over 19.4 million repositories.

As GitHub is designed for groups of developers to collaborate it is reasonable to assume that GitHub's users would make use of a method for keeping track of their projects without having access to a computer.

Proposition

I will develop a GitHub client for Android to implement the features of the GitHub website.

Background

Git

The Git version control system was developed by Linus Torvalds in 2005 and is designed for nonlinear distributed development, whereby multiple developers work on multiple different tasks concurrently.

Git is used to store snapshots of a project, and store them as unique versions. Git makes it easy both to rollback changes to a previous state, and to merge the project with changes made to it elsewhere.

Storage

Git acts as a content-addressable file system.

When content is inserted into the system, the return value is a key which can later be used to retrieve the content.

The key returned is a 40 character (160 bit) SHA-1 checksum of the content and its header.

The probability of a collision occurring across n unique objects is $0.5 * n^2 / 2^{160}$. In order to achieve a 1% probability of collision 1.7×10^{23} objects are required, which is effectively impossible.

In order to store a file system structure, Git uses tree objects.

Each node in the tree is constructed of four elements:

- The unix file permissions
- A pointer to either a blob or another tree
- The hash of the object pointed to
- The filename

Repositories

Each project is stored in a repository.

A repository contains a set of commit objects, and a set of references to commit objects, called heads.

Commits

A commit object contains:

- A set of files, describing the project state at the time of the commit
- References to parent commit objects
- An SHA1 checksum which uniquely identifies the commit object

Heads

A head is a reference to a commit object.

Each head has a name, with the default name being 'master'.

A repository can contain any number of heads, but at any one time there is only a single current head.

Branches

Every branch has a head and every head has a branch. The only difference between the two is that while a branch refers to a head and the entire history of

ancestor commits preceding it, a head is used to refer to a single commit object, the most recent commit in a branch.

When the user switches branches their current head pointer is changed to point to the branches head, and all of their working files are rewritten to match those at that head commit.

Merging

When work has been completed on one branch, the changes need to be brought into another branch in order to allow others to use the changes.

This is done by the merge command.

When asked to merge a branch 'changed' into a branch 'other' Git:

- Finds the common ancestor of 'changed' and 'other'
- If the ancestor commit is equal to the head of 'other'
 - Updates 'other' by adding the commits which have been made to 'changed' since the common ancestor
 - Updates the head pointer of 'other' to point to the same commit as 'changed'
- Otherwise a full merge must be performed. This involves:
 - Determining the changes made between the common ancestor and the head of 'changed'
 - Attempt to merge these changes into 'current'
 - If the merge was successful:
 - Create a new commit with two parents 'changed' and 'other'
 - Set 'other' and the working head to point to this commit
 - Otherwise
 - Insert conflict markers into the file
 - Inform the user of the problem without creating a commit

GitHub

GitHub is hosting service for Git repositories, and the largest host of source code in the world.

GitHub adds numerous features on top of the Git system.

Issues

Issues are a tracker system for 'tasks, enhancements, and bugs'.

Each repository has its own Issues section. Within the issues section of a

repository each issue has its own content. The screenshot below shows the issues section for a repository.

The screenshot shows the GitHub Issues page for a repository. The top navigation bar includes links for Code, Issues (4), Pull requests (0), Projects (4), Wiki, Pulse, Graphs, and Settings. Below the navigation bar, there is a search bar with the query 'is:issue is:closed' and a 'New issue' button. A 'Clear current search query, filters, and sorts' button is also present. The main content area displays a list of 8 issues, all marked as closed. The issues are: 1. 'Only trigger FAB for specific screen region.' (enhancement), 2. 'Fix floating FAB being left transparent.' (bug), 3. 'Fix formatting of body text in skimmer.' (bug, enhancement), 4. 'Single click on floating FAB breaks after long press.' (bug), 5. 'Implement long press scrolling for the floating FAB.' (enhancement), 6. 'Change fabUp behaviour to scroll up if fab is going up.' (enhancement), 7. 'Make floating FAB a setting' (enhancement), 8. 'Add acceleration to floating FAB' (enhancement). Each issue has a title, a label, a number, and a status (closed).

The page is divided into two sections, open and closed.

An issue which is marked as open is one which has not yet been resolved and is either being investigated or to be investigated later.

When an issue is created, it must have a title explaining its purpose.

It is then assigned an automatically incremented number, and linked to the user who created the issue.

While the purpose of an issue may be clear from its title, larger projects are easier to manage when issues are separated into different categories.

This is achieved through labels.

A label is a string of text and a background colour which may be applied to an issue, allowing it to be filtered.

Further, each issue may have up to 10 assignees.

An assignee is a user who has been designated to investigate an issue.

Filtering by assignees allows work to be more easily distributed, as well as identifying issues which are not yet under investigation.

In order to facilitate the investigation of an issue, each issue has a comments section.

The comments section consists of a mixed feed of comments and events as shown below

The screenshot displays a GitHub issue's comment history. It begins with a timeline of events: a user 'tpb1908' adds labels (bug, duplicate, enhancement, help wanted, invalid, question, wontfix) on 30 Dec 2016; the user self-assigns the issue; the user removes labels (duplicate, enhancement, help wanted, question); a user 'tpb1908-test' is assigned by 'tpb1908'; and finally, the issue is closed on 11 Jan. Below this, there are three comment blocks, each with a header bar indicating the user, date, and role (Owner), and a text area for the comment. The first comment, dated 11 Jan, contains the text 'Test comment'. The second comment, dated 27 Jan, contains the text 'Opened the issue from the app'. The third comment, also dated 27 Jan, contains the text 'Test reopening'. Between the comment blocks, there are additional event entries: the issue is reopened on 11 Jan and then closed on 27 Jan.

tpb1908 added **bug** **duplicate** **enhancement** **help wanted** **invalid** **question** **wontfix** labels on 30 Dec 2016

tpb1908 self-assigned this on 30 Dec 2016

tpb1908 removed **duplicate** **enhancement** **help wanted** **question** labels on 30 Dec 2016

tpb1908-test was assigned by tpb1908 on 30 Dec 2016

tpb1908 closed this on 11 Jan

tpb1908 commented on 11 Jan Owner + 🗨️ ✎️ ✕

Test comment

tpb1908 reopened this on 11 Jan

tpb1908 closed this on 27 Jan

tpb1908 commented on 27 Jan Owner + 🗨️ ✎️ ✕

Opened the issue from the app

tpb1908 reopened this on 27 Jan

tpb1908 closed this on 27 Jan

tpb1908 commented on 27 Jan Owner + 🗨️ ✎️ ✕

Test reopening

Pull requests

A pull request is effectively a subclass of issue.

A pull request is designed to notify users about changes which you have made to a repository.

As a pull request follows the issue model it has the same comments section as an issue.

The primary difference is that a pull request also references any number of commits which can be merged into the repository.

Projects


'Projects' were introduced in September 2016.

The aim of projects is to integrate the planning of a project and its features into the development process, as such projects are closely linked with the issue system.

Each repository may contain multiple projects, each of which could be used to plan a particular feature or release.

Each project has title and description, which are displayed in the projects section of a repository page.

New Project

 2 Open ✓ 1 Closed

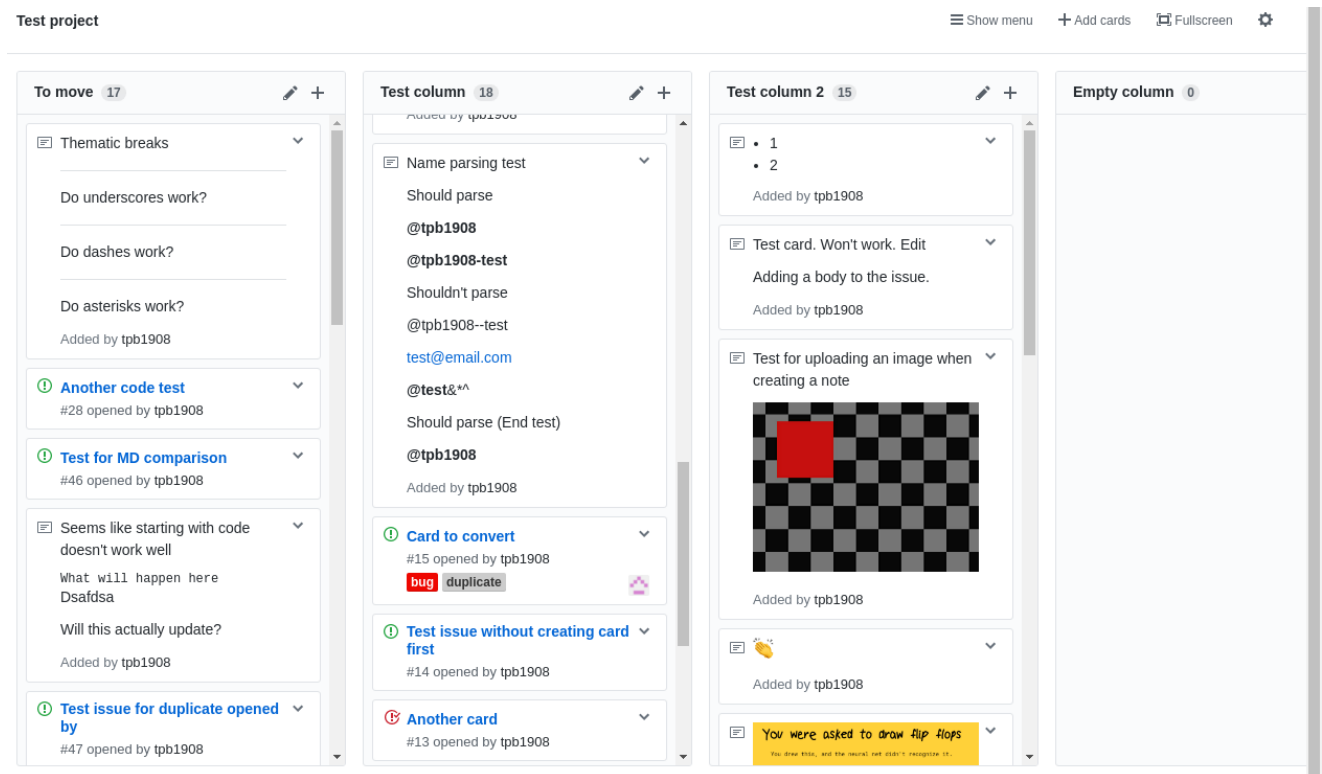
Sort ▼

Many cards Updated 20 days ago	No description	▼
Test project Updated 6 days ago	No description	▼

Since their release projects have been updated to include the same possible states as issues.

A project consists of multiple columns of content.

Each column has a title and can be reordered within the project.



Each item within a column is either a card, which is simply a string of markdown up to 250 characters in length, or a reference to an issue, which displays a link to the issue, its state, labels, and assignees.

Integrations

Integrations are designed to extend GitHub's functionality.

An integration can be installed to a user's account or to a single repository, allowing it hook access.

When an integration has hook access, it is notified of changes to the repository, which might be a new commit being pushed to the repository or an issue being created.

An integration could be used to automatically tag issues based on keywords in their text.

A more common use of integrations is to notify a continuous integration system. Continuous integration is a practice where checked in code is verified against an automated build system, allowing the developer to be notified of a problem without manually running tests.

Continuous integrations are often used as checks on pull requests. If a check fails the pull request will not be merged.

Markdown

Markdown is a markup language with plain text syntax.

Markdown doesn't have a common standard, however there are some universal rules.

```
# Heading level 1
```

```
## Sub-heading
```

```
### Another deeper heading
```

```
Paragraphs are separated  
by a blank line.
```

```
Two spaces at the end of a line leave a  
line break.
```

```
Text attributes _italic_, *italic*, __bold__, **bold**, `monospace`.
```

```
Horizontal rule:
```

```
---
```

```
Bullet list:
```

- ```
* item 1
```
- ```
* item 2
```
- ```
* item 3
```

```
Numbered list:
```

- ```
1. item 1
```
- ```
2. item 2
```
- ```
3. item 3
```

```
A [link](http://example.com).
```

The markdown above will be formatted as

Heading level 1

Sub-heading

Another deeper heading

Paragraphs are separated
by a blank line.

Two spaces at the end of a line leave a line break.

Text attributes *italic*, *italic*, **bold**, **bold**, monospace.
Horizontal rule:

Bullet list:

- item 1
- item 2
- item 3

Numbered list:

1. item 1
2. item 2
3. item 3

A [link](#).

GitHub uses its own markdown structure with some extra features.

Lists

Dashes can be used to create lists:

```
- Item 1  
- Item 2  
- Item 3
```

creates

- Item 1
- Item 2
- Item 3

Nested lists are also supported

```
- Item 1  
  - Item 1 depth 2  
  - Item 2 depth 2  
- Item 2
```

creates

- Item 1

- Item 1 depth 2
 - Item 2 depth 2
- Item 2

Images

Images can be inserted inline

```
![Image description](image_url)
```

should load and display the image

further,

```
![Image description](relative_path)
```

should attempt to load the image from the current repository.

Blockquotes

Blockquotes are displayed in the same way as a HTML blockquote tag

```
> Some quoted text  
> across  
> multiple lines
```

Some quoted text
across
multiple lines

Strikethroughs

Placing two tildes on either side of a sequence will draw a strikethrough through it

```
~~text~~
```

will be displayed as

~~text~~

References

Issue references

A # followed by the number of an issue within the repository will be shown as a link to that issue

User references

An @ followed by a username will be shown as a link to that user

Emoji

Emoji names wrapped in colons are converted to emoji characters

```
:camel:
```

is displayed as the emoji character 🐫

Code

Code can be inserted between triple backticks

```
``` Language
```

Some code

```
```
```

Will display the code without any other formatting

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                // evil floating point bit
level hacking
    i  = 0x5f3759df - ( i >> 1 );        // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
//    y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can
be removed

    return y;
}
```

HTML

The HTML tags which correspond to each of the markdown features can be used in place of the markdown characters.

There are also further tags which do not have a markdown equivalence:

The sup and sub tags display text as ^{super} and _{sub} script respectively.

The font tag can be used to choose a text colour and face

```
<font color="red" face="impact">Formatted text</font>
```

gives **Formatted text**

Android Application Structure

Activities

The Android developer documentation defines an Activity as 'a single, focused thing that the user can do.'

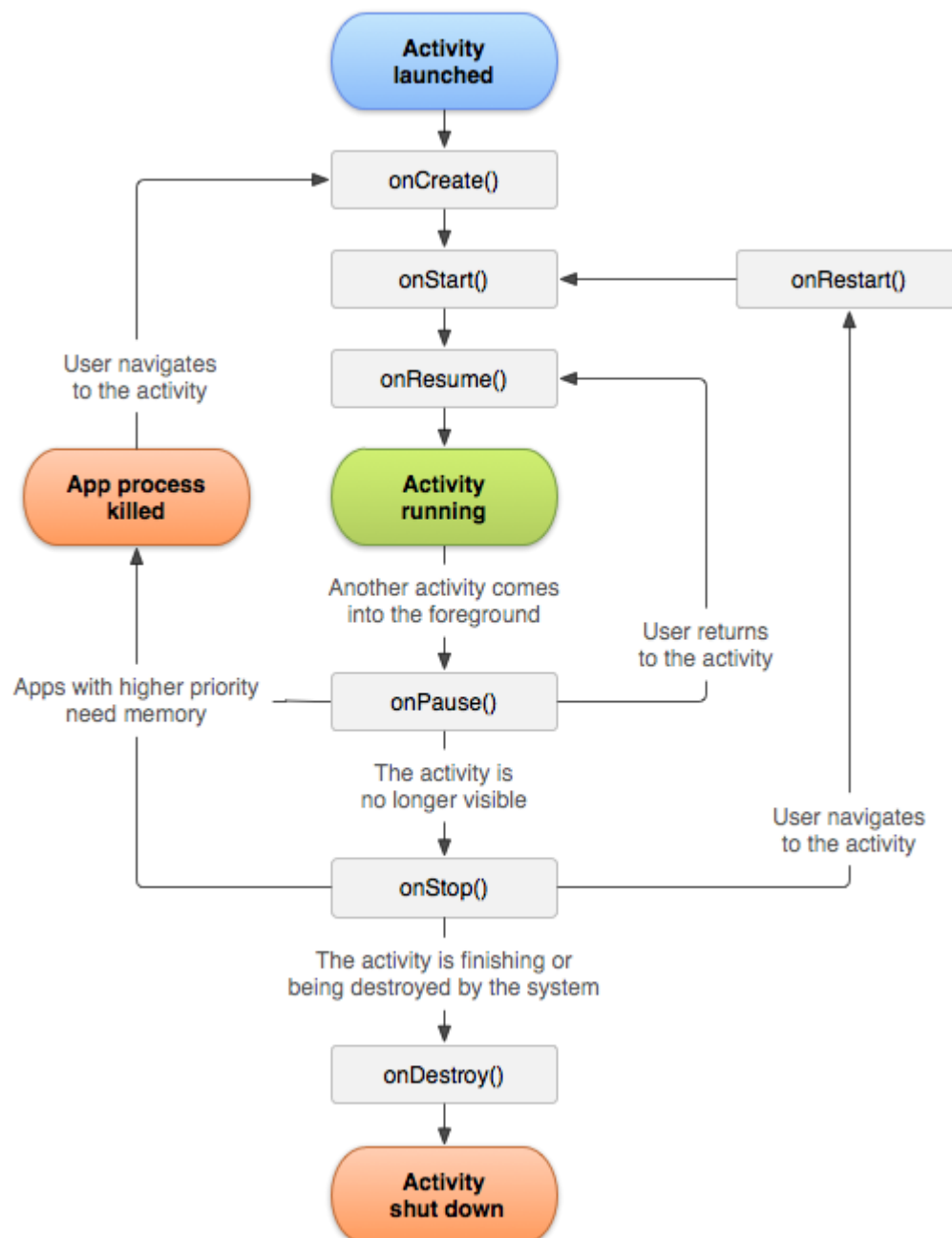
The Activity class is responsible for creating the window in which an application's UI is placed.

When an Activity is launched, the Android system first calls the `onCreate` method, in which the Activity should create its view tree (if applicable) and perform any setup that it needs to.

Next the `onStart` method is called, making the Activity visible to the user.

Finally, the system calls `onResume` when the Activity enters the foreground. In this state the user can interact with the Activity.

The Activity will stay in the resumed state until it exits it, or another Activity



Intents

Activities are launched by Intents.

An Intent is 'an abstract description of an operation to be performed'. Intent objects are used throughout the Android system to communicate both within apps, and between them.

Intents are messaging objects used to request an action from another application component, within the same app or a different one.

An example of a simple implicit Intent is to launch a messaging application to send a string of text.


```
// Create the text message with a string
final Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "Our message");
sendIntent.setType("text/plain");

// Verify that there is an application to handle the Intent
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

If the user has already chosen a default application for `Intent.ACTION_SEND` the Android system will launch the specified Activity, otherwise it will show a chooser dialog.

Intent handling

In order for any Activity to be launched it must be declared in the application's manifest, with an item pointing to the Activity class.

```
<activity android:name=".app.TestActivity" />
```

Declaring the Activity in the manifest allows it to be launched, however to be launched from outside of the app the Activity must specify an Intent filter.

To specify that an Activity is a MAIN Activity, one which can function as an entry point for the app and does not require being launched with data, the MAIN intent filter must be registered.

Further, to be shown in the device's app drawer, allowing the user to easily launch it, the Activity must register the LAUNCHER Intent in its Intent filter.

The simplest primary Activity for an app has the following manifest registry:

```
<activity
    android:name=".app.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

This allows the app to be launched from the user's home-screen.

Layouts

Layouts in Android apps are usually defined in XML.

The base class for an Android View is the `view` class in the `android.view` package. `view` directly extends `object` and contains many methods and interfaces for laying out and drawing the `view` on screen.

All individual views, for example to input text or display an image, extend `view`. However, layouts would be difficult to work with if it were only possible to manipulate individual views.

A `viewGroup` is a subclass of `view` which can contain other views, called children. The `viewGroup` class is the base class for all layouts and view containers.

One of the most commonly used ViewGroups is `LinearLayout` which can align other views in a linear fashion either horizontally or vertically. This would be useful for setting out multiple Views one after another on a device screen. There are also more complicated ViewGroups such as `RelativeLayout` which aligns its children relative to each other and itself.

Layout inflation

While view objects can be, and are, created at run time, it is usually much quicker to write, and more easily understandable to define the layout in a layout resource file.

A layout resource file is an XML file containing the view classes to be shown on screen and their attributes.

A simple login Activity might have the following layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/user_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPersonName"
        android:hint="Username"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPassword"
        android:hint="Password"/>

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Login"/>
</LinearLayout>
```

Each view must have a defined width and height, however other values need not be specified.

While absolute values can be specified, such as the hint and text attributes in the example above, this is not recommended as it is more difficult to refactor and much more difficult to adapt to other languages and locales.

Instead attributes should be specified in the applications values directory.

The values directory is part of the resources directory, which also contains layout

files and other resources such as images.

Hint text is specified in a file called strings.xml

```
<resources>
    <string name="hint_login_username">Username</string>
</resources>
```

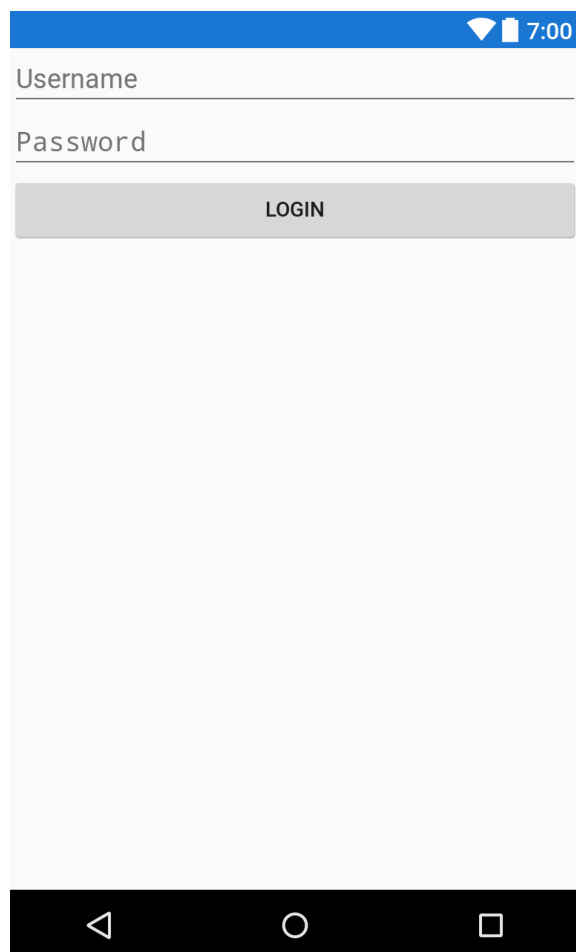
This value can then be referenced elsewhere as

```
<EditText
    android:id="@+id/user_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="textPersonName"
    android:hint="@string/hint_login_username"/>
```

However the values are specified, the XML will then be used by the `LayoutInflater` class to create a tree of view objects with the specified attributes.

The attributes are passed as an `AttributeSet` object which is usually used to obtain a `TypedArray` which contains the resolved attributes.

When the layout above is inflated and set as the root layout in an `Activity` it will look like the image below:



Dimensions

The example above used `match_parent` and `wrap_content` to define the dimensions of each `View`.

These values refer to constants which, as their names suggest, inform the system to layout the `views` either to fill the available space within their parent `ViewGroup` or to the space requested by the instance of `View`.

In order to properly layout a collection of `views`, actual sizes must be specified. `views` can be specified as having a particular dimension in pixels with the `px` unit, which simply draws the `view` across the requested number of pixels.

The three units which directly relate to a physical dimension are as follows

- in Inches based on the size of the screen
- pt 72nds of an inch based on the physical size of the screen
- mm Millimetres based on the size of the screen

In practice, none of the four units above are particularly useful because device dimensions and resolutions vary greatly.

There are therefore sizes independent of the size and resolution of the device:

- dp or dip are density independent pixels based on the physical density (dots per inch) of the screen. Density independent pixels are relative to a screen with a density of 160 dots per inch. This allows a `view` to maintain its proportions on different resolutions and form factors.
- sp are scale independent pixels which are similar to dp, but are also scaled with the user's font size. This unit is recommended for use with text in order to respect both the device dimensions and the user's preference.

View binding

In order to interact with the `view` tree and provide a functional application, the application code must have access to the inflated `view` objects.

This is achieved by assigning ids to each view.

As shown in the layout above, every view can have an `id` attribute. This is a string which must be unique to the current view tree. When the application is built, all of the attributes for each module in the application are collected into a single class named `R`, with subclasses for each of the resource types.

For a complete application the `R` class is usually thousands of lines

```

1  + /.../
7
8  package com.tpb.projects;
9
10 public final class R {
11  + public static final class anim {...}
38  + public static final class animator {...}
41  + public static final class array {
42      public static final int settings_card_actions=0x7f0e0000;
43  }
44  + public static final class attr {...}
3009 + public static final class bool {...}
3016 + public static final class color {...}
3135 + public static final class dimen {...}
3285 + public static final class drawable {...}
3455 + public static final class id {...}
3867 + public static final class integer {...}
3879 + public static final class layout {...}
4003 + public static final class menu {...}
4014 + public static final class mipmap {...}
4017 + public static final class plurals {...}
4025 + public static final class string {...}
4427 + public static final class style {...}
4820 + public static final class xml {...}
4823 + public static final class styleable {...};
12944 }
12945

```

The id class contains the integer values of all the ids used throughout the app.

In our Activity we create member variables for the views which we need to access.

```

package com.tpb.example;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {

    private EditText mUserNameEditor;
    private EditText mPasswordEditor;
    private Button mLoginButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}

```

After calling setContentView we can assign the inflated views to these variables.

```

setContentView(R.layout.activity_main);
mUserNameEditor = (EditText) findViewById(R.id.user_name);
mPasswordEditor = (EditText) findViewById(R.id.password);
mLoginButton = (Button) findViewById(R.id.button);

```

While this method is part of the Android SDK, and the default way to access the view tree, it is verbose and quickly becomes difficult to read when accessing complex layouts.

The most common solution to this problem is ButterKnife.

ButterKnife is an annotation processor which generates the necessary lookups from annotations at build time.

The same Activity class using ButterKnife is as follows

```
@BindView(R.id.user_name) EditText mUserNameEditor;
@BindView(R.id.password) EditText mPasswordEditor;
@BindView(R.id.button) Button mLoginButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ButterKnife.bind(this);
}
```

ButterKnife can also other resources

```
@BindColor(R.color.colorPrimary) int primaryColor;
```

As well as assigning click listeners to Views

```
@OnClick(R.id.button)
void onButtonClick() {
    //Do something
}
```

rather than

```
mLoginButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //Do something
    }
});
```

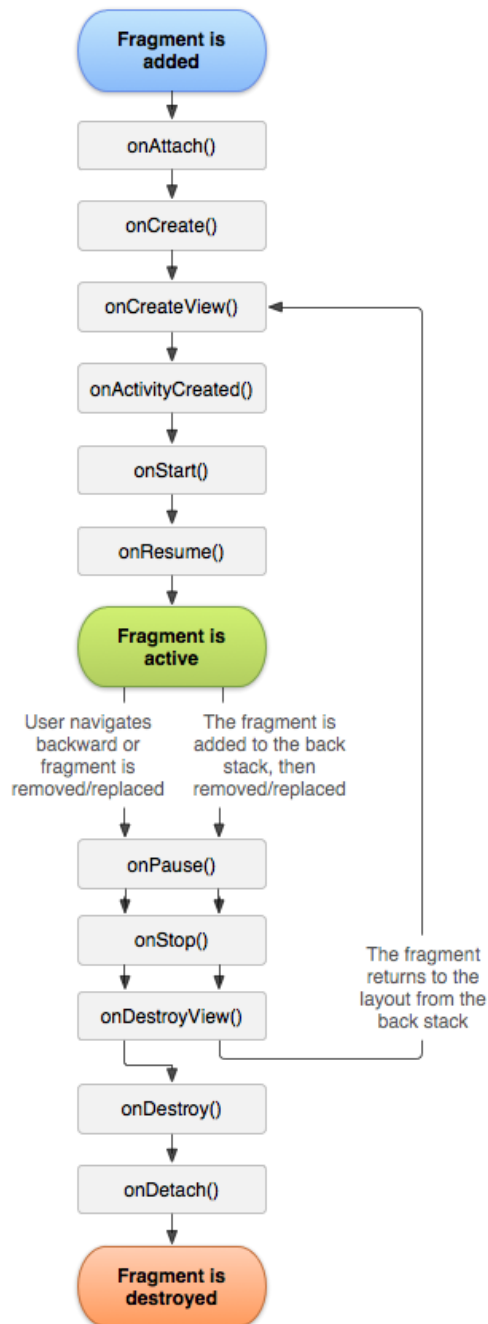
Fragments

In all cases, at least one Activity must be used to display an application, however it is not always necessary, and often adds unnecessary complexity to use a full Activity for each component of the application.

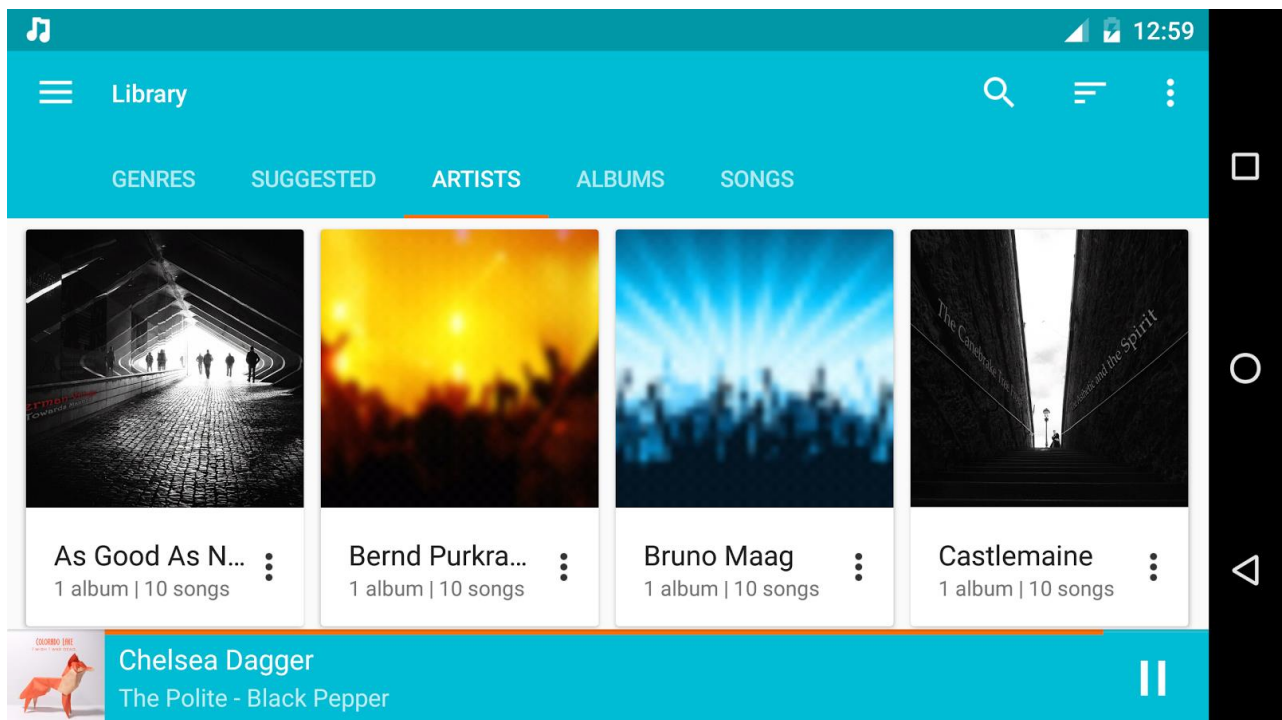
A Fragment is 'a behaviour or a portion of user interface in Activity'.

Multiple Fragments can be combined within an Activity to produce dynamic layouts on different devices, and allow easy reuse of components across different parts of an application.

A Fragment has its own lifecycle separate from its parent Activity



Fragments are commonly used in `viewPagers` to display multiple related sets of content within the same Activity.



RecyclerViews

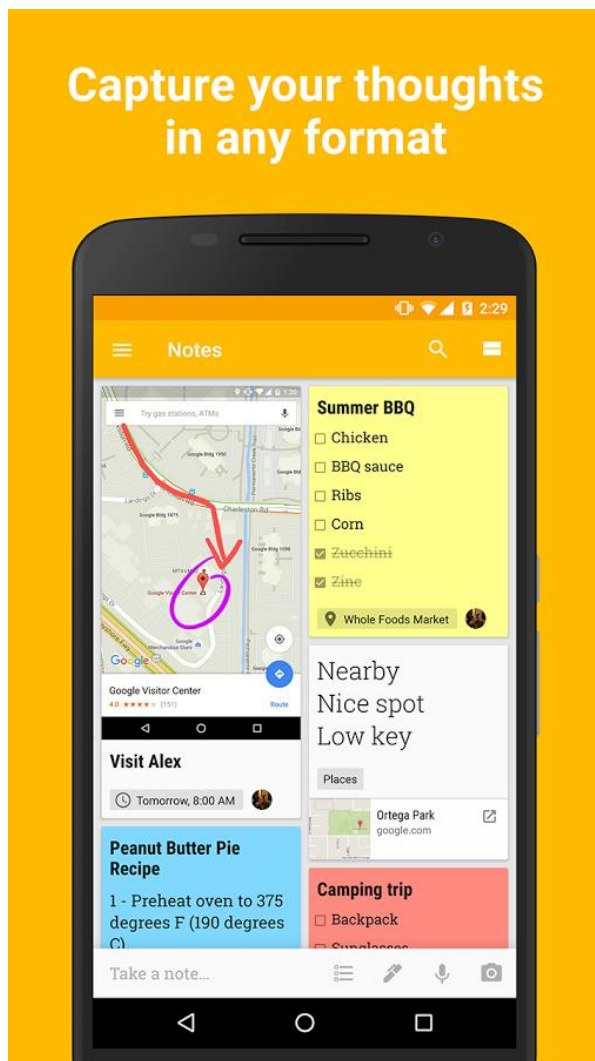
The `ListView` class was added to Android in API level 1.

As its name suggests a `ListView` is used to show a set of views in a scrollable list.

The `RecyclerView` was added in API level 22 (February 2015) with the intention of replacing the `ListView`.

The `RecyclerView` brought three key improvements over the `ListView`

1. In a `RecyclerView` views are always reused as the user scrolls. When the `RecyclerView` is first drawn, it instantiates as many `ViewHolder` instances as are necessary to fill the screen and provide some padding before reusing these views by binding the appropriate data to them rather than creating new instances. This reduces memory usage and stops unnecessary `findViewById` calls which are costly as they require traversing the view tree.
2. Animations are decoupled from the individual views and delegated to an `ItemAnimator` which provides default animations for common actions such as insertion and removal of items.
3. The list itself is decoupled from its containing `ViewGroup` which allows much more complex item layouts with `LayoutManagers` such as the `StaggeredGridLayoutManager` which can view of different sizes as in Google Keep (Below).



Basic storage

While many applications use a variety of database solutions, it is not always necessary or the best solution to store simple values in a full database.

The SharedPreferences system is a persistent set of key value pairs which can be used to store application information such as settings.

Each 'preference' is a key value set stored under a particular name.

Once a SharedPreferences object has been returned for the set, the key value pairs within the set can be read from and written to.

Build system

Android uses the Gradle build system.

Gradle converts each of the files in the project to the necessary type to be packaged in an application, as well as performing minification and obfuscation if told to.

Java files are converted to dex files, and XML files are built into the required resource classes.

Gradle is a plugin based system which makes it particularly useful for Android development which often contains multiple languages other than Java, such as C and C++ through the native development kit, as well as other Java Virtual Machine based languages such as Kotlin and Scala which can replace Java entirely and provide many useful features not present in versions of Java present on older devices.

A build script for a near empty project may appear as follows

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.tpb.example"
        minSdkVersion 21
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
            "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-
        core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.0'
    compile 'com.jakewharton:butterknife:8.5.1'
    annotationProcessor 'com.jakewharton:butterknife-compiler:8.5.1'
    testCompile 'junit:junit:4.12'
}
```

This build script includes the Android support library, ButterKnife, and runners for the built in tests.

All applications have a debug build type, however this build script also specifies a release build type which would result in ProGuard being applied to obfuscate the source, and the test dependencies not being included.

More complicated scripts can also be included:

In order to determine when a bug was introduced, it is useful to have a different versionCode for each change made to the code.

If the Git version control system is being used, this can be achieved by checking the number of commits when the application is built.

This is done by executing a Git command to count the number of revisions.

```
def gitInfo() {
    def commitCount = 0
    def revision = "(unknown revision)"
    try {
        def stdout = new ByteArrayOutputStream()
        exec {
            commandLine 'git', 'rev-list', '--all', '--count'
            standardOutput stdout
        }
        commitCount = stdout.toString().trim().toInteger()
        stdout = new ByteArrayOutputStream()
        exec {
            commandLine 'git', 'rev-list', 'HEAD', '-n', '1'
            standardOutput stdout
        }
        revision = stdout.toString().trim()
    }
    catch (error) {
        println "Error: ${error}"
    }
    return [commitCount: commitCount, revision: revision]
}
```

The versionCode can then be written `versionCode gitInfo().commitCount`.

The Android build system will generate an APK (Android Package Kit) which can be installed on a test device.

Debugging

Whether running their application on a physical device or an emulator, communication with the device is managed through ADB (Android Debugging Bridge).

Once a device has been connected via ADB the Logcat can be used for simple debugging.

The Logcat is used to print messages and exceptions and is usually used through the Log class.

The Log class provides 6 static methods for logging different levels of information.

Each method takes a TAG string which is used to identify the source of the log, and most methods have overloaded methods for logging strings as well as Throwable exceptions.

- V (Verbose) The highest level of logging, generally used when debugging

- I (Information) Used to report useful information, usually that an operation has completed successfully
- D (Debug) Used for debugging purposes only, to track the flow of an application
- W (Warning) Used to notify of unexpected behaviour
- E (Error) Used to log an error which is known to have occurred. Often prints a stack trace.
- WTF (What a terrible failure) Used to report an exception that should never happen.

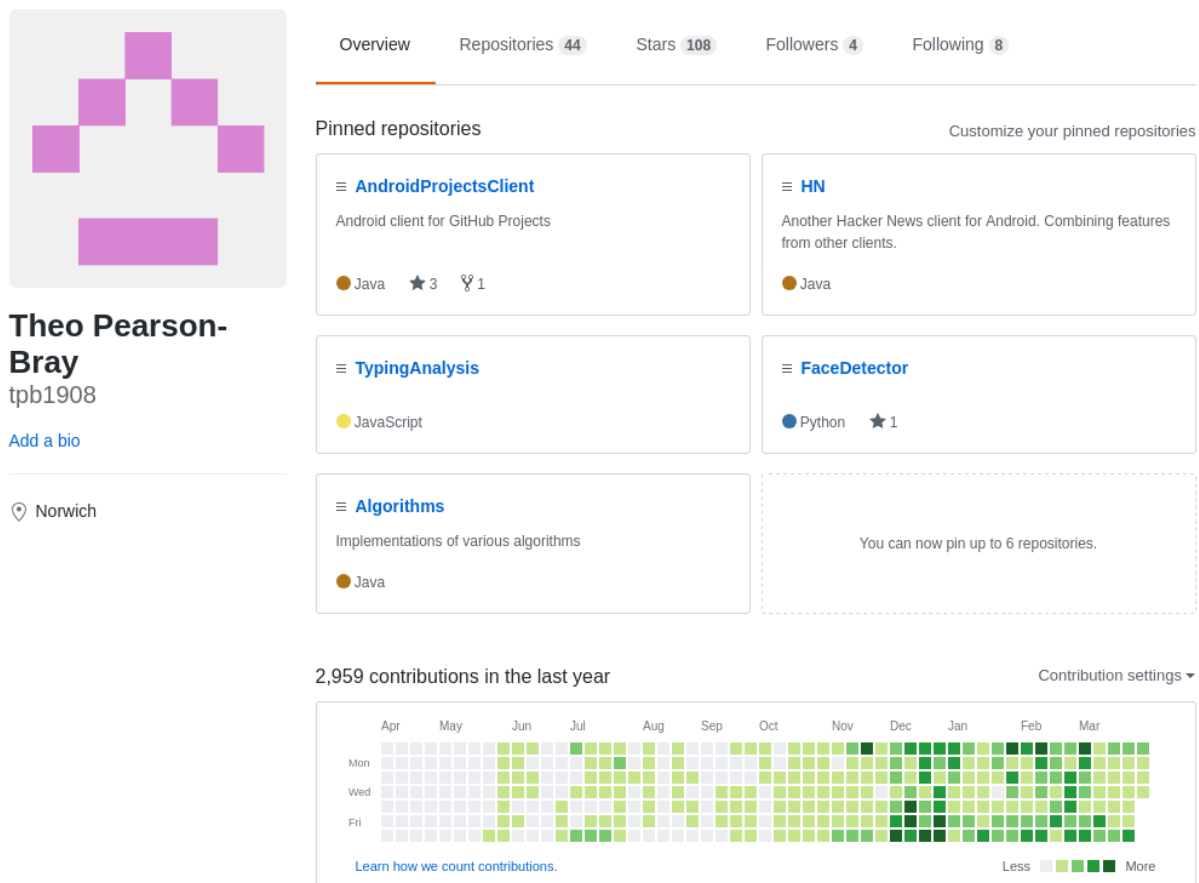
Proposed design

As explained in the background section, an Android app is made up of Activities and Fragments.

In order to implement the functionality listed above the app requires multiple activities to mimic the pages on the GitHub website.

User information Activity

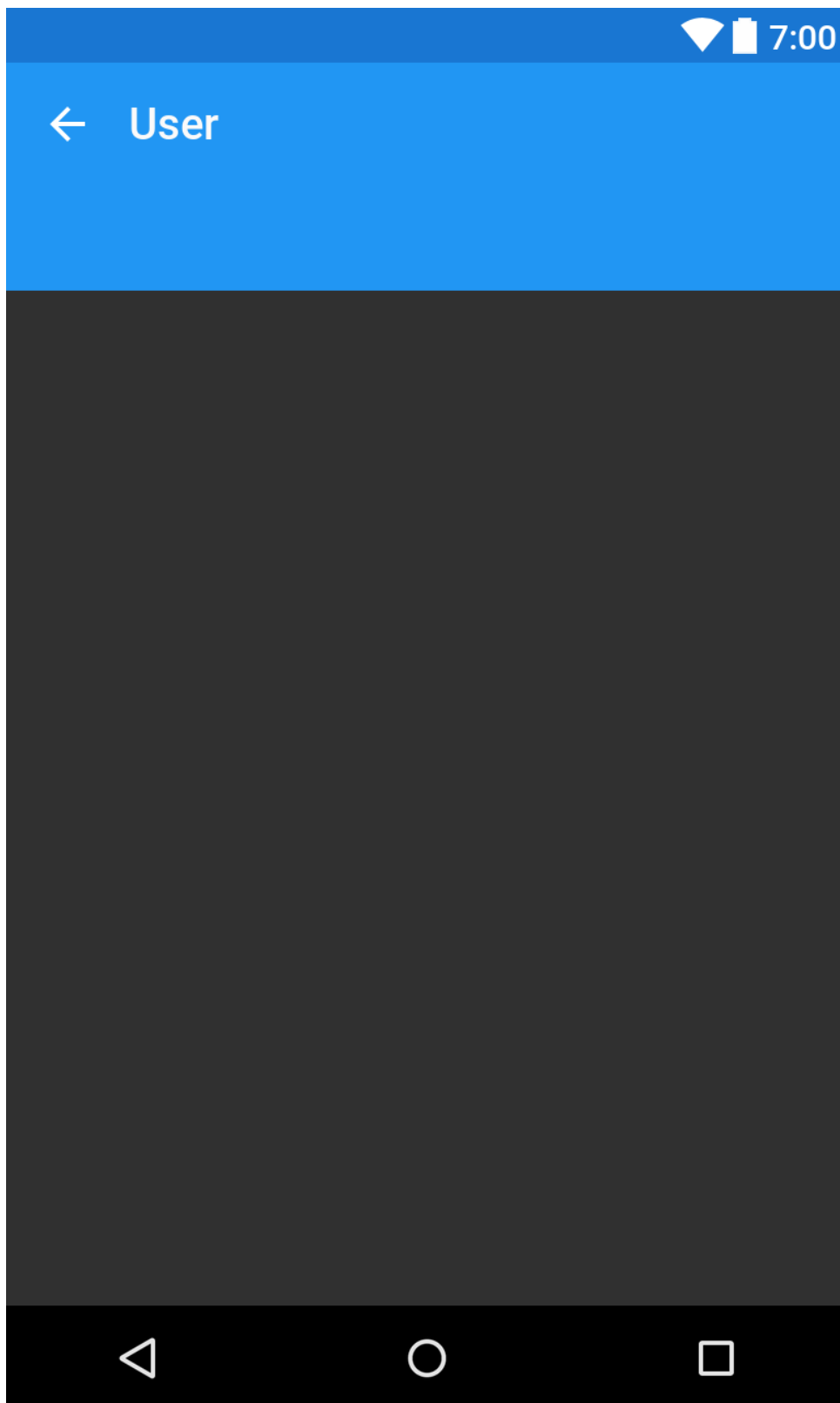
The initial Activity should display the same information that a user would see when they navigated to their own GitHub page.



This page can be easily adapted to a mobile layout as it is already paginated.

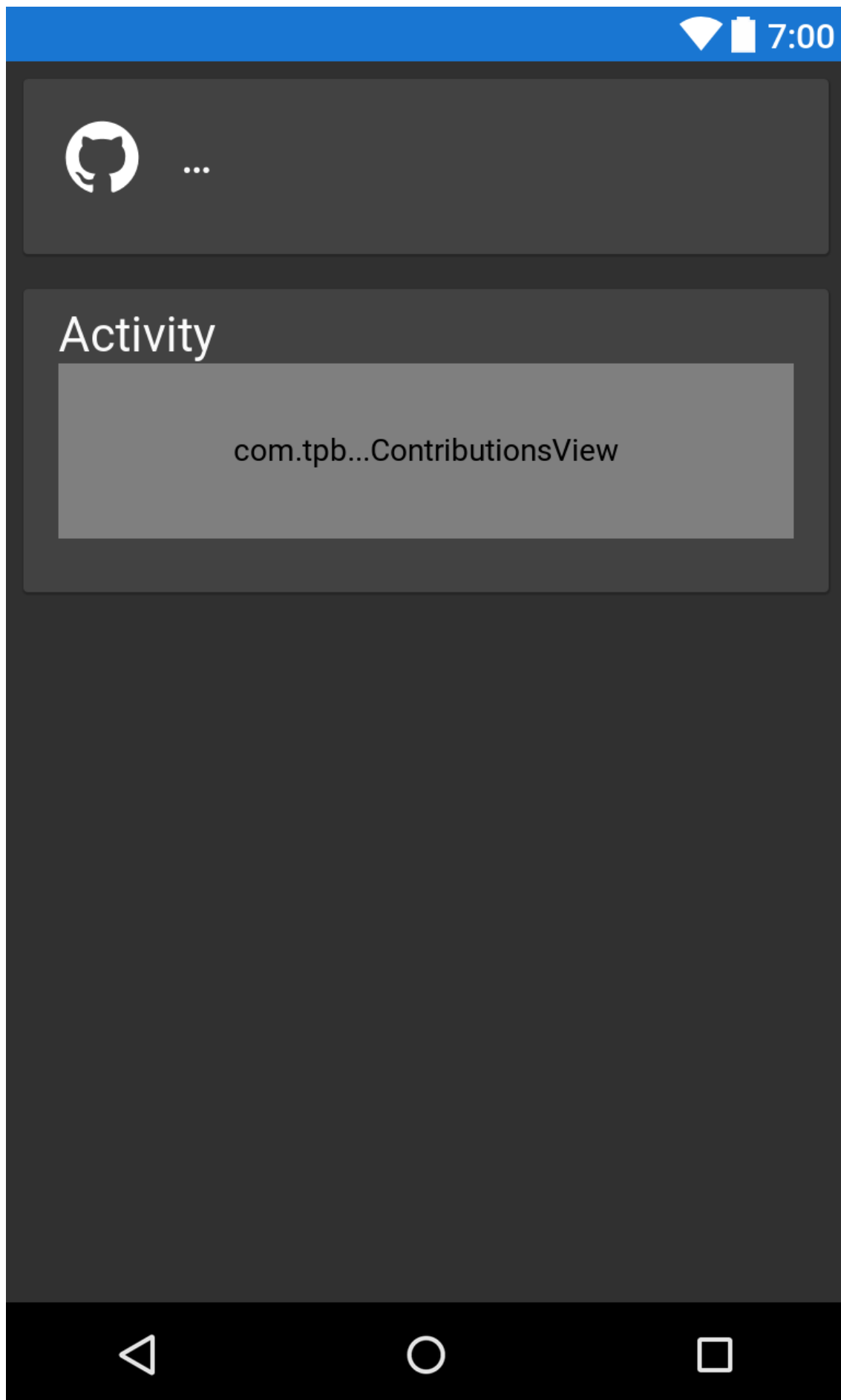
The root activity layout only needs to contain the navigation interface.

This is the toolbar which displays back and settings navigation, and the `ViewPager` layout which displays the different page names.



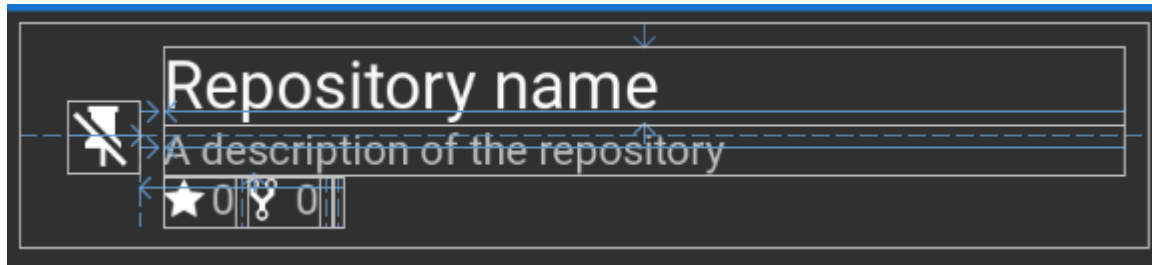
Within the `ViewPager` for this Activity, there will be 6 pages.

The first should display information about the user and their activity



The second should list the user's repositories

A single repository is shown as a list item with the repository name, description, stars and forks, and a button to pin the repository.



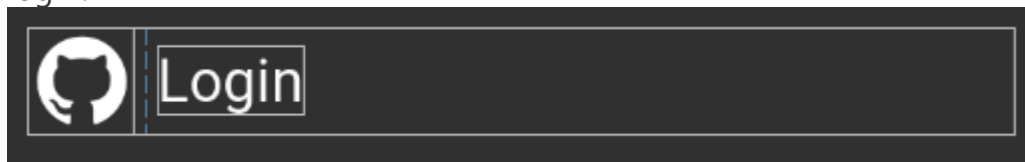
The third lists the user's starred repositories, using the same list item layout as the user's own repositories without the pin button.

The fourth lists the user's Gists, with each item showing the Gist title and description.

The fifth lists the users that the authenticated user is following.

The sixth lists the users that are following the authenticated user.

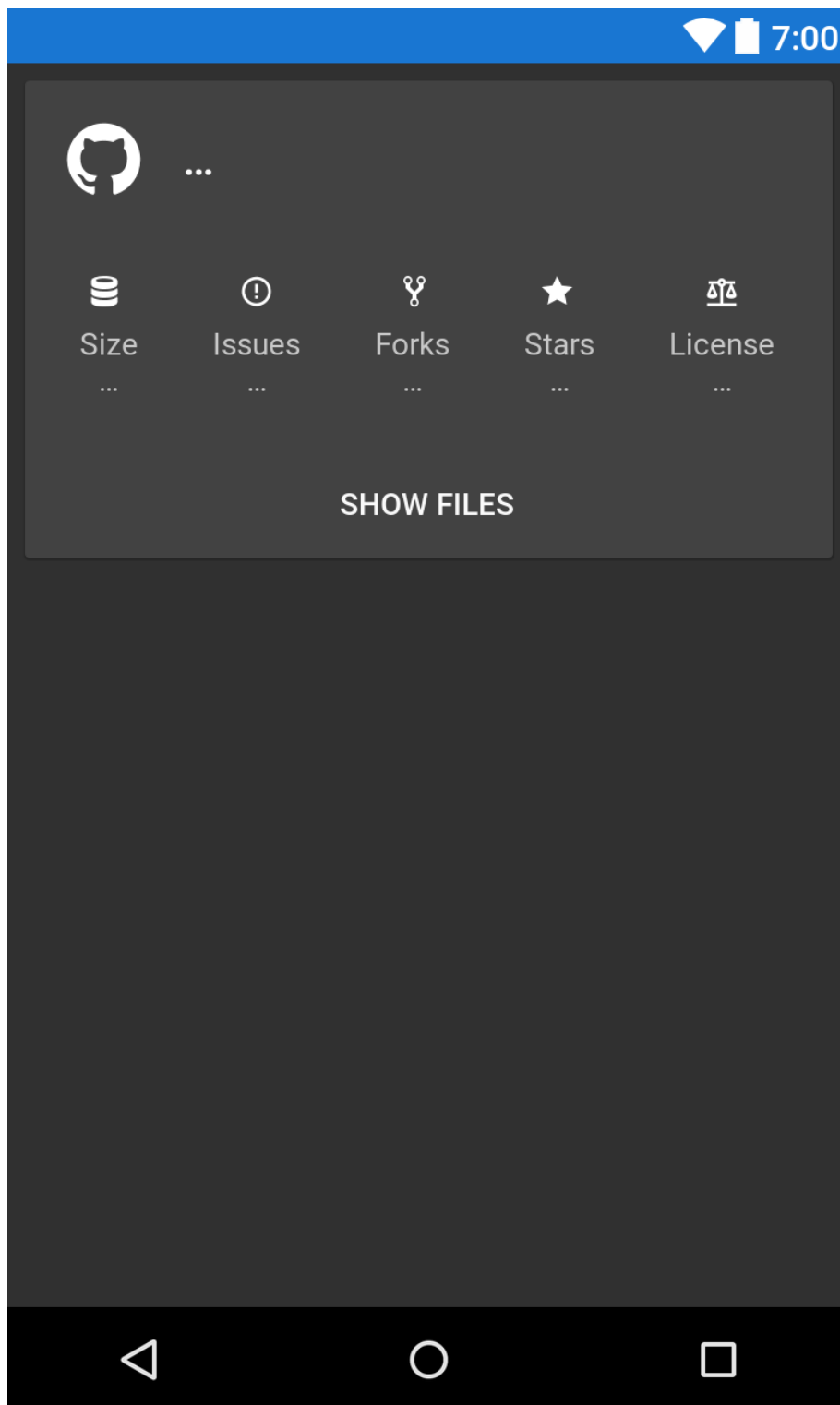
Each user listed in the fourth or fifth `Fragment` is displayed with their avatar and login.



Repository information Activity

The repository information `Activity` is also split into multiple `Fragment`s

The first `Fragment` should display information about the repository, as well as the users related to it.

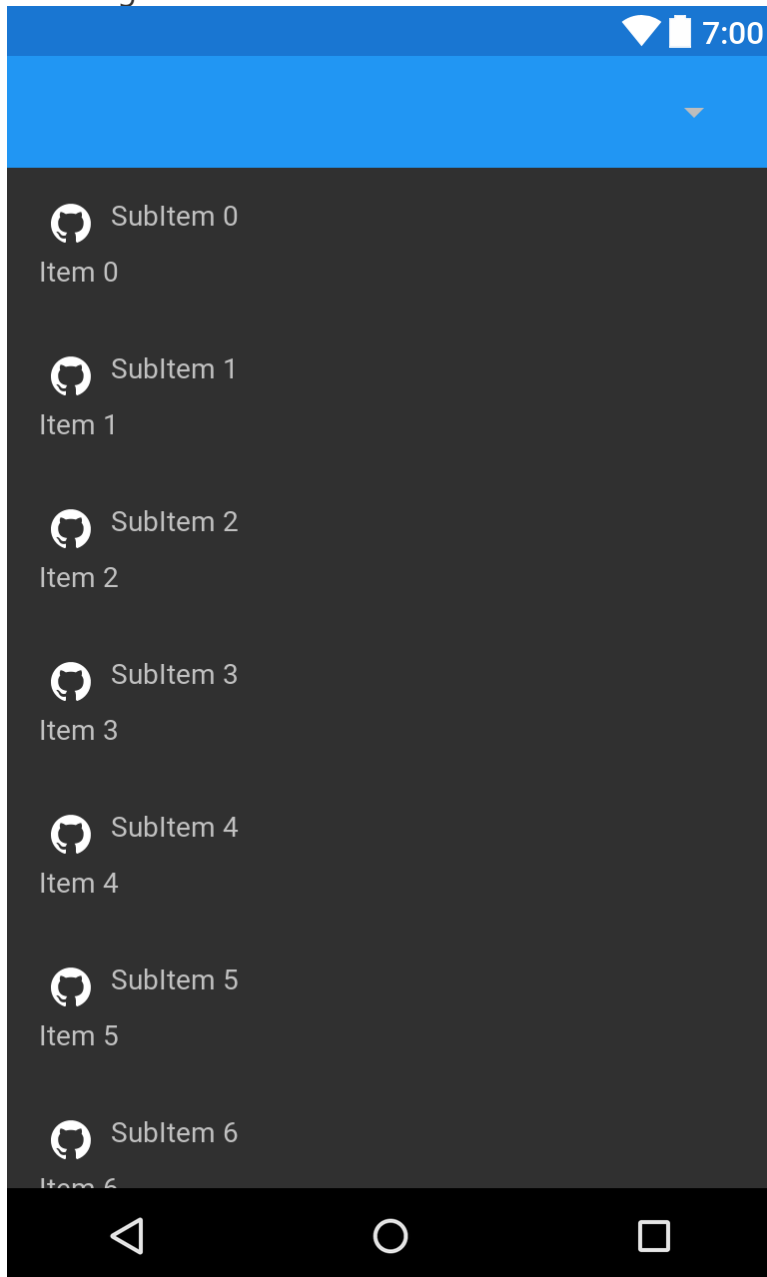


Once populated with data this layout will show the user that created the repository, the size of the repository, the number of issues, the number of forks, the number of stars, and the license.

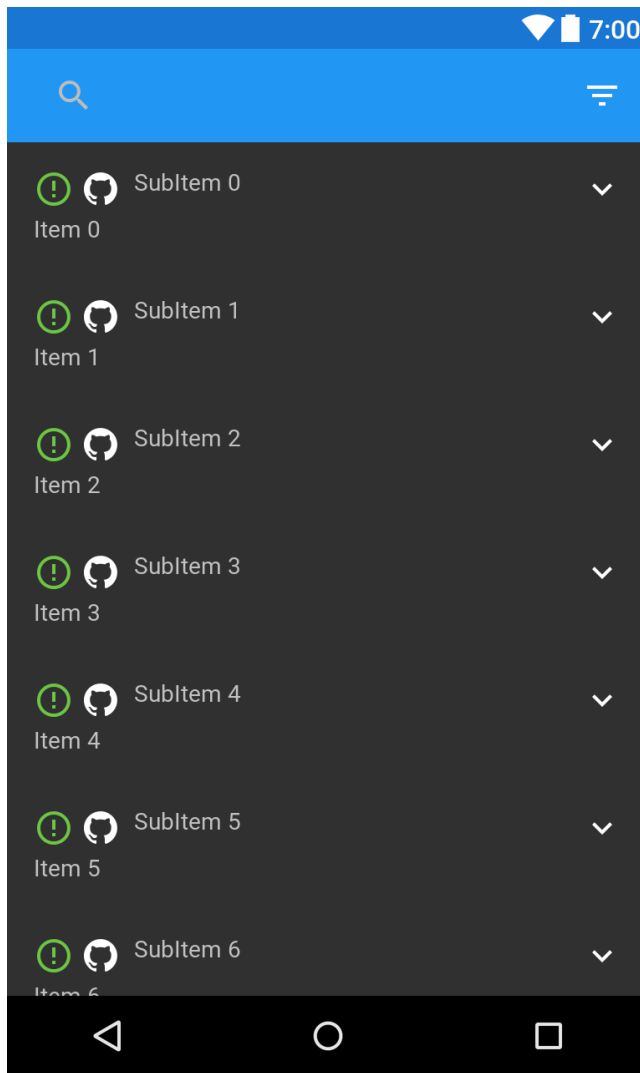
If there are users collaborating on or contributing to the repository they will be shown in a `HorizontalScrollView`.

The second `Fragment` should display the repository's README file.

The third `Fragment` should display the commits made to the repository, and allow selecting the branch for which to view commits.

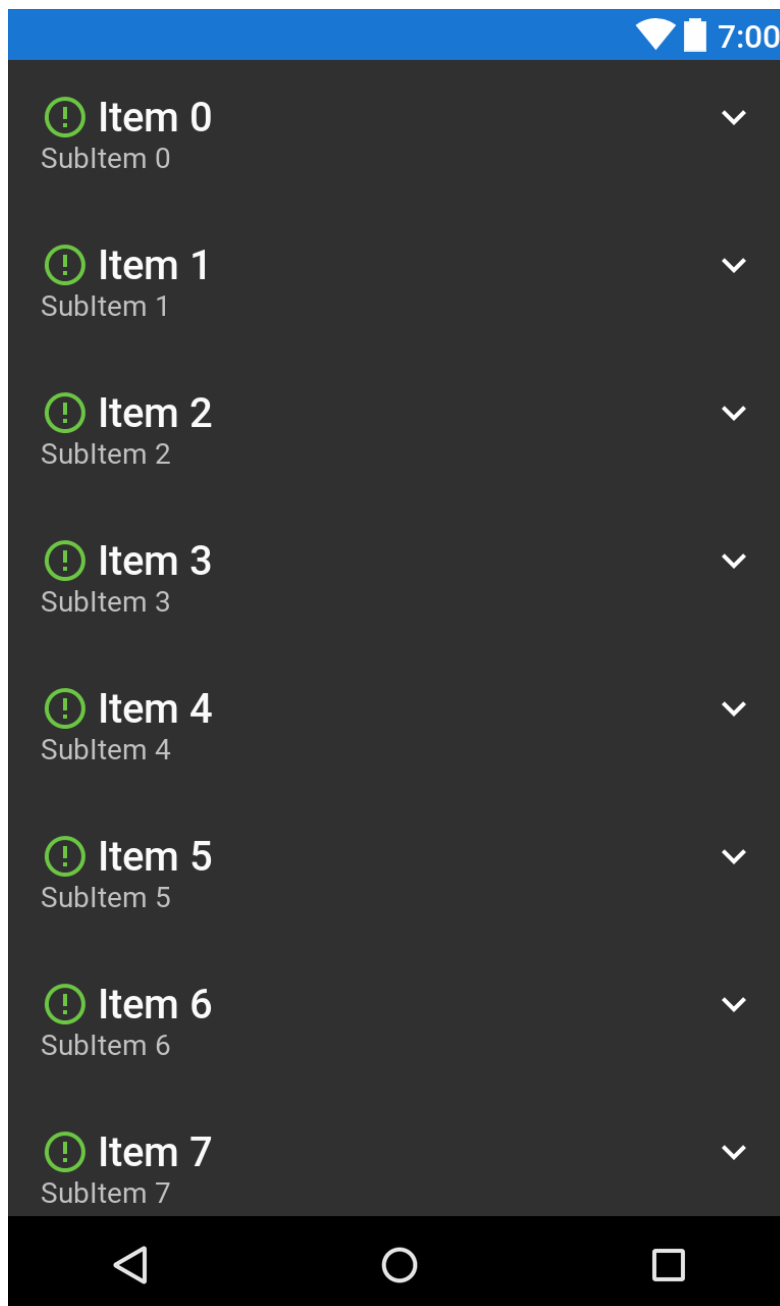


The fourth `Fragment` should display the issues on a repository, and allow filtering as well as searching the issues.



Each list item shows the issue state, the user that created the issue, issue information and an overflow button for performing actions on the issue.

The fifth and final Fragment displays each of the projects associated with the repository.



Each list item displays the project name, its description, the last time it was updated, its state, and an overflow button for performing actions on the project.

In order to be more useful than GitHub's mobile website, and other clients, an Android client must implement most of the functionality of the desktop website.

The client must implement the following features:

1. Sign in
 - a. Allow the user to log in to GitHub with their credentials
 - b. Store the authentication token received from GitHub
2. Users
 - a. Display available information about a user
 - i. Name
 - ii. Avatar
 - iii. Join date
 - iv. Contributions in graphical form
 - v. Statistics on contributions
 - b. Repositories
 - i. List the repositories that the user has created
 - ii. List private repositories for the authenticated user
 - iii. Display the repository primary language
 - iv. Display the last time that the repository was updated
 - v. Display the number of users that have starred or forked the repository
 - vi. Allow a user to pin a repository to the top of their repository list
 - c. Display the list of repositories that a user has starred
 - d. Gists
 - i. Display the gists that the user has created
 - ii. Display private gists for the authenticated user
 - e. Display the users that a user is following
 - f. Display the users that are following a user
 - g. Implement following and unfollowing of users
3. Repositories
 - a. Display information about the repository
 - i. Repository size
 - ii. Number of issues
 - iii. Number of forks
 - iv. Number of stars
 - v. The repository license type
 - vi. Display the text of the license

- b. Repository files
 - i. Display the repository file tree
 - ii. Implement viewing the file tree for different branches
 - iii. Display files within the repository
- c. Display the README in a suitable format for a mobile device
- d. Commits
 - i. List the commits made to a repository
 - ii. Implement selecting the branch for which to show commits
 - iii. Display the user that made the commit, and the commit message
- e. Issues
 - i. List the issues made on a repository
 - ii. Display information about each issue
 - 1. The issue state
 - 2. The issue number
 - 3. The user that opened the issue
 - 4. The user that closed the issue (If applicable)
 - 5. The date at which the issue was opened
 - 6. The user(s) assigned to the issue
 - 7. The tag(s) added to the issue
 - 8. The number of comments made on the issue
 - iii. Implement filtering of the issues list
 - 1. By state
 - a. Open
 - b. Closed
 - c. All
 - 2. By labels
 - 3. By assigned user
 - iv. Implement searching of the issues list
 - 1. Real time searching of the list
 - 2. Fuzzy string matching when searching contents
 - v. Implement toggling issue state
 - vi. Implement editing of issues (See issues section)
 - vii. Implement creation of issues (See issues section)
- f. Projects
 - i. List the projects made on a repository
 - ii. Display information about each project
 - 1. Name

2. Description
 3. State
 - a. Open
 - b. Closed
 4. Last date updated
4. Issues
- a. Display information about the issue
 - i. Title
 - ii. Number
 - iii. State
 - iv. Body
 - v. User that opened the issue
 - vi. Date that the issue was opened
 - vii. User(s) assigned to the issue
 - viii. Labels added to the issue
 - ix. Display the list of events which have occurred on the issue
 1. Closed - When the issue was closed, and the commit if it was closed from a commit message
 2. Reopened- When the issue was re-opened, and by whom
 3. Subscribed- When a user subscribed to the issue, and who subscribed
 4. Merged- When the issue (A pull request) was merged, the commit, and the user that merged the issue
 5. Referenced- When the issue was referenced from a commit message, and the commit
 6. Mentioned- When a user was mentioned in the body, and the user that was mentioned
 7. Assigned- When a user was assigned to the issue, the user that was assigned, and by whom
 8. Unassigned- When a user was unassigned from the issue, the user that was unassigned, and by whom
 9. Labeled- When a label was added to the issue, and by whom
 10. Unlabeled- When a label was removed from the issue, and by whom
 11. Milestoned- When the issue was added to a milestone, and by whom

12. Demilestoned- When the issue was added to a milestone, and by whom
13. Renamed- When the issue was renamed, the old and new names, and the user that renamed the issue
14. Locked- When the issue was locked, and by whom
15. Unlocked- When the issue was unlocked, and by whom
16. Head ref deleted- When the pull request's branch was deleted
17. Head ref restored- When the pull request's branch was restored
18. Review requested- When a user was requested to review the pull request, and by whom
19. Review request dismissed- When a review request was dismissed, and by whom
20. Review request removed- When a request for a user to review the pull request was dismissed, and by whom
21. Added to project- When the issue was added a project board
22. Moved columns in project- When the issue was moved between columns in a project board
23. Removed from project- When the issue was removed from a project board
24. Converted note to issue- When the issue was created by conversion of a project note to an issue

b. Comments

- i. Display the list of comments on the issue
 1. The user that created the comment
 2. The date that the comment was created
 3. The reactions to the comment
 4. The comment body
- ii. Implement editing of comments made by the authenticated user
- iii. Implement creation of comments made by the authenticated user

c. Implement editing of

- i. The issue title
- ii. The issue body
- iii. The issue assignee(s)
- iv. The issue label(s)

5. Commits

- a. Display information about the commit
 - i. The commit message
 - ii. The time that the commit was created
 - iii. The user that created the commit
 - iv. The number of additions and deletions made in the commit
- b. Diffs
 - i. List the changed files
 - ii. Display information about the changed files
 - 1. The file state
 - a. Created
 - b. Deleted
 - c. Modified
 - 2. The number of additions and deletions
 - iii. Display and highlight the changed lines of code
- c. Comments
 - i. Display the list of comments on the commit
 - 1. The user that created the comment
 - 2. The date that the comment was created
 - 3. The reactions to the comment
 - 4. The comment body
 - ii. Implemented editing of comments made by the authenticated user
 - iii. Implement creation of comments by the authenticated user
- d. Statuses
 - i. Display the overall status for a commit
 - ii. Display the integration that created a status
 - iii. Link to the status information

6. Projects

- a. Display information about each column
 - i. The column title
 - ii. The last time that the column was updated
 - iii. The number of cards in the column
- b. Implement creation of new columns
- c. Implement deletion of columns
- d. Display cards for each column
 - i. For note cards display the note text
 - ii. For issue cards display
 - 1. The issue title

2. The issue body
 3. The issue state
 4. The issue number
 5. The number of comments on the issue
 6. The user that opened the issue
 7. The user that closed the issue (If applicable)
 8. The user(s) that are assigned to the issue
 9. The label(s) that are assigned to the issue
- e. Implement editing note cards text
 - f. Implement deleting note cards
 - g. Implement editing issue cards
 - i. Implement editing the issue (See issue section)
 - ii. Implement toggling issue state
 - iii. Implement removal of the issue card from the project
 - h. Implement creation of new cards
 - i. Implement creation of note cards with text limited to the correct length
 - ii. Implement creation of new issue cards
 1. Create a new issue
 2. Create a new card linked to the issue
 - iii. Implement creation of issue cards from pre-existing issues
 - i. Implement searching of project cards
 - i. Search text of cards and issues
 - ii. Fuzzy text matching
 - iii. Jump to and highlight the selected search item
7. Link handling
- a. Handle all links through github.com
 - b. Gracefully reject unsupported links by showing other apps which can handle the link
 - c. Handle username links by opening the user
 - d. Handle repository links and subsections
 - i. Handle repository links by opening the repository
 - ii. Handle issues links by opening the issues section of the repository
 - iii. Handle projects links by opening the projects section of the repository
 - iv. Handle commits links by opening the commits section of the repository

- e. Direct item links
 - i. Open individual issues from their numbers
 - ii. Open individual commits from their hashes
 - iii. Open individual files from their paths
 - iv. Projects
 - 1. Open individual projects from their numbers
 - 2. With a project, jump to a selected card from the id specified in a URL
- 8. Notifications
 - a. Run a background service to check for new notifications
 - b. Allow the user to disable the notification service
 - c. Only load notifications since the last time that they were loaded
 - d. Display different icons and titles for different notification types
 - i. Assign notifications, when the user is assigned to an issue
 - ii. Author notifications, when an update occurs on an item which the user created
 - iii. Comment notifications, when an update occurs on a thread which the user commented on
 - iv. Invitation notifications, when the user accepts an invitation to contribute to a repository
 - v. Manual notifications, when the user has manually subscribed to a thread
 - vi. Mention notifications, when the user has been mentioned in the content of an item
 - vii. State change notifications, when the user changes the state of a thread
 - viii. Subscribed notifications, when a change happens to a repository that the user is subscribed to
- 9. Markdown
 - a. Parse markdown to an Android usable format
 - b. Implement GitHub markdown specific features
 - i. Code blocks
 - 1. Display short code blocks as monospaced text blocks within the text body
 - 2. Display large code blocks as placeholders for a separate view
 - ii. Parse horizontal rules and create the correct spans
 - iii. Images
 - 1. Parse image links

2. Load the images asynchronously
 3. Display the images in the text body, maintaining their aspect
 4. Cache the images for later use
 5. Make image clickable, to show images in a full screen dialog
- iv. Username mentions
 1. Find username mentions in text body
 2. Ensure that the following string matches the GitHub username format
 3. If the username is valid, replace the username text with a link
- v. Issue references
 1. Find issue references in text body
 2. Ensure that the following string is numeric
 3. Replace the issue reference text with an issue link
- vi. Emojis
 1. Find emoji name strings in the text body
 2. If the emoji name is valid, replace it with the correct Unicode character
- vii. Checkboxes
 1. Find GitHub style checkboxes in the text body
 2. Replace the checkboxes with the correct Unicode characters
- viii. When displaying background colours, choose a text colour which contrasts the label colour
- ix. Display tables as placeholders for a separate view
- c. Link handling
 - i. Match the same URIs that Android will normally find
 - ii. Attempt to ignore code strings which may match a URI
- d. Format lists with the required indentation levels
 - i. Support both ordered and unordered lists
 1. Correctly indent lists
 2. Handle nested lists
 - ii. Support ordered list types
 1. Numbered lists
 2. Alphabetic lists
 3. Capitalised alphabetic lists
 4. Roman numeral lists

5. Capitalised Roman numeral lists

10. Markdown editing

- a. Implement toggling of a text editor between raw markdown and formatted markdown
- b. Add utility buttons for markdown features
 - i. Links
 - ii. Bold text
 - iii. Italic text
 - iv. Strikethrough text
 - v. List items
 - vi. Numbered list items
 - vii. Quote blocks
 - viii. Horizontal rules
- c. Add further utility buttons
 - i. Images
 1. Allow the user to choose an image from their device, take a photo, or type a link
 2. Upload the image to a hosting service
 3. Collect the image link and insert it into the text editor
 - ii. Code tags
 - iii. Checkboxes
 - iv. Emoji insertion
 1. Display a list of possible emojis
 2. Allow searching emojis by their names
 3. Insert the chosen emoji text into the text editor
 - v. Unicode insertion
 1. Display a list of possible unicode characters
 2. Allow searching characters by their names
 3. Insert the chosen characters into the text editor

Limitations

Processing power

Unlike the GitHub website, which has the benefit of running on more powerful devices, an Android app must perform the same tasks on a considerably less powerful device, without causing any inconvenience to the user.

One of the key problems will be to ensure that the user interface remains responsive.

By default, all logic will be run on the UI thread, which is usually the only thread which is allowed to modify a view, as only the original thread that created a view hierarchy can touch its views.

In order to stop the UI thread from being blocked by other operations, such as networking or markdown rendering, worker threads should be used to perform the calculations and then call back to the main thread to perform the UI update.

Screen size

Due to the limited screen size, and vertical orientation of mobile devices, some content which has been designed to be viewed on much larger landscape oriented screens may not display properly.

While not ideal, some content such as code and repository READMEs can be displayed such that they can be scrolled in two axis, as shown on the GitHub mobile website below:

Code
Issues 0
Pull requests 0
Pulse

app / src / main / java / io / fabianterhorst / isometric / sample / Main

```

package io.fabianterhorst.isometric.sample;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

import io.fabianterhorst.isometric.Color;
import io.fabianterhorst.isometric.IsometricView;
import io.fabianterhorst.isometric.Point;
import io.fabianterhorst.isometric.shapes.Prism;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        IsometricView isometricView = (IsometricView) findViewById(R.id.isometric_view);
        //Sort false improves performance but requires more memory
        //Sort true also does not support every shape
        //isometricView.setSort(false);
        //isometricView.add(new Cylinder(Point.ORIGIN, 100, 100));
        //isometricView.add(new Prism(Point.ORIGIN, new Point(100, 100, 100), new Point(100, 100, 100)));
        //isometricView.add(new Prism(new Point(100, 100, 100), new Point(200, 200, 200), new Point(200, 200, 200)));
        isometricView.add(new Octahedron(new Point(100, 100, 100)));
        isometricView.add(new Pyramid(new Point(100, 100, 100)));
        isometricView.add(new Stairs(new Point(100, 100, 100)));
        sampleTwo(isometricView);
    }
}

```

API

There are some limitations to the GitHub API.

Rate limiting

For non-authenticated requests to the GitHub API there is a rate limit of 60 requests per hour.

For content which must be loaded individually, this limit can be easily exceeded.

Requests authenticated by a user sign in have a limit of 5000 requests per hour, an amount which a user is unlikely to exceed.

In order to ensure that users do not consistently hit rate limits, they must sign in rather than being able to use the app anonymously.

Lack of endpoints

While the GitHub API provides almost all of the data required to implement the proposed solution there are some features of the GitHub website which cannot be easily duplicated.

Firstly, there is no API endpoint to find a user's pinned repositories.

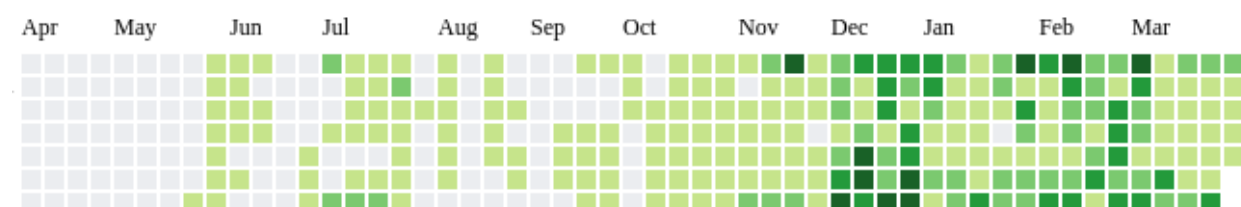
The client can still replicate this feature by pinning repositories within the app.

Secondly, the API endpoint for contributions works by repository, rather than by user.

One possibility might be to use the search API to find repositories that a user has contributed to, and load the contributions for each of them before calculating the total number of contributions made by the user.

This is an awful idea as it will fail completely if a single request fails, as well as being unacceptably slow due to the number of requests made.

Instead, the contributions image can be loaded with a single request



As the contributions image is an SVG containing the contributions information, it can be easily parsed to calculate the number of contributions made per day.

Lack of push API

The GitHub API is purely restful, all data is requested by the client and returned in a JSON format.

As there is no method for GitHub to notify the client of new notifications, the client must repeatedly poll the API for any updates to the user's notifications.

In order to reduce data usage, and to ensure that the impact on rate limiting is reduced, conditional requests can be used.

This is done by passing date-time string under the 'If-Modified-Since' header. If the data has not been modified, the API will return a 304 Not Modified code and the request will not count against the rate limit.