

Analysis

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 filesystem. 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 highly improbable.

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.

[Code](#) [Issues 4](#) [Pull requests 0](#) [Projects 4](#) [Wiki](#) [Pulse](#) [Graphs](#) [Settings](#)

Filters

LabelsMilestones

New issue

☒ Clear current search query, filters, and sorts

<input type="checkbox"/>	4 Open ✓ 27 Closed	Author ▾	Labels ▾	Projects ▾	Milestones ▾	Assignee ▾	Sort ▾
<input type="checkbox"/>	Only trigger FAB for specific screen region. enhancement #64 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Fix floating FAB being left transparent. bug #62 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Fix formatting of body text in skimmer. bug enhancement #59 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Single click on floating FAB breaks after long press. bug #57 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Implement long press scrolling for the floating FAB. enhancement #55 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Change fabUp behaviour to scroll up if fab is going up. enhancement #53 by tpb1908 was closed on 11 Dec 2016						
<input type="checkbox"/>	Make floating FAB a setting enhancement #51 by tpb1908 was closed on 10 Dec 2016						
<input type="checkbox"/>	Add acceleration to floating FAB enhancement #49 by tpb1908 was closed on 10 Dec 2016						
<input type="checkbox"/>	Floating navigation FAB enhancement						

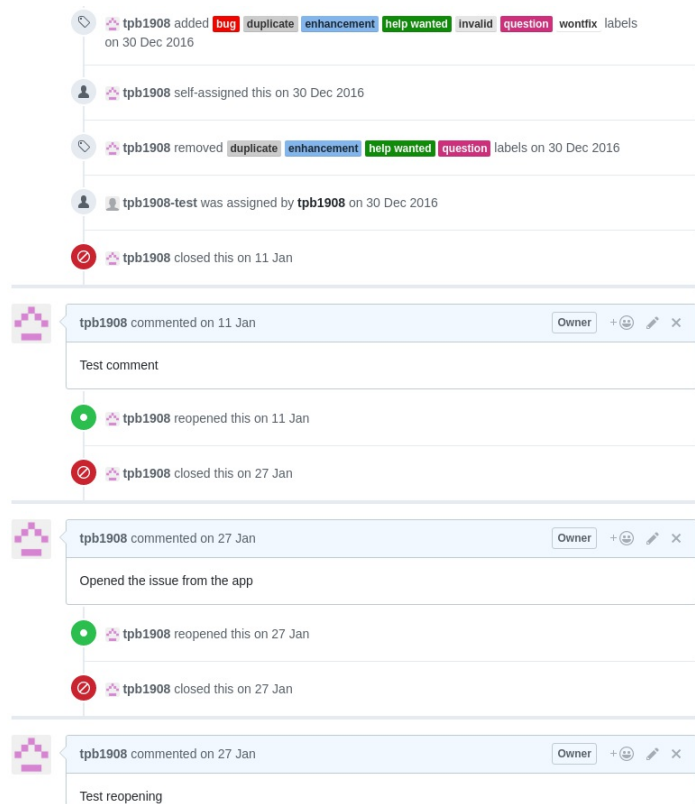
The page is divided into two sections, open and closed. An issue which is marked as open is one which has not yet been resolved and is either being investigated or to be investigated later.

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 manager 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



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.

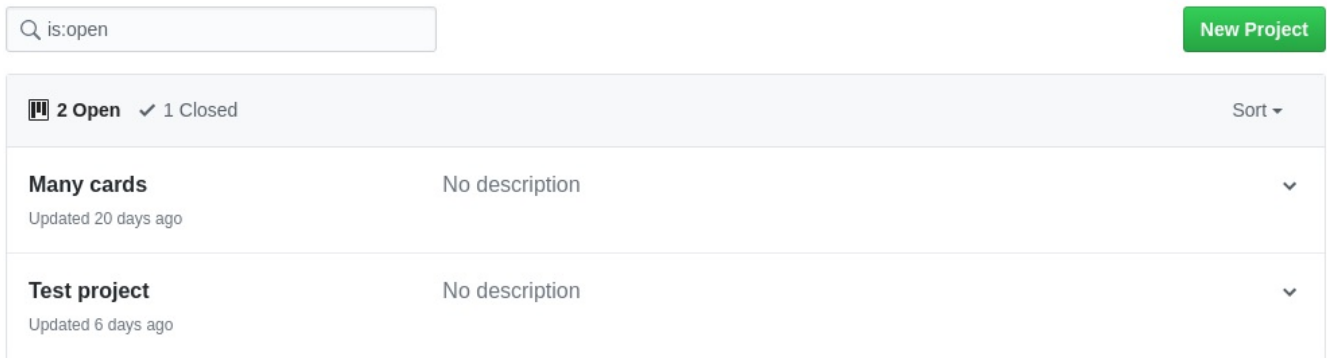
As a pull request follows the issue model it has the same comments section as an issue. The primary difference is that a pull request also references any number of commits which can be merged into the repository.

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

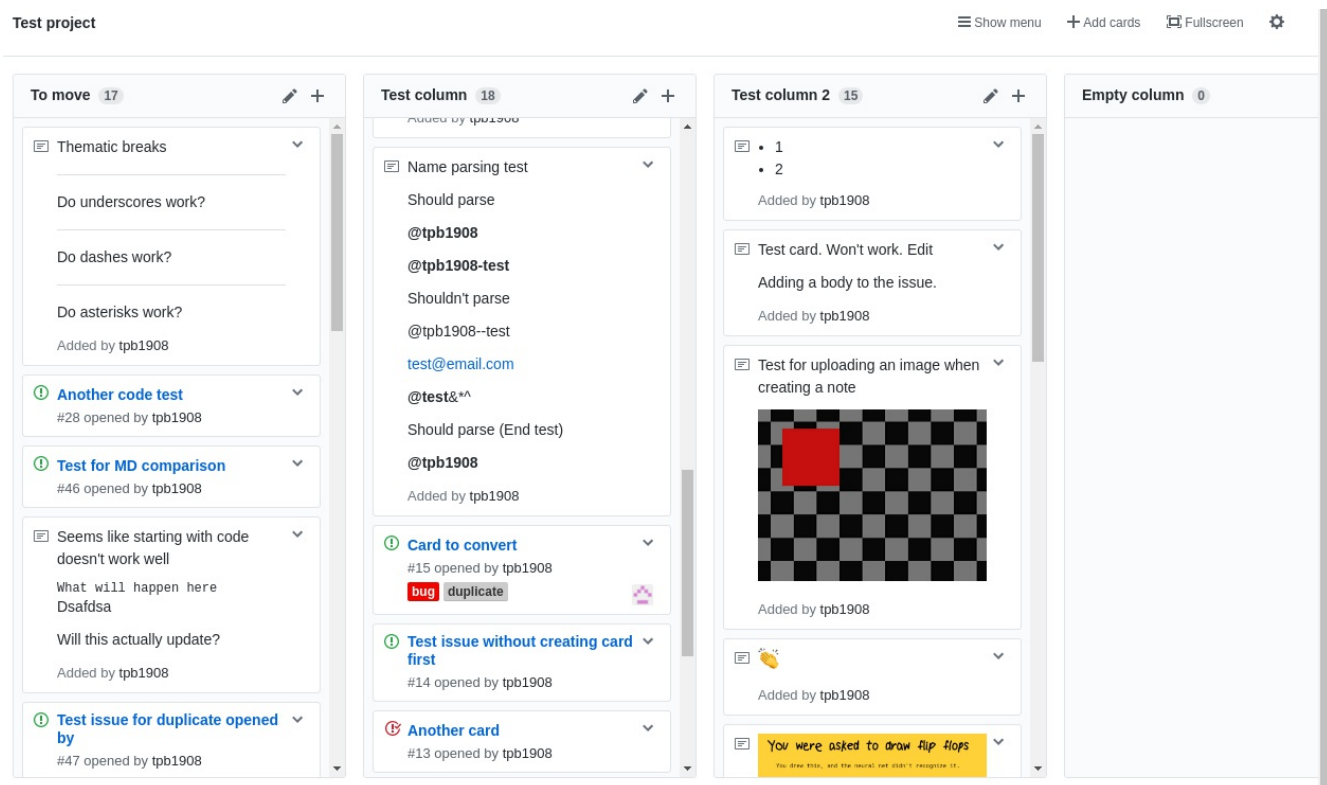
issue system. Each repository may contain multiple projects, each of which could be used to plan a particular feature or release.

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



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 users 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)`.

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 backquotes

```
``` Language
```

```
Some code
```

```
```
```

Will display the code without applying 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 color and face

`Formatted text`

gives **Formatted text**

Android Application Structure

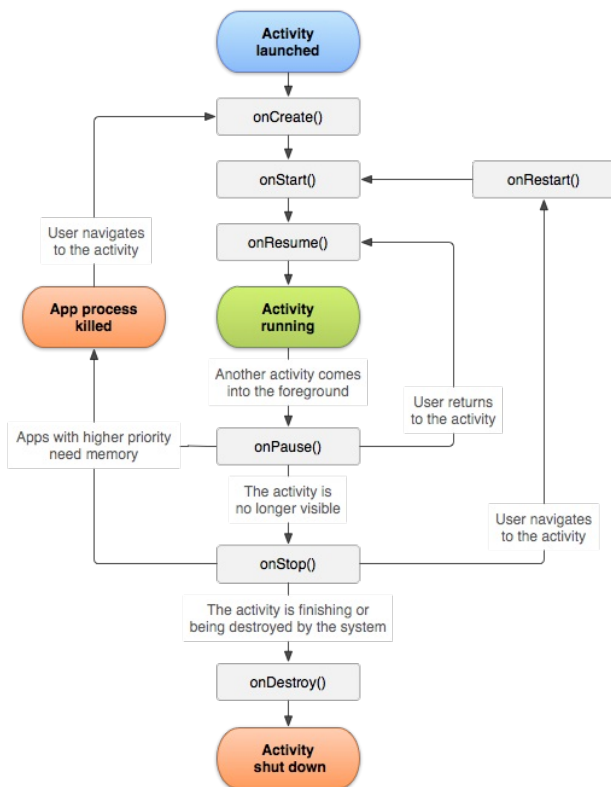
Activities

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 a 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>
```

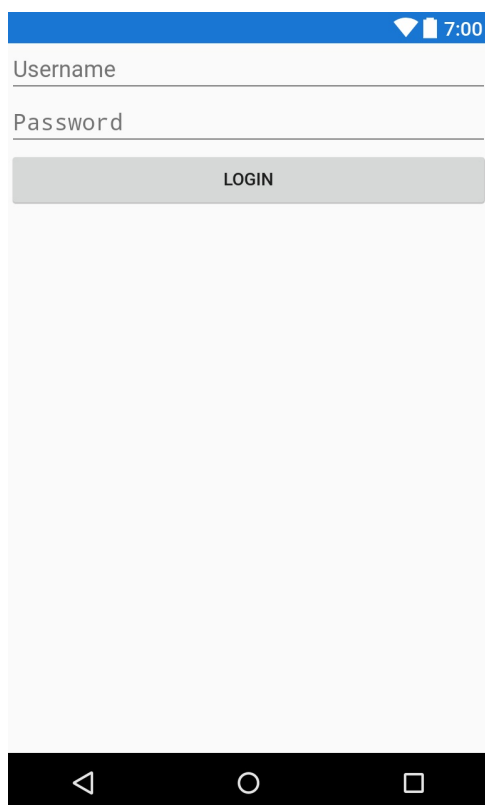
This value can then be referenced elsewhere as

```
<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 users 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 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
    }
});
```

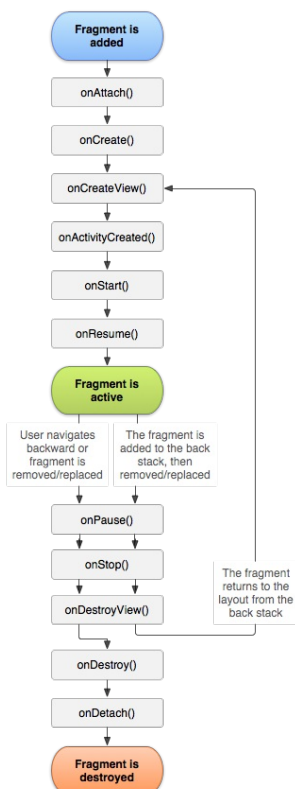
Fragments

In all cases, at least one **Activity** must be used to display an

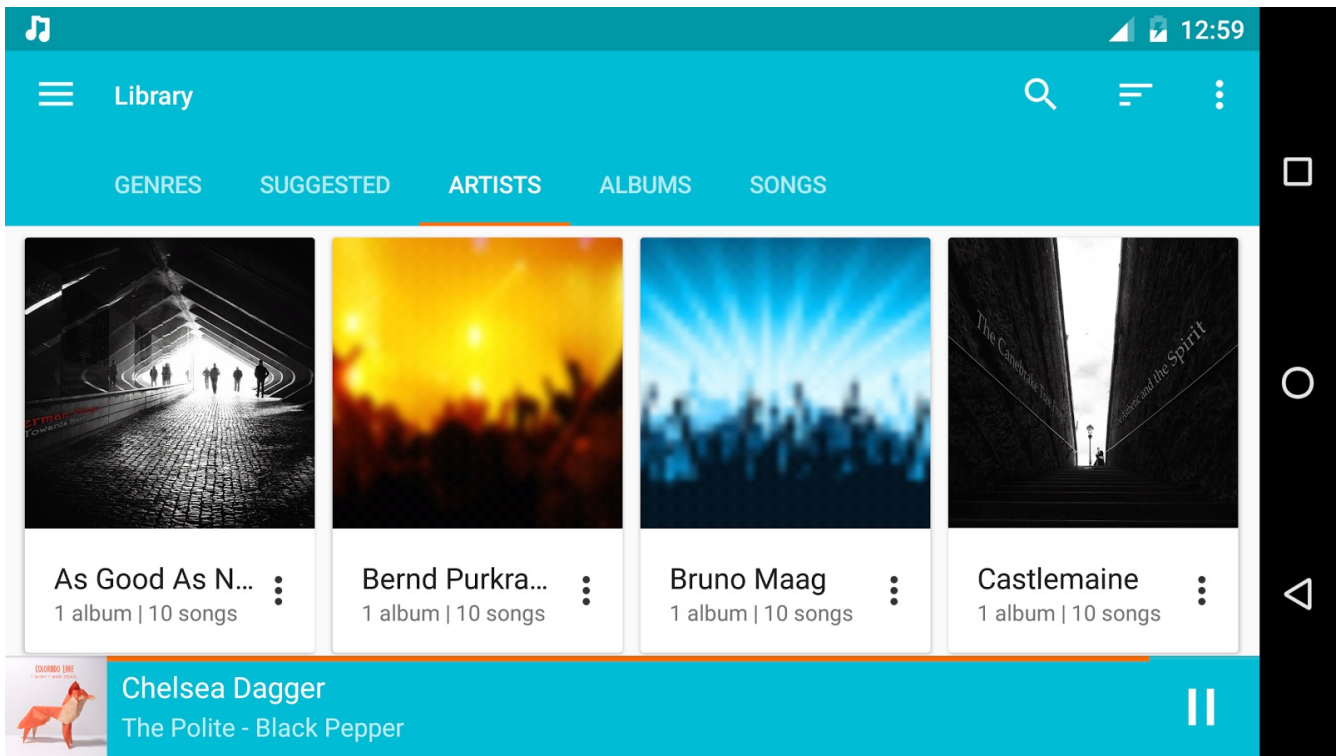
application, however it is not always necessary, and often adds unnecessary complexity to use a full **Activity** for each component of the application.

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**.



RecyclerView

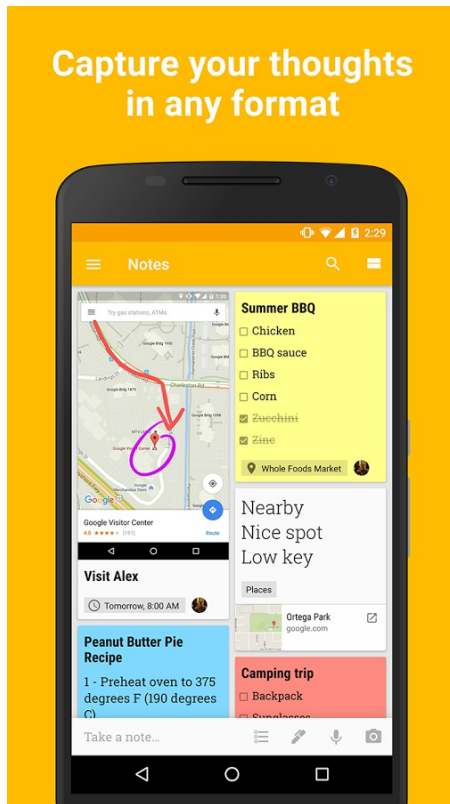
The **ListView** class was added to Android in API level 7. As its name suggests a **ListView** is used to show a set of **Views** in a scrollable list.

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 **Views** 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.

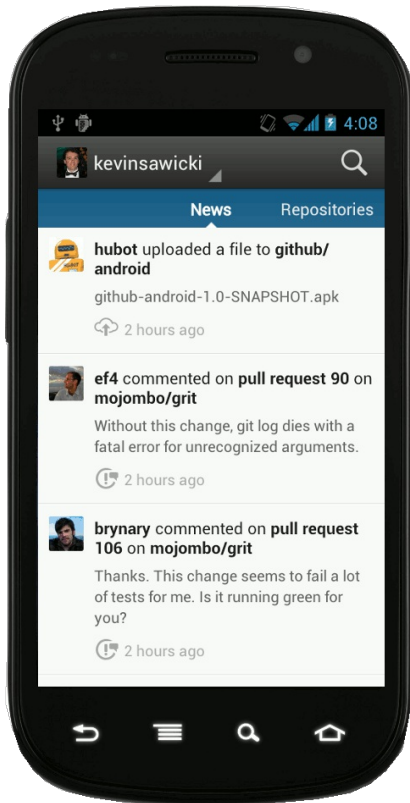
The **Log** class provides 6 static methods for logging different levels of information. Each method takes a **TAG** string which is used to identify the source of the log, and most methods have overloads for logging strings as well as **Throwable** exceptions.

- 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.

Identification of the problem

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

TODO Split frames from GIF



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 and objectives

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

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 a 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. Stars
 - i. Display the list of repositories that a user has starred
- d. Gists
 - i. Display the gists that a user has created
 - ii. Display private gists for the authenticated user
- e. Following
 - i. Display the users that a user is following
- f. Followers
 - ii. Display the users that are following a user
- g. Events
 - i. Display the events relevant to a user
 - ii. Display private events for the authenticated user

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 repositories license type
 - vi. Display the text of the license
- b. Repository files
 - i. Display the repositories file tree
 - ii. Allow viewing the file tree for different branches
 - iii. Display files within the repository
- c. README
 - i. Display the README in a suitable format for a mobile device
- d. Commits
 - i. List the commits made to a repository
 - ii. Allow selecting the branch for which to show commits

- iii. List 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 closed)
 - 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. Allow filtering the issues list
 - 1. By state
 - a. Open
 - b. Closed
 - c. Any
 - 2. By labels
 - 3. By assignee
 - iv. Allow searching the issues list
 - 1. Real time searching
 - 2. Fuzzy string comparison
 - v. Allow toggling issue state
 - vi. Allow editing issues (See issues section)
 - vii. Allow creating 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 update date
 - iii. Allow toggling project state

- iv. Allow deleting projects
 - v. Allowing editing projects and their descriptions
- #### 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
 - ix. Labels added to the issue
 - x. 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 comment body
- ii. Allow editing of comments made by the authenticated user
- iii. Allow creation of comments by the authenticated user

c. Allow 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 comment body
 - ii. Allow editing of comments made by the authenticated user
 - iii. Allow 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. Allow creation of new columns
 - c. Allow deletion of columns
 - d. Display cards for each column
 - i. For note cards display the not 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
 - 8. The user(s) that are assigned to the issue
 - 9. The label(s) assigned to the issue
 - e. Allow editing note cards
 - i. Editing note text
 - ii. Deleting note cards
 - f. Allow editing issue cards
 - i. Allowing editing the issue
 - ii. Allow toggling the issue state
 - iii. Allow removing the issue card
 - g. Allow creating new cards
 - i. Create note cards with note text limited to the correct text length
 - ii. Create issue cards
 - 1. Create a new issue with the same functionality as creating a new issue elsewhere
 - 2. Create a new card linked to the created issue
 - h. Implement searching project cards
 - i. Search text of cards and issues
 - ii. Fuzzy text searching
 - iii. Jump to and highlight selected item
- 7. Link handling
 - a. Handle all links through github.com
 - b. Gracefully reject unsupported links by showing

other apps to handle the link

- c. Handle username links by opening the user

- d. Handle repository links and subsections

- i. repository links by opening the repository

- ii. issues links by opening the issues

section of the repository

- iii. projects links by opening the projects

section of the repository

- iv. commits links by opening the commits

section of the repository

- e. Direct item links

- i. Open individual issues from their numbers

- ii. Projects

- a. Open individual projects from their numbers

- b. Within a project, jump to a selected card from the url

- iii. Open individual files from their links

- f. Open links to project file trees

8. Notifications

- a. Run a background service to periodically check for new notifications

- b. Allow the user to disable the notification service

- c. Only load notifications created since notifications were last 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. Replace longer code blocks with clickable items to display a dialog containing the code

- ii. Strikethroughs

- 1. Parse strikethroughs 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 fullscreen 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. Background colours
 1. When displaying labels, choose a text colour which contrasts the label colour
- 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. Nested lists
 - i. Format lists with the required indentation levels
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
 - c. Add further utility buttons
 - i. Images
 1. Allow the user to choose an image from their device or take a new image
 2. Upload the image to a hosting service
 3. Collect the image link and insert it into the text editor
 - ii. Code tags

- iii. Emoji insertion
 1. Display a list of possible emojis
 2. Allow searching emojis by their names
 3. Insert the correct emoji text into the text editor
- iv. Unicode characters
 1. Display a list of possible unicode characters
 2. Allow searching characters by their names
 3. Insert the 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.

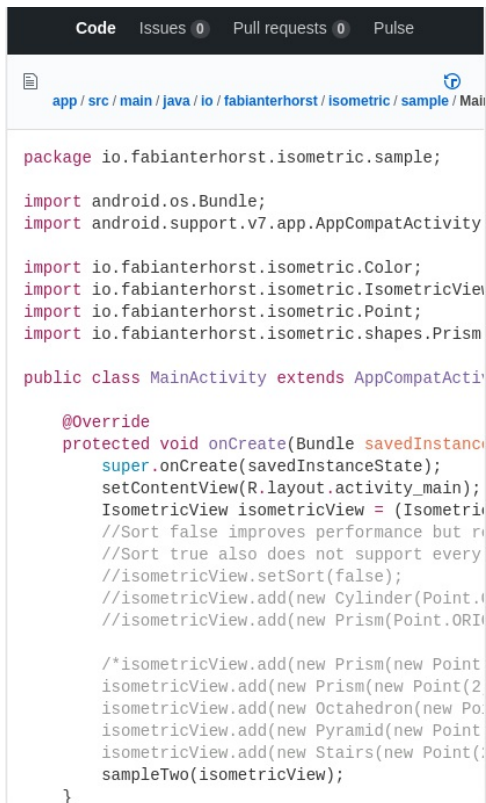
In order to stop the UI thread from being blocked by other operations, such as networking or markdown rendering, worker threads should be used to perform the calculations and then call back to the main thread to perform the UI update.

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:

A screenshot of the GitHub mobile website interface. At the top, there's a dark navigation bar with 'Code', 'Issues 0', 'Pull requests 0', and 'Pulse'. Below this is a breadcrumb trail: 'app / src / main / java / io / fabianterhorst / isometric / sample / MainActivity'. The main content area displays Java code for 'MainActivity'. The code includes package declarations, imports for Android and custom classes, and an '@Override' method 'onCreate' that initializes an 'IsometricView' and adds various 3D shapes like Cylinder, Prism, Octahedron, Pyramid, and Stairs. The code is syntax-highlighted with colors like purple for keywords, blue for class names, and green for comments.

```
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 API level 24
        //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, 100, 100, 100));

        /*isometricView.add(new Prism(new Point(0, 0, 0), new Point(100, 100, 100)));
        isometricView.add(new Prism(new Point(200, 200, 200), new Point(300, 300, 300)));
        isometricView.add(new Octahedron(new Point(400, 400, 400), 100));
        isometricView.add(new Pyramid(new Point(500, 500, 500), 100));
        isometricView.add(new Stairs(new Point(600, 600, 600), 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.

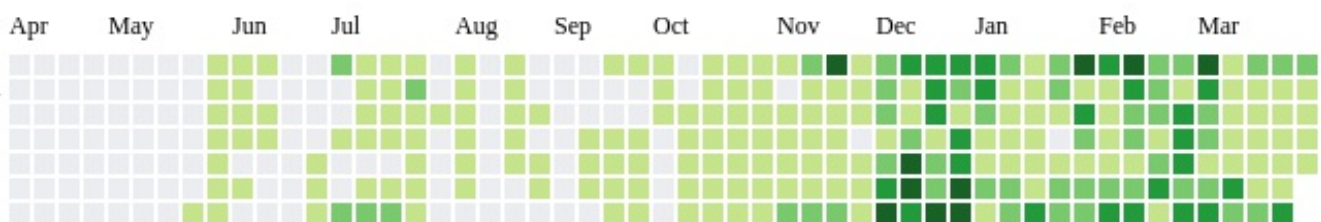
Lack of endpoints

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.

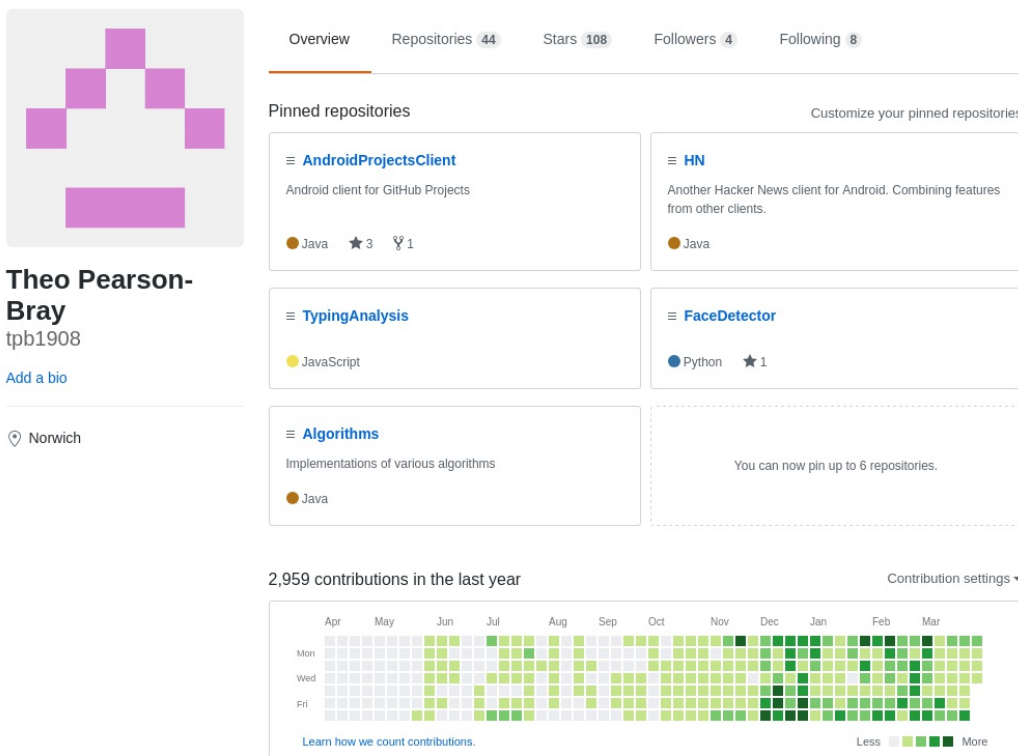
In order to reduce data usage, and to ensure that the impact on rate limiting is reduced, conditional requests can be used. This is done by passing date-time string under the 'If-Modified-Since' header. If the data has not been modified, the API will return a 304 Not Modified code and the request will not count against the rate limit.

Proposed design

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

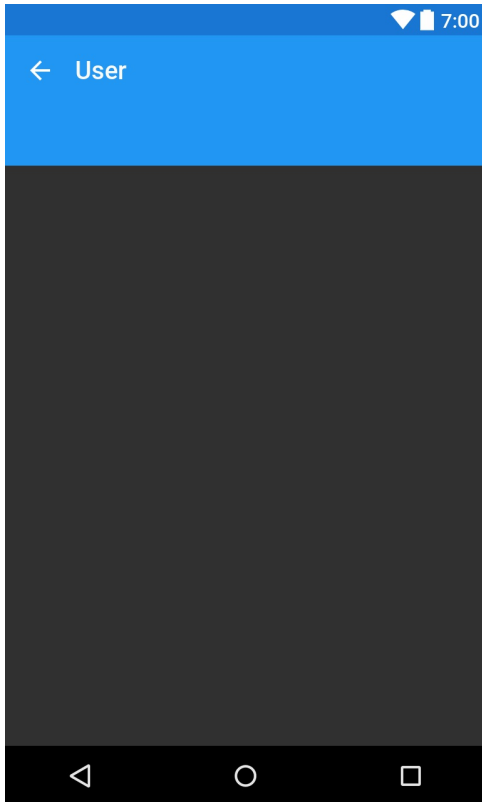
In order to implement the functionality listed above the app requires multiple activities to mimic the pages on the GitHub website.

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



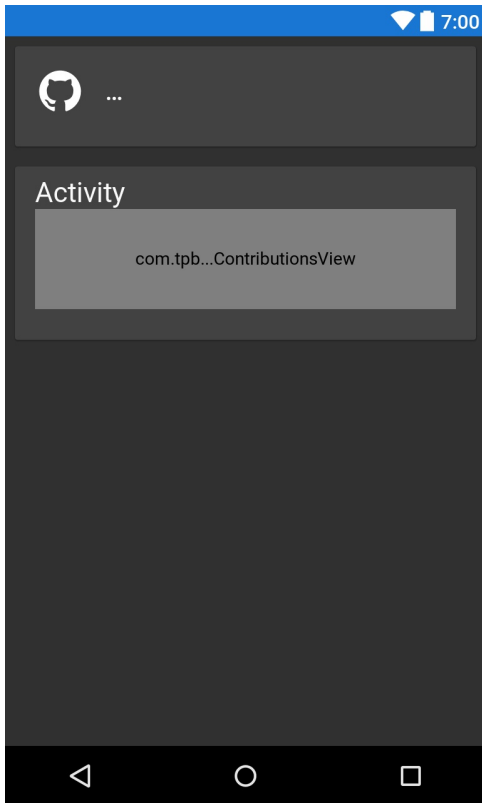
This page can be easily adapted to a mobile layout as it is already paginated.

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.



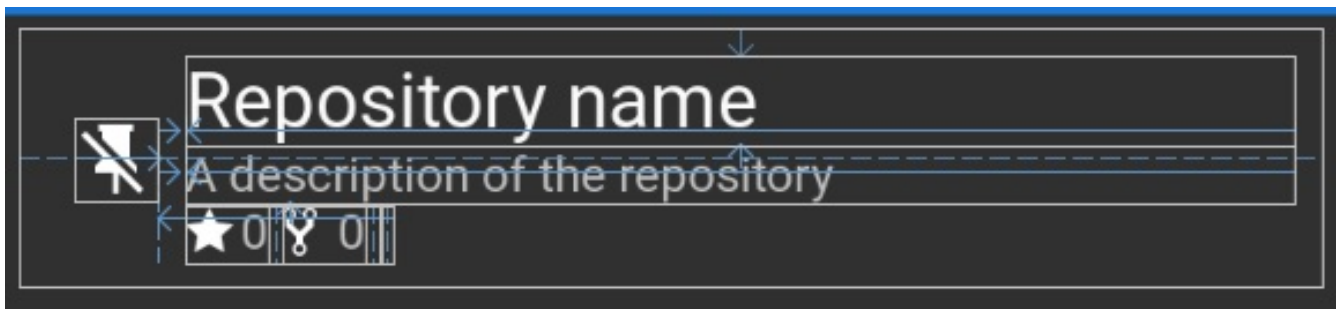
Within the viewpager for this Activity, there will be 6 pages.

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

Design

Data structures and sources

Almost all of the data used in the application is acquired through the GitHub API

Authentication

Basic authentication

Requests can be authenticated by sending the user's username and password in the request header, however this is not an acceptable method of authentication for an Android client.

The first problem is that, while HTTPS requests are encrypted, the request itself is likely to be logged by the Android system, and may pass through another service if the user has a VPN active.

The second problem is that the user's account may require more than a password to authenticate. GitHub supports two factor authentication. If the user has activated two factor authentication, the two factor pin must be sent with each request. This is not a usable experience as the pin changes each minute.

OAuth2 authentication

OAuth2 authentication allows applications to request authorization to a user's account without having access to their password. This method also allows tokens to be limited to specific types of data, and can be revoked by the user.

In order to use the OAuth API, the application must be registered with GitHub.

Register a new OAuth application

Application name

Something users will recognize and trust

Homepage URL

The full URL to your application homepage

Application description

This is displayed to all potential users of your application

Authorization callback URL

Your application's callback URL. Read our [OAuth documentation](#) for more information.

Register application

Cancel

The application is registered with information recognisable to the user to ensure their trust when authorizing the application. The callback URL is the URL which GitHub redirects to once the authorization is complete.

Web authentication flow

1. Redirect users to request access

Display the webpage <https://github.com/login/oauth/authorize>

In order to successfully authenticate we must also pass parameters with the request. The only required parameter is the client id which was received when the application was registered.

The scope parameter is used to specify what level of access is required to the user's account.

2. Redirect

If the user signs in and accepts the authorization request, GitHub

redirects back to your site with a temporary parameter.

The `code` parameter has a limited timeframe to be exchanged for an authorization token.

This is done by posting to

https://github.com/login/oauth/access_token.

The post request must have three parameters:

1. The client id, which must match that provided when loading the authorization page
2. The client secret
3. The code received in

If the request is successful the response will be a string containing the access token in the form:

`access_token=some_base_64_string&scope=the_scope_requested_in_step_1`.

3. Once the access token has been received it can be used for authorization by including the authorization header with each request.

Implementation of the authorization flow

In order to avoid duplication of values used throughout the GitHub API, I have used a single abstract class to contain the headers and path keys used throughout the project

```
package com.tpb.github.data;

import android.content.Context;
import android.support.annotation.Nullable;
import android.support.annotation.StringRes;
```

```

import com.androidnetworking.error.ANError;
import com.tpb.github.R;
import com.tpb.github.data.auth.GitHubSession;

import java.util.HashMap;

/**
 * Created by theo on 18/12/16.
 */

public abstract class APIHandler {
    static final String TAG =
APIHandler.class.getSimpleName();

    protected static final String GIT_BASE =
"https://api.github.com";
    private static final String ACCEPT_HEADER_KEY =
"Accept";
    private static final String ACCEPT_HEADER =
"application/vnd.github.v3+json";
    private static final String
ORGANIZATIONS_PREVIEW_ACCEPT_HEADER =
"application/vnd.github.korra-preview";
    private static final String
PROJECTS_PREVIEW_ACCEPT_HEADER =
"application/vnd.github.inertia-preview+json";
    private static final String
REPO_LICENSE_PREVIEW_ACCEPT_HEADER =
"application/vnd.github.drax-preview+json";
    private static final String
PAGES_PREVIEW_ACCEPT_HEADER =
"application/vnd.github.mister-fantastic-
preview+json";
    private static final String
AUTHORIZATION_HEADER_KEY = "Authorization";
    private static final String
AUTHORIZATION_TOKEN_FORMAT = "token %1$s";
    private static GitHubSession mSession;

```

```

        protected static final HashMap<String, String>
API_AUTH_HEADERS = new HashMap<>();
        static final HashMap<String, String>
PROJECTS_API_AUTH_HEADERS = new HashMap<>();
        static final HashMap<String, String>
ORGANIZATIONS_API_AUTH_HEADERS = new HashMap<>();
        static final HashMap<String, String>
LICENSES_API_AUTH_HEADERS = new HashMap<>();
        static final HashMap<String, String>
PAGES_API_AUTH_HEADERS = new HashMap<>();

        protected static final String SEGMENT_USER =
"/user";
        static final String SEGMENT_USERS = "/users";
        static final String SEGMENT_REPOS = "/repos";
        static final String SEGMENT_README = "/readme";
        static final String SEGMENT_COLLABORATORS =
"/collaborators";
        static final String SEGMENT_LABELS = "/labels";
        static final String SEGMENT_PROJECTS =
"/projects";
        static final String SEGMENT_COLUMNS = "/columns";
        static final String SEGMENT_ISSUES = "/issues";
        static final String SEGMENT_PERMISSION =
"/permission";
        static final String SEGMENT_CARDS = "/cards";
        static final String SEGMENT_MOVES = "/moves";
        static final String SEGMENT_COMMENTS =
"/comments";
        static final String SEGMENT_EVENTS = "/events";
        static final String SEGMENT_STARRED = "/starred";
        static final String SEGMENT_SUBSCRIPTION =
"/subscription";
        static final String SEGMENT_MILESTONES =
"/milestones";
        static final String SEGMENT_GISTS = "/gists";
        static final String SEGMENT_FOLLOWING =
"/following";

```

```

        static final String SEGMENT_FOLLOWERS =
"/followers";
        static final String SEGMENT_COMMITS = "/commits";
        static final String SEGMENT_NOTIFICATIONS =
"/notifications";

        protected APIHandler(Context context) {
            if(mSession == null) {
                mSession =
GitHubSession.getSession(context);
                initHeaders();
            }
        }

        protected final void initHeaders() {
            final String accessToken =
mSession.getAccessToken();

API_AUTH_HEADERS.put(AUTHORIZATION_HEADER_KEY,

String.format(AUTHORIZATION_TOKEN_FORMAT,
accessToken)
                );
            API_AUTH_HEADERS.put(ACCEPT_HEADER_KEY,
ACCEPT_HEADER);

ORGANIZATIONS_API_AUTH_HEADERS.put(AUTHORIZATION_HEAD
ER_KEY,

String.format(AUTHORIZATION_TOKEN_FORMAT,
accessToken)
                );

ORGANIZATIONS_API_AUTH_HEADERS.put(ACCEPT_HEADER_KEY,
ORGANIZATIONS_PREVIEW_ACCEPT_HEADER);

PROJECTS_API_API_AUTH_HEADERS.put(AUTHORIZATION_HEADE

```

```

R_KEY,

String.format(AUTHORIZATION_TOKEN_FORMAT,
accessToken)
    );

PROJECTS_API_API_AUTH_HEADERS.put(ACCEPT_HEADER_KEY,
PROJECTS_PREVIEW_ACCEPT_HEADER);

LICENSES_API_API_AUTH_HEADERS.put(AUTHORIZATION_HEADE
R_KEY,

String.format(AUTHORIZATION_TOKEN_FORMAT,
accessToken)
    );

LICENSES_API_API_AUTH_HEADERS.put(ACCEPT_HEADER_KEY,
REPO_LICENSE_PREVIEW_ACCEPT_HEADER);

PAGES_API_API_AUTH_HEADERS.put(AUTHORIZATION_HEADER_K
EY,

String.format(AUTHORIZATION_TOKEN_FORMAT,
accessToken)
    );

PAGES_API_API_AUTH_HEADERS.put(ACCEPT_HEADER_KEY,
PAGES_PREVIEW_ACCEPT_HEADER);
    }

    private static final String CONNECTION_ERROR =
"connectionError";

    public static final int HTTP_OK_200 = 200; //OK

    public static final String
HTTP_REDIRECT_NEW_LOCATION = "Location";

```

```

    public static final int HTTP_301_REDIRECTED =
301; //Should redirect through the value in location

    public static final int
HTTP_302_TEMPORARY_REDIRECT = 302; //Redirect for
this request only
    public static final int
HTTP_307_TEMPORARY_REDIRECT = 307; //Same as above

    private static final int HTTP_BAD_REQUEST_400 =
400; //Bad request problems parsing JSON

    public static final String KEY_MESSAGE =
"message";
    private static final String
MESSAGE_BAD_CREDENTIALS = "Bad credentials";
    private static final int HTTP_UNAUTHORIZED_401 =
401; //Login required, account locked, permission
error

    private static final String
MESSAGE_MAX_LOGIN_ATTEMPTS = "Maximum number of login
attempts exceeded.";

    public static final String
KEY_HEADER_RATE_LIMIT_RESET = "X-RateLimit-Reset";
    private static final String
MESSAGE_RATE_LIMIT_START = "API rate limit exceeded";
    private static final String MESSAGE_ABUSE_LIMIT =
"You have triggered an abuse detection mechanism";
    private static final int HTTP_FORBIDDEN_403 =
403; //Forbidden server locked or other reasons

    private static final int HTTP_NOT_FOUND_404 =
404;

    private static final int HTTP_NOT_ALLOWED_405 =
405; //Not allowed (managed server)

```

```

    private static final int HTTP_409 = 409;
    //Returned when loading commits for empty repo

    private static final int HTTP_419 = 419; //This
    function can only be executed with an CL-account

    public static final String
    ERROR_MESSAGE_UNPROCESSABLE = "Validation Failed";
    public static final String
    ERROR_MESSAGE_VALIDATION_MISSING = "missing";
    public static final String
    ERROR_MESSAGE_VALIDATION_MISSING_FIELD =
    "missing_field";
    public static final String
    ERROR_MESSAGE_VALIDATION_INVALID = "invalid";
    public static final String
    ERROR_MESSAGE_VALIDATION_ALREADY_EXISTS =
    "already_exists";
    private static final String
    ERROR_MESSAGE_EMPTY_REPOSITORY = "Git Repository is
    empty.";
    private static final int HTTP_UNPROCESSABLE_422 =
    422; // Validation failed

    //600 codes are server codes
    https://github.com/GleSYS/API/wiki/API-Error-
    codes#6xx---server

    //700 codes are ip errors
    https://github.com/GleSYS/API/wiki/API-Error-
    codes#7xx---ip

    //800 codes are archive codes
    https://github.com/GleSYS/API/wiki/API-Error-
    codes#8xx---archive

    //900 domain
    https://github.com/GleSYS/API/wiki/API-Error-
    codes#9xx---domain

```



```
//1000 email  
https://github.com/GleSYS/API/wiki/API-Error-codes#10xx---email
```

```
//1100 livechat  
https://github.com/GleSYS/API/wiki/API-Error-codes#11xx---livechat
```

```
//1200 invoice  
https://github.com/GleSYS/API/wiki/API-Error-codes#11xx---livechat
```

```
//1300 glera  
https://github.com/GleSYS/API/wiki/API-Error-codes#13xx---glera
```

```
//1400 transaction  
https://github.com/GleSYS/API/wiki/API-Error-codes#14xx---transaction
```

```
//1500 vpn  
https://github.com/GleSYS/API/wiki/API-Error-codes#15xx---vpn
```

```
public static final int GIT_LOGIN_FAILED_1601 =  
1601; //Login failed
```

```
public static final int GIT_LOGIN_FAILED_1602 =  
1602; //Login failed unknown
```

```
public static final int  
GIT_GOOGLE_AUTHENTICATOR_OTP_REQUIRED_1603 = 1603;  
//Google auth error
```

```
public static final int GIT_YUBIKEY_1604 = 1604;  
//Yubikey OTP required
```

```
public static final int GIT_NOT_LOGGED_IN_1605 =
```

```

1605; //Not logged in as user

//1700 invite
https://github.com/GleSYS/API/wiki/API-Error-
codes#17xx---invite

//1800 test account
https://github.com/GleSYS/API/wiki/API-Error-
codes#18xx---test-account

//1900 network
https://github.com/GleSYS/API/wiki/API-Error-
codes#19xx---network

static APIError parseError(ANError error) {
    APIError apiError;

    if(CONNECTION_ERROR.equals(error.getErrorDetail())) {
        apiError = APIError.NO_CONNECTION;
    } else {
        switch(error.getErrorCode()) {
            case HTTP_BAD_REQUEST_400:
                apiError = APIError.BAD_REQUEST;
                break;
            case HTTP_UNAUTHORIZED_401:
                apiError = APIError.UNAUTHORIZED;
                if(error.getErrorBody() != null)
{
                    if(error.getErrorBody().contains(MESSAGE_BAD_CREDENTIALS)) {
                        apiError =
APIError.BAD_CREDENTIALS;
                    } else
                    if(error.getErrorBody().contains(MESSAGE_MAX_LOGIN_ATTEMPTS)) {
                        apiError =
APIError.MAX_LOGIN_ATTEMPTS;
                    }
                }
            }
        }
    }
}

```

```

        }
        break;
    case HTTP_FORBIDDEN_403:
        apiError = APIError.FORBIDDEN;
        if(error.getErrorBody() != null)
    {

        if(error.getErrorBody().contains(MESSAGE_RATE_LIMIT_S
TART)) {

            apiError =
APIError.RATE_LIMIT;
        } else
        if(error.getErrorBody().contains(MESSAGE_ABUSE_LIMIT)
) {

            apiError =
APIError.ABUSE_LIMIT;
        }
    }
    break;
    case HTTP_NOT_ALLOWED_405:
        apiError = APIError.NOT_ALLOWED;
        break;
    case HTTP_NOT_FOUND_404:
        apiError = APIError.NOT_FOUND;
        break;
    case HTTP_UNPROCESSABLE_422:
        apiError =
APIError.UNPROCESSABLE;
        break;
    case HTTP_409:
        if(error.getErrorBody() != null
&& error.getErrorBody().contains(
ERROR_MESSAGE_EMPTY_REPOSITORY)) {
            apiError =
APIError.EMPTY_REPOSITORY;
            break;
        }
        default:

```

```

        apiError = APIError.UNKNOWN;
    }
}
apiError.error = error;
return apiError;
}

public enum APIError {
    NO_CONNECTION(R.string.error_no_connection),
    UNAUTHORIZED(R.string.error_unauthorized),
    FORBIDDEN(R.string.error_forbidden),
    NOT_FOUND(R.string.error_not_found),
    UNKNOWN(R.string.error_unknown),
    RATE_LIMIT(R.string.error_rate_limit),
    ABUSE_LIMIT(R.string.error_abuse_limit),

    MAX_LOGIN_ATTEMPTS(R.string.error_max_login_attempts)
    ,
    UNPROCESSABLE(R.string.error_unprocessable),

    BAD_CREDENTIALS(R.string.error_bad_credentials),
    NOT_ALLOWED(R.string.error_not_allowed),
    BAD_REQUEST(R.string.error_bad_request),

    EMPTY_REPOSITORY(R.string.error_empty_repository);

    @StringRes
    public final int resId;

    @Nullable ANError error;

    APIError(@StringRes int resId) {
        this.resId = resId;
    }
}
}

```

The **APIHandler** class is mostly static constants:

- **GIT_BASE** - The base URL for all GitHub API requests
- **ACCEPT_HEADER_KEY** - The key for the content type header
- **ACCEPT_HEADER** - The default content type, for JSON results
- **ORGANIZATIONS_PREVIEW_ACCEPT_HEADER** - A content type header required for some features of the API, specifically requesting collaborators on a repository
- **PROJECTS_PREVIEW_API_HEADER** - A content type header required for requesting information related to projects
- **REPO_LICENSE_PREVIEW_API_HEADER** - A content type header required to load information about a repositories license when loading the repository
- **PAGES_PREVIEW_ACCEPT_HEADER** - A content type header required to load information about a repositories pages when loading the repository
- **AUTHORIZATION_HEADER_KEY** - A header key for providing the authorization token
- **AUTHORIZATION_TOKEN_FORMAT** - A format string used when inserting the authorization key into a header

Next we have the headers themselves. As headers are key value pairs, they are represented as string to string maps.

We then have the **SEGMENT_** constants. These are segments of the API paths which are used across numerous different API requests.

The **APIHandler** constructor checks if the single instance of **GitHubSession** is null, and if so access the singleton session instance before initialising the headers. Next each header map is initialised with the authorization token header, and their own respective accept headers.

The class which actually stores the authorization information is **GitHubSession**, which was used once above in **APIHandler**.

```

package com.tpb.github.data.auth;

import android.content.Context;
import android.content.SharedPreferences;
import android.support.annotation.NonNull;

import com.tpb.github.data.models.User;

import org.json.JSONException;
import org.json.JSONObject;

public class GitHubSession {
    private static final String TAG =
GitHubSession.class.getSimpleName();

    private static GitHubSession session;
    private final SharedPreferences prefs;

    private static final String SHARED =
"GitHub_Preferences";
    private static final String API_LOGIN =
"username";
    private static final String API_ID = "id";
    private static final String API_ACCESS_TOKEN =
"access_token";
    private static final String INFO_USER =
"user_json";

    private GitHubSession(Context context) {
        prefs = context.getSharedPreferences(SHARED,
Context.MODE_PRIVATE);
    }

    public static GitHubSession getSession(Context
context) {
        if(session == null) session = new
GitHubSession(context);
        return session;
    }

```

```

    }

    void storeUser(JSONObject json) {
        final SharedPreferences.Editor editor =
prefs.edit();
        editor.putString(INFO_USER, json.toString());
        final User user = User.parse(json);
        editor.putInt(API_ID, user.getId());
        editor.putString(API_LOGIN, user.getLogin());
        editor.apply();
    }

    void storeAccessToken(@NonNull String
accessToken) {
        final SharedPreferences.Editor editor =
prefs.edit();
        editor.putString(API_ACCESS_TOKEN,
accessToken);
        editor.apply();
    }

    public User getUser() {
        try {
            final JSONObject obj = new
JSONObject(prefs.getString(INFO_USER, ""));
            return User.parse(obj);
        } catch(JSONException jse) {
            return null;
        }
    }

    public String getUserLogin() {
        return prefs.getString(API_LOGIN, null);
    }

    public String getAccessToken() {
        return prefs.getString(API_ACCESS_TOKEN,
null);
    }
}

```

```
        public boolean hasAccessToken() {  
            return getAccessToken() != null;  
        }  
  
    }
```

GitHubSession is a singleton class which saves and loads the user credentials and authorization token to and from shared preferences.

The private constructor is used to initialise the SharedPreferences instance, this either opens the pre-existing map or creates a new one if it does not exist.

When the user authorizes the app the access token is stored with **storeAccessToken(@NonNull String accessToken)**.

Once we have an authorization token, we can load the user's data and store it for later use.

The **LoginActivity** consists of two layouts, only one of which is visible at a time.

The first layout is a **WebView** which is used to display the user authentication page, and the second is a layout to display the user's information once they have signed in.

```
#include "app/src/main/java/com/tpb/projects/login/LoginActivity.java"
```

The **LoginActivity** binds four views.

1. The **WebView** which displays the login webpage
2. The **CardView** which holds the **WebView** and user layout
3. The spinning **ProgressBar** which is display to indicate progress while the **WebView** is loading or the user's information is being loaded.
4. The **LinearLayout** which will be filled with views showing the

user's information

In the `onCreate` method the `WebView` is set up not to allow scrolling, to enable JavaScript, and to use a custom client to override page loading.

```
mWebView.setVerticalScrollBarEnabled(false);
mWebView.setHorizontalScrollBarEnabled(false);
mWebView.setWebViewClient(new
OAuthWebViewClient(OAuthHandler.getListener()));
mWebView.getSettings().setJavaScriptEnabled(true);
mWebView.loadUrl(OAuthHandler.getAuthUrl());
mWebView.setLayoutParams(FILL);
```

The `OAuthWebViewClient` extends `WebViewClient` and is used to capture the code once the user has logged in, as well as ensuring that the user only navigates through the pages required to log in.

The method `onPageStarted(WebView view, String url, Bitmap favicon)` is called whenever a page load begins. The client checks if the url contains `?code=`, and if so, passes the segment after that point to the `OAuthHandler` which then requests the authorization token.

```
package com.tpb.github.data.auth;

/**
 * Created by theo on 15/12/16.
 */

import android.content.Context;
import android.util.Log;

import com.androidnetworking.AndroidNetworking;
import com.androidnetworking.error.ANError;
import
com.androidnetworking.interfaces.JSONObjectRequestLis
```

```

tener;
import
com.androidnetworking.interfaces.StringRequestListene
r;
import com.tpb.github.data.APIHandler;
import com.tpb.github.data.models.User;

import org.json.JSONObject;

public class OAuthHandler extends APIHandler {
    private static final String TAG =
OAuthHandler.class.getSimpleName();

    private final GitHubSession mSession;
    private final OAuthAuthenticationListener
mListener;
    private final String mAuthUrl;
    private final String mTokenUrl;
    private String mAccessToken;

    private static final String AUTH_URL =
"https://github.com/login/oauth/authorize?";
    private static final String TOKEN_URL =
"https://github.com/login/oauth/access_token?";
    private static final String SCOPE = "user
public_repo repo gist";

    private static final String TOKEN_URL_FORMAT =
TOKEN_URL +
"client_id=%1$s&client_secret=%2$s&redirect_uri=%3$s"
;
    private static final String AUTH_URL_FORMAT =
AUTH_URL +
"client_id=%1$s&scope=%2$s&redirect_uri=%3$s";

    public OAuthHandler(Context context, String
clientId, String clientSecret,
                        String callbackUrl,
                        OAuthAuthenticationListener

```

```

listener) {
    super(context);
    mSession = GitHubSession.getSession(context);
    mTokenUrl = String.format(TOKEN_URL_FORMAT,
clientId, clientSecret, callbackUrl);
    mAuthUrl = String.format(AUTH_URL_FORMAT,
clientId, SCOPE, callbackUrl);
    mListener = listener;
}

    public void getAccessToken(final String code) {
        AndroidNetworking.get(mTokenUrl + "&code=" +
code)

                .build()
                .getAsString(new
StringRequestListener() {
                    @Override
                    public void
onResponse(String response) {
                        mAccessToken =
response.substring(
response.indexOf("access_token=") + 13,
response.indexOf("&scope")
                );

mSession.storeAccessToken(mAccessToken);
                initHeaders();

mListener.onSuccess();

                fetchUser();
            }

            @Override
            public void
onError(ANError anError) {
mListener.onFail(anError.getErrorDetail());

```

```

        }
    });
}

private void fetchUser() {
    AndroidNetworking.get(GIT_BASE +
SEGMENT_USER)

    .addHeaders(API_AUTH_HEADERS)
        .build()
        .getAsJSONObject(new
JSONObjectRequestListener() {
            @Override
            public void
onResponse(JSONObject response) {

mSession.storeUser(response);

mListener.userLoaded(mSession.getUser());
            }

            @Override
            public void
onError(ANError anError) {

mListener.onFail(anError.getErrorDetail());
                Log.e(TAG, "onError:
" + anError.getErrorDetail());
            }
        });
}

public String getAuthUrl() {
    return mAuthUrl;
}

public interface OAuthAuthenticationListener {
    void onSuccess();
}

```

```
        void onFail(String error);

        void userLoaded(User user);

    }
}
```

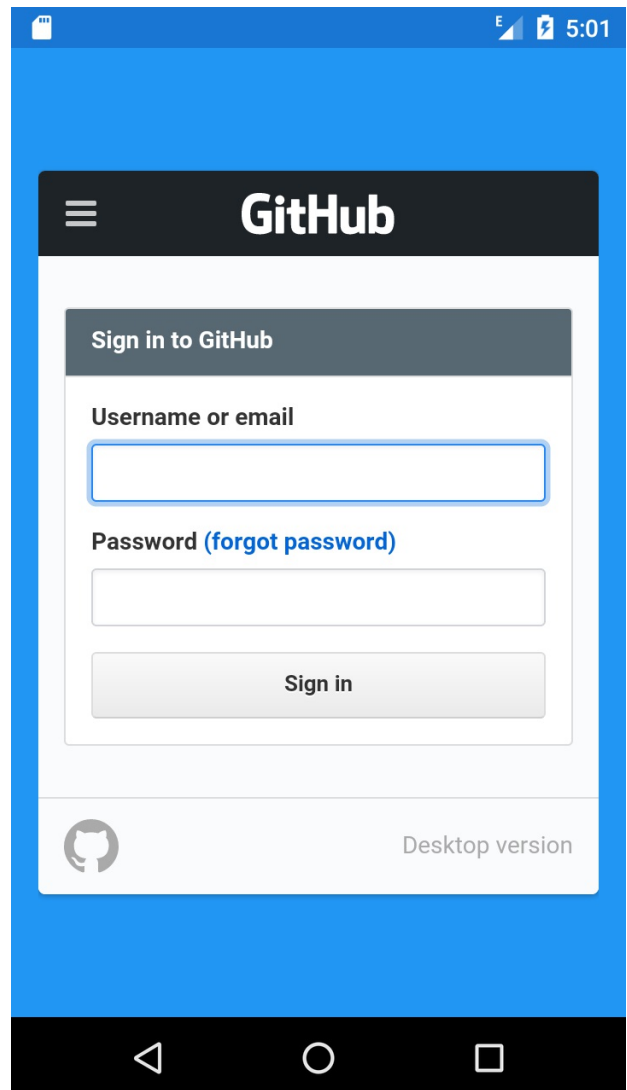
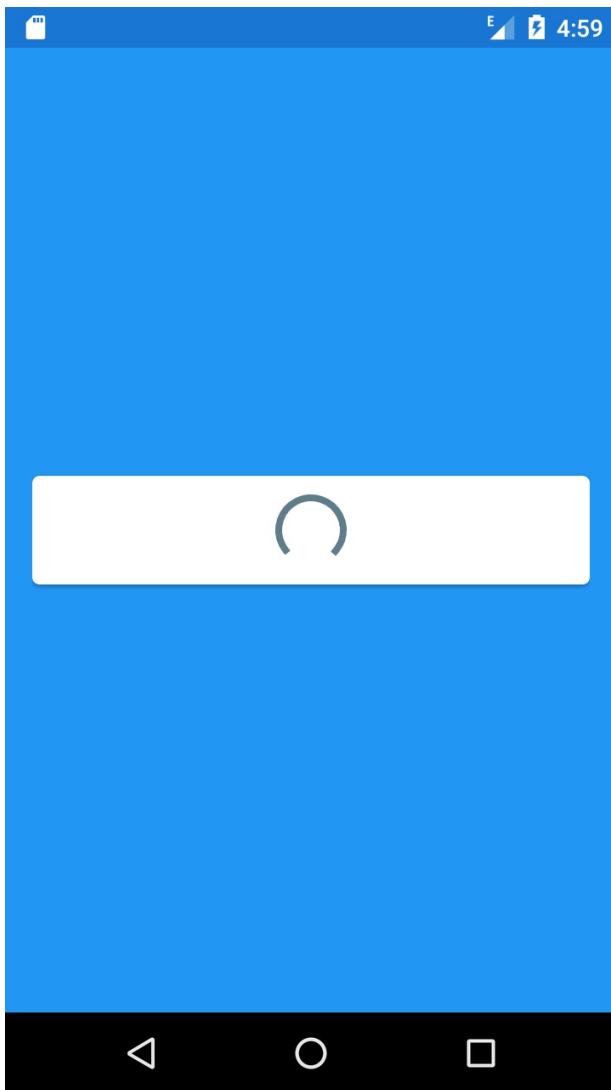
The **OAuthHandler** is used to load the authenticated user for the first time.

getAccessToken(final String code) performs a get request to the formatted token url, and parses the response as a string. The access token is extracted from the string between "access_token=" and "&scope". Once the access token has been extracted:

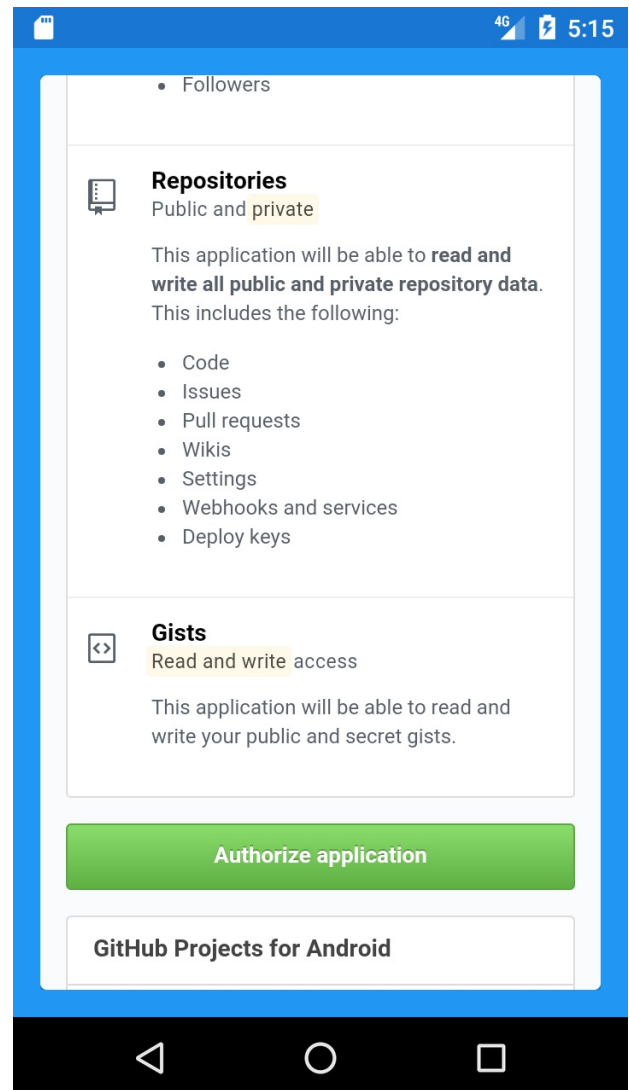
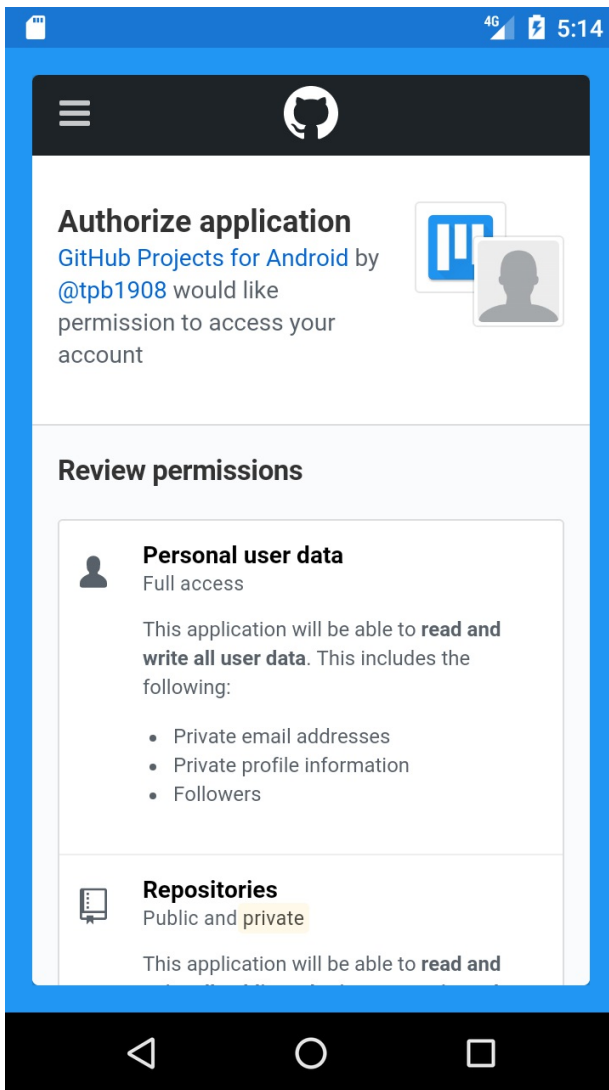
- The token is stored with **GitHubSession**
- The headers are initialised with the token
- The authentication listener (**LoginActivity**) is notified, allowing it to update the **ProgressBar**
- **fetchUser()** is called to load the authenticated user model

fetchUser() performs another request, this to time /user, which loads the authenticated user if provided with an authorization token. The response is returned as a **JSONObject**, Java's built in JSON model, and is passed to **GitHubSession** where it is stored as a string for later use, and parsed into a **User** object. The authentication listener is then called again, with the **User** object.

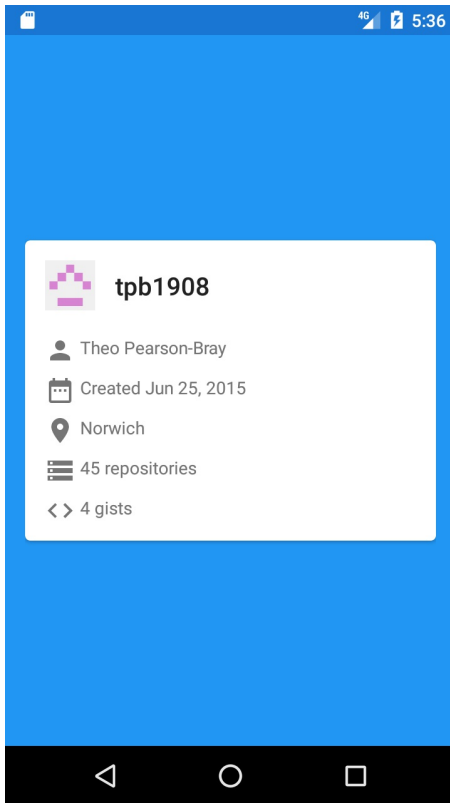
Prior to the **WebView** loading the login page, the **LoginActivity** shows the view with a spinning **ProgressBar**, and once it has finished the login page is displayed



Once the user has logged in, the GitHub authentication page will show the access which the app is asking for, and ask the user to grant access:



Finally, the user is loaded and their information is displayed.



Data models and loading

All of the models from the GitHub API are returned as JSON unless another content type is specified. Each endpoint returns either a single model, or a single dimensional array of models.

All models used extend the abstract class **DataModel** which contains some of the commonly used keys, as well as the object creation date which is used across all models.

```
package com.tpb.github.data.models;

/**
 * Created by theo on 15/12/16.
 */

public abstract class DataModel {

    static final String ID = "id";
    static final String NAME = "name";
}
```



```

    static final String CREATED_AT = "created_at";
    static final String UPDATED_AT = "updated_at";
    static final String URL = "url";
    public static final String JSON_NULL = "null";

    long createdAt;

    public abstract long getCreatedAt();

}

```

Networking is split into two classes extending **APIHandler** is split across four classes.

Loader

The **Loader** class is responsible for almost all get requests sent to the GitHub API.

Each method is responsible for load a single model type, and takes the path or filter parameters required to load the model(s) as well as an implementation of a generic loader.

The first interface **ItemLoader** is used when loading a single model or value.

```

public interface ItemLoader<T> {

    void loadComplete(T data);

    void loadError(APIError error);

}

```

Any class implementing `ItemLoader` must implement `loadComplete(T data)` as well as `loadError(APIError error)`.

Most uses of `ItemLoader` load an instance of `DataModel`.

An example is `loadIssue(@NonNull final ItemLoader<Issue> loader, String repoFullName, int issueNumber, boolean highPriority)`

```
loadIssue(@NonNull final ItemLoader<Issue> loader,
String repoFullName, int issueNumber, boolean
highPriority) {
    AndroidNetworking
        .get(GIT_BASE + SEGMENT_REPOS + "/" +
repoFullName + SEGMENT_ISSUES + "/" + issueNumber)
        .addHeaders(API_AUTH_HEADERS)
        .setPriority(highPriority ?
Priority.HIGH : Priority.MEDIUM)
        .build()
        .getAsJSONObject(new
JSONObjectRequestListener() {
            @Override
            public void onResponse(JSONObject
response) {
                loader.loadComplete(Issue.parse(response));
            }

            @Override
            public void onError(ANError
anError) {
                loader.loadError(parseError(anError));
            }
        });
}
```

this method is used to load a single **Issue** model given a full repository name (user login and repository name) and the issue number.

Some single methods also have prefetching when a null **ItemLoader** is passed to them:

```
loadProject(@Nullable final ItemLoader<Project>
loader, int id) {
    final ANRequest req =
    AndroidNetworking.get(GIT_BASE + SEGMENT_PROJECTS +
"/" + id)

    .addHeaders(PROJECTS_API_API_AUTH_HEADERS)

    .build();
    if(loader == null) {
        req.prefetch();
    } else {
        req.getAsJSONObject(new
JSONObjectRequestListener() {
            @Override
            public void onResponse(JSONObject
response) {

loader.loadComplete(Project.parse(response));
            }

            @Override
            public void onError(ANError anError) {

loader.loadError(parseError(anError));
            }
        });
    }
}
```

In this case the **ANRequest** instance is built and only requested as a **JSONObject** when there is an **ItemLoader** to deal with the model.
