BINARY NINJAZ

Harvest

Coding Standards

Letanyan Arumugam	14228123
Sizo Duma	15245579
Teboho Mokoena	14415888
John Ojo	15096794
Kevin Reid	15008739
Shaun Yates	16007493

STAKEHOLDERS

SAMAC: Barry Christie

Contents

List of Figures

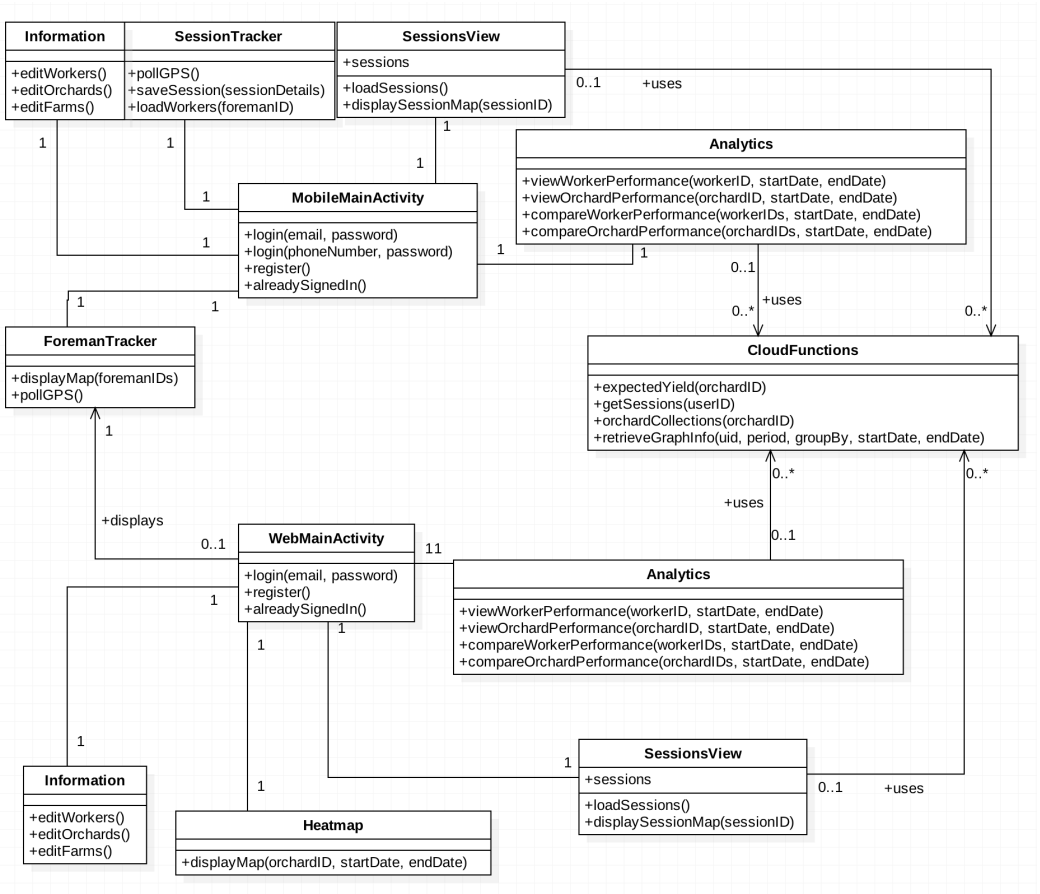

Figure 1: UML Class Diagram

1 Coding Conventions

The project and design we've undertaken means that we will need to use multiple languages and IDE's to complete the project. As such we have decided that the best common practice and standards of each language shall be the standard that is used for that respective language/environment. Linters are also used to enforce many of the rules we follow.

1.1 General Rules

- Attributes:
  special @ sign attributes should be on their own line.

- Class Delegate Pattern:
  Delegate pattern should be implemented using protocols/interfaces.
  In the case of protocols
  the protocol must be a class protocol.

- Closing Brace:
  The closing brace inside a parameter must not be followed by whitespace.
  Incorrect Usage: ‘array.map(.0)‘
  Correct Usage: ‘array.map(.0)‘

- Closure Closing Indentation:
  The closures closing brace must end on the same column that the starting lines column is on.
  Incorrect Usage:

```
func foo() {
  }
```

  Correct Usage:

```
func foo() {
}
```

- Closure Spacing:
  Closures must have at least a single space inside of each brace.

- Comma Spacing:
  A
  comma must not have any space infront of it and only one space after it.

- Conditional Return Positions:
  Any return statement from a conditional branch must be on a newline
  Incorrect Usage:

```
if (true) { return 0; } else { return 1; }
```

  Correct Usage:

```
if (true) {
  return 0;
} else {
  return 1;
}
```

- Cyclomatic Complexity:
  The cy-
  clomatic complexity of a function body should not exceed more than 10.

- Singleton Direct Initialization:
  Singleton classes should not be directly instantiated.

- Optional Boolean:
  Where available an Optional may not be used to wrap a Boolean. Prefer an enum instead.

- Prefer Non-Optional Collection:
  Where available an Optional should not be a preferred method of wrapping a collection. Instead try to
  use a collections emptiness to handle nullity as well.

- Empty Count:
  Use isEmpty property and not check emptiness via count/length property.

- Empty Parameter:
  Prefer Void over an empty parameter

- String isEmpty:
  Prefer using isEmpty over checking equality to an empty string

- Avoid Fallthrough:
  Switch statements should not fallthrough to subsequent cases but instead always break at the end of
  a case.

- File Length:
  File lengths should not exceed more than 2000 lines.

- File Name:
  A
  file name should match the name of a class/struct/extension it contains.

- Function Body Length:
  Length of a function body should not exceed 150 lines.

- Default Parameters at End:
  Default parameters should be the last parameters in a parameter list.

- Function Parameter Count:
  A maximum of 5 parameters per function should be allowed.

- Type Names:
  Type names should start with an uppercase letter and be of length 1 to 20 characters. Each
  character should only be an alphanumeric. Where subsequent words in the name are also
  capitalized.

- Identifier Name:
  Type names should start with an lowercase letter and be of length 1 to 20 characters. Each
  character should only be an alphanumeric. Where subsequent words in the name are capitalized.
  Or names can be all uppercased if deemed fit.

- Tuple Size:
  Where available tuple sizes may not exceed an arity of 3.

- Leading Whitespace:
  Files may not start with whitespace (excluding comments)

- Line Length:
  Lines should not exceed 120 characters.

- Mark Comments:
  Mark comments should follow "MARK: ..." or "MARK: -..."

- Fix Me Comments:
  Fix me comments should be treated as warnings and follow the format "FIXME: ..."

- Todo Comments:
  Todo comments should be treated as warnings and follow the format "TODO: ..."

- Type Nesting:
  Types should not be nested more than one type scope deep.

- Open Brace Spacing:
  Open braces should
  be on the same line as the declaration and be preceded by a single space.

- Operator Whitespace:
  Operators should be surrounded by an equal number of whitespace characters.

- Redundant Void Return:
  Void returning functions should not be explicitly stated if not needed.

- Modify Assign Operators:
  Prefer operators such as +=, *=, -=, /= over constructs such as ‘a = a + 42‘

- Sorted Imports:
  Import list should be sorted alphabetically

- Statement Positions:
  Else and catch statements should be on their own line.

- Switch Case Statement Indentation:
  Case statement indentation should be the same as the switch statements indentation.
  Incorrect Usage:

```
switch f {
  case a:
}
```

  Correct Usage:

```
switch f {
case a:
}
```

- Trailing Newline:
  Files should have a single newline at the end.

- Type Body Length:
  A types body length should not exceed 500 lines.

- Unneeded Break:
  Switch statements should not have unnecessary break statements.

- Vertical Whitespace:
  Limit vertical whitespace to a single empty line.

1.2 Conventions used for Specific Languages

1.2.1 HTML/CSS

The HTML and CSS standards we shall follow will be the ones of Google's HTML/CSS style guide

1.2.2 JavaScript

For JavaScript we will be using Google's JavaScript style guide.

1.2.3 Java

Similarly we will use Google's Java style guide.

1.2.4 Swift

For Swift we will use Apple design guidelines.

2 File Structure

It is important to note that `Web/app` is where the main functionality of the website source code can be found and that `Web/functions` is strictly used for cloud functions.

2.0.1 Branching

An example structure is shown above, showing that `master` is never worked on directly. It is important to note that branch names given of feature branches are not necessarily set branches, but are examples to show branching structure. Each subsystem (Android, iOS, Website) has its own respective branch, and these branches branch off `Developer`, and in turn any changes done to a branch would then branch off that specific branch. A `Developer` branch is used, to act in much the same way as `master`, but as a proxy, so that before merging that branch to master, the entire system can be reviewed there before a more solid commitment is made.

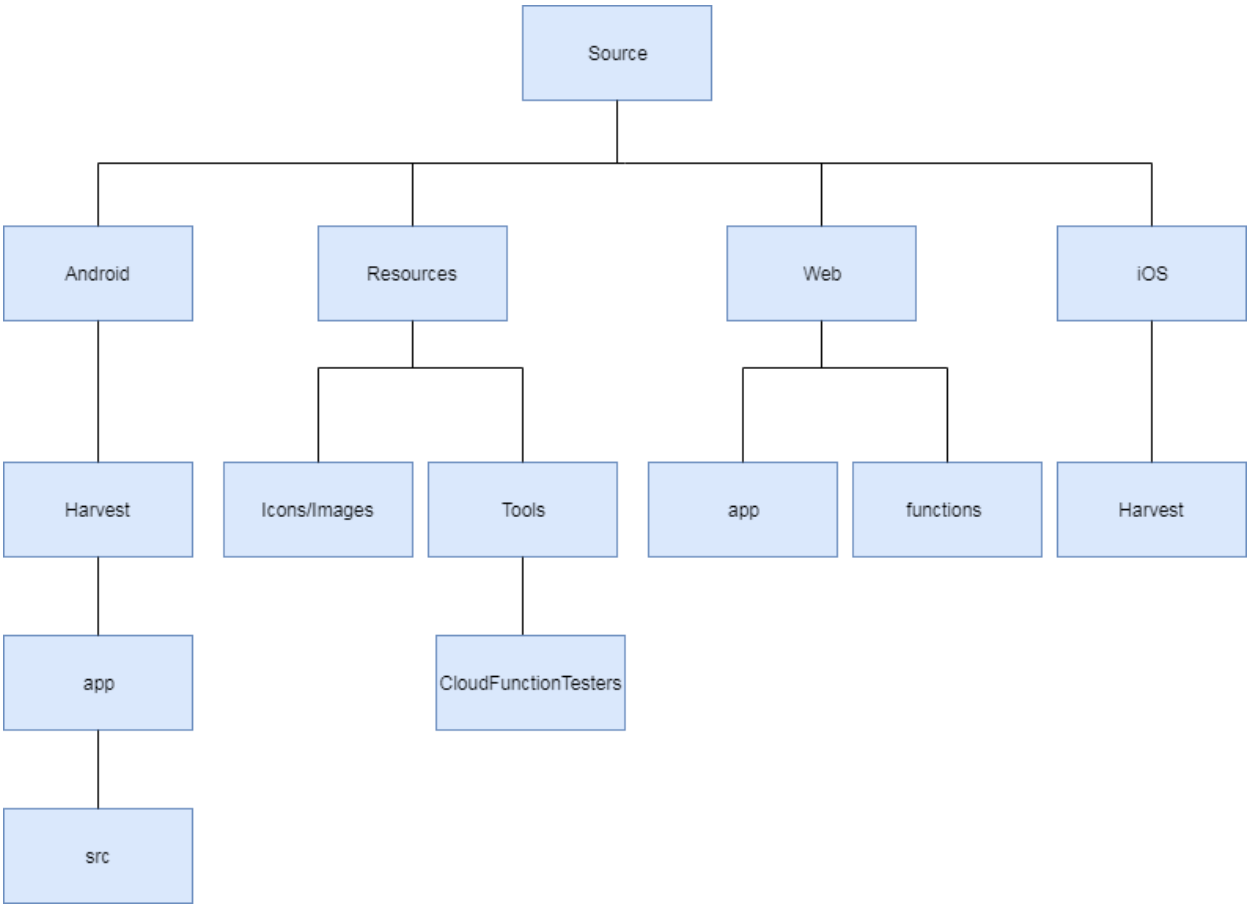

Figure 2: File Structure

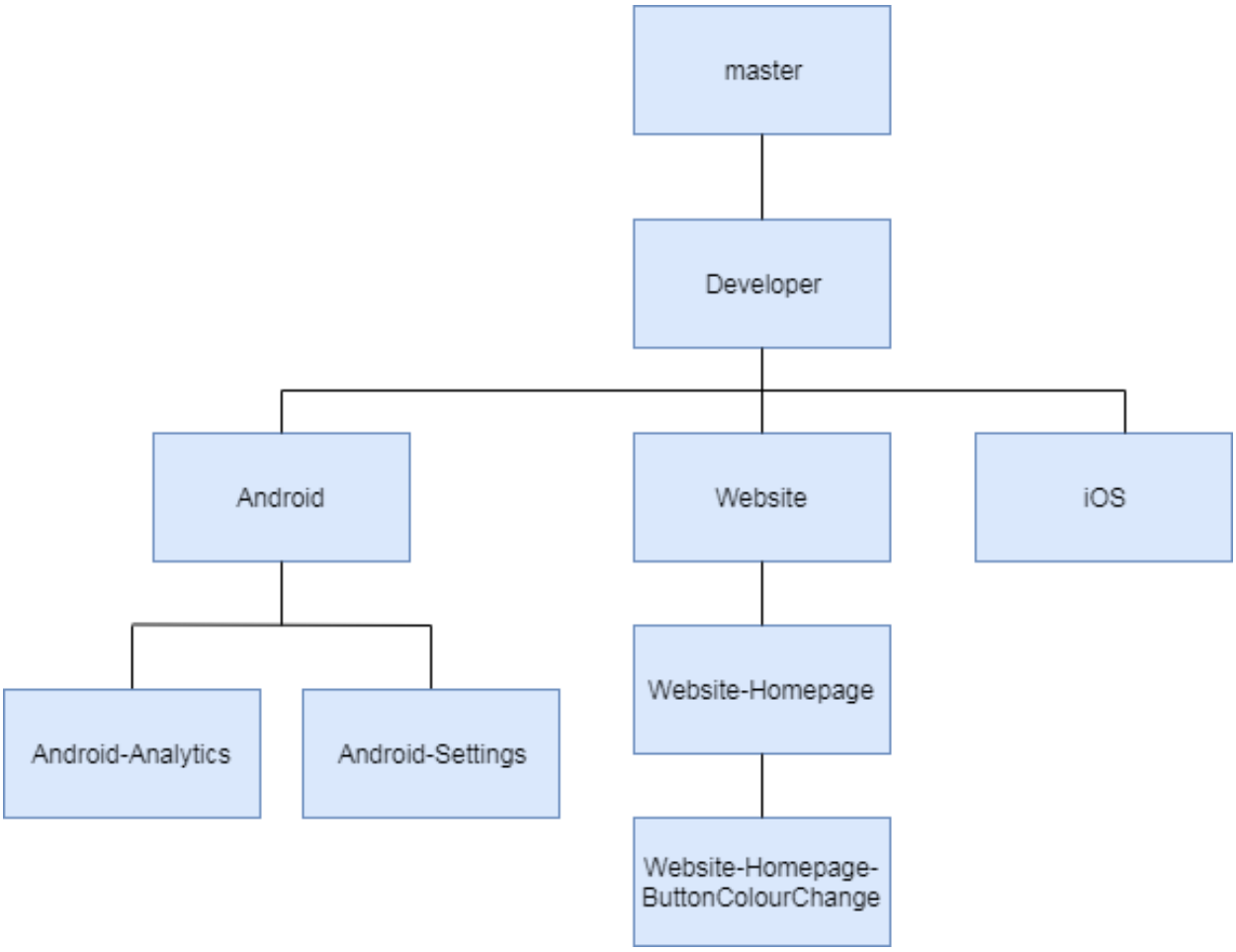

Figure 3: Git Branch Structure

2.0.2 Naming

A branch shall never contain an individuals name, rather it must be informative as to what the intention of the branch is whilst also remaining short. An example of a name that would be deemed unacceptable is `Joe`; another would be `JoeWebsiteChanges`, where the name contains a member name, and gives very little information as to what is going on in the branch. Bad, but barely acceptable names are `WebsiteChanges`, or `WebsiteExperimentation`. Ideal names are as follows: in the case of a branch where the website is ultimately being assembled, by merging other branches into it, and the only modifications taking place are to structure the files, or a quick fix, such as changing the colour of a button, `Website`; in the case that a new homepage is created for the website, where the homepage is created, and merged into `Website` on completion, `Website-Homepage`. Finally, if a user wants to propose a change to the `Website-Homepage` branch, where he wants to change the colour of a button, but the user has not been assigned, or is not the primary benefactor to `Website-Homepage`, he creates a new `Website-Homepage-ButtonColourChange` branch. Notice that in the ideal examples it is possible to track down the source of a branch, to understand exactly what it's purpose is.

3 Code Review Process

All code reviewing is done on GitHub, and occurs every time there are changes done to code, or if new code has been implemented. Currently, pre-commit reviews are utilized and although it is a slower process compared to post-commit reviews, it always for more people to be aware of changes to the code and could possibly lead to better implementation as more eyes usually means more ideas.
After changes have been made on the branch, the creator, who wants to merge, will then make a pull request on GitHub, and announce the request on Slack. After which it shall become the responsibility of the entire team, but more importantly that of the original branch maintainer to analyze and comment on the changes. The exact location of the discussion is trivial, however GitHub provides a better, and more concrete platform for this discussion to take place, where inline comments can be made. After a sufficient amount of time and discussion has passed, either the branch maintainer (if it is a minor change), or the entire team (if it is to master, or a major change) shall decide to merge or not.