Bae: Before Anyone Else; The Answer to Mobile Dating for the African Diaspora

Jordan S. Kunzika

Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.dartmouth.edu/senior_theses/115

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Bae :: Before Anyone Else

The Answer to Mobile Dating for the African Diaspora

By Jordan Kunzika

Dartmouth Computer Science Technical Report TR2016-810

A thesis submitted in partial fulfillment for the degree of Bachelor of Arts

in the

Department of Computer Science

June 2016
# Table of Contents

## Abstract

### Chapter 1: Background of Bae

- The State of the Online Dating Industry ................................................................. 1
- The Difficulties & Opportunities of Online Dating for African-Americans ............. 1
  - Figure 1.1: Stats regarding the difficulties African Americans face with online dating .................................................. 2
  - Figure 1.2: Stats regarding the opportunities with targeting African Americans .......... 2
  - Figure 1.3: Market Fit & Competition ......................................................................... 3
  - Figure 1.4: African Diaspora Smartphone Market ..................................................... 4

- Prior Work ...................................................................................................................... 5
  - Figure 1.5: Other Dating App Competitors ................................................................. 5
- Interesting Features to Explore .................................................................................... 5
- How do we measure success? ...................................................................................... 6

### Building the MVP

- Figure 1.6: Showcase of Initial Designs for Bae ....................................................... 8
- Figure 1.7: Development & Business Stack ............................................................... 9

## Chapter 2: Android Specific App Evolutions & Challenges

- Crashes ............................................................................................................................ 11
  - Uncaught Exceptions ............................................................................................... 11
  - Nullity Annotations .................................................................................................. 12
    - Figure 2.1: Applying Nullity Annotations ............................................................... 13
    - Figure 2.2: Examples of Nullity Annotation Implications ....................................... 13
  - Out of Memory Errors ................................................................................................ 14
  - Tools .............................................................................................................................. 15
    - Memory analysis tools in Android Studio ............................................................... 15
      - Figure 2.3: Memory Monitor in Android Studio .................................................. 15
      - Figure 2.4: Heap Viewer in Android Studio ....................................................... 16
    - Eclipse Memory Analyzer ....................................................................................... 17
    - LeakCanary ............................................................................................................ 18
      - Figure 2.6: LeakCanary Logo & Leak Detection Example ................................... 18
    - Implementing Memory Improvements .................................................................. 19
  - Image Loading ........................................................................................................... 19
    - Using the Universal Image Loader Library .......................................................... 20
      - Figure 2.7: Possibilities for DisplayImageOptions, a class that specifies how to display images with the imagerloader .......... 20
      - Figure 2.8: Possibilities for ImageLoaderConfiguration, a class that defines how the imagerloader should function ................. 20
      - Figure 2.9: Load and display task flow for images based on cache states ............. 21
    - Strategies for Balancing Memory and Load Time ................................................. 22
  - Build Variants ............................................................................................................ 23
    - Figure 2.10: File/Folder Structure For Build Variants .......................................... 25
As online dating becomes increasingly popular, millions of people across the country utilize various dating services that allow them to meet, chat, date, and develop meaningful relationships with people they would probably not have met otherwise. While mass market dating apps work well for the majority of online daters, and while many niche dating apps have been created to target a number of specific demographics such as the Jewish population, Southeast Asian population, farmers, homosexuals, etc., the African-American market has been largely underserved. African-Americans have the worst experience on mass market dating apps due to negative racial bias, and before Bae, there was no modern, ubiquitous dating app that specifically targeted the African American population. Bae attempts to fill the market gap by providing a mobile first, modern dating app curated specifically for African-Americans. I will discuss our process of building our MVP product, and the technical challenges of improving on it to provide the best user experience and achieve scale with a growing user base. Finally, I will discuss Bae’s successes, areas for improvement, and possible next steps.
Chapter 1: Background of Bae

The State of the Online Dating Industry

In this day and age, more and more people are trying to find meaningful relationships online. According to data from the Pew Research Center, over 30 million Americans have used an online dating service or mobile dating app (Smith & Duggan). Some of the most popular platforms include dating apps and sites such as Tinder, Match.com, OkCupid, Coffee Meets Bagel, Plenty of Fish, and many others. With the proliferation of dating apps and services, mobile dating has become a multi-billion dollar industry, proving to be a great platform through which people are able to meet, chat, and date.

The Difficulties & Opportunities of Online Dating for African-Americans

However, while many people seem to be getting a lot of value out of the services these sites and apps provide, minorities and people of color are largely left out in significant ways. As shown in Figure 1.1, which includes stats from a study done by OkCupid, 82% of non-Black men have a negative bias against Black women, a cohort of users who also have to send about 10 times as many messages as any other ethnic group just to get a single reply (Rudder). It’s also hard for people of color to find each other on these mainstream dating apps due to the fact that they are also minorities on those platforms as well.
Figure 1.1: Stats regarding the difficulties African Americans face with online dating

While the stats prove how disadvantaged African-Americans are when it comes to mobile dating, there are several stats shown in Figure 1.2 that showcase the unique reasons why the African-American market is a valuable niche to consider. Overall, the data shows that generally African-Americans are young, single, highly engaged on mobile apps and social media, and would highly benefit from a service that connects them with people of a similar background.

Figure 1.2: Stats regarding the opportunities with targeting African Americans
The concept of creating a dating app catered to the African-American population prompted analysis of the dating landscape to consider how this implementation would complement and/or compete with other online dating services. As shown in Figure 1.3, the dating services on the market are separated into four categories: mass dating sites, mass dating apps, niche dating sites, and niche dating apps. The mass dating services tend to market themselves as solutions that fit many demographics of people in several geographies while the niche dating services tend to be focused on a certain segment of population which can be based on many factors such as geography, race, religion, sexual preference, etc. The sites are categorized as dating services that may have initially been developed as desktop-first platforms before possibly being ported to mobile versions while the apps are services that were developed from the beginning as mobile-first platforms. Each category (mass market, niche, site, app) has various implications on how the dating service situates itself amongst the dating landscape.

![Figure 1.3: Market Fit & Competition](image)

Since an African-American focused dating app seemed to fit best with the niche dating app category, other dating apps using race as a niche category were studied for comparison. One
particular example of this was another niche dating app called JSwipe, which is a successful platform specifically made for the Jewish population. JSwipe boasts over 375,000 users in 70 countries, proving that there is value and potential in making a dating app that has a niche target market (Yarus). While there are a couple of dating services that attempt to cater directly to the African-American market, until recently, there has yet to be a ubiquitous dating app that caters to the African-American market in the same way that JSwipe caters to the Jewish market. In addition, as Figure 1.4 signifies, the market opportunity regarding African-Americans becomes much larger when considering that the African Diaspora smartphone market has over 190 million members, which is projected to grow to over 700 million in the next 2 years. Due to all the factors listed above, Bae was created with the aim of becoming the number 1 dating app for the African Diaspora around the world.

Figure 1.4: African Diaspora Smartphone Market
Prior Work

While creating Bae, other dating apps that were utilized and studied for the purposes of comparing various features and competition analysis are shown here in Figure 1.5 (some of which are also listed in Figure 1.3). This is not an exhaustive list, but it showcases the primary alternative dating apps that were used for comparison:

![Other Dating App Competitors](image)

*Figure 1.5: Other Dating App Competitors*

Interesting Features to Explore

Outside of simply providing a means of creating meaningful relationships for members of the African Diaspora, there were a few other features of interest that were deemed valuable for enhancing the experience with and beyond mobile dating. For instance, since, as stated in Figure 1.2, African Americans are 39% more likely than members of any other race to make a purchase
based on ads viewed from a cell phone, and because events are a great way to facilitate having people meet each other in real life, we wanted to explore the concepts of performing dynamic event engagement & in-app advertising (Experian Marketing Services). Furthermore, since a large part of the success of any app is keeping users engaged, we wanted to explore unique ways of engaging our users via email & push notifications for our demographic.

**How do we measure success?**

In the process of creating Bae, analytics from several sources were used to help determine how to measure success with specific aspects of the app. This list contains examples of some of the metrics we try to review occasionally:

- Daily/Weekly/Monthly Active Users in relation to total users
- Crash Percentages
- App Store ratings & reviews
- Click/install rates of marketing links
- Open/click rates of engagement emails
Building the MVP

With my role as CTO & Co-Founder of Bae, the majority of my direct contributions were within the realms of Android and backend development, while I assisted in the management of the iOS development. Therefore, while all the aspects of the app will be touched upon, stronger emphasis will be made on aspects pertaining to Android and the backend development.

Figure 1.6 below shows the initial designs created for the first iteration of Bae. For the MVP, upon logging in with Facebook, the user could create their profile by adding up to 5 photos from Facebook, writing a description in the about me section, and deciding to hide or show various information we automatically retrieve from Facebook such as hometown, education, occupation, and current location. In addition, a user could enter their settings to control the potential matches they see in terms of how far they are and what their gender and age is. After creating their profile, a user can simply swipe right on a user to select them, left on a user to reject them, and can click on the user to view their profile pictures and/or information. When two users swipe right on each other, they are matched and they can chat within the app for free to foster the relationship. This differs from dating apps such as BlackPeopleMeet in which users need to pay to message others.
When initially launching the app, the decision was made to limit the swipes people could use to 10 total swipes every 24 hours. This decision was prompted from a number of differing factors. First, since the app hadn’t been launched yet, the small cohort of early users would have run out of potential matches very quickly. Second, the goal was to move away from the unlimited swiping model that Tinder had at the time and use a model more akin to Hinge where users have a certain amount of swipes allotted per day. The hope was that by limiting the swipes, users would be forced to think more about their potential matches before swiping left or right, which could ideally lead to more meaningful relationships.

Bae was launched on April 1, 2015 at an event hosted at Howard University that held around 2,000 students. Two weeks after launching Bae, we won 1st place and the People’s Choice Prize at the Dartmouth Entrepreneur’s Forum (formerly known as Dartmouth Ventures), winning $27,500 in prize money. This momentum allowed us to reach 17,000 downloads at the end of our first month, and soon caused us to be a Top 50 Lifestyle app in several African and Caribbean countries.
In building Bae as both an app and a business, we used the stack shown in Figure 1.7. The elements in bold were the aspects of the app that we used and were present in our MVP product, while the other elements were slowly added in over time as the product evolved.

<table>
<thead>
<tr>
<th>Frontend</th>
<th>iOS &amp; Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backend &amp; APIs</td>
<td>Parse, Facebook</td>
</tr>
<tr>
<td>Build Distribution</td>
<td>HockeyApp, TestFlight, Google Play Beta</td>
</tr>
<tr>
<td>Analytics &amp; Marketing</td>
<td>Mixpanel, Parse Analytics, Google Analytics, Fabric.io, Branch.io</td>
</tr>
<tr>
<td>Team Communication</td>
<td>Slack &amp; Skype</td>
</tr>
<tr>
<td>Product Management &amp; Customer Support</td>
<td>Apptentive &amp; JIRA</td>
</tr>
</tbody>
</table>

*Figure 1.7: Development & Business Stack*
Having launched the MVP product and received both validation and feedback from users and the greater community, the aim was to iterate on Bae in order to solidify our MVP. These were the goals that were set to help us improve both as an app and a business:

- **Clean up bugs and issues from our original MVP and iterate on the user experience to create one that satisfies the majority of our users (to be measured via ratings, reviews, and crash/bug reports)**
- Create a streamlined process for dealing with customer support, technical issues, and overall product management
- Create an organized process by which new features and bug fixes are developed, tested & validated internally through alpha testing, and tested & validated externally through a robust beta testing group before being released to the public
- Increase user engagement and retention with the app
- Get users to their first match as soon as possible
- Accelerate our user growth & increase the incentives for and the ability for users to share the app with friends to help grow the user base

While I did contribute towards working on all of these goals, the rest of the thesis will focus on the goal boldered above, the particular challenges I encountered while iterating on the Android app and our backend framework, and how I overcame those challenges.
Chapter 2: Android Specific App Evolutions & Challenges

Upon improving the MVP for the Android application, the following questions came up:

1. How could the Android app for Bae become at least 98% crash-free?
2. Since Bae is at its core a photo-viewing app, how could the app be structured such that images are loaded in a timely, yet memory-efficient manner?
3. How could the app be developed to be flexible for various kinds of configurations such app versions used for alpha/internal testing, beta testing, and public release on the Play Store?

Crashes

The first question took precedent because crashes are the highest priority issues that users could face. Using Crashlytics as the primary platform for tracking crash data, it turned out that most crashes could be placed into one of two categories: uncaught exceptions and out of memory errors.

Uncaught Exceptions

Exceptions in Java are essentially errors that occur during the execution of a function which disrupt the normal flow of a program’s instructions. When a method throws an exception, the runtime attempts to find an exception handler to catch the exception and run some code to handle the error. When the exception remains uncaught after the runtime system exhausts its search for an exception handler, the application terminates, which is colloquially recognized as a crash from the user’s perspective. Exceptions come in all kinds of flavors such as IndexOutOfBoundsExceptions, which occur when trying to access an object from an array at an invalid index, or IOExceptions, which reveal errors that could occur when trying to read or write to files. However, the most frequently occurring exception was the NullPointerException, which is thrown by methods that try to manipulate or access methods from variables that are currently set to null. Many of these originate from a level of naiveté that leads one to code in an optimistic fashion that expects values to always be non-null. Therefore, the discovery of the frequency of NullPointerExceptions led to the
following sub-question: **how does one code defensively to prevent uncaught NullPointerExceptions from occurring?**

The part of this question that addresses the means of coding defensively has a couple of relatively straightforward answers. The first approach is to block NullPointerExceptions by wrapping lines of code that deal with a potentially nullable object with a try-catch block that catches NullPointerExceptions when they are thrown. The second approach is to dodge a NullPointerException and prevent it from happening by performing a null check using an if-else statement that performs some code if the nullable object is equal to null and some other code if the nullable object is, in fact, null. While these are the seemingly obvious ways to prevent NullPointerExceptions, simply knowing about what to do to prevent NullPointerExceptions does not solve the part of the question that addresses how one can ensure that they are actually coding defensively.

**Nullity Annotations**

The concept of nullity annotations was utilized. Nullity annotations are part of larger annotations feature set provided by IntelliJ IDEA, the Java IDE that was used to build Android Studio. What they allow an Android developer to do is to label functions and variables with annotations such as @Nullable and @NotNull to declare whether a function or variable could return null or whether a function or variable is not allowed to return null, respectively. In the case of functions or variables that are labelled as @Nullable, these annotations are useful for reminding developers to perform null checks in their code (Inferring Nullity). As for the @NotNull case, the IDE will warn the developer when a variable that is nullable or a function that can return null with this label exists so that the proper action can be taken. On top of that, Android Studio has a feature called “Infer Nullity” that will scan either 1 file or the whole project and will automatically decorate one’s code with the proper annotations. The nullity annotations feature has definitely helped in...
making smart decisions about where to make null checks and how to write code defensively to prevent uncaught NullPointerExceptions, an occurrence which drastically decreased over time.

![Image](image1.png)

**Figure 2.1: Applying Nullity Annotations**

```java
@NonNull
public void onLocationChanged(Location location) {
    if (location != null) {
        processLocation(location);
    }
}
```

The top left and right sections show what occurs when the parameter location is labelled as being @NonNull. It treats the statement “location != null” as unnecessary because the annotation of @NonNull means that one should be able to assume that location is a non-null value. The bottom left and right sections show what occurs when the parameter location is labelled as being @Nullable. Because location is considered nullable, it does not complain about the presence of a null check. However, in the bottom right, it warns the developer that location might be null, which also happens because processLocation is a function that expects an @NonNull location object.

![Image](image2.png)

**Figure 2.2: Examples of Nullity Annotation Implications**
**Out of Memory Errors**

Every application has a set amount of memory that it is allowed to use based on a certain percentage of the RAM available on the phone. As a result, out of memory errors occur when an allocation of memory is made that causes the app to exceed its allowable amount of memory. These kinds of errors usually point to memory leaks, inefficiencies with loading images into memory, and potentially the need for other forms of optimization throughout the app. Their frequency also varies based on the amount of memory that a particular device has available, meaning that memory inefficiencies can cause more problems on devices that have less memory available overall. These issues led to the following sub-question: **what tools should be used to investigate the root causes of the memory issues, and what is the best way to implement viable improvements to reduce the frequency of out of memory errors?**
Tools

Three different tools were used to gain some perspective on where memory issues may lie:

Memory analysis tools in Android Studio

![Figure 2.3: Memory Monitor in Android Studio](image-url)
As shown in Figure 2.3, one of the memory profiling tools from Android Studio that was used for investigating out of memory errors was the memory monitor. The way it works is that if your Android device is plugged into the computer while you are running your app, it will show a real-time graph showcasing the amount of memory currently being used, and the amount of memory that the app has left. From this view you can also explicitly cause a garbage collection to occur, or you can perform a memory dump. Performing a memory dump will capture a record of the current state of the heap, which one can view in the heap viewer, shown in Figure 2.4. The heap viewer is also very useful because it allows you to view the heap and see how many instances of each class was contained in the heap, how much memory those instances consumed, and you can also explore the instances themselves and view their corresponding inner variables and references to other objects within the heap.
The combination of these tools was used to gain insight into the memory related issues. The memory monitor showed which user-performed actions on the app correlated with certain patterns of memory consumption while the heap viewer allowed one to deeply analyze which kinds of objects were taking up the most amount of memory so that effective plans could be made for optimizing the code to reduce out of memory errors.

![Image of Leak Hunter and Heap Viewer in Eclipse Memory Analyzer]

**Figure 2.5: Leak Hunter and Heap Viewer in Eclipse Memory Analyzer**

**Eclipse Memory Analyzer**

The Eclipse Memory Analyzer is a Java heap analyzer that was built for finding memory leaks and reduce memory consumption. While the heap viewer available in Android Studio was good for seeing the raw data about which classes consume the most memory, it became very tedious to manually analyze the heap to determine where the memory leaks are. The Eclipse Memory Analyzer was more useful because besides containing a more robust heap viewer, it includes a leak hunter feature that analyzes the hprof files produced by memory dumps performed from the memory monitor in Android Studio and determines a list of instances within the heap that seem to serve as suspects for a memory leak. Use of this tool revealed that some of the activities in Bae were leaking in ways that were not previously detected using the manual form of analysis, a revelation that led to further reductions in out of memory errors.
LeakCanary

LeakCanary is an Android library developed by Square that detects the occurrence of memory leaks in real time (Square). By default, it will detect if activities and/or fragments get leaked, but if one wants to ensure that certain variables aren’t leaked, they can explicitly have the library watch certain variables to see if the variable ever reaches a state when the memory is no longer freeable. When LeakCanary detects the memory leak, it performs a memory dump on the device itself and provides an app that one can open to view what the latest memory leak was, which references were leaked, and the stacktrace related to the memory leak. LeakCanary has been very useful for reducing out of memory errors because while the other tools provide the raw data which would need to be analyzed manually to determine if a leak occurred, LeakCanary gives developers insight into which specific parts of the app leak, and what actions directly cause leaks, especially since the detection occurs in real time.

![LeakCanary Logo & Leak Detection Example](Image)

*Figure 2.6: LeakCanary Logo & Leak Detection Example*
Implementing Memory Improvements

While the tools described above assisted with the detection of memory leaks, there still remained the issue of how to implement the memory improvements. For instances where entire activities or fragments leaked, it often had to do with references to views not being cleared when the activities get destroyed. The reason why this causes leaks is that each view that one instantiates for the purpose of programmatically manipulating it internally contains a strong reference to the activity it is attached to preventing the activity itself from being garbage collected. Therefore, for these issues, it was imperative to set the view-related references to null in the onDestroy method for each activity to ensure that the activities could properly be freed. There were also instances where the heap viewers revealed that many more instances of a particular type of object than previously expected or that certain objects were holding much more information than expected. These cases led to optimizations that allowed those objects to be more lightweight in the information they contained. While some of these memory issues had rather straightforward solutions, the next section will highlight some of the challenges with preventing out of memory errors that occurred due to image loading.

Image Loading

With a dating app, where experience is centered around one’s ability to view and interact with potential matches on the platform, profile pictures serve as the cornerstone of that experience. Even in a recent in-app survey conducted with our users using Mixpanel, 68.4% of responders said that great photos are the number 1 reason that they would swipe another user right. Given how crucial photos are to one’s experience with Bae, it was important to make sure that high quality photos loaded quickly without taking up too much memory, which prompted the following
question: **how could the app be structured such that images are loaded in a timely, yet memory-efficient manner?**

**Using the Universal Image Loader Library**

Due to early reviews stating that the images were loading slowly, the initial focus was on improving the image load time. The process for showing images was originally set up such that images were being downloaded from the web as they were being displayed. With this setup across different phone types and varying network strengths, images would take anywhere between 1-2 seconds to 10 seconds to load into view. Several attempts to solve this issue led to the discovery of an open source image processing library called Universal Image Loader (Tarasevich). This library provides a suite of configurable utility functions that assist with downloading, manipulating, caching, and displaying images. Here are some code snippets that illustrate the configurable functionality that this library provides:

```java
DisplayImageOptions options = new DisplayImageOptions.Builder()
    .showImageOnLoading(R.drawable.ic_stub) // resource or drawable
    .showImageForEmptyUri(R.drawable.ic_empty) // resource or drawable
    .showImageOnFail(R.drawable.ic_error) // resource or drawable
    .resetViewBeforeLoading(false) // default
    .delayBeforeLoading(1000)
    .cacheInMemory(false) // default
    .cacheOnDisk(false) // default
    .preProcessor(...)
    .postProcessor(...)
    .extraForDownloader(...)
    .considerExifParams(false) // default
    .imageScaleType(ImageScaleType.IN_SAMPLE_POWER_OF_2) // default
    .bitmapConfig(Bitmap.Config.ARGB_8888) // default
    .decodingOptions(...)
    .displayOnNewThread(true) // default
    .handler(new Handler()) // default
    .build();
```

**Figure 2.7: Possibilities for DisplayImageOptions, a class that specifies how to display images with the imageloader.**
Figure 2.8: Possibilities for ImageLoaderConfiguration, a class that defines how the imageloader should function.

The ImageLoaderConfiguration class is used to set up parameters for how this library's image loader will function across the app and the DisplayImageOptions class is for setting up specific parameters for how images will be loaded into corresponding imageviews. Given that improving the load time for images was the primary focus, strategies for how to utilize this library were largely based on the concepts contained in the following figure provided from the library's documentation:

```java
File cacheDir = StorageUtils.getCacheDirectory(context);
ImageLoaderConfiguration config = new ImageLoaderConfiguration.Builder(context)
    .memoryCacheExtraOptions(480, 800) // default = device screen dimensions
    .diskCacheExtraOptions(480, 800, null)
    .taskExecutor(...)
    .taskExecutorForCachedImages(...)
    .threadPoolSize(3) // default
    .threadPriority(Thread.NORM_PRIORITY - 2) // default
    .tasksProcessingOrder(KeepProcessingOrderType.FIFO) // default
    .denyCacheMultipleSizesInMemory()
    .memoryCache(new LruMemoryCache(2 * 1024 * 1024))
    .memoryCacheSize(2 * 1024 * 1024)
    .memoryCacheSizePercentage(13) // default
    .diskCache(new UnlimitedDiskCache(cacheDir)) // default
    .diskCacheSize(50 * 1024 * 1024)
    .diskCacheFileCount(100)
    .diskCacheFileNameGenerator(new HashCodeFileNameGenerator()) // default
    .imageDownloader(new BaseImageDownloader(context)) // default
    .imageDecoder(new BaseImageDecoder()) // default
    .defaultDisplayImageOptions(DisplayImageOptions.createSimple()) // default
    .writeDebugLogs();
    .build();
```
Figure 2.9: Load and display task flow for images based on cache states

Strategies for Balancing Memory and Load Time

This chart shows that the image loader library uses a hierarchy of caches to manage how images ultimately get displayed. If an image has not been downloaded before, the imageloader must initially download it before the image can optionally be stored in a disk cache or a memory cache. On the next usage of that image, if the system detects that the image has been cached on disk in a temporary file or as a bitmap in memory, then it will pull the image from one of those two sources instead of downloading it directly from the internet.

The initial strategy was to display all images with the memory cache enabled because as one can tell from the chart, an image cached in memory goes through the least amount of steps to display the image. Upon implementing, there was a significant improvement in the time it took to view images for profiles one had already matched with as well as for one's own profile. However, over time, this method proved to have some caveats.

For one thing, for profiles that users were swiping through for the first time, the images still took somewhat long to load, primarily because even though memory cache was enabled, the images still had to be downloaded once from the internet. An AsyncTask was developed in an attempt to
resolve this issue. The AsyncTask runs as soon as the app receives the next batch of profiles to swipe through, and it opens a separate thread within which it loops through all the profiles, downloads the images, and stores them into the memory cache ahead of time. By doing this asynchronous process, it ensured that even the new profiles that a user is about to swipe on can load images much faster than before.

Unfortunately, while improvements were made with the timely aspect of the image loading challenge, the memory aspect of this challenge was pretty significant. Even though part of the strategy to reduce out of memory errors overall involved using the tools described in the previous section to detect memory leaks, a significant amount of out of memory errors occurred while the app was trying to allocate memory to display an image. With regards to the solutions developed for making image loading faster, the process of storing all the images ahead of time in the memory cache tended to cause more out of memory errors to occur on phones with smaller amounts of RAM. In order to solve this, the display image options were modified such that memory caching was disabled while disk caching was enabled. Performing this modification proved to produce the best balance of faster image processing coupled with a reduction in memory related errors due to image processing.

**Build Variants**

This section addresses question number 3, which was: How could the app be developed to be flexible for various kinds of configurations such app versions used for alpha/internal testing, beta testing, and public release on the Play Store?

Bae was designed from the start to be able to interact with two separate Parse backend environments, one used for development and another used for production. In order configure the app to point to one backend or the other, it was necessary to switch between separate sets of API keys to do so. This pattern began to extend to other services that were adopted such as various analytics platforms because it was important to keep any data that was collected by using
development versions of the app from polluting the data being produced by real users in the production environments.

In the beginning, the act of switching between app configurations was handled via a single global boolean variable in code called isProduction, which would be set to true or false depending on the desired configuration the app before building it. However, as the configurations became more complex, it became clear that using this as a means of switching between app configurations was both limiting and error-prone. For instance, in order to specify a version of the app that points to the production Parse backend, yet routes analytics data to our testing environments for those platforms, it was necessary to create more boolean variables to manage that nuance. Also, in the imaginary case where a version of the app that points to our development Parse backend is built and accidentally gets published to the Play Store, it would be disastrous because real users would start logging in to our test database, leading to several user experience issues. Therefore, question number 3 comes from a desire to have an efficient way to build the app for various configurations that does not rely on remembering to manually edit variables within the codebase.

The concept of build variants was used as the solution to this problem. Build variants are configurations of an app that allow developers to easily swap out layouts, variables, and other resources without having to manually change variables to represent what kind of app one would like to build (Gradle Plugin User Guide).
Figure 2.10: File/Folder Structure For Build Variants

The image on the far left illustrates selecting which kind of build to create. The other three images illustrate how the file/folder structure for build variants works. While the default files are contained in the “main” folder, the other build variant specific folders contain files that have the same names as equivalent files in the main folder but they have different values that are specific to that build variant. When Android Studio builds a particular build variant, it combines identical versions of the same file from the main folder and the corresponding build variant folder, considering the parameters from the build variant version as higher priority when faced with parameters that have the same name but different values.

Figure 2.11: Examples of bool.xml (left) and strings.xml (right) from the perspective of build variant stagingTest
Figure 2.12: Example of build variants defined in build.gradle file

Thanks to the Android Studio build variant system, the manual configuration system was replaced with the following 5 product flavors:

<table>
<thead>
<tr>
<th>Product Flavor</th>
<th>Result of Build Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>stagingTest</td>
<td>App called Bae-Staging that points to our test Parse environment. Used for Hockey app alpha testing builds.</td>
</tr>
<tr>
<td>stagingProd</td>
<td>App called Bae-Staging that points to our production Parse environment. Used for Hockey app alpha testing builds.</td>
</tr>
<tr>
<td>stagingProdUser</td>
<td>App called Bae-Staging that points to our production Parse environment, with added ability to login as any user without needing Facebook account information. Only used for internal testing and debugging purposes for specific user profiles.</td>
</tr>
<tr>
<td>stagingTestPostParse</td>
<td>App called Bae-Staging that points to our self-hosted backend environment that represents our app after migrating from the regularly hosted Parse. This is used for testing and validating how the app works with our own server.</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>production</td>
<td>App called Bae that points to our production Parse environment. Used for Google Play Beta and live production builds.</td>
</tr>
</tbody>
</table>
Chapter 3: Parse Backend Specific Evolutions & Challenges

Upon improving the MVP for the Parse Backend, the following questions came up, with the first being the most important:

1. **How could Parse Cloud Code be used to create a robust, general system of large-scale, recursive querying that could be used in various kinds of long running background jobs?**
2. How could essential queries be structured and performed to optimize efficiency of time and resources?
3. With the hosted version of Parse reaching its end-of-life, what should be considered for migrating the backend?

Large-Scale, Recursive Queries

Challenges of Manipulating Large Amounts of Data

As improvements to the MVP were being made over time, certain business decisions led to the need for various server side tasks to act on the behalf of increasingly large amounts of data. For example, when making decisions that alter the data schemas such as adding a new field of information for users, not only does this require structuring server and/or client side code to populate data into the new field properly, but in many cases, this also requires needing to perform long running background jobs to pre-populate that field with initial information in order to ensure that all the users have that information moving forward. Also, when attempting to engage with certain segments of the user base with custom, relevant push notifications or emails, these kinds of tasks require doing repeatable large scale queries since they could involve actions that need to be applied to thousands of users at a time.
Limitations of Parse

By default, Parse queries are limited to returning 100 results, which can be increased to a maximum of 1000 results. It is possible for one to page through results by using a method that allows you to skip a specified number of results, but one can only skip up to 10,000 results. Beyond that, there are few ways to query more than 10,000 items.

Means of Performing Large-Scale Querying in Parse

One way is to use the “each” method from the Parse.Query JavaScript object, which will allow one to specify a certain task that should run for each object that fits the query. However, this method doesn’t allow one to limit the number of results, meaning that if there are 100,000 objects that match your query, it will attempt to simply run your task on each object. With background jobs in Parse having a limit of 15 minutes before being terminated, doing this can lead to the background job failing before completing the task.

Another way to query for more than 10,000 items is to page through results by using constraints on the creation date field. For instance, one could use the “lessThan” method from the Parse.Query JavaScript object to specify that they want to look at the 1000 entries that were created before a certain date. As long as the results are received in order of descending creation date (where earliest entries have the latest creation date), upon receiving those results, one simply needs to extract the date from the last entry and repeat the query in order to get the next 1000 entries. Since this is not bound by the 10,000 object skip limitation, one could do this repeatedly until the query returns less than 1,000 items, which would signify that there are no more results that fit that query.

While paging through results via the creation date field provides more functionality, there are couple of limitations that prevent this from being used so simply for background jobs that need to manipulate large datasets. First, due to the background job limit of 15 minutes stated earlier, this method is also subject to failing if it takes too long to process all the data available. Second, when
performing queries with specific constraints within processes that manipulate the results in a way such that the results would no longer fit the constraints of the original query, paging by creation date can lead to skipping over data.

Take the scenario where one desires to query for users who do not have a number filled into a newly added “matchesCount” field, and the underlying process aims to perform some calculation for the purposes of saving a number into that field. If one queries for the first 1000 results that were created before today’s date and manipulates each of those results in this way, then if one were to re-run the original query, this would naturally yield the next 1000 results since the first 1000 no longer match the query parameters. However, this means that if one were to page by creation date by updating the query to look at all results created before the last result of the previous query, many valid results would have been skipped.

Addressing the Difficulties with Data-Intensive Processes in Parse

The above limitations and difficulties were reached in the process of designing long-running background jobs for various functions, such as data migrations across the entire user base and sending custom push notifications and emails to specific user segments. This led to the question:

How could Parse Cloud Code be used to create a robust, general system of large-scale, recursive querying that could be used in various kinds of long running background jobs?

Creating General Functions for Exhaustive Querying & Data Manipulation

Due to the need for functions that can be used for exhaustively querying large datasets within the limits and nuances of Parse mentioned above, the first task was to develop a couple of functions specialized for certain use cases. The first use case was to have a function that allows for paging through results without manipulating the results themselves. As stated before, since the regular way of paging through results only allows for processing up to 10,000 results, there was a need to use the method of paging through results using the creation date as a constraint. Besides
this, since some of our datasets are so large that they would take more than 15 minutes to look through them, it was necessary to provide an easy way to specify how much of the data to process at once.

Research, trial, and error led to the development of the getAllRecordsByCreationDate function, as seen in figure 3.1. This function takes the following parameters:

- The name of a background job
- A variable that keeps track of how many times the function has been recursively called
- The query parameters to search for
- The limit for how many results the query should return at once
- The maximum number of results to process before terminating the function
- The date before which the results should have been created (used for the creation date constraint)
- The request parameters that the job was executed with
- A status object used for indicating progress, successes, or errors for the background job
- An optional function that will be executed once for each object in the list of results
- An optional function that will be executed for once for all of the objects in the list of results

Using these parameters, the getAllRecordsByCreationDate function was designed to run a given query to find a set of matching results. After a set of results is retrieved, depending on parameters provided, it will run a custom function to process those results. It is then up to the processing function to decide whether to terminate or re-execute the corresponding background job using an updated set of parameters that would represent an attempt to get the next set of results. The very first time the function is called, it looks for objects created before the current date, but on subsequent, recursive calls, it will take the creation date of the last object in the previous set and use that as the new constraint for the next one.
Figure 3.1: Example of a background job that utilizes general querying functions to recursively count the number users that match a specific set of query parameters
As shown in Figure 3.1, the `getAllRecordsByCreationDate` can be used when merely querying over data sets for the purposes of analyzing the data without manipulating the data being analyzed. However, as mentioned before, using this function becomes problematic if the data is being manipulated since it might cause the query to skip data. Due to this, it became necessary to create a similar function called `getAllRecordsByRecursiveJob`. This function is very similar to `getAllRecordsByCreationDate`, except that this function uses a particular variable within the request parameters called “isSelfDepleting”, which should be set to true for any background job that manipulates its results in a way such that the objects no longer fit the original query parameters. The reason why this is needed is that the manipulation itself serves as the mechanism for paging through the results, so in the cases where “isSelfDepleting” is set to true, the creation date constraint is no longer used.

**Making Background Jobs Recursive**

Some of the limitations with doing large-scale querying through background jobs included that trying to process all of the available data for a certain query at once within a single background
function could fail due to the 15-minute maximum imposed by Parse. This led to changing the strategy from having a single background job that attempts to process large amounts of data to having multiple background jobs that process smaller portions of data. Parse’s dashboard for managing background jobs allows developers to run background jobs once or schedule them to run at a certain interval. While the scheduling functionality is sufficient for certain kinds of background jobs, having a system of multiple, sequential background jobs that represent a single long-running process is not compatible with the concept of scheduling since each background job needs to know some information about the previous background job so that it can know which portion of the dataset to process. In addition, background jobs need to have the property of being recursive, which, here, refers to the ability for a background job to spawn another instance of itself with different conditions at the end of its own execution. This led to the sub-question: **how could the implementation of background jobs be designed to be recursive for the sake of supporting multiple, sequential background jobs that represent a single long-running process?**

Research into this issue led to Parse’s REST API, their platform for accessing Parse functionality using HTTP requests. Among the available endpoints, one particularly useful endpoint has the following URL structure: [https://api.parse.com/1/jobs/<name>](https://api.parse.com/1/jobs/<name>) (REST API Developers Guide). Designed to be used as an HTTP POST request, this request can trigger a background job when invoked with the name of the desired job. This API endpoint was used to create a server side function called executeJob (as shown in Figure 3.3), which can execute background jobs given the name of the job, parameters with which to start the job, and optional functions to run in the case of a successful or faulty job execution.
```javascript
executeJob = function(job, params, status, successFunction, errorFunction) {

  var app_id = "";
  var master = "";

  if(constants.TestEnvironment){
    // Test environment
    app_id = "<redacted>";
    master = "<redacted>";
  } else {
    // Production
    app_id = "<redacted>";
    master = "<redacted>";
  }

  var parameters = {
    url: 'https://api.parse.com/1/jobs/' + job,
    method: 'POST',
    body: params,
    headers: {
      'X-Parse-Application-Id': app_id,
      'X-Parse-Master-Key': master,
      'Content-Type': 'application/json'
    }
  };

  return Parse.Cloud.httpRequest(parameters).then(function(httpResponse){
    console.error("job " + job + " was successfully executed!");
    console.error(httpResponse);

    if(successFunction !== null)
      successFunction(httpResponse);

    status.message("job " + job + " successfully executed");

  }, function(httpResponse){
    console.error(httpResponse.text);

    if(errorFunction !== null)
      errorFunction(httpResponse);

    status.error("error executing job " + job);
  });
}
```

*Figure 3.3: Function to execute background jobs using the REST API endpoint*
The executeJob function in conjunction with the two functions developed for general exhaustive querying, getAllRecordsByCreationDate and getAllRecordsWithRecursiveJob, allowed for the development of several background jobs, such as the “countX” background job showcased in Figures 3.1 and 3.2, which were designed to be a single long-running process that is ultimately represented by many sequential background jobs that each process a portion of the dataset. As shown in 3.1, the countX background job can be configured to use either the getAllRecordsByCreationDate function or the getAllRecordsWithRecursiveJob function based on whether the query can be considered “self-depleting”. The optional processing function “getMoreObjects”, which gets called once after each set of results, invokes the executeJob function as long as the number of objects that have been processed so far has not exceeded the maximum number of results as represented by the stop variable (in cases when the stop is equal to -1, there is no maximum and the job will ideally continue to run until there are no more results to process).

Since the general querying functions keep track of relevant variables such as the amount of results processed so far and the creation date of the last object that was processed, and since those variables are passed as parameters into the executeJob function, the subsequent call to the “countX” background job will utilize the intermediate variables rather than the initial variables set when running the background job initially from the dashboard as done in Figure 3.2. Therefore, the functions executeJob, getAllRecordsByCreationDate, and getAllRecordsWithRecursiveJob serve as the essential building blocks for the robust, general system of large-scale, recursive querying that was necessary for various kinds of long running background jobs, thus achieving the goal.
Chapter 4: Successes, Areas of Improvement, & Next Steps

Android

As for the Android app in particular, Figure 4.1 illustrates how the rate of crash-free users has evolved over time. Crashlytics/Fabric.io provided data about the percentage of users that used the app on a given day that were crash-free for each build (or app version) that had a significant amount of users. These daily percentages were then calculated into overall lifetime crash-free user averages for each build as well as for all the builds combined. Crashes here refer to any error that causes the app to stop responding and automatically quit. As shown in Figure 4.1, in general, as the app matured, it became relatively stable by maintaining a crash-free user rate of 96-98%.

![Average Percentages of Crash Free Users Per Build](image)

*Figure 4.1: Chart showing the average percentage of crash free users per build*
Figure 4.2 shows the ratings for Bae as seen on the Play Store. While some of the poor ratings were due to crashes that were fixed over time as shown in Figure 4.1, others had to do with users who wanted the experience to be different in a number of ways. For instance, at the beginning, Bae was set up to give users 10 swipes every 24 hours and this was done to prevent users from running out of people to swipe on when we had just recently launched. Due to this restriction, many people wrote reviews asking us to raise this restriction. As the user base grew, this restriction gradually decreased until it eventually became unlimited left swipes, unlimited right swipes until your first match, and 100 right swipes every 8 hours after your first match (as of 5-20-2016), a change that seemed to be received positively by our users. Another significant reason for poor ratings was simply the association between the app and Facebook. Many users were annoyed that they had to use Login with Facebook to use the app, and in some cases, people who would have been users merely deleted the app as soon as they learned that there was Facebook Login. For some users, they stated that they either did not have Facebook accounts or that they did not feel comfortable using their Facebook accounts with Bae. This was interesting considering that most of our competitors were also exclusively Facebook Login as well. While Bae is still managed via Facebook Login today, there are plans to extend beyond this, which will be mentioned in the “Next Steps” section.

![Figure 4.2: Rating on the Play Store as of 5-20-2016](chart.png)
As for Parse, due to the work done to create a robust, general system of large-scale querying and recursive background jobs, one way to represent the success of this is to discuss what this new system was able to produce and support. This system allowed for the creation of 60 different background jobs with varying functions such as large-scale data migrations, data analysis queries, user engagement campaigns, and in-app event advertising.

Regarding the user engagement campaigns, this was critical because background jobs could be created to generate a dynamic email engagement series in which emails were being sent to users based on their position within the Bae user onboarding flow. For instance, the periodically running background jobs would detect when users recently joined the app to send a welcome email; if the user had not finished uploading pictures after being on the app for 3 days, they were sent another email to remind them to do so; if the user recently got a match and their match sent them a message but the user had not responded yet, the user would be sent an email to remind them to talk to their match; lastly, if the user had not logged on to Bae in about a week, and there were people who had swiped that user right recently, that user would be sent an email to remind them to come back to the app and swipe on the potential matches that were waiting for them.

Besides the formal email engagement series, the extended background job functionality allowed for jobs to be created for generating in-app advertising for events. For instance, there have been a number of times when Bae has hosted an event such as a launch party or sponsored an event in collaboration with partners. While some of these events are focused on acquiring more users, other events are meant to celebrate and foster community within the current user base. Even though it would be relatively simple to just send emails to current users about events, we wanted a more interactive, engaging way to advertise the events. Because of this, for each event Bae hosted, Bae profiles were created with relevant pictures and description about the event. The extended background job functionality allowed for the creation of jobs that would query for thousands of
users within a radius from a certain location, automatically generate matches between those users and the event profile, automatically have the event profile sent a message to the user, and send emails and push notifications to the user as well. In some cases, the messages would include a particular Eventbrite link and a discount code that users could use for buying tickets to our events. This functionality was very effective for advertising about our internal events and the second event this was used for actually sold out after doing this. While it has been a great way to get the word out about Bae internal events, in the future, this could also be used for third party advertising.

**Overall Business Success**

Since launching Bae at Howard University on April 1, 2015, we have grown a lot as an app as well as a business. We have over 160,000 downloads with roughly 20-30k monthly active users, and as shown in Figure 4.3, the growth we had within our first month (which was 17,000 downloads) was greater than the growth attained by one of our dating app competitors, Hinge, which was about 1.5 years after their launch.
Also, over time, Bae as a company has made several partnerships with other companies and we have also been featured in a lot of press from major media outlets since we launched as shown in Figure 4.4. Bae recently got into Facebook’s FbStart Accelerate Program, which provides startups that use Facebook products with up to $80,000 in resources and direct contacts for support from Facebook engineers and marketing partners. Bae has also organically grown to become a Top 50 Lifestyle app in over 20 African and Caribbean countries. This validates not only the fact that the African Diaspora is in need of a solution for online dating, but also that Bae is well positioned to become that solution.
Next Steps

As Bae continues to grow, there are a number of things we are looking towards achieving. For one thing, due to the amount of users who crave a different means of login outside of Facebook Login, we are already working towards a new solution. Facebook recently released a new product called Account Kit, which is a framework that allows developers to build apps that support email and/or phone number login without passwords. Bae's role in the Facebook FbStart Accelerate Program allowed us to be chosen to beta test this product and start using this feature before it was released to the rest of the world. At their recent developer's conference in April called F8, Facebook even featured us on a keynote slide among many other companies who were also beta partners for this product, as shown in Figure 4.5. This functionality will allow us to reach more users and will be live in the app stores within the next month or so.

![Figure 4.5: Keynote slide shown at Facebook's F8 Developer Conference featuring Bae as a Beta Partner for AccountKit](image)

Besides login functionality, we hope to add in features that both enhance the experience and serve as a means for monetization. For instance, one could imagine being able to purchase a virtual gift to send to a potential match to increase the likelihood that the other person would decide to match with you or being able to purchase unlimited swipes. Also, it would be great to build upon
our in-app event advertising functionalities to create a third party advertisement platform that could potentially generate revenue as well.

While there are many features we would like to add in the future, a lot of focus needs to be placed on the backend both in the near and long term future. In January 2016, Facebook announced that it was shutting down our backend service, Parse, and that all apps that utilized Parse needed to go through the process of migrating their app to a self-hosted backend service such as Heroku, AWS, Microsoft Azure, etc. While Parse open sourced their framework, allowing developers to keep much of their client and server-side code, Parse shutting down also means that much of their functionalities need to be refactored such as push notifications, cloud functions, background jobs, etc. Bae’s database has already been migrated to a self-hosted MongoDB instance hosted on mLab.com, but the server-side code still needs to be refactored before transferring that to AWS, which is our current backend of choice to migrate the server to.

Last, but not least, since Bae has already had a lot of traction overseas, we would love to be able to both capture the US market as well as be able to launch internationally at scale. This would require us to really understand the market of various African countries specifically, and determine the ways in which we would need to differentiate in African that might differ to how we need to differentiate here in the US to be successful. In addition, I would need to ensure that our backend could handle potentially millions of users internationally. As difficult as this might be, being able to do this successfully could truly allow Bae to become the answer to mobile dating for the African Diaspora.
Bibliography


