Nikolay Morev
iOS Developer @ Mail.Ru
Sep 14, 2016 · 21 min read

# A record-breaking story of boosting the launch time of Mail.Ru's email app for iOS

Launch time is an important factor of long-term success for every application. And it turns critical for applications like Mail.Ru Email that users tend to launch many times a day just to check their inbox.

This article is about optimizing a code-intensive application with a long development history and lots of features, both public as well as hidden from users. Our task was to decrease the application's launch time without sacrificing the functionality.

## What can and should be optimized?

Saying "launch time" we imply the interval between a user's tapping our application's icon—and the moment when the user can perform the desired actions, i.e. those why the user has actually launched the application. The most important criterion here is the user's subjective time perception, so developers can resort to all kinds of tricks that involve no real code optimization. However, we cannot completely ignore objective timings, i.e. the elapsed time of the basic launch steps, that do affect the subjective perception too. Besides, they allow estimating the effect of specific changes upon the result.

Here we analyze the case when our application does not reside in the device's memory because opening an already launched application takes notably less time. Ideally, we must've made our test conditions even stricter to avoid the influence of various system caches, but then we would've had to reboot the device before each measurement, and the process would've been too time-wasting and harder to automate.

An application's launch time comprises two basic steps: first—the time lag from tapping the icon till the first call of the application code (these are typically *+load* methods), second—the execution time of the application code itself, including actions performed by system frameworks (launching the run loop, loading the launch screen and the main UI) and our own code.

## Loading an application

The first step includes loading the application's executable code to the device's virtual memory, loading all required system and user dynamic libraries, adjusting addresses that point to symbols inside and outside the application's executable file, initializing Objective-C runtime environment. Some system frameworks may have their own *+load* methods or constructor functions that are also executed within this step.

This step was discussed in every detail at WWDC this year: there was a great talk, 406 Optimizing App Startup Time, and you can find a summary in this blog, Use Your Loaf—Slow App Startup Times.

Here are key recommendations from that talk:

- When launching your application, use the environment variable *DYLD_PRINT_STATISTICS* (to enable it for iOS 9, activate the checkbox "Dynamic Linker API Usage") to get time statistics for all actions performed in the first step. Here is an example of such breakdown for iPhone 5 running iOS 9.2.1:

```
total time: 2.1 seconds (100.0%)
total images loaded:  309 (304 from dyld shared cache)
total segments mapped: 14, into 2352 pages with 140 pages p
total images loading time: 842.08 milliseconds (39.3%)
total dtrace DOF registration time: 0.19 milliseconds (0.0%
total rebase fixups:  310,006
total rebase fixups time: 51.52 milliseconds (2.4%)
total binding fixups: 376,990
total binding fixups time: 598.68 milliseconds (27.9%)
total weak binding fixups time: 6.55 milliseconds (0.3%)
total bindings lazily fixed up: 0 of 0
total time in initializers and ObjC setup: 639.21 milliseco
                        libSystem.B.dylib : 130.68 milliseco
              libBacktraceRecording.dylib : 1.05 millisecond
                           libc++.1.dylib : 0.16 millisecond
                           CoreFoundation : 1.74 millisecond
                                CFNetwork : 0.02 millisecond
                                   vImage : 0.00 millisecond
                          libGLImage.dylib : 0.15 millisecond
```

- To decrease *total images loading time*, use as fewer dynamic frameworks as you can. Most of them (about 200–300) are normally system frameworks shipped together with iOS, so their loading time is already optimized. But if you add your own frameworks, this may affect your application's launch time. By the way, standard Swift libraries in this respect should be treated as user rather than system ones.
- The timings of *rebase fixups*, *binding fixups* and *ObjC setup* are affected by the number of symbols in Objective-C code: classes, selectors, categories. For existing applications, it would be hard to recommend anything that doesn't require refactoring the entire application or reducing its functionality. But if you still have the chance, you can try writing most of the code in plain Swift, that is without using Objective-C runtime at all.

These recommendations can help accelerate the first step pretty much, so they're worth observing. But it would be much more challenging to analyze the second step where our own code is executed and, subsequently, we have all the power to profile, investigate and improve. Here is our story.

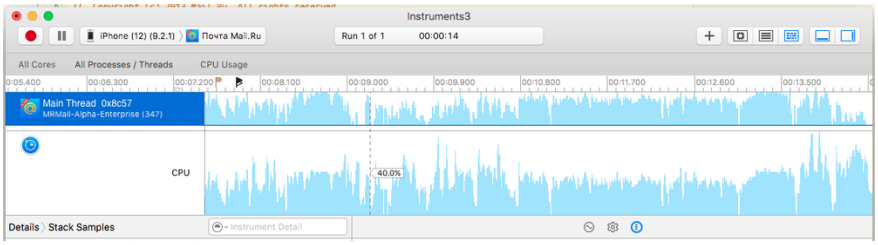
## Looking for better optimization methods

First off, to assess the efficiency of our optimizations, we needed to come up with some criteria. We decided to focus on the most popular scenario, when the user was already logged in their email account, so after launch our application displayed the user's inbox. Next, we defined a number of substeps in our application's internal logic: these substeps were hidden from users, and we wanted to get a better idea about which substeps were more time-consuming. These were: loading the application's settings, loading account data, loading a cached list of folders, displaying the email list controller view, initializing background screens, and some others.

## Time Profiler

No doubt, every competent iOS developer would turn to the Xcode Time Profiler should any performance troubles occur. That's what we started with. Our initial expectation of the troubleshooting process was a clear sequence of the following actions: we look into the profiler, notice that most of the time is "eaten up" by two or three (okay, ten at most) calls, and consolidate our efforts for optimizing those calls. Well, here is what we actually saw in the profiler:

- The call tree revealed no explicit time-eaters that we could easily sweep away. Most of the time was occupied by loading the UI from xib files, building the UI object tree, rendering the layout. We couldn't simply abandon this code since the user's aim of launching our application was to see the UI after all.
- Leaving out the UI loaders, the remaining time was evenly distributed among all substeps. Every single optimization attempt brought us a ridiculous gain here, 50ms being the top result.
- The CPU usage graph showed that our application didn't use 100% of the CPU all the time (or at least 50% that would correspond to single-threaded execution), and at times there were dips and standstills on the graph. Time Profiler could help us very little with investigating what happened at those dip-and-standstill points, and we could only consult the list of stack samples and analyze the stack traces which preceded those points. Most often, this was I/O activity and interaction with system services over XPC.




CPU usage graph

Nonetheless, Time Profiler gave us some useful insights.

First, it gave us a better idea about which code inside system frameworks, especially UI Kit, took much time to execute. Based on this, we could make decisions on using specific UI Kit features and reordering our UI initialization pipeline.

Second, after spotting the key events of the launch process, we could inspect what code was executed between those events. Without Time Profiler it was not obvious sometimes, because the launch order was determined by complex interdependencies of various subsystems and classes (from the business logic and the UI together), asynchronous callbacks and concurrent code execution. Our method was as follows: in the list of stack samples, we found samples that corresponded to the investigated event, added flags for those points on the graphs (Edit > Add Flag), selected the area under two flags and investigated the call tree.

Finally, Time Profiler allowed us to find functions and methods that were executed quickly, but throttled the overall launch process due to a big number of calls. For this purpose, we enabled the checkboxes “Invert Call Tree” and “Top Functions” in the call tree settings, ordered the call tree by “Self Weight” or “Self Count” and watched it top down, skipping irrelevant calls with the help of the options “Charge XXX to callers” or “Prune XXX and subtrees”. This way we learned that a single call to the *objc_msgSend* function cost us roughly 228ms, which was the general overhead for calling Objective-C methods, and the cost of calling *CFRelease* was 106ms, which was the overhead for memory management. We were unlikely to optimize this, but we still had chances to discover more bottlenecks with this method. This happened to be calls to *+[UIImage imageNamed:]* since we performed the appearance configuration for all our screens at startup.

## Embedded application logs

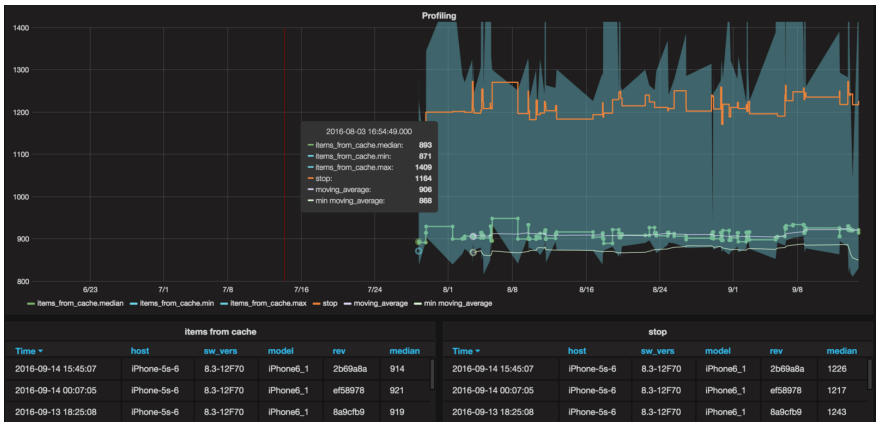
With Time Profiler, we didn't have any straightforward way to measure the total time spent for a complex sequence of asynchronous calls. So, the investigation method described above—searching the list of trace samples and adding flags—could not be automated. That's why we changed our strategy and decided to add logs at all critical points of the launch process. Each record in these logs included the name of the event, the absolute time elapsed since the measurement start, and the relative time since the previous event. Here is an example of such a log:

```
load    0   0
main    44  44
did finish launching    295 250
did init BIOD   489 194
will load accounts  1145    655
ELVC view did appear    1663    518
did load accounts   1933    269
did load cached folders 2075    142
items from cache    5547    3471
```

How did we identify where to add a log? As long as some substeps of the launch process could be performed concurrently, we paid more attention to the substeps that took more time and that had to be completed first before other substeps could be started. For example, we could load the list of database messages and the view controller xib file concurrently, but we couldn't load the list of messages from the database until we opened the database itself. To identify a critical path of key events that were to be completed, we used an Xcode feature that allowed viewing the current stack trace together with all the previous stack traces in the sequence of asynchronous calls. All we needed was to add a breakpoint at the string that we considered the end of our launch process, switch the Xcode Debug Navigator into the mode "View Process By Queue"—and here you are, a critical path from *main* or *application:didFinishLaunchingWithOptions:* to the breakpoint target!

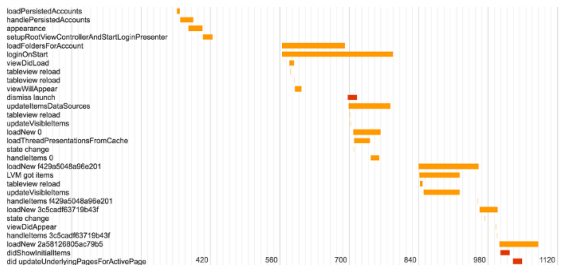
Although the logs had lower granularity than the data in Time Profiler, they offered a number of benefits.

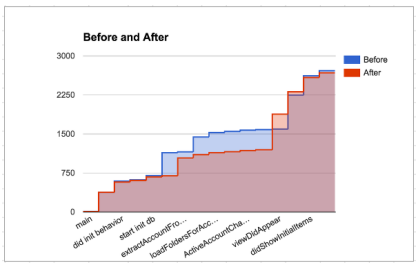
- Log-based measurements were easy to automate. This allowed assessing the effect of our changes faster, as well as including the launch speed check into our continuous integration process.

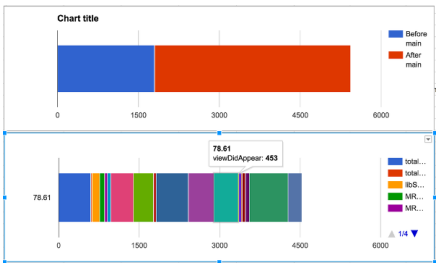



Grafana

- Perhaps, you noticed that the execution time of the same piece of code might differ tangibly from launch to launch. That's because the execution time is generally influenced by all sorts of optimizations in the operating system (like disk cache) and other process running concurrently on the same device. To make our result more stable and comparison-friendly, we made a series of similar measurements and then took the average value or the median.
- We could copy the logs to Google Sheets, and then analyze and compare them in a convenient way, and even build smart-looking diagrams that illustrated the sequence and duration of each substep.






This kind of log-based profiling has some pitfalls though. If you are in our track and notice that some substep in the sequence takes unexpectedly much time, don't rush in optimizing it. First make sure that the entire time gap is occupied solely by performing this substep. Chances are that the substep's completion is throttled by waiting for *dispatch_async* in the main queue, while the main queue is currently busy with other tasks. If you remove *dispatch_async* here, the substep may now take less time, while the total time taken by all substeps will be redistributed and will so remain the same.

Remember that the time required to execute a certain function is not enough to spot an optimization candidate. You also need to consider your application's high-level architecture. It's important to understand the interdependencies of the substeps, i.e. whether the substeps can be performed concurrently or only consecutively. Knowing this, you can get a speed gain if you parallelize consecutive substeps, or put off optional substeps to be performed later. That's where log-based profiling can be useful.

## Input/output

Reading files from disk, even when you deal with flash memory, takes a dozen times longer than when you work with RAM or CPU, so it makes sense to minimize the number of disk operations at startup. Throttling caused by reading data from disk may be displayed in Time Profiler as gaps, but more often it's not displayed there at all, so other methods are required to spot throttling.

- The "I/O Activity" instrument in Xcode (see the "System Usage" profiling template) works only on real devices. It captures I/O events and displays information about all I/O-related system calls, paths to accessed files and stack traces for the code that was executed during these operations.
- If you are unhappy with the above-suggested method, you can obtain similar information by adding a breakpoint on the *__open* function. In this case, you can obtain the name of the accessed file with the following LLDB command: *p (char *)$ro* (the name of the register where the first parameter for this call is stored will depend on the architecture).

## Layout and stuff

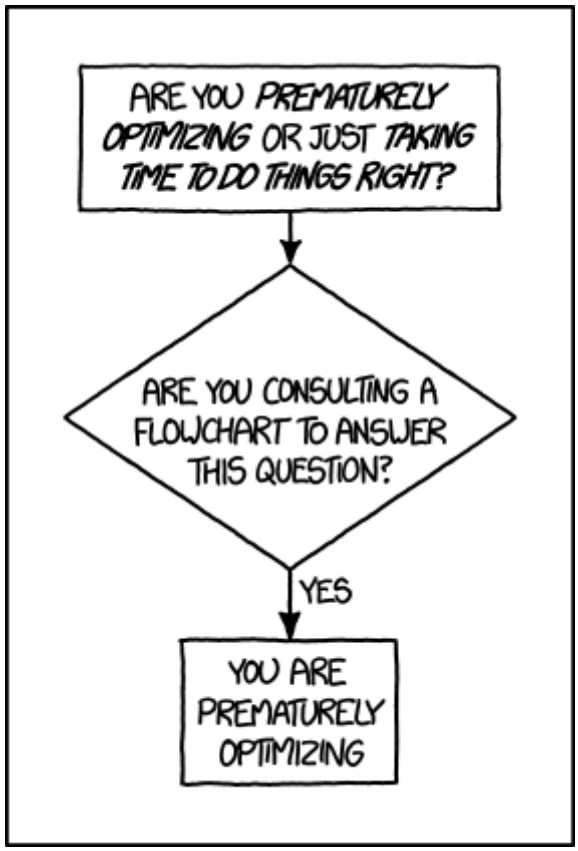
There may happen some CPU-intensive calls that can't be analyzed with Time Profiler, because their stack traces don't always contain the information indicating the part of your application to which these calls belong. As an example, these are layout calls for traversing the view hierarchy. Their stack traces often include only system methods, while we're interested to know which views have time-consuming layout calls. You can use swizzling to measure such calls and match them to the objects they manipulate. To be more specific, you can add some code before and after the call, and this code will display in console some information about processed objects and the duration of each call's execution. With this log, it's easy to create a Google Sheets table showing how the time spent for each view's layout is allocated. For swizzling, we used the Aspects library.

```
static void LayoutLoggingForClassSelector(Class cls, SEL se
    static NSMutableDictionary *counters = nil;
    if (!counters) {
        counters = [NSMutableDictionary dictionary];
    }

    SEL selector = NSSelectorFromString(selectorName);
    [cls aspect_hookSelector:selector withOptions:AspectPos
        TLLOG(NL(@"lob %s %p"), class_getName([[info instan
    } error:nil];
    [cls aspect_hookSelector:selector withOptions:AspectPos
        NSValue *key = [NSValue valueWithPointer:(void *)[i
        NSNumber *counter = counters[key];
        if (!counter) {
            counter = @(0);
```

## Optimization

Here are some optimization tips that come from our experience of decreasing the launch time of our email application. We hope that colleague developers may find these tips useful.

Now and then, as we analyze the collected performance data, we may interpret the data incorrectly. Running the wrong hare, we try to fix what actually needn't be fixed. Such optimization attempts often make the code more complicated and less reliable, increase the number of edge cases that must be considered, etc. This is why, when you are in for some optimization, remember not to act blindfold but to come up with an accurate measurement procedure that yields stable results and helps you assess correctly the usefulness of each change.




Premature optimization

Before you plunge into optimizing the launch time, remember to determine the shortest launch time possible by removing all code except those pieces that only make your application look like launched (preserving the functionality is not the aim here). In our case, the minimal application was a single screen containing a *UINavigationController* with a navigation bar and buttons, a *UITableView* and several cells.

For testing, take the application version which is as close to the App Store package as possible, disable all runtime checks, logs and asserts, and enable optimization in the package settings. In our case, the package for internal testing included swizzling for many methods to enable runtime checks for these methods in the main thread. Of course, this influenced the test results a great deal.

The first thing to optimize is the code executed in the main thread, since the aim of the entire launch process is to display something on the screen, and this can be made only in the main thread. Background threads are also worth taking into account, because the capacities of hardware parallelization are limited: most often, a device has only two cores, and CPU-intensive background code may considerably affect the total launch time. We faced this when investigating the influence of some analytics library on our application's launch time.

After initialization, this library worked entirely in a background thread, nonetheless we got an inspiring time gain after the library was disabled.

Fishing around for more ways to optimize the UI launch time, we found out that loading, configuring and displaying cells in the email list made up a fat time-eater altogether. There were several reasons for that: we had plenty of cells, each containing multiple subviews, and the main time-eaters there were icons and labels. Icons caused a slowdown because image loading was generally slower than other operations, and the slowdown with labels was due to calculating their sizes to fit the labels' content.

Icons displayed in an email's cell indicated the email's attributes (unread, flagged, has an attachment) and possible actions (displayed in an action panel on swiping the email). We changed the logic for creating the action panel and made it "lazy": the panel was not created until a swipe occurred. Then we made a similar change for the attribute icons: specific icons were not loaded if there were no emails with corresponding attributes in the list.

In fact, the "lazy load" optimization comes in handy for all actions performed at launch. Anything you don't really need for launch should be created in a lazy (or delayed) manner: all sorts of views displayed only in certain conditions, as well as stubs, icons and view controllers that remain invisible in the initially displayed UI. A good example in our case was loading images with *+[UIImage imageNamed:]*. At first glance, *imageNamed:* shouldn't have taken long because the images were actually loaded and decoded only on display. However, due to a big number of calls, their total execution time got accumulated. In our application, the visual appearance of all UI elements was configured centrally at launch (this logic was implemented in the class *AppearanceConfigurator*). Ideally, we must've carried out this configuration in a lazy manner too, but we failed to find a lean solution that would let us take away all settings into one place out of the configured view classes and could be as well applied in a lazy manner on the first hit at the corresponding view or controller. To optimize *imageNamed:*, we made two changes: we replaced the call to *imageNamed:* with a call to *imageNamed:inBundle:compatibleWithTraitCollection:*, which took slightly less time, and removed all direct calls to *imageNamed:* at launch. Instead, we delegated all configuration logic calls to proxy objects that forwarded these calls to a real *UIImage*, which was created in a lazy manner on the first call to any of its methods.

```
+ (UIImage *)imageWithBlock:(UIImage *(^)(void))block {
    MRLazyImage *lazyImage = [(MRLazyImage *)[self alloc] i
    return (UIImage *)lazyImage;
}

- (UIImage *)image {
    if (!_image && self.block) {
        _image = self.block();
        self.block = nil;
    }
    return _image;
}

- (void)forwardInvocation:(NSInvocation *)invocation {
```

Configuring the visual appearance with *UIAppearance* at launch can still prove time-consuming if you need to configure too many objects. We succeeded to cut most of this time thanks to “lazy images”.

We tried optimizing the display of *UILabel* in the cells with AsyncDisplayKit. ASDK is a fairly flexible framework, which you can use as intensively as you like: from building the entire UI to using just a few ASDK classes for particular UI pieces. We abandoned right away the idea to migrate the display logic of the entire email list to ASDK, because this re-write would go far beyond the time limits we had for optimizing our application’s launch time. Instead, we decided to start with replacing all *UILabel* objects in a cell with *ASTextNode*. We assumed that this would let us display the email cells without waiting for the labels display to be completed. Besides, we hoped that the alternative rendering implementation would work faster than the standard SDK implementation. *ASTextNode* could render text in three modes: synchronous, asynchronous, and asynchronous displaying placeholders (i.e. gray boxes instead of the text). Alas, none of these modes lived up to our expectations. Asynchronous rendering looked inattractive and user-unfriendly, because users were primarily interested to see the email message text, while the synchronous and asynchronous-displaying-placeholders modes were about as fast, or even slower than *UILabel* (which was no wonder since in both cases text rendering was based on CoreText).

The final nail in the coffin of the idea to use ASDK was the raw and unstable feeling that the framework produced. The latest ASDK version that we used for testing kept crashing and displaying screens

of log messages in console even for the simplest use cases, including those shipped together with the framework; the behavior of many *ASTextNode* attributes was very different from that of *UILabel* attributes, and so on.

Another cell-related opportunity for optimization that caught our eyes in Time Profiler was the time spent for loading a cell from xib. Of course, time after time we heard about developers who abandoned using Xcode Interface Builder for many reasons. Why? One reason was probably the low speed of creating the view hierarchy. We decided to check this hypothesis and re-wrote our entire xib content as code. Strange as it might seem, our measurements revealed that the cell creation time changed only slightly, and it grew a bit up instead of growing down. Similar results were published in the Cocoa with Love blog in the far-away 2010. In that post, you can also find a link to the project that was used for measurements and double-check the results yourself.

Still another xib-related optimization opportunity—this time it's the one that really works and is quite useful for cells—is caching *UINib* objects in the memory. The *UINib* class already has an embedded function that frees some memory if there is a memory shortage, so writing this kind of cache is easy as pie.

One more pitfall we got into was that using some standard frameworks could throttle the launch process, and this would be displayed in Time Profiler as a standstill several dozens of milliseconds long. This related to actions like working with the address book, keychain, Touch ID or Pasteboard, as well as checking all sorts of access privileges like access to the photo library or geolocation services. Where possible, we tried to move such calls to some later substep in our launch sequence, namely after the basic UI was displayed.

Authors of third-party libraries for analytics or access to various services often recommend initializing their libraries right in *application:didFinishLaunchingWithOptions:*. Before you make up your mind to follow these recommendations, check whether this affects your application's launch time, and if needed, similarly postpone library initialization until the basic UI is displayed.

If your application works with data via Core Data or reads the data directly from a SQLite database, you have a chance to reduce the launch time a little bit due to optimizing the overhead for initializing

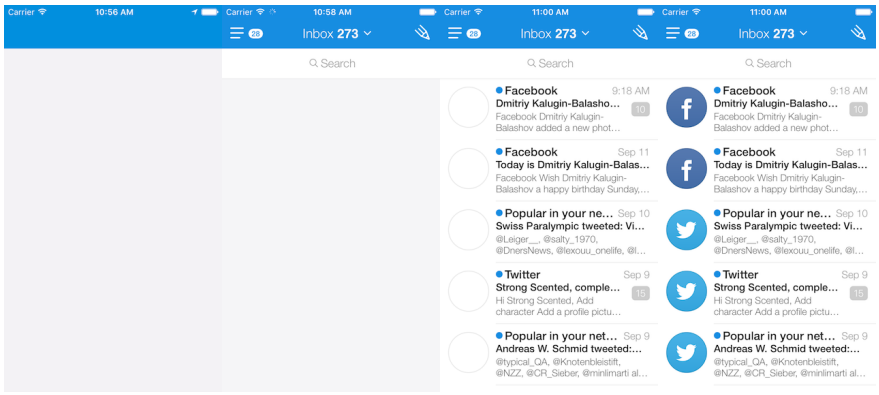
the Core Data stack or opening the database. To do so, save the minimal set of data required to display the basic UI into a file using some simple, easily parseable format and move database opening to a later substep. The main point here is to keep the balance between the gain and the cache-related sophistications introduced into the code.

How does Swift-based code influence the launch time? Our application code was mostly written in Objective-C, and we used Swift only as an experiment to implement several small-sized classes, intending to add maybe more Swift-based classes over time. Theoretically, Swift-based code could help somehow reduce the rebase/bind phase and the overhead for sending messages in the Objective-C runtime. On the other hand, Swift-based code slightly increased the launch time due to extra dynamic frameworks. We eventually decided to abandon using Swift-based code altogether when solving a different task—the one to decrease the size of the loaded application. Most likely, we'll get back to Swift again no sooner than next year, when a stable ABI arrives at last.

## Subjective perception

From the user's point of view, an application launch is completed when the user can see an updated list of emails on the screen. And the user's subjective perception of the launch speed is particularly influenced by how fast the following substeps are performed.

- Displaying the UI outline.
- Displaying the basic UI elements: folder list button, new email button (notice that these elements needn't be activated outright: the user will start using them only in a while, so there'll be time to activate all these elements after launch).
- Displaying a cached list of emails—before the application retrieves an updated list from the server.




Here are some tips for managing the user's subjective perception at launch.

- Fairly often developers abuse the following recommendation from iOS Human Interface Guidelines: "The launch screen isn't an opportunity for artistic expression. It's solely intended to enhance the perception of your app as quick to launch and immediately ready for use." We too used to abuse this recommendation, displaying a handsome Mail.Ru logo at startup, but in the newer version of our email app we put this recommendation into action.
- Be careful with animated load indicators. According to some research, animations make users blame your application for being slow.
- A common practice is to display an extra launch screen after the main screen. The extra screen helps show a load indicator, or make an animated transition from the launch screen to the basic UI, or simply keep the system from stopping the application if the initialization takes too long. If your priority is the launch speed, and you have no special reasons for an extra launch screen, consider avoiding extra screens as they require extra time to load and display.

## Measurement automation

An application's launch time tends to grow over time, gradually and humbly, as new features are added and the application code is refactored. Each time increase may be just a matter of milliseconds, so the cumulative effect gets noticeable only in a while. If you took much pain to optimize your application's launch time, you'd surely hate the idea that the extra time may humbly get back. Our way to avoid such kickbacks was building a graph that displayed our application's

launch time on each commit to the main branch of the repository, so we could easily spot all changes.

We recommend running launch time measurements on the slowest device you can find, but never using a simulator.

Our automated measurements were launched via Jenkins, similarly to all the other tasks within the continuous integration scope. For this purpose, we had a dedicated iPhone. We tried two test scenarios: with jailbreak and without. In the long run, we opted the scenario with jailbreak as it gave us the following benefits.

- No need to keep the device permanently connected via USB to one of our Jenkins servers, while the developers didn't have physical access to most of those servers. With jailbreak, it was possible to interact with the device via **ssh**.
- No need to associate tasks in Jenkins with a particular slave to which the device was connected.
- No need to install any special software for Jenkins slaves to enable interaction with the device via USB.
- Fewer problems with signatures and profiles when building an application package.

Application packages are built similarly, no matter whether you use a device with or without jailbreak. The only difference is that in case of jailbreak you have less strict requirements to signatures and profiles. We were building our Xcode project with **xcodebuild** using the "App Store Release" configuration, with logging enabled for all launch substeps. *ENABLE_TIME_LOGGER* not only enabled logging, but also let us have our application stopped automatically when the launch process was completed.

```
1  $XCODEBUILD -project MRMail.xcodeproj -target "$TARGET" -con
2      -destination "platform=iOS" -parallelizeTargets -jobs 4
3      CODE_SIGN_IDENTITY="iPhone Developer" \
4      MAIN_INFOPLIST_FILE="tools/profiler/Info.plist" \
```

We used a separate *Info.plist* to enable "iTunes File Sharing", which allowed accessing the log file in the "Documents" folder on devices without jailbreak. We also disabled the Flurry library to avoid the following pitfall: if an application was launched and stopped earlier

than Flurry had sent all its events, then after a series of such launch-and-stop actions Flurry accumulated a heap of temporary files and tried to read them all at launch, increasing the application launch time a great deal. To install our application package to a device without jailbreak, we used the **ios-deploy** utility:

```
APP_BUNDLE="$PROJECT_ROOT/build/${CONFIGURATION}-iphoneos/$P
$IOS_DEPLOY --bundle "$APP_BUNDLE" --id "$DEVICE_ID" \
    --noninteractive --justlaunch
```

To install our application on a device with jailbreak, it was enough to archive the *.app* folder as *.ipa* and copy it via **ssh** to the device (in our script, we indicated *localhost* as the device address, because **ssh** worked via a tunnel). After that, we installed our application using the **ipainstaller** utility from Cydia.

```
APP_BUNDLE="$PROJECT_ROOT/build/${CONFIGURATION}-iphoneos/$P

cd "$PROJECT_ROOT/build"
rm -rf Payload; mkdir -p Payload
cp -a "$APP_BUNDLE" Payload/
rm -f MRMail.ipa; zip -r MRMail.ipa Payload

```

For measurements, we used a test scenario with an already enabled account, so the very first installation of our application was done manually in order to log into the test email account. Fortunately, we had to do it just once, during the initial device setup, so we didn't even try to automate this.

Next, we launched our application as many times as needed. After each launch, we downloaded the log file from the device and saved it for further analysis. To launch our application on a device without jailbreak, we used the **idevicedebug** utility, and to download the results via USB, we used the **ifuse** utility.

```
for i in $(seq 1 $NUMBER_OF_RUNS)
do
    $IDEVICEDEBUG --udid "$DEVICE_ID" run "$BUNDLE_ID" >/de

    COMPLETION_PATH="$MOUNTPOINT_PATH/$COMPLETION_INDICATOR
    LOG_PATH="$MOUNTPOINT_PATH/$LOG_NAME"

    for j in $(seq 1 5)
    do
        sleep $MOUNT_SECONDS_PEDIOD
        $UMOUNT "$MOUNTPOINT_PATH" 2>/dev/null || true
        $MKDIR "$MOUNTPOINT_PATH"
        $IFUSE --documents "$BUNDLE_ID" --udid "$DEVICE_ID"
        sleep $AFTER_MOUNT_SECONDS_PEDIOD

        if [ -f "$COMPLETION_PATH" ] && [ -f "$LOG_PATH" ];
            break
```

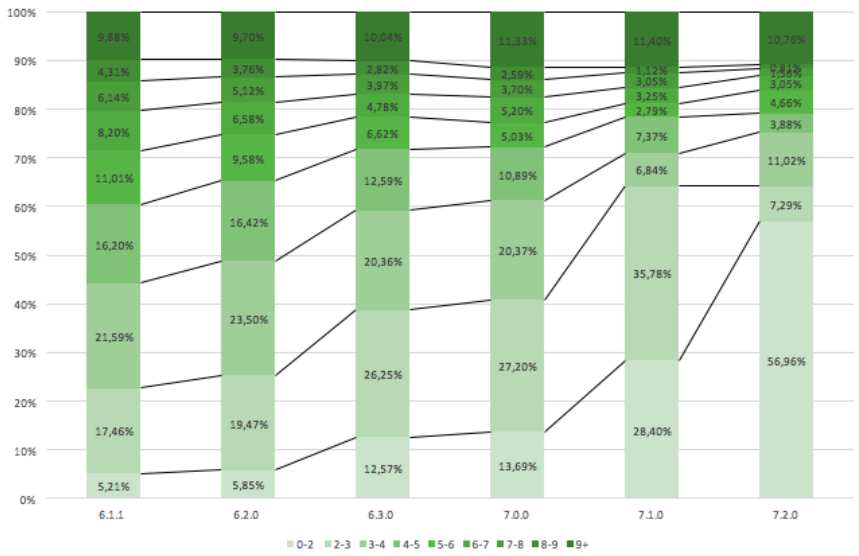
Things were somewhat simpler for devices with jailbreak. To launch our application, we used the **open** utility from Cydia and then copied the results via **ssh**.

```
SANDBOX_PATH=`SSH_TO_DEVICE ipainstaller -i "$BUNDLE_ID" |
SANDBOX_PATH="${SANDBOX_PATH//[$'\t\r\n ']}"
COMPLETION_PATH="$SANDBOX_PATH/Documents/$COMPLETION_INDICA
LOG_PATH="$SANDBOX_PATH/Documents/$LOG_NAME"

for i in $(seq 1 $NUMBER_OF_RUNS)
do
    SSH_TO_DEVICE open "$BUNDLE_ID"
    sleep $MOUNT_SECONDS_PEDIOD

    for j in $(seq 1 5)
    do
        if SSH_TO_DEVICE test -f "$COMPLETION_PATH" && \
            SSH_TO_DEVICE test -f "$LOG_PATH"
        then
```

Further on, a trivial Ruby script collected the launch results, calculated statistics for all events (minimal time, maximal time, average, median, quantiles) and sent the data to Influxdb, which then represented the data as a graph in a Grafana-based dashboard.

Measurements on real devices are non error-prone. Our question was: how many times should we run the measurements to obtain a stable result with a specified error? We used a formula for sample size determination. Parameters for this formula were calculated based on 10,000 measurements (in our case, the result was roughly 270 measurements), but even after 10 measurements the error was acceptable to let us notice an increasing trend over a big time gap.

Measurements embedded into our continuous integration were of limited value, however, because they reflected only one of all possible launch scenarios for only one device/IoS-version configuration. To get ampler statistics, we also collected them from real users to get an idea about the whole variety of cases. To compare the launch time of different application versions, we grouped all launches by the number of seconds.




Statistics

## Results

In the course of our optimization campaign, we decreased the real launch time of our email application by about one third and improved the users' subjective perception of the application's launch speed, which was backed up by an improved retention rate. But an even more important achievement was the mechanism for controlling and avoiding launch time kickbacks in future.