Toheed Jahan Khan

Non-Functional Requirements in Mobile Applications

NFR

Table of Contents

[Why NFR is needed? 2](#_Toc114759276)

[What are non-functional requirements? 3](#_Toc114759277)

[What are the key types of non-functional requirements? 5](#_Toc114759278)

[Performance 5](#_Toc114759279)

[Scalability 6](#_Toc114759280)

[Responsiveness 6](#_Toc114759281)

[Use-ability 6](#_Toc114759282)

[Reliability 6](#_Toc114759283)

[Security 7](#_Toc114759284)

[Availability 7](#_Toc114759285)

[Screen Adaption 7](#_Toc114759286)

[Network Coverage 7](#_Toc114759287)

[Accessibility 7](#_Toc114759288)

[Extensibility 8](#_Toc114759289)

[Localization 8](#_Toc114759290)

[Portability 8](#_Toc114759291)

[Compatibility 8](#_Toc114759292)

[Maintainability 8](#_Toc114759293)

[Restaurant Business Example 9](#_Toc114759294)

[Some Important factors to consider before finalizing NFRs 10](#_Toc114759295)

[CPU & Memory 10](#_Toc114759296)

[Network Conditions 10](#_Toc114759297)

[Screen resolution and screen form factors 12](#_Toc114759298)

[Battery usage 13](#_Toc114759299)

[Support for operating system versions 14](#_Toc114759300)

[Media capabilities 15](#_Toc114759301)

[Sensor characteristics 15](#_Toc114759302)

[Maximum Sample rate: 15](#_Toc114759303)

[Sensitivity: 15](#_Toc114759304)

[Non-Functional Testing Aspects of Mobile Apps 16](#_Toc114759305)

[Performance testing: 16](#_Toc114759306)

[Stress testing: 16](#_Toc114759307)

[Load testing: 16](#_Toc114759308)

[Volume testing: 16](#_Toc114759309)

[Usability testing: 16](#_Toc114759310)

[UI testing: 17](#_Toc114759311)

[Compatibility testing: 17](#_Toc114759312)

[Documentation testing: 17](#_Toc114759313)

[Conclusion 17](#_Toc114759314)

*Imagine you’re buying a motorcycle. What features do you have in mind? Do you expect it to travel at 170 miles an hour and not to fall apart? Can you attach a sidecar to it or expand luggage space by attaching a pull-behind trailer? And let’s not forget about security systems. While these requirements don’t directly describe the vehicle’s primary function – delivering a person from point A to point B – they are still important to satisfy your needs as the driver.*

*Like motorcycles or any other kind of machinery, any software product has its own non-functional requirements. Be it a website, a mobile, or a desktop app, it should have a set of quality attributes to meet end-user needs.*

The correct specification and adherence of non-functional requirements similarly plays at least an equal, if not a greater role in the success of mobile applications.

This is due to the following reasons:

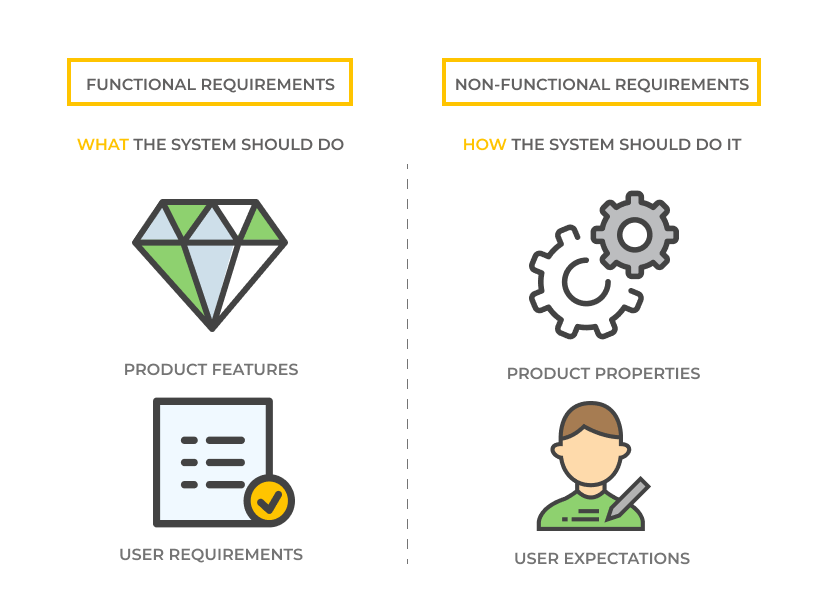
Mobile devices are uniquely constrained in several aspects such as multi-tasking support, available network bandwidth, screen real estate etc. These constraints translate into strict bounds being imposed on the operating characteristics of an application running on a mobile device.

Mobile applications need to operate successfully (or degrade gracefully) within a wide spectrum of operating conditions, such as a range of supported screen resolutions and form factors, network bandwidth situations and network types (2G, 3G, 4G, WiFi) etc.

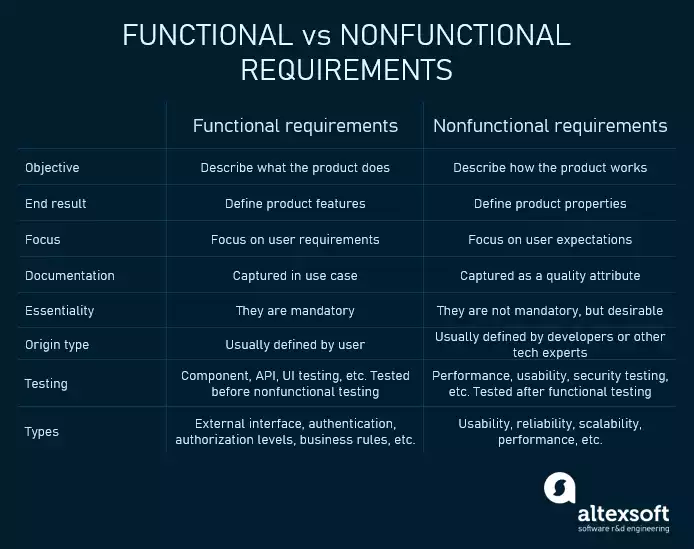
Mobile applications sometimes need to interact with the device’s sensors such as GPS, accelerometer, the ambient light sensor, camera etc. The application must respect the sensor’s operating characteristics such as its operating range, sensitivity, accuracy, minimum polling interval etc.

# Why NFR is needed?

Well you can develop application only with Functional requirement but without NFR the product will be buggy , non-reliable and incomplete.



Here’s a tabulated look at key differences between functional and non-functional requirements:



# What are non-functional requirements?

By definition, a non-functional requirement is a requirement that indicates how the system performs a particular function.

While functional requirements describe the functionality expected from a software system, non-functional requirements (NFRs) describe how well the system should function. NFRs are quality attributes of a software system.

You can deliver an application to an end-user with the functional requirements alone. It will deliver functionality even if you don’t meet the NFRs. However, the application might not deliver a good user experience.

When non-functional requirements are well defined and executed, they can make the system easy to use and enhance its performance. Non-functional requirements are product properties, so they mostly focus on user expectations.

Non-functional requirements like “The feature must not compromise security” or “Performance metrics can’t fall below X”. These non-functional requirements usually fall under the quality domain.

If defined well, NFRs can be a great checklist for quality and performance for any software.

Non-functional requirements are requirements that define ‘how’ the app must perform a certain function. In essence, they are the quality attributes of an app that define the user experience of the app. They are also known as non-behavioral requirements and are to be implemented according to their priority to the app function. This makes them flexible to an extent, making it possible to skip a few in case of time, budget or technology constraints.

Some examples of non-functional requirements would include:

* Loading speed
* Time taken to deliver server response
* User response time
* Data consumption limits
* Database security should be HIPAA-compliant.
* Users should be able to reach their profile data from any page within three clicks.
* The system should require users to enter their passwords every 60 days.
* The system must accommodate a minimum of three million concurrent users.
* Time to put this all in perspective. For example, the functional requirement for a food ordering app would include:
* Displaying restaurants in the area
* Displaying menus
* Placing orders
* Processing transactions, etc.

Non-functional requirements of the food delivery app would include:

* Refresh restaurant listings every 5 minutes
* Load menu within 10 seconds of clicking on menu button
* Order confirmation must be sent within 2 seconds of payment

As you can tell, functional requirements are indispensable features of an app that must be included in the development. Does that mean though, that non-functional requirements aren’t crucial after all?

Non-functional requirements are in fact extremely important for the app’s performance and user experience. If an app’s functional requirement is to pull up a list of sushi restaurants in the neighbourhood, but the app takes over 30 seconds to do so, the user is likely to abandon the app, even though the functional requirement was ultimately fulfilled.

So to make sure that an app development project is successful, within budget and on time, having a good balance of functional and non-functional requirements is important. This way, if push comes to shove and some app attributes need to be dropped, you can efficiently decide which ones to keep and which can be excused for the moment.

While NFRs don’t concern themselves with user stories, they are important in software engineering. You need good requirements engineering skills to capture NFRs effectively. Gathering NFRs can be hard.

While they fulfil long-term user needs, there’s a high degree of subjectivity in NFRs. The architect needs to explain NFRs clearly to the development team.

# What are the key types of non-functional requirements?

Non Functional Requirements

Scalability

Usability

Performance

Portability

Availability

Localization

Compatibility

Reliability

Maintainability

Security

Accessibility

Responsiveness

## Performance

How fast does the system return results? how many simultaneous users or transactions the system is to service and its response time.

When it comes to performance, the team should examine the speed and responsiveness of the app, its CPU and memory usage, how it contributes to battery drain as well as the app’s utilization of on-device and memory card storage. Tester should notice the loading of the images, content, lists with in the app. It is preferred to have a quick and easy app with less functionality than a slow and difficult application with many features.

The performance of the Application can be determined by it responsive time , time to complete the given task.

For example when Application is made to start up it shouldn’t take more than 3 second to load initial screen. Also it should be made sure that app will not hindrance to the user Input.

## Scalability

How much will performance change with higher workloads? Ways to expand the system and avoid

adversely affected performance. It should determine if the system will be able to handle future needs.

App should able to adopt itself to increased usage or able to handle more data as time progress.

For example when the user data(caches , stored data etc) increases app should be capable of handling them without delay by optimising the way storage is done and accessed

## Responsiveness

Application should be responsive to the user Input or to any external interrupt which is of highest priority and return back to same state.

For example :- When app gets interrupted by call , then app should able to save state and return to same state/ page which was there before it got interrupted.

## Use-ability

How easy is it for a customer to use the system?

Ease of use and user-friendly interface, that allow users to seamlessly interact with the product.

User should be able to understand the flow of App easily i.e. users should able to use App without any guideline or help from experts/manuals. If user experience needs to be explained then its not good UX.

There are many types of usability criteria. One of the most popular is by [Nielsen Norman Group](https://www.nngroup.com/articles/usability-101-introduction-to-usability/) that suggests evaluating usability with five dimensions:

**Learnability**: How fast is it for users to complete the main actions once they see the interface?

**Efficiency:**. How quickly can users reach their goals?

**Memorability**: Can users return to the interface after some time and start efficiently working with it right away?

**Errors**: How often do users make mistakes?

**Satisfaction**: Is the design pleasant to use?

Example of usability requirements:

The error rate of users submitting their payment details at the checkout page mustn’t exceed 10 percent. With that in mind, consider how to make these requirements measurable.

## Reliability

How often does the system experience critical failures?

App behaviour in case of alarm status, for example, automatic restart and operation recovery.

The application should be reliable to perform the business , i.e. when user perform some important action it should be acknowledged with confirmation.

Reliability specifies how likely the system or its element would run without a failure for a given period of time under predefined conditions. Traditionally, this probability is expressed in percentages. For instance, if the system has 85 percent reliability for a month, this means that during this month, under normal usage conditions, there’s an 85 percent chance that the system won’t experience critical failure.

As you may have guessed, it’s fairly tricky to define critical failure, time, and normal usage conditions. Another, somewhat simpler approach to that metric is to count the number of critical bugs found in production for some period of time or calculate a mean time to failure.

Example of reliability requirements:

The system must perform without failure in 95 percent of use cases during a month.

## Security

How well are the system and its data protected against attacks?

App operation and use of safety requirements related to access control, private data processing, and external attack risk reduction.

Security is a non-functional requirement assuring all data inside the system or its part will be protected against malware attacks or unauthorized access. But there’s a catch. The lion’s share/largest part of security non-functional requirements can be translated into concrete functional counterparts. If you want to protect the admin panel from unauthorized access, you would define the login flow and different user roles as system behaviour or user actions.

All the app data should be secured and be encrypted with minimum needs so that it’s protected from outside environment also from internal attack.

For example :- All authentication token should be saved on local device for comparison and need user permission to gain access

## Availability

Availability describes how likely the system is accessible to a user at a given point in time. While it can be expressed as an expected percentage of successful requests, you may also define it as a percentage of time the system is accessible for operation during some time period. For instance, the system may be available 98 percent of the time during a month.

There should be a common place where the user can access your application to install and look for regular updates give feedback

For example :- Apple’s App Store and Googles Play Store

The above mentioned points cover only few important areas of the Non Functional Requirement Apart from this there are much more concepts to be covered and need to be taken care of Some of honours mentions are

## Screen Adaption

Now a days lot of mobile devices comes with different screen sizes and layout, So your application should be able to render it’s layout to different screen sizes. Along with automatic adjustment of Font size and image rendering.

## Network Coverage

As we all know all Apps work well with Wi-Fi but also care should be taken care to handle slow connection while experience Wi-Fi black spots or when connected to mobile Network. App should be able to look out for Wi-Fi if not available then automatically switch to mobile network.

## Accessibility

It is a feature which makes physically challenged people make use of your Application.

## Extensibility

Requirements for app extensibility in case there is a need to add new functional requirements.

## Localization

The localization attribute defines how well a system or its element falls in line with the context of the local market-to-be. The context includes local languages, laws, currencies, cultures, spellings, and other aspects. The more a product sticks with it, the more success it should have with a particular target audience.

Example of a localization requirement:

The date format must be as follows: month.date.year.

## Portability

Portability determines how a system or its element can be launched within one environment or another. It usually includes hardware, software, or other usage platform specifications. Put simply, it establishes how well actions performed via one platform are run on another. Also, it prescribes how well system elements may be accessed and may interact from two different environments.

Example of portability requirements:

A app running on iOS 12 must be able to run on iOS 13 without any change in its behaviour and performance.

## Compatibility

Compatibility, as an additional aspect of portability, defines how a system can coexist with another system in the same environment. For instance, software installed on an operating system must be compatible with its firewall or antivirus protection.

Example of compatibility requirements:

The iOS application must support iPhone devices running on OS versions:

11, 12, 13

Portability and compatibility are established in terms of operating systems, hardware devices, browsers, software systems, and their versions. For now, a cross-platform, cross-browsing, and mobile-responsive solution is a common standard for web applications.

Portability non-functional requirements are usually based on preliminary market research, field research activities, or analytics reports on the types of software and devices the target audience has. If you are working within a corporate environment and the software will be accessed through a documented list of devices and operating systems, it’s quite easy to define compatibility and portability.

## Maintainability

Maintainability defines the time required for a solution or its component to be fixed, changed to increase performance or other qualities, or adapted to a changing environment. Like reliability, it can be expressed as a probability of repair during some time. For example, if you have 75 percent maintainability for 24 hours, this means that there’s a 75 percent chance the component can be fixed in 24 hours. Maintainability is often measured with a metric like MTTRS — the mean time to restore the system.

Example of maintainability requirements:

The mean time to restore the system (MTTRS) following a system failure must not be greater than 10 minutes. MTTRS includes all corrective maintenance time and delay time.

## Restaurant Business Example

Let’s take a real life Example to further understand the importance of the Non-Functional Requirement

Consider you are in to restaurant business and you have an idea for mobile Application and defined all its feature and business logic now it time to cross verify it with above points.

**Performance**

When user opens the app the app should able to load food menu with in 5 seconds with all thumbnail images , you doesn’t want to make hungry customer wait for app to respond for long time.

**Scalability**

Your application should be able to show or recommend the user of the dishes they liked or previously ordered. So that they can make a quick check out. Also user doesn’t need to go through entire menu to order his regular item.

Also Application should be able to recommend similar dishes base on user previous order

**Responsiveness**

When user selects any food it should be easy for him to add to the cart and mention the quantity of the food , also customise if available with few touches and this should happen in fewer seconds or instantaneous with user touch.

**Reliability**

When user is done with selecting the menu and proceeding to check out there should be a way for user to see summary of order and once transaction complete he or she should get confirmation via text or notification.

App should not make a wrong order to restaurant end.

**Availability**

User should be aware that your restaurant application and can be able to install from restaurant website or redirect to store(App Store or play store) to download application.

User should be able to rate the app and contact necessary person via app.

**Security**

Users info like personal contact , payment methods should be protected and should not be accessible to unauthorised personals and also there should not be a way for user to manipulate the application for their gain or bypass necessary means.

# Some Important factors to consider before finalizing NFRs

## CPU & Memory

When an application s developed to run on a particular software platform such as Android, iOS etc. it can in theory be installed and run on any device that supports that OS platform. However, for any given OS, the supported devices could have a very wide range of capabilities in terms of CPU speed and available memory. For example, an Android app can be theoretically run on both a low end Android phone having limited RAM and a single core CPU, and on a high end device sporting several gigabytes of RAM and an Octa-core processor. However, if the app has been designed to require a certain minimum CPU power and memory, it will in practice, fail on the low end Android phone. If your application makes extensive use of arithmetic and logic operations such as those involved in streaming and decompression of audio and video and in rich animations, or it lets the user view and manipulate large sets of information or images, you should specify minimum CPU and memory requirements for the application in exactly the same way it is done for desktop applications.

To determine these requirements objectively, you can evaluate each feature in your application from the point of view of its CPU and memory consumption by running it in a profiling tool, and accordingly arrive at a lower bound for the entire application from this analysis.

Here’s a list of the CPU and memory profiling tools you can use for some of the common smart phone platforms:

**iOS**: The XCode development environment comes with a rich suite of performance measurement, analysis and monitoring tools under the Performance Tools package. Chief among them is the Instruments performance measurement tool, and several analysis and monitoring tools that will together provide you everything you need to measure, analyse and monitor performance at the finest grain in memory, CPU, network usage and much more.

**Android**: Android Developer Tools (ADT) comes with the Dalvik Debug Monitor Server (DDMS) which will show you a variety of information about the running state of the Android app such as thread and heap information, process information, the LogCat display etc. Additionally, you can use the Traceview and dmtracedump tools to trace method calls and track which methods your code has spent most time in. The Systrace tool will let you see the bigger picture on your app’s performance by showing the performance characteristics of your application’s processes alongside the Android system processes on a common timeline. You can use the very powerful HPROF tool or DDMS itself to analyze heap usage and see which method calls are contributing the most to heap usage.

## Network Conditions

There are four things to consider in this area:

Support for different network protocols: Mobile devices can communicate with the network on one or more protocols such as SMS, USSD, Wi-Fi, EDGE, UMTS, LTE etc. Certain functions in your application may not perform well (or not perform at all) on certain protocols. If you wish to support only the high capacity protocols such as Wi-Fi & LTE, you will risk shutting out the low-end devices which may not support these protocols. On the other hand, if you do not specify which are the required (or recommended) protocols for your application, it may result in the users inadvertently trying your application’s high-bandwidth features on a low-bandwidth protocol such as EDGE, thereby frustrating themselves out of the application.

For a networked application, it is crucial that you evaluate each network-enabled feature of the application from the point of view of determining the protocol that it is most likely to require. For features such as streaming media, you may want to require the user to switch on a high capacity protocol such as Wi-Fi, UMTS (3G) or LTE (4G). Alternately, you may design the feature to degrade or fail gracefully, if the user operates it on a low-capacity protocol. For example, you may want to serve the media in a low bandwidth format such as 3GP over EDGE and in a high bandwidth format such as MP4 over a WiFi or a 3G/4G protocol. A Twitter client, on the other hand can be written to work just as efficiently on a very low capacity protocol such as SMS (assuming you have a SMS translation gateway in place that your application can talk to), or a high capacity channel such as WiFi. However, when operating over the SMS channel, you should turn off the transmission or receipt of anything that is not pure ASCII text, thereby ensuring a graceful degradation.

Finally, be sure to inform the user if a particular feature is likely to degrade, or not be available on a certain channel. Also inform them about additional usage charges if choosing to operate over certain channels such as SMS so as to prevent a billing surprise.

Signal drop or signal strength reduction: Be sure to evaluate each network-enabled feature in your application in the situation, where the protocol over which it is operating becomes unavailable or its signal strength reduces. Ensure that the feature is either network-fault tolerant or degrades or fails gracefully in such a situation.

Network protocol transition: This network condition deals with the mobile application’s behavior when the device transitions from one protocol to another. A classic scenario is one where the user walks out of their office building and the device transitions from WiFi to 3G. If there is a transaction underway in the mobile application when this happens, how will the application react to such a protocol transition? It is useful to evaluate the application’s functionality in such situations and design it for a seamless transition to the new protocol or for a graceful degradation or failure.

Support for multiple protocols: This network condition relates to the mobile application’s behavior when there are multiple network protocol radios active at a time. Note that most smart phone operating systems today automatically prefer WiFi if it is available, rather than using cellular data (e.g. EDGE/UMTS/CDMA2000/LTE). Android and iOS display this behaviour. Windows Phone 7 had a bug that prevented WiFi from being given a preference but that appears to have been fixed now in later versions. In the older versions of Blackberry (definitely up through Blackberry OS 6), one could programmatically specify that in one’s app that WiFi was desired.

In general, you will not have this option today to choose a specific radio – Wi-Fi or cellular – in your app.

However, note that the SDKs of most major mobile platforms provide a means to register for listening to one or more network related events or notifications when a network protocol radio becomes available or unavailable or when the signal strength on that protocol changes. You should register for such notifications and perform network condition handling inside the network listeners. Also, program all network calls defensively via extensive use of exception blocks and return value checking so as to ensure a graceful degradation or failure of the call.

Interrupts, notifications and multi-tasking

When a phone call, SMS or some type of notification (such as a calendar notification) arrives, your mobile device will usually inform your application of this event. If the user chooses to respond to the event, the OS may either background your application or, in case of non-multi-tasking mobile operating systems, simply terminate your application. In either case, the OS will usually give your application a chance to respond to the pause, background or termination event by invoking a handler method that you should implement.

It is important that your mobile application handles the interrupt in such a way that:

It does not come in the way of the OS’s processing the user’s decision to respond to the interrupt (such as accepting a call or reading an SMS), and

It does not result in any damage to your application’s ability to function normally after the OS ‘foregrounds’ i.e. resumes your application after the user finishes handling the interrupt or after they choose to ignore the interrupt.

You should evaluate each feature in the application from the point of view of determining how it would, and should function if the application gets sent into the background by the OS, or made dormant while that feature is executing, and how it will recover from this interrupt condition after the OS or the user brings the application back into the foreground after servicing the interruption.

## Screen resolution and screen form factors

If you design your app for only low resolution screens, one of two things will happen if a user runs it on a high resolution screen phone – either the high resolution phone will not apply automatic pixel doubling, in which case, your app will occupy a tiny area of the screen of the high resolution phone, or the high res phone will apply pixel doubling in which case your app’s UI might look odd in places where images and fonts are unnaturally large. Either way, things are likely to look bad, even though, technically the app functions correctly. Similarly, an application designed for a high-resolution (high Dots Per Inch – DPI) screen may have a major part of its UI go off screen on a low DPI screen phone, if the OS does not do automatic pixel filtering, leading to quite an annoying experience at best.

Open platforms such as Android that runs on hundreds of different types of phone and tablet devices are particularly susceptible to screen DPI issues.

Hence it’s important to keep a set of target screen resolutions in mind while designing the user interface of a mobile application. Practically, you will need to identify one or more screen resolution (or alternately screen size) families that you want to support. For example, let’s say you want your app to support all kinds of popularly available display sizes. It is useful to divide this wide range into a) 4.x to 6.x inch displays, b) 7.x to 9.x inch mini tablets and c) 9.x to 12.x inch full size tablets. Now, find out the screen resolution range for each family size. Finally, identify one or two representative devices in each family that you want to test the app on.

Normally, you will find that the following kinds of changes will be needed for supporting different Screen resolutions

**Font adjustment**: The text font size may need to be adjusted up (for high resolution screens) or down (for low resolution screens) so as to keep the text readable.

**Layout tweaks:** The layout may need to be adjusted to increase or decrease the spacing between and around labels and widgets shown on the screen so as to prevent them from getting clustered together on high-res screens or spaced apart too much on low-res screens.

Image changes: Background images or background art may have to be provided in two different versions: a large size/high resolution version and a small size/low resolution version so that it properly fills the amount of physical space available on the screen.

Most smart phone SDKs have a fairly well-defined methodology for specifying which screen sizes/resolutions your app supports and then a way to specify the imagery, fonts and layouts for each screen size family. Your app should make full use of these OS and SDK provided mechanisms to support different screen sizes  of devices running that particular mobile OS.

Touch and non-touch screens: If you are supporting platforms such as Symbian and Blackberry that can work on both touch, non-touch and dual-input i.e. physical keyboard + touch devices, you should ensure that touch is enabled for all features in your application when the application is operating in touch screen mode. There is usually API support available to your application to detect whether the application is operating on a touch enabled screen.

Portrait and Landscape modes: Avoid adding the complexity of supporting both portrait and landscape modes if your application does not need to work in both modes. However, if you decide to add support for both modes, try to add this support for all screens in the application and not just some of the screens so as prevent a usability surprise.

## Battery usage

On mobile devices, the battery is a scarce and valuable resource.

There are three major sources of battery drain in a mobile device:

* The main processor,
* The device’s screen (and therefore also the Graphics Processor a.k.a. the GPU), and
* The communications processors (such as the ones used for interacting with the cellular network, WiFi, GPS and Bluetooth at regular intervals).

On a mobile phone, the battery should remain maximally available for the phone application i.e. for making and receiving phone calls. Your application may therefore fall by the wayside or even get uninstalled by the user, if it drains too much battery. The definition of what constitutes ‘too much’ may vary depending on the nature of the application and the hardware capabilities of the platform it is running on. A good way to evaluate the battery usage characteristics is to evaluate your application against a ‘control’ application which is usually a ‘well-known application’ in its class. For example, if you write a universal Instant Messaging app for Android, you may want to evaluate its battery usage against that of something like the Nimbuzz IM app or WhatsApp.

There are three main battery power tests that you must perform:

Normal use test: Start on a full battery and use the application for 6-12 hours and measure the battery level at the end of each ½ or 1 hour. You may use an automated testing tool to do this so as to keep the test running for the required time interval. This test will tell you how quickly your application drains the battery when in ‘normal’ use, with all the foreground and background features of the application running normally.

**Idle run test**: Turn off the screen lock and power saver modes on the device. Then start on a full battery. Keep your application running on its main, home or dashboard screen as appropriate, and measure the battery level at ½ or 1 hour intervals. This test will measure the battery drain due to such things as intentional or unintentional automatic screen refreshes, and due to the background threads or services running in your application.

**Screen lock test**: Perform the idle run test again but with the device screen in locked mode. This will allow you to test if your application is consuming any CPU and/or network resources (and hence also the battery) when the app is not viewable to the user.

In each case, you may want to plot the battery level against time so as to get a quick visual indication of the overall trend. Perform the same tests for the ‘control’ application in order to see how your application’s battery drain characteristics compare with that of the control application.

While performing the battery tests ensure that no other application is running on the device and turn off the telephone feature so as to avoid receiving a call while the test is underway. If possible also turn off applications such as email and chat clients that tend to run in the background.

## Support for operating system versions

Most of the commonly used mobile phone operating systems are designed so as to be “forward compatible”. The OS writers try hard to ensure that as long as your application uses the official developer SDK in a manner prescribed by the SDK documentation, your application that is written for OS version 1.0 will function correctly on version 2.0 and so forth. Of course beyond a point this forward compatibility breaks down and the application creator needs to at least recompile the application for a later OS version. Users tend to upgrade the OS on their mobile devices very frequently. Therefore forward compatibility is a very important thing to ensure in your mobile app. If your app stops working after the user upgrades their mobile device to a newer OS version, the user may simply uninstall the app and not bother to download the newer upgraded version, even if it’s available.

To ensure that your mobile app continues to run on later OS versions you should follow the same conventions and so as to ensure compatibility across OS versions: viz:

* Avoid extensive use of third party libraries,
* Avoid use of deprecated APIs,
* Use best practices in using APIs and avoid non-standard usage of API methods,
* Test your application on all OS versions that you claim to support,

If using features specific to an OS, ensure that they degrade or fail gracefully on previous OS versions.

Finally, every time your app starts, check whether a newer version is available and ask the user to download the same (or do an automatic update from within the app). Normally, the app store will inform the user via a push notification whenever a newer version of your app is available on the store. However, you may additionally want to proactively perform the version check test anyways upon application start. If you notice that the user is too far behind in the app version from what is the latest version available on the app store, you may want to force the user to upgrade the app before letting them use it.

## Media capabilities

The mobile device’s operating system and the device’s hardware that your mobile application runs on, will usually determine what audio and video capabilities your application can provide. For example, which audio/video formats (such as 3GP, MP3, AAC, AAC+ etc) your application supports, whether it will support full HD video or not, whether it can support multi-channel stereo surround sound etc., are all decided by the mobile OS’s capabilities and the characteristics of the device’s hardware.

Before you decide to support specific audio/video features, ensure the following:

Be sure to find out if the version of the mobile operating system that your app will run on, contains support for any specialized media capabilities such as surround sound that your app wants to support. Target your app at only versions at and above that version. In most cases you will also need programmatic access to these features, therefore also ensure that there is SDK support available to operate these features from within your application code.

If the OS supports the feature but there is no out-of-the-box support present for them in the OS’s SDK, ensure that there is strong support available via third party libraries, or be prepared to incur the often substantial cost of rolling out support by writing your own software library.

Finally, ensure that the mobile hardware you are targeting the application for has the hardware capability for running the feature you wish to have.

## Sensor characteristics

A large number of phones (even low end feature phones) come with a camera, which can be considered to be a type of sensor. Smart phones and tablet devices typically contain many more kinds of sensors such as GPS, accelerometer, gyroscope, ambient light & proximity sensor and also the ability to connect to an external sensor via USB or Bluetooth. If your mobile application makes use of a sensor to receive information, you need to pay attention to at least the following characteristics of the sensor:

Maximum Sample rate: This is the maximum rate at which your application code can ask the sensor for a useful reading. Sometimes, this value is expressed in Hertz – in this case a measure of readings per second (as with the accelerometer sensor), or in the form of a minimum delay between two consecutive readings. This value may be available via the sensor’s API.

Operating range: Most sensors have a maximum range beyond which they will not provide a useful value. For example for an accelerometer, this value will be specified as +/- X g which means that if your application tries to use the sensor in situations where the g-force is more than +/- X, the sensor will not provide you with a useful measurement.

Sensitivity: This is a measure of how much the sensor’s reading changes in response to changing conditions. Sensitivity information is normally not available directly to application developers. The sensor’s sensitivity also usually changes over the operating range of the sensor, making it a complex parameter to work with. Normally, as long as your application respects the recommended operating range of the sensor, you will get the best possible sensitivity from the sensor.

Accuracy: Be sure to know about the accuracy of the sensor you are using. APIs will sometimes provide this value to you for certain sensors. There are also mobile applications available that a user can install on their mobile device and which will enable the user to calibrate a sensor so as to improve its accuracy. If your mobile application is critically dependant on a particular sensor, you may wish to advise the user to calibrate it before they use your application.

# Non-Functional Testing Aspects of Mobile Apps

The term “non-functional testing” refers to testing those aspects of a software application, that may not be connected with a defined user action or function (for example, security, scalability, behavior).

So let’s go through some of the features of an app that aren’t related to functionality:

* App performance in normal scenarios
* App behaviour when the system is loaded with many users
* App handling of system stress
* App security
* App performance from OS to OS / device to device
* App recovery from failure

Therefore, it is important to discuss the non-functional testing techniques that address these aspects.

Performance testing: Assesses the overall system performance, a process that involves measuring system response time, and measuring the response time of the crucial app elements. It can also be carried out as part of system testing as well as integration testing.

Stress testing: Validates the system performance in the context of scarce resources, which involves running tests on low storage / memory configurations to identify bugs that may be undetectable in normal circumstances. It also validates the app consistency when a number of users are executing the same action with the same data set, as well as many client machines connected to a number of servers, all subjected to varying degrees of system stress.

Load testing: Measures performance in scenarios of normal usage, repeated with a number of users to assess the consistency of the app response times. This should ideally be carried out at specified, customized, dedicated servers that closely simulate the real-world environment and expected usage scenarios.

Volume testing: Judges performance in the context of enormous amounts of data, involving an identification of where exactly the app fails, at what volume of data the system is unable to continue running. A database is also created at the largest size possible, and many client queries are fed into the system, in order to test how the app handles this kind of volume.

Usability testing: Basically to assess user-friendliness, GUI consistency, error reportage, and correct output in line with the business-specified requirements.

UI testing: Issues addressed here include layout, data movement from one page to another, and pop-ups for assistance if the system concludes that a user needs guidance.

Recovery testing: Validates if the app shuts down during failure without glitches and without affecting the system, and that the data is not lost. Such testing involves premature interruption or termination of data processes, manual dismantling of database keys and fields, as well as even turning off routers, servers, and disconnecting the wires, in order to assess the effectiveness of the app recovery, when all the systems are rebooted.

Compatibility testing: Checks overall compatibility with a range of operating systems, browsers, and devices, at varying strengths of configuration. Also has to ensure that the test cases executed in functional testing are the same ones used here.

Instability testing: Checks the smoothness of installs and uninstalls, and confirms that the app behaviour remains steady if the disk space is limited. Also confirms whether all the app components are correctly installed, and that updates take place at the designated intervals.

Documentation testing: Confirms the presence of guides, instructions, read-me, online assistance, release notes, etc. as part of the app package.

Inferring from all the above points, it is therefore very important to consider the indispensability of non-functional testing procedures in the app development. This requires a concerted strategy from the outset (incorporating NFT requirements in the initial testing plans). The objective for all QA teams should be, to extend their coverage, to include all the non-functional aspects of their software, to ensure that their end-product reflects their organization’s commitment to excellence, and of course, their loyal customer base.

# Conclusion

As stated above those are few of NFR which needs to be considered while developing a software Application. When it’s come to implementation not all criteria can be satisfied so choose carefully.

For example in order to have proper security the data must to encrypted and then saved and again to read the data should be decrypted and then read , even thought it ensure security but this process may take time and effect the performance of the application by taking more time to do simple read and write operation.

So it’s better to choose wisely where you need to be compromised.

When it comes to business app development, every resource is precious. Understanding your functional and non-functional requirements can help save time, money and human resources in app development. Defining functional and non-functional requirements clearly is akin to having a detailed roadmap with pit-stops clearly marked out. This way, you know exactly how long you need to hold down that can of soda you just had.

Once you know your requirements, you can get started on the functional requirements right away and leave room for modifications based on iterations and client feedback. When the time comes to scrape deadlines and drop off a few bells and whistles, you’ll know all the important components you want to keep.

Therefore, clearly map out your functional and non-functional requirements from the start and you will find yourself sailing through the development process without a hitch.