10 MOBILE UX DESIGN PRINCIPLES YOU SHOULD KNOW

Beyond communication and information gathering, services and features such as banking, placing orders, healthcare tracking, and even home security have become part of the smartphone experience.
But designing a usable and enjoyable mobile experience is more than just good aesthetics. In order to trigger positive reactions, you have to satisfy the basic usability principles of interface design.
Learnability: How easily first-time users complete basic tasks.
Efficiency: How quickly users perform basic tasks once they've learned the design.
Memorability: The ability of users to remember how to use the system.
Errors: The amount and severity of errors users make and how easily they can correct them.
Satisfaction: How pleasant the experience of using the design was.
Here are 10 tried and tested UX design principles that are key to creating great mobile user experiences.

1. Content Prioritization

Human attention spans are 8 second. This makes it extremely important to grab your users' attention within the first few seconds of interaction with your products.
Less is more. When designing your products, keep interface elements to a minimum. Simple designs are what keep the user engaged and at ease with products.
Display only essential content and functionalities the user needs. The content should be available through a menu. To further reduce clutter, use icons instead of text wherever possible. Prioritize content and remember that notifications for new content should appear without interrupting primary content!

2. Make Navigation Intuitive

Learnability of your design is key for its success. If it takes too much time or effort to discover how to navigate through your product, chances are you're just going to lose your users.
User journeys should be logical enough that a task is accomplished within a single app. Don't make users have to switch between pages and apps to get something done. Simplify the process and have all the information needed readily available. When navigating through your app, the user should always know where they are without wondering how they got there, or what they are to do next.

3. Touchscreen Target Sizes

Apple's iPhone Human Interface Guidelines suggest a minimum target size of 44 pixels wide x 44 pixels tall. If action buttons are too close to each other, you risk the user making undesired actions leading to frustration. It is especially important

to space out contradicting action buttons, such as the save and delete buttons to avoid errors. At the end of the day, we're all human and make mistakes, even when target size and spacing are scaled properly. That's why including an undo button in your designs will relieve many people (including yourself) when mistakes are made.

4. Provide User Control

Allow users to make decisions to personalize their journeys. Changing settings, controlling notifications and cancelling actions provides the user with a sense of control over the system. Apps can suggest actions or provide warnings, but they shouldn't be the ones doing the decision-making.

5. Legible Text Content

The key to mobile typography is readability.
The strategy for optimal mobile typography is a balance between legibility and space conservation. Generally, anything small than 16 pixels becomes challeging to read for any screen. Crowding a small space with too much content makes it difficult and straining for users to read. A good rule of thumb is to use 30–40 characters per line for mobile.

6. Make Interface Elements Clearly Visible

It's important to have sufficient contrast between content and background in your designs so it's legible, in any setting, even outside in the sunlight.

7. Hand Position Controls

Common features should be placed in easily accessible regions, while actions such as delete buttons should be placed in areas harder to reach to avoid errors.

8. Minimize Data Input

Typing on small devices can be annoying and unfortunately there really isn't a direct solution to it, other than autocorrect (which let's be honest, isn't always accurate). But what you can do is work around the problem by minimizing the need to enter data in your designs. Reduce typing required by shortening forms, removing unnecessary fields and using 'remember me' options for future use. Relieve users by providing autocomplete, recent search history and location detection to reduce data entry requirements and accelerate the experience. Display keyboard variations depending on the required data, for example for phone numbers provide the numeric keypad for faster input.

9. Create a Seamless Experience

Functional, flexible, and responsive mobile designs are what users today expect to encounter. Focus on key user goals by reducing friction, minimizing steps and page loads to decrease interaction time. Make content accessible even without an online connection and provide alternative paths to avoid dead-ends. Make use of the mobile phone's features, like the camera to scan barcodes, GPS for identifying locations and touch ID in place of passcodes to simplify journeys. Users appreciate smooth interactions with designs and are satisfied when their needs are met effortlessly.
Synchronization across devices is a key priority for creating seamless experiences.

10. Test Your Design

Test early. Test often.
The key to any successful product is to continuously test and optimize.
Usability testing is essential for the success of your products. Test different features, layouts and variations of your designs to see what works best. Build your products with a user-centered approach by testing with real users. With each round of testing, you uncover new ways to improve your design to meet your user's needs.
The earlier you discover your problems, the easier and cheaper it is to fix them.

Introduction to web APIs

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place. They generally fall into two categories:

Browser APIs are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it.

Third-party APIs are not built into the browser by default, and you generally have to retrieve their code and information from somewhere on the Web. Let's recap this to make it clearer, and also mention where other JavaScript tools fit in:

JavaScript — A high-level scripting language built into browsers that allows you to implement functionality on web pages/apps.
Browser APIs — constructs built into the browser that sits on top of the JavaScript language and allows you to implement functionality more easily.

Third-party APIs — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).
JavaScript libraries — Usually one or more JavaScript files containing custom functions that you can attach to your web page to speed up or enable writing common functionality. Examples include jQuery, Mootools and React.
JavaScript frameworks — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch. The key difference between a library and a framework is "Inversion of Control". When calling a method from a library, the developer is in control. With a framework, the control is inverted: the framework calls the developer's code.

In particular, the most common categories of browser APIs you'll use (and which we'll cover in this module in greater detail) are:
APIs for manipulating documents loaded into the browser. Which allows you to manipulate HTML and CSS — creating, removing and changing HTML, dynamically applying new styles to your page, etc.
APIs that fetch data from the server to update small sections of a webpage on their own are very commonly used. This seemingly small detail has had a huge impact on the performance and behavior of sites.
APIs for drawing and manipulating graphics are widely supported in browsers.

Device APIs enable you to interact with device hardware: for example, accessing the device GPS to find the user's position using the Geolocation Api.
Client-side storage APIs enable you to store data on the client-side, so you can create an app that will save its state between page loads, and perhaps even work when the device is offline.

Scheduling: setTimeout and setInterval

We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call".

There are two methods for it:

- **setTimeout allows us to run a function once after the interval of time.**
- **setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.**

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

settimeout

The syntax:

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
Parameters:
func|code
Function or a string of code to execute. Usually, that's a function. For historical reasons, a string of code can be passed, but that's not recommended.
delay
The delay before run, in milliseconds (1000 ms = 1 second), by default 0.
arg1, arg2...
Arguments for the function

EXAMPLE

For instance, this code calls sayHi() after one second:

```
function sayHi() {
    alert('Hello') ;
}

setTimeout(sayHi, 1000);
 With arguments:
function sayHi(phrase, who) {
alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

If the first argument is a string, then JavaScript creates a function from it.
So, this will also work:
setTimeout("alert('Hello')", 1000);

- **But using strings is not recommended, use arrow functions instead of them, like this:**

setTimeout(() => alert('Hello'), 1000);
Pass a function, but don't run it
Novice developers sometimes make a mistake by adding brackets () after the function:
// wrong!
setTimeout(sayHi(), 1000);
That doesn't work, because setTimeout expects a reference to a function. And here sayHi() runs the function, and the result of its execution is passed to setTimeout. In our case the result of sayHi() is undefined (the function returns nothing), so nothing is scheduled.

canceling-with-cleartimeout

A call to setTimeout returns a "timer identifier" timerId that we can use to cancel the execution.
The syntax to cancel:
let timerId = setTimeout(...);
clearTimeout(timerId);

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
As we can see from alert output, in a browser the timer identifier is a number. In other environments, this can be something else. For instance, Node.js returns a timer object with additional methods.
Again, there is no universal specification for these methods, so that's fine.

Nested settimeout

There are two ways of running something regularly.
One is setInterval. The other one is a nested setTimeout, like this:
/ instead of:**
let timerId = setInterval(() => alert('tick'), 2000);
***/**

```
let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

The setTimeout above schedules the next call right at the end of the current one (*).
The nested setTimeout is a more flexible method than setInterval. This way the next call may be scheduled differently, depending on the results of the current one.
For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds...
Here's the pseudocode:
let delay = 5000;

```
let timerId = setTimeout(function request() {
  ...send request...
```

```
  if (request failed due to server overload) {
    // increase the interval to the next run
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

And if the functions that we're scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.
Nested setTimeout allows to set the delay between the executions more precisely than setInterval.

EXAMPLE

Let's compare two code fragments. The first one uses setInterval:

```
let i = 1;
setInterval(function() {
  func(i++);
}, 100);
```

The second one uses nested setTimeout:

```
let i = 1;

setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);
```

For setInterval the internal scheduler will run func(i++) every 100ms:
Did you notice?
The real delay between func calls for setInterval is less than in the code!
That's normal, because the time taken by func's execution "consumes" a part of the interval.

Zero delay settimeout

There's a special use case : setTimeout(func, 0), or just setTimeout(func).
This schedules the execution of func as soon as possible. But the scheduler will invoke it only after the currently executing script is complete.
So the function is scheduled to run "right after" the current script.
For instance, this outputs "Hello", then immediately "World" :

```
setTimeout(() => alert("World"));
```