Chapter 10

Optimizing iOS WebApps

"Perfection is achieved not when there is nothing more to add, but when there is nothing more to take away."

—Antoine de Saint-Exupéry

This chapter is about web optimization and search engine optimization (SEO). First we talk about iPhone and iPad compatibility, and then we show how to optimize the performance of a WebApp. We also suggest some rules for optimizing the code, reducing HTTP requests, and minimizing DOM access.

We then demonstrate how to compress a WebApp, optimize its usability, and make it capable of working offline. Finally we look at a mobile SEO approach to WebApps, analyzing first the anatomy of a search engine and then exploring how to implement a search engine oriented design. We also look at the principles behind the Google algorithm and some useful mobile SEO tools.

iPad and iPhone Compatibility

Beside the fact that the user experience is totally different between iPhone and iPad users, most of the concepts behind good optimization are common to both devices.

Some of these concepts are implemented in different ways in order to optimize specific aspects of the device, whereas others are equally applied in order to increase the level of user experience.

Performance Optimization

Optimizing the performance of our WebApp is not a development approach that we can perform only at the end of our project workflow. It is something that, exactly as for the test phase, is applied for the duration of the project. Obviously, at the end of the Development Phase we apply some optimization techniques to our WebApp, but it is

A. Picchi, *Pro iOS Design and Development*

most efficient to incorporate some good habits from the beginning in order to reduce mistakes and shorten the overall development time (See Figure 10–1).

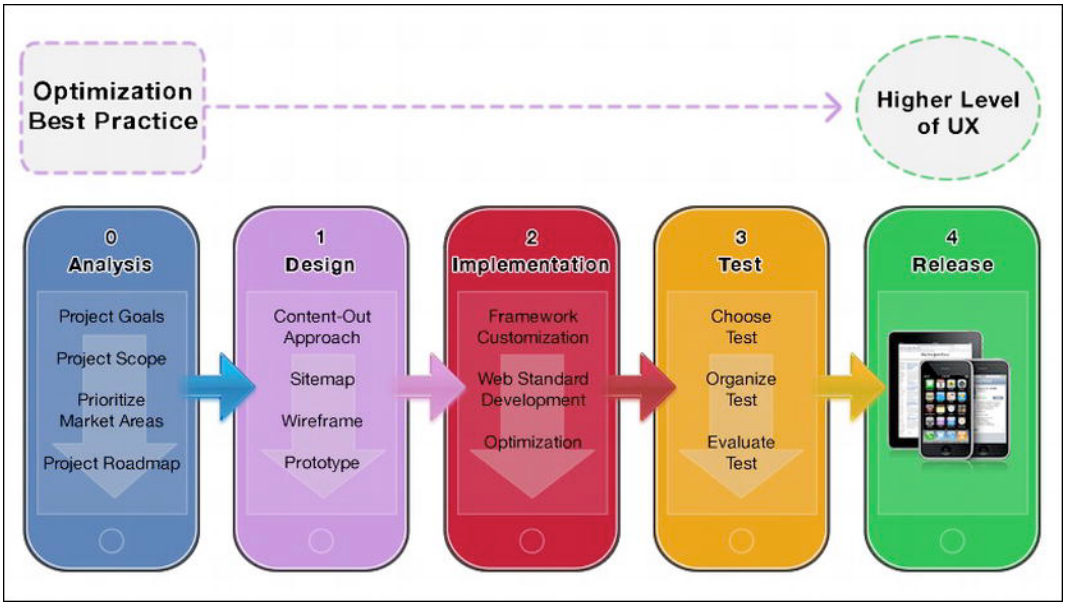

Figure 10–1. *Optimization best practice applied along the whole project workflow process*

When we optimize our web pages, it is important to know what can be optimized. For those who know the Vilfredo Pareto Principle, you also know that 80% of the consequences come from 20% of the causes, which means that it will be hard to get positive results without knowing exactly what to target with our optimization process.

Coming up we'll look at some of the best practices, presented as rules, in order to clearly present a pragmatic approach to a performance optimization process applicable to our WebApp.

Code Optimization

Code optimization is the first step of any type of optimization technique because everything is based on code—everything is coded in our web pages. Good code can save bandwidth, reduce rendering delay, and improve the page's readability and maintainability overtime.

The following are some best practices to keep in mind when writing any type of code in our WebApp.

Rule 1: Use Web Standards Complaint Code

Use HTML5, CSS3, and JavaScript compliant code. Besides clean HTML5 syntax, this also means inserting our style sheet in the `<head>` part of the page and (except the link

to the iWebKit Framework) the JavaScript in the bottom of our web pages. This is because style sheets in the top of the page significantly speed up the loading time. On the other hand, insert the JavaScript in the bottom of the web page so the JavaScript code does not block HTTP requests. This is because when JavaScript is downloading, the browser will not start any other resource downloads, even if the resource is on a different hostname.

> **NOTE:** An alternative to this rule is to insert the Desktop-Mobile Redirect JavaScript code at the top of the page. We can do that because in this case it is more important to execute the script than render and load the web page.

This rule helps the parser work faster and helps reduce the overall rendering delay.

Rule 2: Write Slim Code

Write slim code. Remove unnecessary or redundant parts of the code and avoid using tab and space where it is not strictly necessary. Don't use CSS expressions if you can achieve the same result with other techniques. CSS rules are evaluated more frequently than we can imagine and can negatively affect the performance of our web page.

> **NOTE:** In our use-case, for illustration purposes, we overwrite many CSS rules in order to present both the original iWebKit Framework and our use-case custom code. In a real project, keep the number of overwritten CSS rules to a minimum.

Choose short and meaningful names for comments and CSS ID and class or JavaScript variables and functions. Don't hesitate to adopt xHTML5 syntax, if you like to write XHTML code, and combine CSS rules to a good level of factorization inside your style sheet. Use Gzip compression or minify HTML5, CSS, and JavaScript code, but always remember to store an uncompressed version for development purpose in the project.

This rule reduces the overall weight of our web page and, by default, the rendering and loading delays.

Rule 3: Reduce HTTP Requests

It is important to always keep one eye on the number of imported resources (images included). More files imported into our web page equals more rendering and loading latency from the browser. Minimizing the number of HTTP requests speeds up the web page loading time. With this in mind, it may be a good idea to consider adding HTTP caching to our web pages.

HTTP caching, also known as web cache, is based on good principles, but it is almost unusable in Apple Safari because of its specification limits. The following list summarizes some of the main limits of HTTP caching:

- **Single resource must be less than 15kB (non-compressed)**
  Web pages designed for iPhones should reduce the size of each component to 15kB (25kB before iOS3) or less for optimal caching behavior. The iPhone is able to cache 105 15kB components. Attempting to cache one more file results in removing an existing one from the cache.
- **Global cached resources must be less than 1.5MB**
  Although the iPhone is able to cache multiple components, the maximum cache limit for multiple components is around 1.5MB (500kB before iOS3). Maximum bytes available in the cache are around 105 * 15 = 1575kB.
- **Powering off the device clears the HTTP cache**
  If the user needs to force a hard reset, components in the cache will be lost. The reason is that, on the iPhone, Safari allocates memory from the system memory to create cached components but does not save the cached components in persistent storage.
- **Closing the tab clears the HTTP cache**
  Closing all tabs except the blank one and then closing Safari clears the cache.

We can see from a development point of view that this type of cache is unreliable because it is cleared too often and can't cache the majority of the resources in a modern web page. Even the most compressed JavaScript Framework or CSS are a struggle to get under 15K, and none of the images used in almost every WebApp are under this limit. The offline features provided by HTML5 are a better option for our goals, and we introduce them later in this chapter.

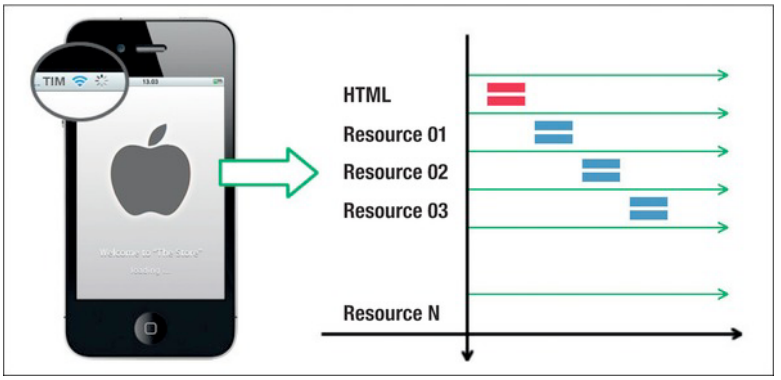

Figure 10–2. *WebApp resource requests according to the HTTP/1.1 protocol*

The best reason for following this rule, besides reducing the rendering time of our web pages, is that the HTTP/1.1 protocol specifies that a browser can download only two resources in parallel per hostname, as shown in Figure 10–2.

A workaround for this sort of bottleneck is to spread our external resources over multiple hostnames. Lastly we can't forget to avoid all HTTP redirects in our web pages. The HTTP redirect is accomplished using the 301 or 302 status code, and in both cases it adds a delay to the average page loading time, thereby decreasing the quality of user experience.

This rule reduces the loading time by reducing the communications delays between the client side and server side.

Rule 4: Combine CSS and JavaScript Files

This rule must take into account the project's complexity, but the basic idea is that we combine all our CSS rules and JavaScript code into one single file instead of having multiple files. This will reduce the HTTP header's weight and the latency of imported multiple resources in our web pages due to TCP slow starts, as shown in Figure 10–3.

A side effect of this approach is that we are forced to update larger files, even for small code updates; however, this is often a path that brings more positive effects than negative.

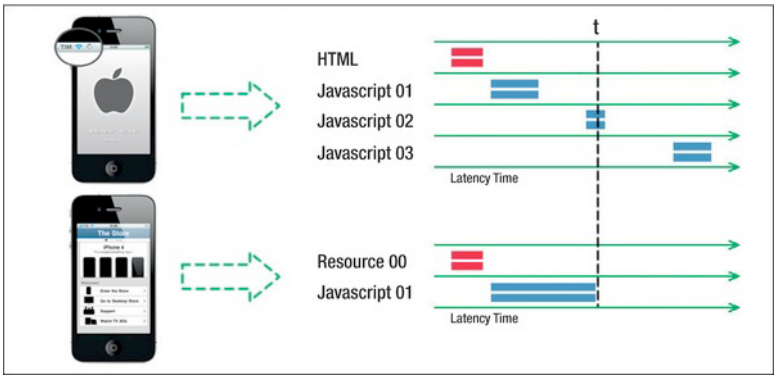

Figure 10–3. *Transfer latency time: comparison between single and multiple JavaScript files*

In our "The Store" use-case, the iWebKit JavaScript Framework Core and the CSS are all inside a single .js and .css file minified with an optimization program. We can keep the logical structure of our code in a development version (non-minified) of these files and subsequently add header and single comments to make the code maintenances and its feature updates easier. In the resource section at the end of this chapter, you will find some online minify resources.

Rule 5: Minimize DOM

In our project, the hard work is done by the framework, but we still need to write JavaScript code to accomplish some of the project requirements. In this case, the rule is simple: minimize the DOM access and the number of DOM objects.

This rule will reduce the web page loading time and user experience delays every time the web page runs a JavaScript.

Image Optimization

An important step in the Optimization Phase is image optimization. Image optimization is another example of a good habit that doesn't contain any great secret. Simply put, optimizing the images of our WebApp can dramatically improve the performance of our web pages by making them lighter and reducing loading delays.

The following are some best practices to keep in mind when we work on our WebApp's images.

Rule 6: Optimize Color Depth

After we design an image, we need to optimize its weight by exporting it using the right image format. If it's a photo, we need to use a good compression ratio in JPG format. If it's a user interface image, it is important to check the number of colors used. If we use fewer than 256 colors, we can export it in PNG8. In most cases, exporting in PNG8 renders a smaller image than exporting as a 256-color GIF. Using similar colors also helps to keep the color count and the image weight low.

We should also stress that exporting images using a graphic program like Adobe Photoshop, Fireworks, or Gimp will add unwanted metadata that will increase the image weight. We can see the metadata imposed on an image in Fireworks by accessing the Metadata Panel and browsing **File ➤ File Info (T)** or using ^⇧⌘F.

A workaround is to optimize our images using a program like PNGOut that will make them as slim as possible.

This rule reduces the web page loading time and increases the level of user experience.

Rule 7: Use CSS Sprites

The word "sprite" might remind you of the'80s, when people played all day with Commodore 64 or ZX Spectrum games. Because in computer science everything that is old sooner or later will become new again, web developers adopted the idea behind the old Sprite management and brought it to the CSS world. Look at the following Figure (10–4) for an example.

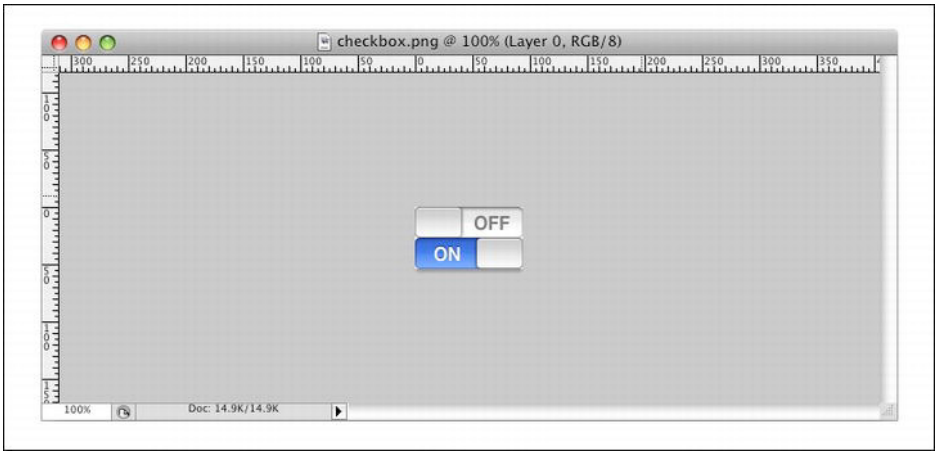

Figure 10–4. *The CSS Sprite techniques used for Design Checkbox in the iWebKit Framework*

To use the CSS Sprite technique, first we group two or more images into a single background image, then we set via CSS the single image width and height, and finally we adjust the background position using the CSS margin rule to display only the portion necessary. With this approach we can use a single background image and display several different graphics (single images) with it, thereby saving server requests and speeding up page load times.

```
/* from framework style.css stylesheet */
input[type="checkbox"] {
  width: 94px;
  height: 27px;
  background: url('../images/checkbox.png');
  -webkit-appearance: none;
  border: 0;
  float: right;
  margin: 8px 4px 0 0;
}

input[type="checkbox"]:checked {
  background-position: 0 27px;
}
```

The CSS background rule shows every image from the coordinates 0px, 0px; this guarantees that if we set a height of 27px, the OFF state would be showed by default. In this case, the Sprite technique shows the ON state by using an offset of 27px, demonstrated by the second CSS rule.

If we use many images for our user interface, the CSS Sprite technique can help to reduce the global loading time of our web pages and avoid the typical white flash of the traditional rollover technique. Because the image loading time is larger than the rendering time, using a traditional image rollover technique creates a white flash every time the browser loads the rollover image for the first time.

SPRITE EXERCISE

In our "The Store" use-case, we design the Breadcrumb Bar using a few images. Implement the Sprite techniques to speed up the rendering time.

- Use a Sprite with all the Breadcrumb images.
- Group two or more Sprite Breadcrumb images and compare the rendering time with the single Sprite approach.

Compare the result and determine which approach is best for our specific use-case.

This rule reduces the web page loading time and the user experience delays every time the web page runs JavaScript.

Rule 8: Use CSS Rules Instead of Images

This rule may sound strange, but because the image optimization process aims to reduce the weight of images globally, use CSS rules every time it is possible instead of bitmap images.

CSS TEXT EXERCISE

In our "The Store" use-case, we design the Breadcrumb Bar using few images. Implement the CSS technique to speed up the rendering time even more.

- Use text instead of images for all the Breadcrumb links.
- Align the House Icon with the Breadcrumb text.

Compare the rendering time of both the Sprite and CSS approaches.

We need to use CSS rules for everything involved in our user interface and insert images only in rare cases. If we must use an image for a user interface element, it must have its color depth optimized. If we need to insert many images we need to insert these images into CSS Sprites. We should also use CSS rules for each small design detail such as borders, backgrounds, or gradients.

This rule reduces the web page loading time and user experience delays every time the web page runs a JavaScript.

Rule 9: Never Scale Images

Always use images with appropriate dimensions according to the device viewport or design element width and height. It is never a good idea to rely on Safari to scale an image for the right fit. The only exception to this rule is when we want to insert an image inside a single device WebApp (only for iPhone or iPad). In this case inserting an image

with a width value of 100% will fit both the landscape (bigger) and portrait (smaller) orientations.

This rule reduces the web page loading time and user experience delays every time the web page runs a JavaScript. While it is important to follow this rule, remember that it is also important to specify the image width and height, as this will also help to reduce rendering times.

IMAGE OPTIMIZATION EXERCISE

All the images used in our "The Store" use-case are in PNG format. Try to determine when it is possible to optimize some of these images using another format like JPG or GIF. Don't forget that some formats don't support the Alpha Transparency.

Choose a graphic program and open some of the images used in the "The Store" use-case located in the directory "/images" and inside the directory "/pics".

- Export the images using a different format.
- Export the images using the same format with a different setting.

Compare the image weight and the image quality, and then see whether you can replace some of these images with optimized images.

Application Compressing

Safari supports GZIP compression (RFC 1952), so compressing some of the resources of our WebApp is often a good idea as this will result in an increase in the level of user experience. We can decide when to compress our HTML5 documents, CSS3 style sheets, or JavaScript code, whereas we don't want to compress images or PDF files because these are already compressed. Compressing images or PDF files adds CPU overhead and potentially increases the file size.

From the server side, in order to use GZIP-compressed resources in our WebApp, the server must be configured to provide compressed resources when requested. From the other side the client must be able support this type of files.

The Request/Respond process represented in Figure 10–4 can be resumed in the following three steps:

- **Client**

  Connect to the server

  Send a request with GZIP support: "accept-Encoding: gzip"

- **Server**

  Acknowledge GZIP support

  Compress resource with Gzip algorithm

Send GZIP-encoded resource: “content-Encoding: gzip”

- **Client**

  Receive GZIP-encoded resource

  Decompress GZIP-encoded resource

  Display (or use) the resource

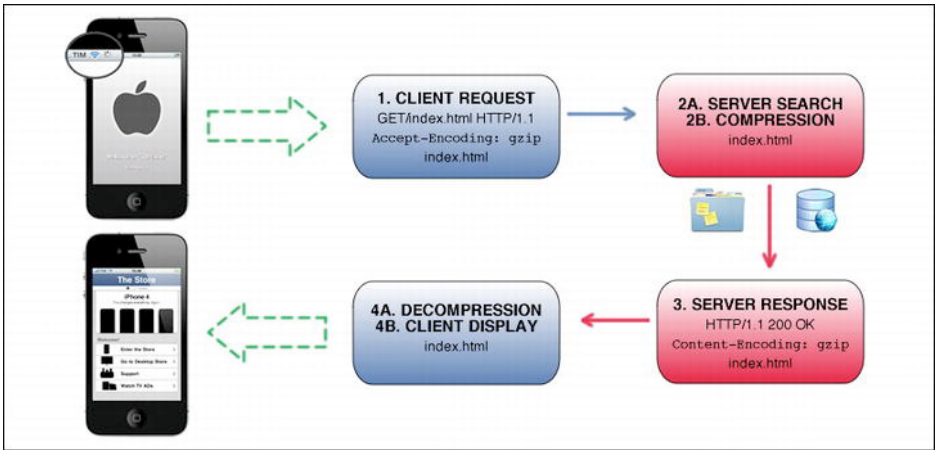

Figure 10–5. *The GZIP-compressed resource request: the HTTP/1.1 protocol in action*

The following code is an example of the header of a HTTP/1.1 request and response for a GZIP resource (also shown in Figure 10–5).

```
GET / HTTP/1.1
... ... ...
Accept-Encoding: gzip
... ... ...
```

After the server receives the client request, determine whether the requested resource is available in a compressed version. If yes, the server sends it to the client adding the following string to the response.

```
HTTP/1.1 200 OK
... ... ...
Content-Encoding: gzip
... ... ...
```

There is not a limit to the file that we can compress using GZIP, and this is the easiest way to achieve a significant reduction of the web page weight. The GZIP compression can reduce the weight by approximately 70%.

Despite that, and because perfection doesn’t exist in this world, generally speaking GZIP compression has a few negative aspects.

- First we need to work with a browser that supports GZIP compression. In our context this is not an issue because Safari and WebKit-based browsers support GZip.
- Second, as previously stated, we can't compress images or PDF files because they are already compressed.
- Third, it is important to remember that because Safari needs to decompress these resources on-the-fly, in some cases this process can add CPU cycles and overhead to the application and eliminate the possible benefits. Perform a test in order to ensure that this overhead does not eliminate the possible benefits gained.

Usability Optimization

Usability is a fundamental necessity of our project, and it is always a good idea to test usability before arriving at the end of the project flow. Testing our work during a certain phase in the project flow can tell us whether we match the project requirement and give us a feedback on the achieved level of usability.

In Chapter 2, we saw that an error can be propagated through the project flow and how its cost increases along with its propagation. A good phase of testing eliminates, or at least mitigates, this cause-effect process.

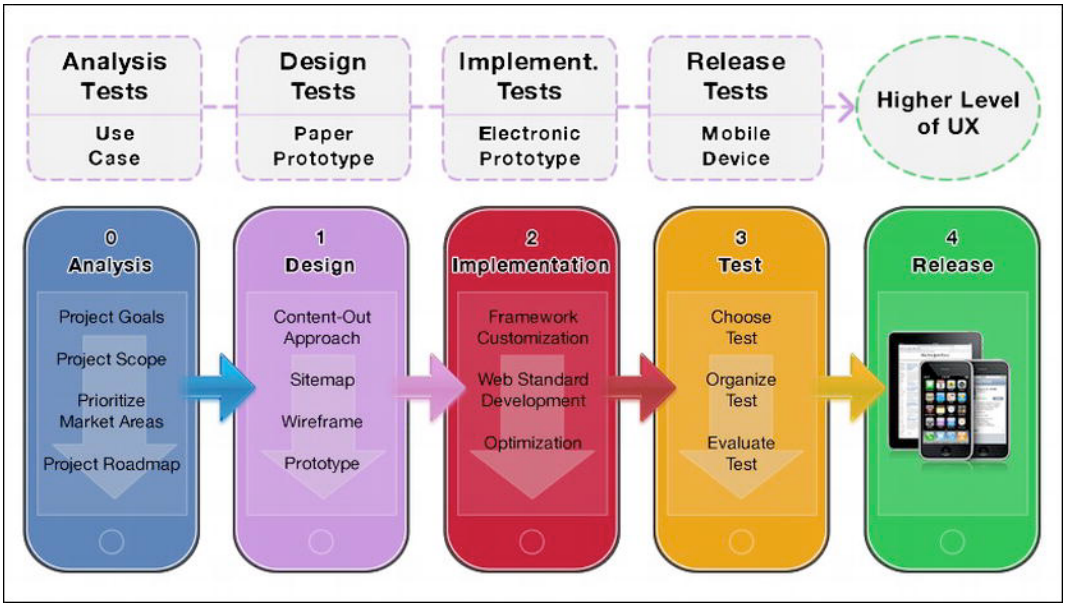

Figure 10–6. *The usability optimization: tests scheduled on each project flow's phase*

We can perform our usability optimization at different levels in every step of our project flow, and we can do it again at the end of the project flow before the final release of our WebApp. Following our diverse approach, we can schedule different types of tests

according to the level of detail of the project. Figure 10–6 shows a type of approach that begins with the Paper Prototype Test and the Electronic Prototype Test and schedules a live test on a real mobile device for the Pre-Release Test Phase.

Compared to a simple web site, a WebApp needs a more accurate phase of testing, so we encourage you never to overlook this phase. Even if you work on a simple project, experience will tell you how hard or soft your project will need to be tested.

We have two types of usability tests depending on the subjects involved.

- **Usability inspection**

  Typically performed by an evaluator that is not the designer or the developer and has no involvement with the project. Usability inspection should be performed from the early phase of design. An example of usability inspection is the *Cognitive Walkthrough*, in which the evaluator simulates the user's problem-solving process for a specific task.

- **Usability test**

  Typically performed by the designer or the developer on a user. Usability tests are performed in the Design Phase, Implementation Phase, and Release Phase using different types of tests. An example of a usability test is the *Prototype Test* where the designer or developer tests the multiple user aspect of the design, including services and specific functions.

The Cognitive Walkthrough is an inexpensive form of test; however, although this approach is used more often in software development than web development, the Prototype Test is a valid option for our mobile design and development context. For this reason, we present the Prototype Tests in Chapter 11, in which we see in detail how to organize, perform, and evaluate the test.

For now it is important to introduce the anatomy of a usability test as well as a few more important concepts. A usability test, like a Prototype Test, is structured by the following steps:

1. **Choose testing environment.**
   We need to choose the test environment according to the type of prototype test we choose to perform and the project requirements.

2. **Create use-case.**
   We need to create a use-case that will define a task for the user that will verify one or more use-case requirements from the project requirements.

3. **Prepare test assets.**
   We need to prepare and re-use the assets that we will use to perform the test.

4. **Select users.**
   We need to choose the right user according to the use-case requirements.

5. **Perform test session.**
   We need to run the test to verify the use-case requirements.
6. **Debrief test.**
   We need to debrief the test with the user and with the observers.
7. **Evaluate test.**
   We need to evaluate the test according to the use-case requirements.
8. **Create findings and recommendations.**
   We need to provide findings and recommendations that will drive designers and developers to improve the project.

These eight steps require us to choose the users according to the application profile. However, we don't know how many users we need in order to gather reliable data for the test. We answer this question in the following section.

How Usability Problems Affect Users

We can define a problem as something that is difficult to deal with, solve, or overcome. Testing a project usually means finding something that can represent a problem for the user.

If we choose the "right" user for our use-case, even a single user test will give us reliable information to improve our project. However, no matter how "right" one user can be, their voice will still remain one in the crowd. The risk that a user performed a certain action by accident or was influenced by personal un-representative contexts is too high to create an entire test based on a single piece of user feedback.

The logical conclusion might be to add as many users as possible to discover as many problems as possible. While this approach may look like the right one, it is not. Those who have some probability and statistic knowledge know that there is a value that represents the best ratio between effort and result and that behind this value the result is minimal compared with the effort. For this reason, choosing a large group of users will not be the best approach to the problem.

It's best to choose a smaller group as a sample size to discover as many problems as possible. This path brings us to Jim Lewis, who published a study in 1982 that described how binomial distribution can be used to model the sample size. This study was supported by Robert Virzi in 1992. Virzi found that 80% of usability problems are found by the first four to five users and that severe problems are more likely to be discovered by the first few users.

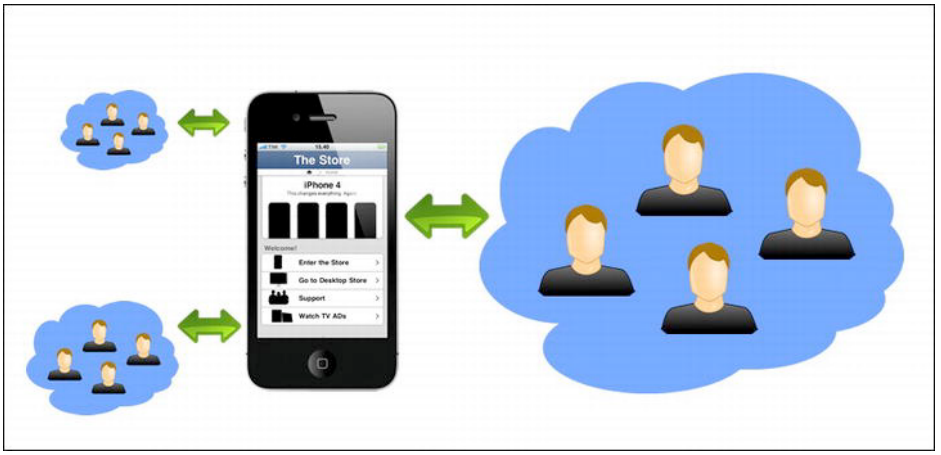

Figure 10–7. *Usability problems: different groups could discover different types of problems*

The problem of selecting a sample size seems to be solved because four to five users should be the right number for our test session. Even the Nielsen studies in early 1990s confirmed this size. Unfortunately, problems don't uniformly affect users, as Woolrych and Cockton showed in their studies in 2001. This means that a simple estimate of problem frequency with the binomial distribution is misleading.

For this reason, the best approach we can choose for a medium to large project is to set a few four-to-five user groups in order to represent different category of users, as shown in Figure 10–7, and discover as many problems as possible. In practice we need to perform a test with a group, fix the discovered problems, and then re-perform the test with another group. Probability and statistic studies on usability problems are long and complex, but by simplifying the conclusions (as in Figure 10–8) we can see that with an average number of 18 users we can discover 85% of the problems. This is supported by Jakob Nielsen and Thomas Landauer in their speech "A mathematical model of the finding of usability problems," at the Proceedings of ACM INTERCHI Conference (Amsterdam, The Netherlands).

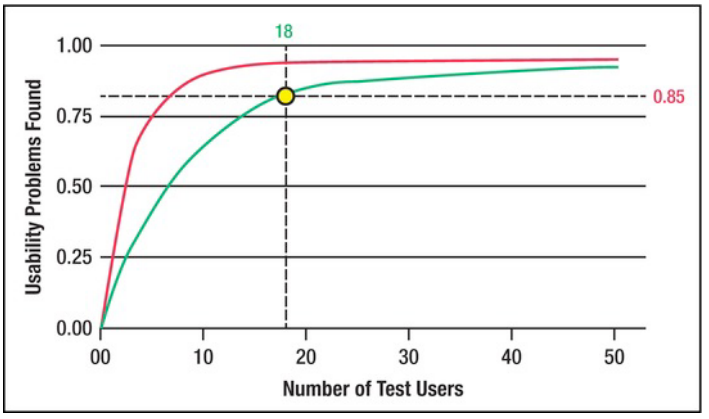

Figure 10–8. *Usability Problems Study: We need 18 users to discover the 85% of the usability problems.*

For a simple project, like a web site or a low-complex WebApp, we can rely on a single group of four to five users and fill the remaining gap by using our experience. We can also apply the cycle approach to the single group by testing the first three users, fixing the problems, and then testing the other two users. The data will not be as complete as in a multi-group approach, but the approach will be more agile and the feedback will still be useful.

Offline WebApp

In this book we always focus on emulating the native application environment and behavior with our web pages. It's obvious that our web pages are dependent on Internet access in order to provide any kind of service, but they are also dependent on the Internet in order to retrieve the various design elements of the web page itself.

Using the HTML5 Offline feature, we can address this issue by storing any type of resource inside the cache of our WebApp. The files that need to be cached are declared in a file called *Manifest File*. Once the files are cached, Safari looks for the Manifest File before beginning any server-side processing while avoiding downloading files previously downloaded and stored.

> **NOTE:** Safari evaluates the content of the Manifest File to determine whether or not to update it. The file date or any other attribute will not be evaluated as we used to see in an HTTP Conditional GET Request. If we want to force an update, we can do it via JavaScript.

The application cache persists between browser sessions, which means that a previously cached resource can be viewed or continue to work without any network support or if the iOS is in Airplane Mode.

The Manifest File

A manifest file is a simple text file hosted on the application's web server that lists all the static resources that need to be downloaded and cached by our WebApp. A Manifest File is composed by two main parts and one optional part:

- Cache Manifest Declaration
- Cache Manifest URL List
- Cache Types Declaration (not required)

The iWebKit framework doesn't use any Manifest File because from the 5.04 version it replaced most of its user interface images with a CSS3 approach. Despite that, from our use-case it is important to cache at least all the product's images and provide the users offline access to the catalogue.

The Cache Manifest file should begin with the uppercase prefix "CACHE MANIFEST." Below that we can define, always using an uppercase prefix, three (sub) section headers

corresponding to three types of different behaviors according to the WebApp requirements:

- **CACHE MANIFEST**
  This is the Cache Manifest header.
- **CACHE**
  Resources are always loaded from the cache, even in online mode.
- **NETWORK**
  Resources are always loaded from the server, even if the file is listed under the CACHE header. This is an exception to the CACHE rule.
- **FALLBACK**
  Resources are used as replacements for other resources that fail to load or load incompletely.

If we list the resources right after the "CACHE MANIFEST" declaration header without specifying any of the three types of (sub) headers, the default CACHE type will be applied to all the listed resources. A typical cache manifest file looks like the following code:

```
CACHE MANIFEST
CACHE
# Comment on Cache Rule Files
fileO1
fileO2
fileN
NETORK
# Comment on Network Rule Files
fileO1
fileO2
fileN
FALLBACK
# Comment on Cache Rule Files
fileO1
fileO2
fileN
```

Below the header we can also insert comments using the prefix "#". This feature is often used to mark the cache version, modify the Manifest File, and force an update of the cache. The following code shows the Manifest File of our "The Store" use-case. If we need to cache an entire folder, like in "The Store" use-case, we can simply insert the absolute folder path and all the files will be added by default to the cache manifest whitelist.

```
CACHE MANIFEST
# WebApp Images inside the pic folder
http://www.thestore.com/images
# WebApp Images inside the images folder
http://www.thestore.com/images
```

In this code we use an absolute path, but a relative path is also allowed; it's totally up to you. After creating the Manifest File, we need to save it using the extension ".manifest".

For the "The Store" use-case, we use "cache-iphone.manifest" and we save in the application root directory.

The next step is to link the Manifest File in our web pages inside the `<html>` tag using the attribute `manifest` as shown in the following code:

```
<html manifest="cache-iphone.manifest">
```

The Manifest File must be served using the "text/cache-manifest" MIME type, so the last step is to add the "text/cache-manifest" content type inside an ".htaccess" file, which is in turn placed inside the web root directory. If it's not done in this order, Safari will not recognize the Manifest File.

```
AddType text/cache-manifest .manifest
```

Now everything is in place, but because our application looks to the resource list in the Manifest File to understand whether the manifest file needs to be updated or not, we need to use JavaScript if we want to force this update process.

> **NOTE:** We can make a change in the Manifest File by using a single comment line in the file. This generates an update from our WebApp the next time the Manifest File is checked. However the JavaScript approach is recommended because it provides more possibilities to the developer. If a failure occurs while downloading the manifest file (its parent file or a resource specified in the cache manifest file), the entire download/update process fails.

We can access the cache using the `window.applicationCache` JavaScript object and update it in three steps using the `update()` and `swapCache()` methods:

1. Test whether the (old) cache is ready to be updated.
2. Update the (new) cache.
3. Swap the old cache with the updated cache.

In Table 10–1 we can see the three items used for the cache update process.

Table 10–1. *The JavaScript object and the two JavaScript methods involved in the cache updated process*

Name	Description
`window.applicationCache.status`	Check whether the cache is ready to be updated
`update()`	Update the cache
`swapCache()`	Swap the old cache with the updated cache

When we check the `applicationCache` object using the `status` property, we observe different returns according to the cache status. Table 10–2 shows the possible value returned by the `status` property.

Table 10–2. *The values returned by the status property applied to the "applicationCache" object*

Name	Value	Description
window.applicationCache.UNCACHED	0	Cache is not available
window.applicationCache.IDLE	1	Cache is up to date
window.applicationCache.CHECKING	2	Manifest File checked for update
window.applicationCache.DOWNLOADING	3	Downloading the new cache
window.applicationCache.UPDATEREADY	4	New cache ready to be updated
window.applicationCache.OBSOLETE	5	Cache deleted because obsolete

Now we are ready to put everything into practice by writing an `if` statement to test the value of the status of the `applicationCache` object and, if ready, perform a cache update:

```
if (window.applicationCache.status == window.applicationCache.UPDATEREADY)
{
    applicationCache.update();
    applicationCache.swapCache();
}
```

At this point of the workflow our WebApp is almost ready for the Release Phase, but before we think about releasing it we need to take care of another important aspect of the optimization, an aspect that is based on the relationship between our WebApp and a search engine. We see how to work on this part in the next paragraph.

Mobile SEO

SEO is an important step of our project workflow. SEO is more fundamental for a web site than for a WebApp because compared to a web site, a WebApp rarely relies on search engine results to promote itself. Despite this fact, SEO should be behind every stage of a WebApp design as well.

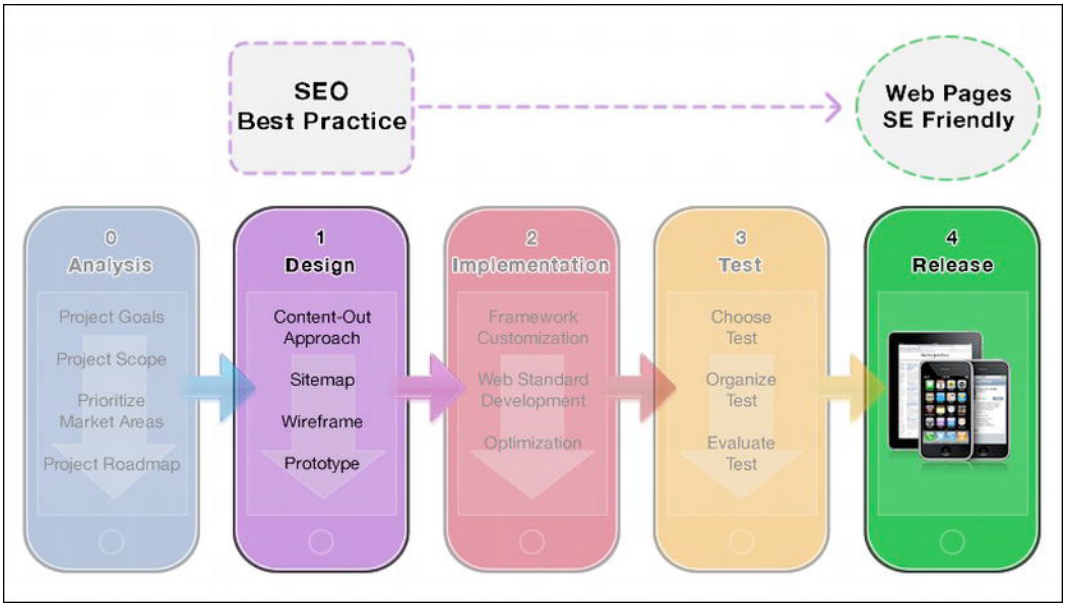

Figure 10–9. *SEO best practices applied along and behind the whole project workflow process*

The SEO Phase brings to the table some rules that are valuable for both web sites and WebApps. As we previously saw for accessibility, usability, and code or image optimization, optimizing our web pages for search engines is an approach that runs from the beginning to the end and behind our project workflow. Working on a complete SEO plan is beyond the scope of this book, but in the following sections we see some key points that make our web site rank higher and our WebApp friendlier to major search engines.

Anatomy of a Search Engine

There is much more behind the minimal user interface of a search engine like Google. Unfortunately we can't know every detail of how a search engine works because this is proprietary information. In spite of this fact, every search engine is composed of a few known parts that, in a general way, can help us to understand how they work.

- **Crawlers, Spiders, and Robots**

  Crawlers, spiders, and robots are programs that crawl the web in search of web pages to index in a database. Google is an example of a crawler-based search engine, and its crawler scans the web collecting information about every URL.

- **User Interface (UI)**

  User interface is where the user writes his query. The minimal user interface offered by Google is just one example of the front-end of every search engine.

Search Engine Database

Search engine databases contain multiple data points about each stored URL. These data could be arranged in many different ways and every search engine has its own way to accomplish this job. How every search engine arranges these types of data is a closely guarded secret; an example is the PageRank method used by Google.

- **Search Engine Algorithm**

  Search Engine Algorithm is the heart of every search engine, which is the part that makes everything works. This algorithm evaluates one or more inputs (words inserted by the user in the search engine user interface) and generates an output and searches the database where URLs and keywords are stored. This algorithm, which can be catalogued as a Problem Solver Algorithm, is composed of multiple algorithms that analyze different web site parts. Every search engine has its own implementation of this algorithm.

- **Search Engine Result Page (SERP)**

  The Search Engine Result Page, besides the search engine user interface, is the only part that is visible to the user. This page is a collection of links catalogued in a specific order by the Search Engine Algorithm.

In the following section, we see how to design and implement our web pages in order to be more search engine friendly.

Search Engine Oriented Design

Optimizing our web page for a search engine is important from the early phases of our project workflow. A search engine oriented design is a title that stands for an approach used during our Design Phase. Let's take a look at the steps of the Design Phase.

Domain Title

The first step of any SEO approach starts even before we can open our graphic program or code editor. Choosing the wrong domain can ruin all the future efforts to gain a good positioning on a search engine. Despite the crucial moment of this step, the solution is simple: insert the primary keyword in the domain name.

```
http://iphone.thestore.com     /* iPhone Third Level Domain Name */
```

This is an example of the hypothetic name of our "The Store" use-case. Inserting the primary keyword in the domain name guarantees that our WebApp will be stored using a word that will be used as primary keyword in the next steps of the SEO Optimization Phase.

Page Title

The HTML page title is one of the most important tags to optimize. The page title displays as the first line in the SERP, and it's the most meaningful source for our WebApp. A good title is short and includes the main keyword or keywords that identify our web page.

```
<title>The Store</title>                                          /* Store Index
Page Title */
<title>The Store (U.S.)</title>                                  /* US Home Page Title */
<title>The Store (U.S.) | Contact Us</title>            /* Contacts Page Title */
```

We must write a unique title for each page and every title must include the name of the WebApp. The code shows three examples for the "The Store" use-case.

Meta Tags

There was a time where meta tags were the holy grail of SEO, but nowadays the situation has changed due to the abuse of this type of tag from webmasters. An important tag to optimize is the description meta tag. A search engine like Google uses this tag to display a description of our web page as the second and third (if the text is long enough) line in the SERP. A good description tag includes our keyword (or keywords) and is informative. The following code is an example from the "The Store" use-case.

```
<meta name="description" content="Apple designs and creates iPod and iTunes,É
 Mac laptop and desktop computers, the OS X operating system, and theÉ
 revolutionary iPhone and iPad." />        /* Store Index Page Description Metatag */
```

Another important meta tag is the keyword tag; it's not fundamental but it's still important. As we can see from the following code, in our "The Store" use-case the choice is easy because a big brand like Apple needs only one keyword: Apple. In other projects, more than one keyword works fine; just don't abuse it.

```
<meta name="Keywords" content="Apple" />        /* Store Index Page Keywords Metatag */
```

Keywords must match the words and phrases that potential visitors will use when searching for your site.

Content

Content is important to optimize our web pages. In the end, what users are looking for is just a small piece of content to read. The point is that now we need to distinguish the case where the user lands on the Compatible page, on the iPad page, or on the iPhone page.

Generally, in a SEO content optimization, we need to use the keyword(s) specified in the keyword's metatag in a few strategic points of our web page:

- The page header (primary keyword)
- The page tagline (secondary keyword)

- The page content (primary and secondary keyword)
- The page links (primary keyword, only wherever is possible)

If we think in a Google-oriented way, it is important to use our keyword(s) in the upper part of our web page because it's the most important (meaningful) part for the crawlers. Because the iPad, and even more the iPhone, version has strictly prioritized contents, this can make our job both easier and harder.

Prioritized content can make our job easier because we assume that this type of content is based on important keywords and short meaningful paragraphs in order to deliver the message in the most direct and fastest way.

At the same time, prioritized content can make our job harder because sometimes the content is so short that it is practically impossible to organize in a meaningful way for both humans and crawlers.

Figure 10–10. *Search engine oriented design: an example of the use of the primary key "iPhone"*

The way to approach the situation is to stick with our prioritized content on our mobile version and play a little bit more with the content of the compatible version where we have more space and a chance to achieve good results.

The last thing we need to avoid is the pitfall of keyword stuffing. Keyword stuffing, simply put, is using a keyword too many times or forcing it in a paragraph with the sole purpose of increasing its usage. Don't use keywords if they don't make sense in context. This can lower the quality score of our webpage.

Links

A web page without links is like a lost island in the ocean; it is there, but almost no one knows that it exists or, if they do, how to reach it. The role of a link is to connect our web page with other relevant information, both internal and external to the web page itself. Another reason why links are so important is that links have a great value on the final weight of our WebApp SEO score. More precisely, inbound and outbound links have a "weight" whereas internal links serve only a better "crawl" of the site.

Google with its famous algorithm, developed by Larry Page and Sergey Bring in 1998 and patented by Stanford University, was the first to assign dynamic and different values to inbound and outbound links in a web page. The real specifics of the Google page rank algorithm are unknown because it is one of the closed secrets of the company; despite that, some details are known.

The Google PageRank Concept

The probability that a random user visits a web page is called its *PageRank*. The Google PageRank concept uses Google's global link structure to determine an individual page's value. The PageRank gives an approximation of a page's importance and quality.

The algorithm that implements this concept interprets a link from web page A to web page B as a vote, by web page A, for web page B. The Google PageRank Algorithm looks at more than the sheer volume of votes or links a page receives and analyzes the page that casts the vote. Votes cast by web pages that have high a PageRank value because they are themselves important or are favorably viewed as "established firms" in the Web community weigh more heavily and help to make other pages look established as well.

The Google PageRank Algorithm Concept

The PageRank value of web page A is given as follows:

PR(A) = (1-d) + d(PR(T1)/C(T1) + … … … + PR(Tn)/C(Tn))

Here, the PageRank given by an outbound link equals the document's PageRank score PR(Ti) divided by the (normalized) number of outbound links C(Ti).

- **PR(A):** PageRank value of page A
- **PR(T1):** PageRank value of page 1 pointing to age A
- **C(T1):** Number of links off page 1, which points to page A
- **PR(Tn):** PageRank value of a page n pointing to page A
- **C(Tn):** Number of links off page n, which points to page A

- **d:** Dampening factor: The probability that, on each page, a random surfer will request another random page. Nominally this value is set to 0.85 and could be set between 0 and 1.

Let's assume a small universe of four web pages with the following relationship, as also shown in Figure 10–11.

- **Page A:** Doesn't link any page
- **Page B:** Link to page A and page C
- **Page C:** Link to page A
- **Page D:** Link to page A, page B, and page C

With a universe of four web pages, the initial approximation value of the PageRank would be evenly divided between these four web pages and is 0.25 (0.25 * 4 = 1). Assuming a damping factor of 0.85 for each page provides the following equation:

PR(A) = (1-d) + d(PR(B)/2 + PR(C)/1 + PR(D)/3)

PR(A) = (0.15) + 0.85(0.25/2 + 0.25/1 + 0.25/3)

PR(A) = (0.15) + 0.85(0.125 + 0.250 + 0.083)

PR(A) = (0.15) + 0.85(0.458)

PR(A) = (0.15) + 0.3893

PR(A) = 0.5393

In Figure 10–11 we can see the calculation of the PageRank value from our example using a mathematical notation.

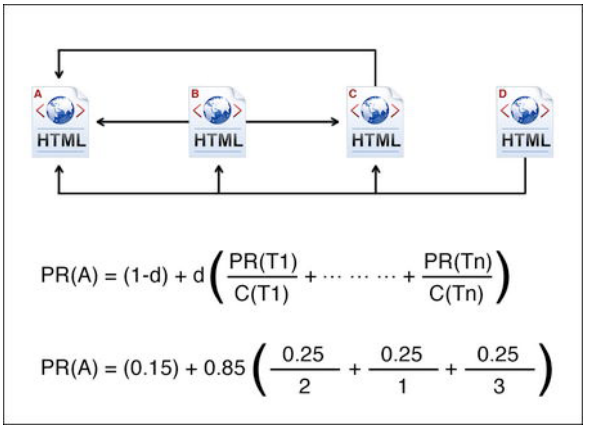

Figure 10–11. *The Google page rank algorithm: web page B, C, D add their PageRank value to web page A*

Figure 10–12 shows the same concept in which web page A receives a Page Rank value according to the PageRank value of each other web page that is linked to it.

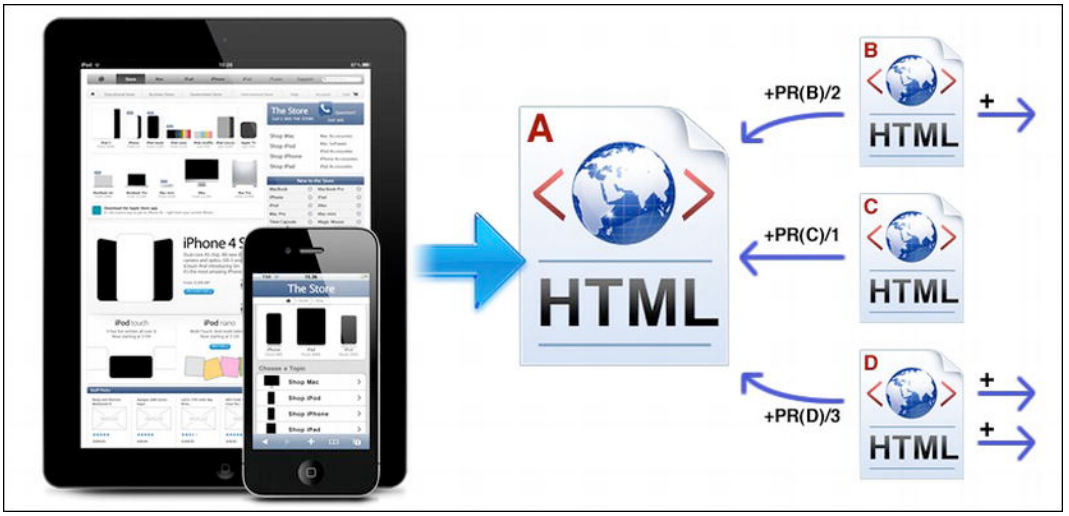

Figure 10–12. *The Google Page Rank Algorithm: web page B, C, D add their PageRank value to web page A*

Ingoing and outgoing links play an important role in the life of a WebApp, but links do much more than link other web pages. Links are also catalogued as internal and external. External links leave the web page and help the crawler to reach every page of your WebApp. In this case external links play the same role of a sitemap. That's why a sitemap is highly suggested in any web project. The same role is played by Breadcrumbs that link to many other (relevant) pages. On the other hand, an internal link is a link that, instead of pointing to a different web page, points to the web page itself.

Images

Search engines see web pages as text pages, which means that they don't understand images. Images have a fundamental role in our web sites and WebApps because as human beings we understand images much better than text. For this reason, we never avoid images in our projects.

The point here is to not rely on images when we need to give meaning to the web page. For example, don't insert important text messages in our images, such as calls-to-action or important titles. The role of the image is to support the content with a different, and possibly more powerful, series of symbols which readers can interpret.

Consider adding text messages as a companion to every image in a web page and an `alt` attribute to communicate with the crawler. Following the code is in the `<div>` that wraps the Hero Content in our "The Store" use-case home page.

```
<a href="#"><img src="pics/hero_iphone4.png" alt="The New iPhone4"/></a>
<a href="#"><img src="pics/hero_ipad.png" alt="The Revolutionary iPad"/></a>
<a href="#"><img src="pics/hero_ipod.png" alt="The New iPod Touch"/></a>
```

If we need to insert extra information about an element, we can choose to add the title attribute. We can also choose not to insert the `alt` attribute when the image doesn't

have any relevant meaning for the crawler and thus the web page. An example of this is the following code from the images used as icons in the edge-to-edge menu. The following piece of code refers to only one menu entry.

```
<li class="menu"><a href="#">
<img src="pics/menu_mac.png" />
<span class="name">Shop Mac</span>
<span class="arrow"></span></a></li>
```

This image doesn't specify any `alt` attribute in the `<img>` tag. However, if you add a tag with a description, it will not be considered a mistake.

```
<li class="menu"><a href="#">
<img src="pics/menu_mac.png" alt="Shop Mac" />
<span class="name">Shop Mac</span>
<span class="arrow"></span></a></li>
```

That piece of code shows a suitable description for this example.

JavaScript Code

JavaScript helps us to build a better WebApp and to emulate the native-app look, but this doesn't mean that it is always SEO-friendly. The solution is to externalize our JavaScript code exactly as we did for the Framework Core used in our "The Store" use-case.

```
<script src="javascript/functions.js" type="text/javascript"></script>
```

Apart from rare cases, we need to import all our JavaScript code, wherever included, in order to make our web page more SEO-friendly. On the other hand, this will add loading latency to our web page, but as we have established, perfection doesn't belong in this world. In these cases we need to interpret the context and choose the right approach according to our project scope, goals, and dimension.

Mobile SEO Tools

Nowadays we have many tools to monitor our web pages, from WebApps like Google Analytics to a few native iOS applications. Google analytics is a fast an easy way to monitor traffic and have a clear idea about how our WebApp interacts with users.

Google Analytics was developed by Urchin in 2005 and has been publically available to users since 2006. The benefits that come from using a tool like Google Analytics are various. Google Analytics helps you determine exactly which is the most effective web page, understand the average amount of time spent browsing our web pages, or and even understand which visitor became an effective user. These and many other types of data are organized in textual and graphic reports that are easy to analyze.

Google Analytics

After registering a Google account, we are able to log in to the Google analytics page located at http://www.google.com/analytics. Once logged in, we are able to add our WebApp from the Google Analytics Control Panel in few steps.

At this point, it is important to create and add a Sitemap to our WebApp in order to be sure that every URL will be discovered by the Google's normal crawling process. The Sitemap can also be used to provide metadata to Google about specific types of content like images, videos, news, and so on.

The last step before the data from our project is collected by Google Analytics is to add a snippet of code from the Control Panel to all our web pages just before the head closing tag. Information is collected and visualized using a few different Dashboard views. We have a Dashboard where all the information is visualized together for a glance at all the information at once, but we can also switch minimizing information in a single view. Some of these views are as follows:

- Content Overview
- Pageview Display
- Visits View
- Bounce Rate View
- Traffic Sources View
- Referring Sites View
- Search Engine View

It is important to remember that data are not collected in real time, and the statistics aren't available until midnight PST of each day. Google Analytics also takes a few hours to fully update all the statistics entries. In case our web pages are able to generate more than a million page views per month, it is useful to remind you that besides the free service offered by Google Analytics, there exists a Premium version of the service available for larger sites.

Resource on Optimization and SEO

In Table 10–3 we have some of the tools used in this chapter to optimize our project. If you are new to one or more of these technologies, we recommend you to continue the project using the following service.

Table 10–3. *Tools Used for Optimization and SEO*

Name	Type	URL	Operative System
Minify CSS	WebApp	http://www.minifyjavascript.com/	OSX – Win - Linux
Minify CSS	WebApp	http://www.minifycss.com/	OSX – Win - Linux
SpriteMe	WebApp	http://spriteme.org/	OSX – Win - Linux
Yahoo SmushIt	WebApp	http://www.smushit.com/	OSX – Win - Linux
PNGOut	Application	http://advsys.net/ken/utils.htm	OSX – Win - Linux
Google Webmaster Tools	WebApp	https://www.google.com/webmasters/tools/	OSX – Win - Linux
Google Analytics	WebApp	http://www.google.com/analytics/	OSX – Win - Linux

Summary

In this chapter we worked on optimization. In the beginning of the chapter, we established the proper way to work on performance-oriented optimization by first explaining how to write and produce good images and then learning how to compress our WebApp.

In the second part of the chapter we worked on usability optimization and introduced two types of approaches and tests also standardized in UML (Unified Modeling Language). We also saw how problems really affect our users and how to choose the right user sample for our purposes.

The third part of the chapter dealt with offline applications, and we saw how to use the Manifest File to cache single or multiple files of our WebApp.

In the final section we worked on a search engine oriented optimization in which we first introduced the concept behind the Mobile SEO and then introduced the anatomy of a search engine. We ended by working on the part of our web pages that needs to be optimized to make it search engine friendly.

We also introduced the concept behind the famous Google algorithm and we saw how a tool like Google Analytics can help us to gather important information that can be used to plan the right mobile strategy and make important decisions on our WebApp.