Mobile Development

Dr. Dobb's Journal

APRIL 2012

7 Steps to a Killer Mobile UI

Next

ALSO INSIDE

Writing Chrome Extensions >>

Fail-Safe Code? >>

From the Vault:
Displaying Tabular Data on iPhones >>

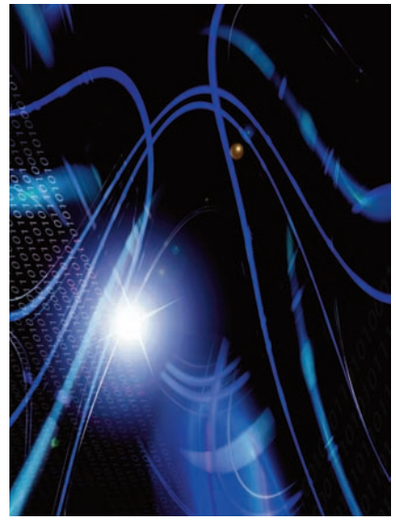
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

April 2012



COVER STORY

7 7 Steps to a Killer Mobile UI
By Tiago Simões
Providing a good user experience in mobile applications is paramount. This article offers seven steps to get you on the right path to delivering great mobile apps.

10 Writing Chrome Extensions
By Kausar Munshi and Basavaraju M
Extensions to Chrome enable you to extend the browser's capabilities, change how pages are rendered, and interact with the server in a controlled manner. For all their power, Chrome extensions turn out to be comparatively simple to write and build once you know how.

5 Guest Editorial
By Ron Wilson
No tool or combination of tools can assure fail-safe code.

18 From the Vault: Concurrency Runtime (iPhone) — The Task Scheduler
By Lahlali Issam
Visual C++ 2010 came with new features and enhancements to simplify native programming. The Concurrency Runtime (iPhone), for instance, is a framework that simplifies parallel programming and helps you write robust, scalable, and responsive parallel applications. This article explores the Task Scheduler, which schedules and coordinates tasks at runtime.

3 Letters
By you
Readers weigh in on the VS 11 Beta, Flash, the JVM, and more.

26 Links
Snapshots of the most interesting items on drdobbs.com including Cojac, CUDA-based plugins, and Android.

27 Editorial and Business Contacts

More on DrDobbs.com

Mastering Threads on MacOS X
The MacOS X platform has four thread solutions — two of which are obsolete as of version 10.7. Your choices are now POSIX threads or Cocoa's `NSThread` class. Here's how they compare and how to get up-to-speed on them quickly.
http://drdobbs.com/parallel/232602177

Looking for USB
Engineers hate reading directions almost as much as they hate asking for directions.
http://drdobbs.com/blogs/embedded-systems/232602380

Verdict on Visual Studio 11 Beta: Not Ready for Prime Time
Despite the benefits of this new release, its unattractive UI and poor usability ruin the product.
http://drdobbs.com/windows/232602205

Agile Success Factors
Scott Ambler's survey finds that at many sites, Agile approaches work at scale, although it's harder for them to succeed at scale.
http://drdobbs.com/architecture-and-design/232601858

Creating a Custom Output Cache Provider in ASP.NET 4
ASP.NET 4 provides support for cache extensibility through a provider model that you can write yourself. The primary benefit of this approach is that you can cache objects outside of the ASP.NET Worker Process. Getting the performance benefits of this customized cache control requires a few basic steps, presented here.
http://drdobbs.com/windows/232601001

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Mailbag

Visual Studio 11 Beta, Flash appreciation, the JVM, and CISC/RISC divide

First Look at VS 11

In response to Andrew Binstock's editorial (http://drdobbs.com/windows/232602205) expressing concern about the Visual Studio 11 beta release , we received:

The title of the editorial caught my attention: "Verdict on Visual Studio 11 Beta: Not Ready for Prime Time." After reading the article, I thought the title was misleading since the title doesn't specify only the UI but implies a broader scope. Perhaps "Verdict on VS2011 Beta: Drab and Ugly" instead.

— Ryan Van Slooten

On Mobile Driving Language Adoption

In response to the editorial, "The Rise and Fall of Languages in 2011" (http://drdobbs.com/mobile/232400093), which concluded that mobile development was largely driving recent language adoption:

While hardware gets better, faster, and cheaper, application development productivity is becoming more complicated, slower, and more expensive. There was a reason Adobe Flash was king. It was better, faster, and cheaper for software development. Write it once, deploy it everywhere. That was until the Apple empire became so powerful that they could force the end of efficient multiplatform development. If Apple allowed the creative development community to deploy a Flash player for iDevices, developers would have continued on the path of better, faster, cheaper for software. There is no reason to use a proprietary low-level language like Objective-C, other than you have to. This sounds like the old IBM game that lead to their downfall. They were so big that companies did what they demanded, until companies got smarter and did what they wanted, taking advantage of the whole open platform world. Companies might follow Apple, too, while they have the controlling market share, but nothing lasts forever. Somebody, somewhere is working on the next new thing, and it is unlikely to require Objective-C. I want better, faster, and cheaper for my software developers.

— Steve Wilfe
Denver, Colorado

The JVM as a Language Incubator

In response to the editorial discussing the JVM as a fertile platform for the development of new languages (http://drdobbs.com/jvm/232600846), and that these languages tend to be entirely new efforts, better versions of Java, or simply ports of other languages to the JVM :

I think that the category mappings in the article don't really fit. For example, NetRexx is really just the JVM version of Rexx as far as I am aware; wouldn't it belong in the "ports" category? Additionally, Mirah,

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

while heavily inspired by Ruby, is not trying to be another Ruby; so I'm curious as to why it would belong in the "ports" category. I think some people may quibble about whether or not Scala should be in the "better Java" or "its own thing" category.

I think there is also one notable omission: Rhino — the JavaScript implementation for the JVM. It has been around for ages. It powers a fair bit of server-side JavaScript on the Web. Common uses include server side JavaScript processing for initial page hits.

Other interesting omissions: Erjang, and Erlang port, and there are at least two versions of PHP ports for the JVM that I've heard of. Erlang has some unique features, and if it could be successfully married to all of the rocket science and investment that go into Hotspot, it could produce something very powerful, however unlikely that may be. PHP is worth noting if only for its ubiquity.

I hope you get to Clojure soon. There's a lot more to it than in many of the other languages mentioned in the article.

— Bill Robertson

CISC vs. RISC on Intel Processors

In response to a comment that appeared in this column on page 3 of the March issue, in which Andrew Binstock gave a rough time frame for Intel's adoption of RISC instructions in the innermost CPU execution instruction, we received:

The response in the "Mailbag" was well wide of the truth, and misunderstands what the distinction between RISC and CISC is. To describe a CISC MUL instruction as being converted to RISC shift/addition is to miss the point. CISC vs. RISC was not about circuit design, but about reducing instruction decode time by reducing the number of instructions to the ones most commonly used. For example, instead of {8,16,32} bit versions of multiply, you reduce to the one used 99% of the time and save on decode-time for every instruction. The original idea was that you could do away with microcode altogether and deal with incompatibilities in the compilers…but it did not work-out that way.

Is RISC/CISC a meaningful distinction? No: CISC processors have enough transistors to dispense with microcode (if they wanted to) and RISC processors need complex instructions for multicore locking. The difference between an ARM and an Atom processor is more about circuit design than which processor allows a WORD load from an non-WORD-aligned address.

—Steve Channell

To which Andrew Binstock responds: "I don't disagree with the second part of your analysis; namely, that CISC vs. RISC is a distinction of diminishing relevance. As to the first point, Intel was definitely doing RISC instructions at the time I mentioned. Here is the description written by Hennessy and Patterson (the fathers of RISC) in *Computer Architecture — A Quantitative Approach*, 3rd Ed. (http://is.gd/qqitUh): "The P6 [so, Pentium III — *Ed*.] translates each IA-32 instruction into a series of micro-operations (uops) that are executed by the pipeline: the uops are similar to typical RISC instructions." I didn't and don't claim that the MUL instruction was implemented in RISC, only that its ability to execute in half a clock cycle occurred at the same time that Intel was using RISC instructions in its processors. I stand by that assertion and all my attempts to find counter arguments suggest that it is in fact correct."

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

No Tool or Combination of Tools Can Assure Fail-Safe Code

The government's analysis of Toyota's automotive code revealed errors and failures. But despite extensive overlapping forms of testing and modeling, it could not prove or clear the software as a contributor to unintended acceleration.

By Ron Wilson

Last year, the U.S. National Highway Traffic Safety Administration (NHTSA) released an enormous report on its investigation into unintended acceleration in Toyota cars. It all seems very familiar, if you remember the Audi 5000. New electronic throttle control reaches the market. Reports of failure emerge from a few users. The vendor denies any problem. A few engineers quietly report having reproduced the problem. But an intensive publicly funded investigation finds nothing.

What makes this report particularly interesting is that the NHTSA called in an evaluation team from the space agency, NASA, to do the heavy lifting. And that team included a software evaluation group. While the hardware folks were shaking, baking, and irradiating cars and car parts, the software team had at the Engine Controller Module code for the four-cylinder 2005 Camry — all 280K lines of ANSI C. The team's report, included as part of the larger "Report on Unintended Acceleration in Toyota Vehicles" (www.nhtsa.gov/UA) could be a case study.

The NASA team applied static source-code analysis, formal logic model checking, and algorithm analysis through simulation. The report states, "The team's experience is that there is no single analysis technique today that can reliably intercept all vulnerabilities, but that it is strongly recommended to deploy a range of different leading tools."

For code analysis, the team used Coverity, CodeSonar, and Bell Labs' Uno (http://spinroot.com/uno/) to identify "common coding defects and suspicious coding patterns." They also used CodeSonar to compare Toyota's code against a Jet Propulsion Lab coding standard.

For model checking, the team used open-source Spin (http://spinroot.com/spin/whatispin.html#A) and its companion tool Swarm (http://spinroot.com/swarm/). Here, the tale gets more interesting. To use a formal model checker, you first have to write formal models. The team decided to build models only for those software modules they believed could be culprits — so the formal analysis depended upon human judgment of possible fault modes.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

The algorithm analysis started with — once again — building models, this time in Matlab. That process was undertaken by reading Toyota documentation and talking with Toyota engineers. It then progressed to analyzing the source code, and finally testing the models against actual Camrys. Once the NASA team was satisfied with the models, they explored failure scenarios in Simulink and checked delays with AbsInt aiT.

Some conclusions suggest themselves. First, there are no silver bullets: Effective debugging means using everything you've got. Second, even when it's grounded in exhaustive and formal techniques, an evaluation is circumscribed by the evaluators' beliefs about the possible behavior of the system. Third, there is no certainty. Despite Toyota's great care in developing their code, NASA's analysis found significant errors, including serious underestimates of delays in the multiprocessing system. But the investigation could not link those errors to any proposed mechanism for unintended acceleration. Contrary to what you probably read in the papers, the NASA Executive Summary stated, "Because proof that the ETCS-I [that is, the Electronic Throttle Control — *Ed*.] caused the reported UAs [unintended accelerations] was not found does not mean it could not occur."

— Ron Wilson is the guest Editorial director of Embedded.com, EDN, and the Designlines at UBM Electronics. This is a slightly revised version of this article, which first appeared in EETimes.

Comment

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

7 Steps To A Killer Mobile UI

Here's how to write a successful mobile app

By Tiago Simões

Odds are, you have a smartphone and without realizing it, have developed a special, personal relationship with it. We only recently learned that Steve Jobs charged the team he assembled to create the first iPhone to "Create a phone that people will fall in love with." Whether we admit it or not, all of us have a special connection with our iPhone, Blackberry, Android, or Windows Phone.

But love is fickle. A recent Harris Interactive study revealed that bad mobile apps can dramatically damage a brand's reputation. Almost one-third of respondents said they told others when they had a bad experience with a mobile app. Still, more than half say they've recommended an app based on a positive experience.

Providing a good user experience in mobile applications is paramount. Here are seven steps to get you on the right path to delivering great mobile apps.

1. Define Your Goals

What are you trying to accomplish with the mobile app? Most importantly, what will your users want from it? You must define the goals your app will address. For this, you need to understand the day-to-day activities of your users related to this app, and their goals and motivations around it.

A good technique is to create personas — fictitious characters that represent your users — and write your specs as agile user stories. This approach puts you in the correct mindset to state requirements for your app. Something like this: "As a security officer, Johnny Bravo must check-in using his mobile phone to find out the next checkpoint that he should walk to so he can complete the randomized patrol tour." This tactic lets you see the application from the user's point of view.

There are, of course, differences between user stories for mobile and desktop applications, particularly when considering the mobile

context. For instance, a mobile user might be walking or using one hand.

After gathering several of these stories, you must prioritize them according to their frequency. I'll explain why this is important shortly.

2. Analyze Existing Apps

If it's the first time you're creating a mobile app, take a close look at the interfaces of existing ones. Two obvious things stand out: The screen size is much smaller and since many mobile devices use touch screen, the elements on the screen must be larger, so they're easy to touch.

The smaller screen and the larger elements mean that you can only put a limited number of items on the screen.

It can be a challenge to select what should go on the mobile screen; some usability experts even advocate that the mobile version of a website be created before the desktop version.

Prioritizing your user stories can help with this. You'll want to take a hard look at the features that your top priority users are using. Figuring out the 20% of the features that will be used 80% of the time is the goal of every usability expert. If you do this correctly with mobile, the desktop will also have its priorities right.

3. Native App or Mobile Web?

You'll need to decide whether to use HTML5 or native APIs. This decision has more implications for the technical implementation than for the user interface, however. Native apps usually run faster and are the best option for games, offline, and hardware-intensive apps. Mobile Web apps are quicker to implement, easier to maintain, and often better suited for enterprise applications. A hybrid approach makes sense if you want mobile Web apps easy maintenance but need specific hardware capabilities, such as the phone's camera or GPS.

An early Palm Pilot prototype side by side with the real thing.

Taking advantage of existing frameworks and platforms can be a solution, but whichever option you choose, make sure you can create and modify the UIs quickly, and that you'll be able to iterate often.

4. Prototype Quickly

A fast way to test a mobile interface at a project's start is to use a low-fidelity prototype. Before the Palm Pilot's launch, it's said that the inventor carried around a block of wood the size of the device to see how it would feel in his pocket. Later, he'd have several rough versions of the UI sketched on it.

Prototyping a mobile app is easy: A pencil and paper are all you need, and because screens are small, you will not have much to draw. Test your prototypes by asking users to try to accomplish tasks on them. See what they do, and ask them what they're thinking. The results will be surprising and help you improve the design. A great design mantra is "Don't Make Me Think!" If your mobile design fails this test, your app will fail as well.

5. Avoid Errors That Ruin the UX

When people start creating mobile applications, they typically make some known errors. First, if they're creating the mobile version of an existing website, developers often try to replicate every feature of the desktop, failing to consider the intent of the mobile user.

Another common error is to have a navigation structure with too many levels. Deep structures aren't well suited to mobile; simpler models work better. Expecting users to be able to easily input text is another mistake: Entering text is much harder on mobile devices, so you'll want to minimize the need for input.

The applications that work best on mobile let users read content more than write it. Think about it: Do any of your favorite mobile apps require you to enter a lot of text?

6. Add Delightful Details

A mobile phone is always with you and knows where you are, which is why mapping applications are so successful. Phones can hear what you hear and see what you see, which explains the popularity of apps like Shazam and Instagram. Using location, the camera, and the microphone are all smart ways to get the same context as the user, and you can take advantage of that context in unique ways.

Even if you aren't using the mobile device hardware, there are several tricks you can use to delight users. Animations can add excitement to your application. If you're creating a mobile Web app, cache static content, and consider using a content delivery network so pages load faster.

Another welcome touch for mobile Web apps is to let users add them to their home screen, making them look and feel like a native app. If you do this, remember to include a back button on your UIs, giving users an accessible exit option.

Remembering user selections across sessions (for example, a recent items list) is another smart touch that will make the app more context aware and free users from having to repeat navigation steps. Together, these usability quick-wins work to make the user's life easier.

7. Fail Early, Recover Fast

Even with a good team of designers and developers, your first attempt at creating a mobile app will likely fail. (A recent survey showed that 38% of people are unsatisfied with the branded apps they tried.) The best strategy is to fail early and iterate quickly, learning how people are using your app and improving it continuously. Agile methodologies are a great way to align the mobile UIs you deliver with what users need.

Making the killer mobile UI is hard, but the strategies outlined in this article will help. Follow them faithfully, and you're on your way to writing some great mobile apps!

— Tiago Simões is a senior software engineer in the R&D group at OutSystems, a cloud-based application developer.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Writing Chrome Extensions

Extensions to Chrome enable you to extend the browser's capabilities, change how pages are rendered, and interact with the server in a controlled manner. For all their power, Chrome extensions turn out to be comparatively simple to write and build once you know how.

By Kausar Munshi and Basavraju M

Chrome Extensions are the code components used for adding new or enhanced features in the Chrome Web browser. In other words, the functionality of the browser relies substantially on add-ons or plugins to Chrome. In this article, we explain what an extension consists of, how it can be developed, and then we provide a sample extension to illustrate the techniques.

The Anatomy of Chrome Extensions

A Chrome extension is a zipped bundle of files — HTML, CSS, JavaScript, images, and related extension files. Extensions are essentially Web pages, and they can use all the APIs that Chrome provides to Web pages, from `XMLHttpRequest` to JSON to HTML5.

A Chrome extension comprises the following files:

- A manifest file (mandatory).
- One or more HTML files (mandatory) such as background.html, popup.html.
- One or more JavaScript files (optional).
- Any other files your extension needs — for example, image files (optional).

Before going into the details of the extension's components, let's see how to load and view an extension. Assuming you have all the extension-related files:

- Create a folder anywhere on your system to hold the above files..
- Go to the Extensions Management page in Chrome by clicking the wrench icon and choosing Tools > Extensions, and click on Developer mode option.
- Use the Load unpacked extension button and specify the extension folder. This step will load the extension.
- If the extension is valid, its icon appears next to the address bar, and information about the extension appears in the extensions page.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

The extension is now ready to be used. Clicking on the extension's icon (next to the address bar) enables interaction with the extension. Note that this page provides options to Reload/Uninstall options for the extension. Now, we'll examine the contents of the extensions.

The Manifest File

The manifest file, called "manifest.json," provides information about the extension such as the most important files and the capabilities that the extension uses.

If you're not familiar with JSON , it stands for JavaScript Object Notation, and it's a lightweight text-based open standard designed for human-readable data interchange in JavaScript. In this way, it serves as an alternative to XML. It is used for representing simple data structures and associative arrays, called objects. It is language-independent, with parsers available for most languages. JSON files are denoted by the extension ".json." JSON is often used for serializing and transmitting structured data over a network connection. Here's a typical manifest file:

```
manifest.json
{
  "name": "Kz_FACE_LIFT",
  "version": "1.1",
  "description": "Kz FACE_LIFT PLUG-IN.",
  "background_page": "background.html",
  "options_page": "options.html",
  "browser_action": {
    "default_icon": "images/clock.png",
    "popup": "popup.html"
  },

"content_scripts": [
{
  "matches": ["<all_urls>"],
  "js": ["page.js"]
}

"icons" : {
  "48": "a_48.jpg",
  "128": "a_128.jpg"
}
```

The manifest file is the heart of the Chrome extension. It contains several name-value pairs specifying what functions are to be performed by whom. The attribute's name, version and description are metadata for the extension, whereas the attribute's `background_page`, `options_page`, and `content-scripts` give information about the files that will be holding the corresponding logic (which will be described later).

The `browser_action` attribute, which is a composite JSON object (that is, made up of one or more JSON objects), is a way of specifying the user interface (UI) to the extension. It holds the details of the image button or the UI that will be provided to the user to access the extension: It has the child attribute `default_icon` that specifies the icon to be used next to the URL in the address bar; and the child attribute `popup` contains the name of the HTML file that will be displayed when you click on the icon in the address bar.

The composite attribute icons specify the images that are to be displayed at various places. The child attribute `48` specifies the image/icon that is to be displayed during the installation and uninstallation process of the extension. The child attribute `128` specifies the image/icon that is to be displayed in the chrome://extensions page where details of all the installed extensions are listed. The child attribute `16` has the same functionality as the `default_icon` attribute.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Extensions get the user interface in the following two ways:

- Browser action: An extension that enhances the functionality of the browser irrespective of the pages. As long as the browser is running, the extension will be there. It adds an action item to the browser. The action item is added in the browser's toolbar in the aforementioned sample manifest file.
- Page action: An extension that enhances the functionality of the browser depending on the Web page being visited. It will be active as long as that page (or any page from that domain) is open in the browser. It adds an action item to some of the pages that depends on the program logic. The action item is added in the address bar for valid page addresses.

At most, an extension can have either one browser action or one page action.

Background.html

The main logic of the extension resides in the background.html file. When an extension needs to have a single long-running script to perform some task, background pages are the solution. Although it is an HTML file, its purpose is not used to add a GUI to the extension. Everything in this file is housed inside a `<script>` tag or contains a reference to a JavaScript file that holds the main logic.

This file exists for the lifetime of extension, and only one instance of it per extension is active at a time. For extensions with only background work to do, the UI is implemented by dumb views. That is, the view collects any data from the background page as needed and displays it on click. The manifest.json file specifies which HTML file will hold the background processing logic. It does this via the attribute `background_page`.

A sample background.html is shown here:

```
<html>
  <body>
     <script src="script.js"></script>
     <script type="text/javascript">
     chrome.tabs.onSelectionChanged.addListener(myFun4);
     chrome.tabs.onUpdated.addListener(myFun4);
     var val = 10 ; //declare a sample variable
  </script>
</body>
```

Popup.html

Popup.html is basically a simple view that extracts data from background pages or content scripts (which are JavaScript files) and displays them. Thus, it provides a way of displaying the results of the operations performed by the extension or the state of the extension.

In the manifest.json file, we describe which HTML file will act as the UI using the attribute `popup`. Note: There are many ways of displaying results to the user. We can use other HTML pages. The only difference is that we need to load those additional pages programmatically as needed, whereas with popup.html, we do not need to write any logic for loading because it is attached with one of the standard Chrome event "browser action/page action" extensions — whenever we click the icon of the extension, popup.html will be displayed.

A sample popup.html is shown next, which just displays "Hello World" text when the extension icon is clicked (it extracts data from background.html):

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

```
<body>
  <h1>Hello World</h1>
  <script>
  var newwin = chrome.extension.getBackgroundPage();
  alert(newwin.val);//where val is a var declared in
                    //the background page.
  </script>
</body>
```

The `chrome.extension.getBackgroundPage()` returns the window object of the background page from which we can access all the variables of the background page that are in the window scope.

There is also an alternative way for the extensions pages to interact. They can use message passing, as will be discussed in the section on content scripts .

Options.html

This file enables the user to customize the behavior of the extension. The options page should be included in the manifest.json file. It provides a link to the options page from the extensions management page at chrome://extensions. In the manifest.json file, you describe which file will be used as the options page using the attribute `options_page`.

Content Scripts

A content script is a JavaScript file that executes in the context of a Web page that's been loaded into the browser. Think of a content script as part of that loaded page, not as part of the extension it was packaged with. It interacts with the Web pages visited by the browser. It can also interact with the extension pages by exchanging messages (message passing). In the manifest.json file, you describe which file will be used as the content script using the composite attribute `content_scripts`. It consists of a child attribute named `js` that tells which file acts as the content script. The child attribute `matches`, which holds a regex-pattern/url-value, tells the extension on which domain names and URLs the content script is valid.

A sample of content script file is shown next, along with the message-passing method of interaction. It sends a request to the background.html page passing `status=2` as data, and waits for response through the handler in which it displays the `counter` value from the response received.

```
var payload = {status: 2};
chrome.extension.sendRequest(payload,
  function handler(response) {
      var a = response.counter;
      alert(a);
});
```

The background.html file shown next handles the above request sent by the content script.

```
function onRequest(request, sender, sendResponse)
{
  var num1 = request.status;
  if (num1==2)
  {
      Num2=20;
      sendResponse({counter:Num2});
  }
}
//register this function
chrome.extension.onRequest.addListener(onRequest);
```

When the background page receives the request, it sends a response back if the value received is `2`. In addition to the exchange of data with

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

the background page, the content scripts can do much more by accessing the data of the loaded Web page, such as the document object and its properties, and also modify them as needed.

Chrome API

In addition to having access to all the APIs that Web pages and apps can use, extensions can also use Chrome-only APIs (often called chrome.* APIs) that allow tight integration with the browser. For example, any extension or Web app can use the standard `window.open()` method to open a URL. But if you want to specify which window that URL should be displayed in, your extension can use the Chrome-only `chrome.tabs.create()` method instead.

Content.js

Table 1 lists the supported chrome.* APIs.

Asynchronous vs. Synchronous Methods

Most of the methods in the chrome.* APIs are asynchronous: They return control back to the browser/script without waiting for the operation to finish and while the operation continues to happen in the background. To get the outcome of an operation after it is complete, a `callback` function is passed upfront to the method, which will be executed later (potentially much later), after the operation is complete. Here's an example of the signature for an asynchronous method:

```
chrome.tabs.create(object createProperties,
  function callback)
```

Few chrome.* methods are synchronous. Synchronous methods will not have a callback because they don't return until they've completed all their work. Often, synchronous methods have a return type. For example:

bookmarks	idle
browserAction	management
contextMenus	omnibox
cookies	pageAction
extension	proxy
fileBrowserHandler	tabs
history	types
il8n	windows

Table 1: The supported Chrome-specific APIs.

```
DOMWindow chrome.extension.getBackgroundPage()
```

This method has no callback and a return type of `DOMWindow` — it synchronously returns the background page and performs no other asynchronous work.

Using a Callback

To navigate the user's currently selected tab to a new URL, the current tab's ID is acquired (using `chrome.tabs.getSelected()`) and then that tab is directed to the new URL using `chrome.tabs.update()`. A synchronous `getSelected()` could be written as below:

```
var tab = chrome.tabs.getSeleceted(null); // WRONG
chrome.tabs.update(tab.id, {url:"http://abc.com"});
.....
// THIS CODE DOESN'T WORK
```

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

This approach fails because `getSelected()` is asynchronous. It returns without waiting for its work to complete, and it doesn't even return a value (although some asynchronous methods do). `getSelected()`, or any method in general, is identified as asynchronous by the presence of the callback parameter in its signature. For example: `chrome.tabs.getSelected(integer windowId, function callback)`.

The following code shows how to define a `callback` function that gets the results from `getSelected()` (as a parameter named `tab`) and calls `update()`.

```
chrome.tabs.getSeleceted(null, function(tab){
chrome.tabs.update(tab.id, {url:"http://abc.com"});
.....
});
//THIS CODE WORKS FINE
```

The `callback` function specified to `getSelected` is called only after information about the currently selected tab is available, which is sometime after `getSelected()` returns. Also note that although `update()` is asynchronous, this example doesn't use its callback parameter, as we don't do anything with the results of the update.

Cross-Origin XMLHttpRequest

Extensions can interact with the outside world using `XMLHttpRequest` objects available in JavaScript. Unlike regular Web pages, which can use `XMLHttpRequest` to send and receive data from remote servers, but are limited by the same origin policy, extensions aren't limited this way. An extension can talk to remote servers outside of its origin as long as it has requested cross-origin permissions in its definition.

Extension Origin

Each running extension exists within its own separate security origin. Without requesting additional privileges, the extension can use `XMLHttpRequest` to get resources within its installation. For example, if an extension contains a text file "abc.txt," the extension can retrieve the file's contents as shown in the following code.

```
var xmlUnameRequest = new XMLHttpRequest();
var url = "/abc.txt";
xmlUnameRequest.open("GET", url, true);
xmlUnameRequest.onreadystatechange = handler;
xmlUnameRequest.send(null);
function handler(evtXHR){
  if (xmlhttp.readyState == 4&&xmlhttp.status == 200) {
      alert(xmlhttp.responseText);
  }
  else {
      alert("Error "+xmlhttp.status);
  }
}
```

If the extension attempts to use a security origin other than itself, say http://www.google.com, the browser disallows it unless the extension has requested the appropriate cross-origin permissions.The same permission can be made available to the extension by specifying those host names in the manifest file as below.

```
{
"name" : "Kz_Face_Lift",
"permissions":[
   http://www.google.com/,
   http://*/*,
   https://*/
  ]
}
```

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

A match pattern of `http://*/*` allows HTTP access to all reachable domains. Match patterns are similar to content script match patterns, but any path information following the host is ignored. Access is granted both by host and by scheme. If an extension wants both secure and non-secure HTTP access to a given host or set of hosts, the permissions must be declared separately with `https` and `http` as in the aforementioned example. While using an `XmlHttpRequest` object in the plugin pages, you can pass the destination URL (http://www.mysite.com/getData) after ensuring the same was specified in the manifest file.

Packaging and Deployment

To deploy the extension, the contents of the folder can be packaged into a special ZIP file that has a .crx suffix.

1. Bring up the extensions management page by going to the following URL: chrome://extensions
2. Go to Developer Mode and Click the pack extension button. A dialog appears.
3. In the extension root directory field, specify the path to the extension's folder — for example, c:\myext. (Ignore the other field; you don't specify a private key file the first time you package a particular extension.)
4. Click OK. The packager creates two files: a .crx file, which is the actual extension that can be installed; and a .pem file, which contains the private key.
5. The .crx file can be hosted on a Web server for the end users to download onto their Chrome browsers.

The .pem file must update the extension and upload the extension to the Chrome Web Store. On successful packaging of the extension, a dialog box is displayed providing the information regarding the location of .crx and .pem files. (Figure 1.)

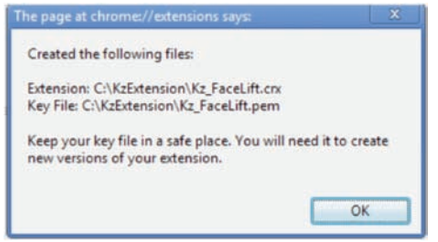


Figure 1: The dialog box confirming successful packaging of an extension.

Packaging also can be done at the command line by specifying the folder where the plugin resides; and upon successful packaging, the same dialog box will pop up.

```
chrome.exe --pack-extension= C:\KzExtension\Kz_FaceLift
```

Hosting

The Web server that hosts .crx files must use appropriate HTTP headers, so that end users can install the file onto their browsers by clicking a link to it. Google Chrome considers a file to be installable if the file has the content type application/x-chrome-extension or the file suffix is .crx and both of the following are true: The file is not served with the HTTP header `X-Content-Type-Options: nosniff` option, and it is served with one of these content types:

- empty string
- "text/plain"
- "application/octet-stream"
- "unknown/unknown"
- "application/unknown"
- "*/*"

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

The most common reason for failing to recognize an installable file is that the server sends the header `XContent-Type-Options: nosniff`. The second most common reason is that the server sends an unknown content type — one that isn't in the previous list. To fix an HTTP header issue, either change the configuration of the server or try hosting the .crx file on another server.

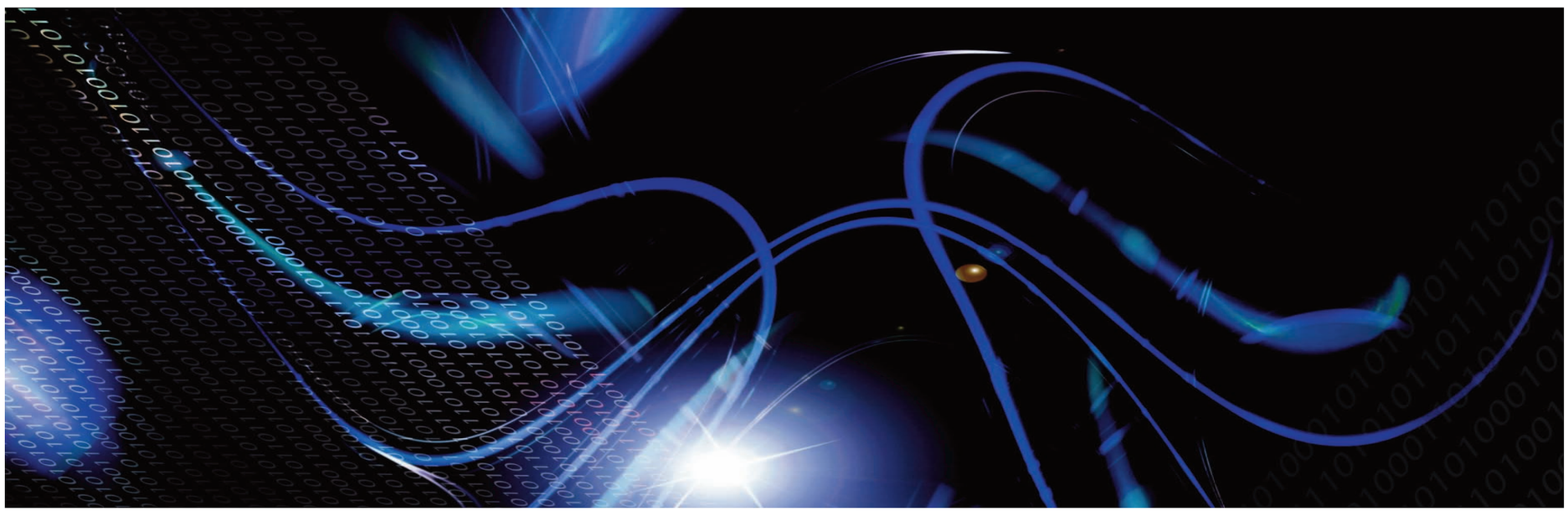
With this introduction, you should be well on your way to building fun and useful extensions to Chrome.

Useful References

http://code.google.com/chrome/extensions/getstarted.html
http://en.wikipedia.org/wiki/JSON
http://code.google.com/chrome/extensions/api_index.html
http://www.w3.org/TR/XMLHttpRequest/

— Kausar Munshi works as a developer in Java and Web Technologies. Basavaraju graduated from IIT Madras and has approximately 15 years of experience in IT.

Comment



From the Vault

Displaying Tabular Data on iPhones

This handy article from last year tackles the big problem of handling lots of data on a small screen. — *DDJ*

By Tom Thompson

One of the great challenges facing developers of business apps that run on mobile handsets is the presentation of tabular data in an area slightly larger than a business card. Presenting sufficient data at a size that is conveniently readable is a difficult task, which must be done correctly. Flub this aspect and no one will use the app, regardless of how good its other properties might be.

Fortunately, Cocoa Touch, the object-oriented framework used in programming for Apple devices such as the iPhone, iPod Touch, and iPad, provides classes that display data in a compact, tabular format. One of the classes, `UITableView`, helps arrange such data into a hierarchic structure of lists, with each list presented as a table on the screen. Other supporting classes and methods enable a user to navigate easily through these tables (screens) of information. The user can then quickly drill down through a complex data set to reach the desired information.

Because `UITableView` is good at managing and displaying information, it often serves as the workhorse in data-centric applications. Unfortunately, the information on using `UITableView` is scattered throughout volumes of documentation in the Apple iOS SDK. This article consolidates this information and introduces you to `UITableView` and its related classes. In addition, I demonstrate how to use these classes through two example iOS apps.

Table Terminology and Types

In the abstract, table data is presented as rows of content, and the rows can be formatted in a specific style. Exactly what constitutes a row and a column though is a looser concept on mobile devices than it is on the desktop. An iOS table *row* is a horizontal strip of content drawn across the width of the screen. Rows are often referred to as cells, from the `UITableViewCell` class that implements them. The content of a row can be a mix of text or graphics, and you can specify the row's height in points. However, because a row spans the width of the screen, a table displays only one column of information.

Rows of related content can be grouped into *sections*. Sections are delineated by a distinctive header strip that appears at the top of each group and contains a title. An optional footer can appear below each group; it also has a title. The titles can consist of strings or graphics. Like the cells, you can specify the height of headers and footers. The entire table can also have its own optional header and footer.

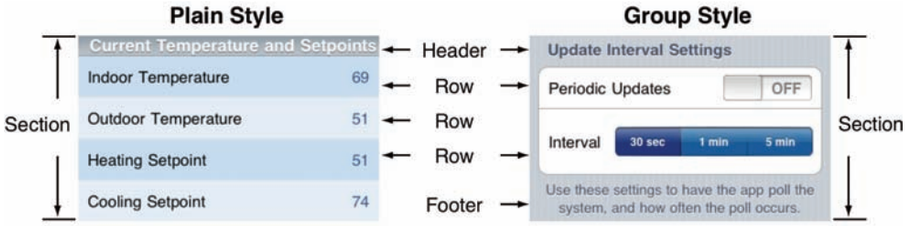


Figure 1: Comparing plain and group-style tables in iOS.

iOS tables come in two styles: plain and grouped. For the plain style, a table consists of section headers, followed by rows of content displayed as lists. For example, the Mailboxes screen of the iOS Mail app demonstrates this style. For the grouped style, the sections occupy distinct screen regions, with the header and footer content appearing above and below them. The Settings app is an example of this style.

Figure 1 shows these two table styles, and indicates the sections, headers, and footers. Note that while the plain style does support footers, they are often not used.

When you first create a table, you specify its style, either directly in the code or in Interface Builder. Once set, a table's style cannot be changed.

Table rows (cells) have four predefined styles: default, style1, style2, and subtitle. These styles vary slightly in the number of built-in text fields they offer and whether an optional image is displayed. Different cell styles can appear in a table, and a cell's style can change on the fly, as driven by user actions.

Cells can also be customized to present additional text labels or images, or present controls such as sliders, buttons, and switches. For example, in Figure 1 the grouped table (right) has cells that contain a switch and a segment control.

Table Classes

A key class for managing an iOS device screen is the `UIView`. It is responsible for painting content onto the screen and capturing user events (typically touches). I have discussed the `UIView` class and shown how to use it in the *Dr. Dobb's* article, "The iPhone Isn't Easy" (http://drdobbs.com/mobile/224200243).

I mention the `UIView` class here because `UITableView` and `UITableViewCell` subclass it. Therefore, they inherit the ability to display content and respond to touch events. However, these classes do not store data. They only display the data and enable the user to interact with the app's data repository (which is known as the data model).

`UITableView` is responsible for implementing a table's layout and certain behaviors. It uses cell objects (described next) to organize and display the data. As Figure 2 shows, `UITableView` also inherits from `UIScrollView`, which allows the table to be scrolled. However, the table can scroll vertically only.

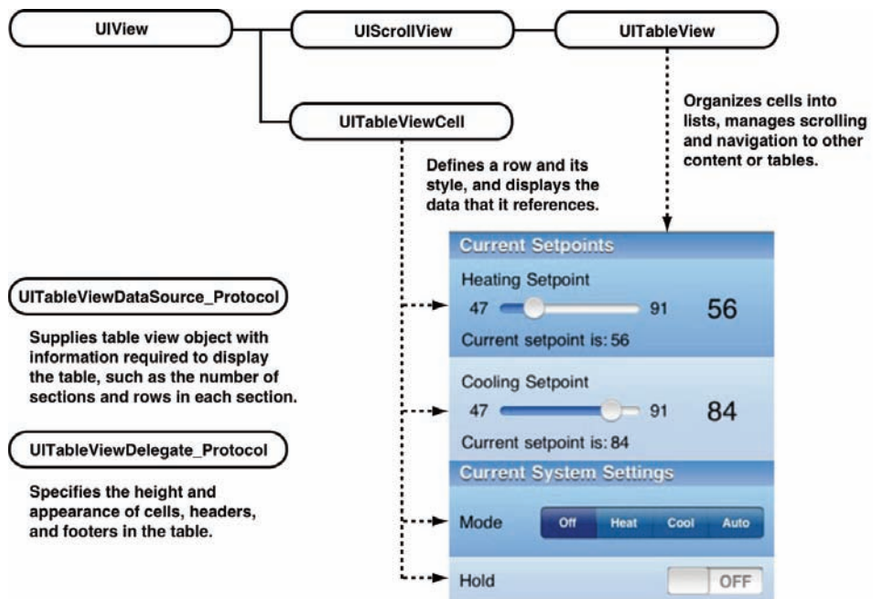


Figure 2: How the Cocoa Touch objects and protocols interact to lay out a table.

The `UITableViewCell` class is responsible for displaying a row and its content. This class serves as a template that clones as many instances of a cell as is required to make up a section. In this sense, `UITableViewCell` functions similarly to a Java factory method. Each cell can have a different style and displays the information that it references in the data model. As mentioned earlier, a cell can display text, images, or controls. Cells are complex objects with many built-in views, including several content-specific views, a background view, and an accessory view. Figure 3 illustrates this. The accessory view displays controls or optional images that assist in navigating to another screen.

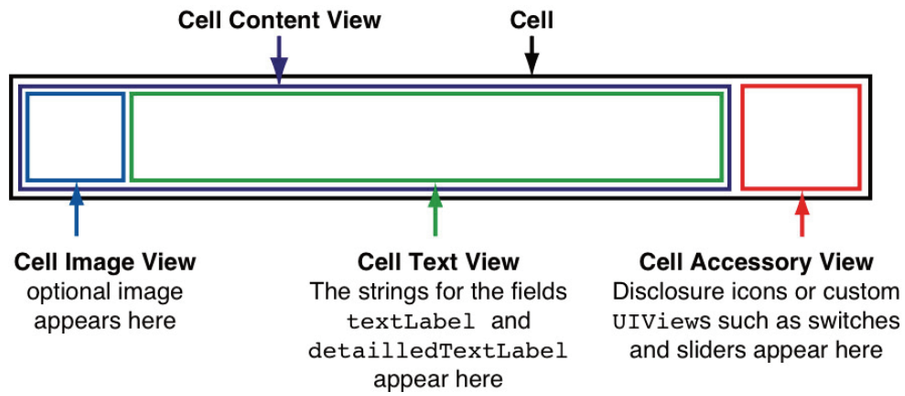


Figure 3: Some of the key views in a cell using the default style.

Table Protocols

In iOS parlance, a protocol defines method declarations and the desired responses when these methods are invoked. It's up to the developer to assign a delegate object that implements the protocol. An Objective-C protocol therefore functions like a Java interface. The iOS runtime is made aware that a class is a delegate object by the addition of a suitable protocol declaration in the class's header file.

Tables use two protocols: `UITableViewDataSource_Protocol`, and `UITableViewDelegate_Protocol`. The `UITableViewDataSource_Protocol` obtains information for `UITableView` about table layout and how to supply data to it for display. For table layout, this protocol defines two required methods. The first, `numberOfRowsInSection`, queries the app about the number of rows a section has. The second method, `cellForRowAtIndexPath`, retrieves a specific cell in the table so that it can be matched up to its data source and the data displayed.

The `UITableViewDelegate_Protocol` describes how the table behaves and its appearance. This protocol is employed to customize the appearance of headers and footers, modify a cell's height, and specify a background color or image. All methods in this protocol are optional.

A `UITableViewController` class is a variant of the `NUIViewController` class that's designed to manage a table. It packages housekeeping functions and provides the stub routines for the data source and delegate protocol methods into one convenient view controller class. This design allows the table's event handling, data display, data updates, and custom behavior to be contained in one omnibus object. `UITableViewController` also performs duties such as loading the table's data, flashing the scroll indicators, and clearing the selection choice. It exposes a property, `tableview`, which allows access to the table view's elements and properties.

A final class, `NSIndexPath`, which has little to do with tables but is crucial to their operation, describes a path to a node in a tree of arrays. Recall that tables are organized as a hierarchy of lists, where each list corresponds to an array. For tables, an instance of `NSIndexPath` provides the section and row information that references a specific cell,

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

and can be used to also reference its data. Many table view methods described here use `NSIndexPath` instances as arguments, or they return results as an `NSIndexPath`.

For more information on each class, consult the iOS SDK documentation (http://is.gd/4gUK4l).

Making a Simple Table

Let's write some code. The first decision is whether to use a `NUIViewController` to manage the instance of `UITableView`, or a `UITableViewController`. The `NUIViewController` choice gives you fine-grained control over every aspect of the table's appearance and operation. However, such flexibility comes at a price: You have to manage every detail of the table's operation, such as assigning and implementing the protocol delegate objects. In contrast, `UITableViewController` manages a lot of these details for you because it assigns itself as the delegate object for both protocols, and provides some housekeeping routines. The tradeoff is that some flexibility in table design is lost because this class implements certain default behaviors. However, I have used `UITableViewControllers` for data display and in many cases I was able to customize the tables' appearance without too much effort. Unless you require an extremely customized UI, I would recommend starting with a `UITableViewController`. The example apps use `UITableViewController` for table management, so that is the class I'll discuss. We'll start out simply by presenting a list of existing Apple products.

From within Xcode, Apple's IDE for writing and debugging Mac OS X and iOS apps, choose File > New Project. In the New Project window, choose iPhone OS Application and Window-based Application for the template. These choices will generate a minimalist app framework where you can add your own code and observe the behavior. Click Choose, name the project TestTable or something similar, and click Save. Xcode generates a basic iOS app template that you can build and run. All the app displays at this point is a blank screen, but that's fine. We're going to add the table as we go, and if something breaks along the way, we'll know where.

Now choose File > New File. In the dialog, select `Cocoa Touch` class, and `NUIViewController` subclass for the file's code template. Under the options, make sure that `UITableViewController` subclass is checked, and that With XIB for user interface is unchecked. This choice directs Xcode to add a `UITableViewController` class to the app's template. Since we aren't including a nib file, we'll use Interface Builder to add the UI elements later. Click Next, and name the subclass file something descriptive like Table.m. When you build the app this time, two errors appear.

These errors flag the data source delegate methods `numberOfSectionsInTableView` and `numberOfRowsInSection`. Xcode has provided stub routines for these data source protocols in the code template, but it can only go so far: You must supply information about the table layout. As mentioned previously, `numberOfRowsInSection` is a required method that requests the number of rows in a section. Oddly, `numberOfSectionsInTableView`, which asks for the number of sections in the table, is not. However, if you fail to provide a value greater than zero for the section number, the app crashes. For simple tables, you can hard-code these values, but for greater flexibility their values should be derived from the app's data model. That way when the data model changes, so do the tables. Leave these alone for now, since we need to create some data to display.

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Make a new file with Xcode, but this time specify an Objective-C class. For the "Subclass of" option, choose `NSObject`. Name this file TableData.m. Create a data array `productNames`, and add data strings to it, similar to that in Listing One.

Listing One

```
// TableData.m for a plain table
#import "TableData.h"

//
@implementation TableData
@synthesize productNames;

-(void) initProductNames {
    productNames = [[NSMutableArray alloc]
       initWithObjects:@"iPhone", @"iPhone 3G",
       @"iPhone 3GS", @"iPhone 4",
       @"iPod classic", @"iPod nano", @"iPod shuffle",
       @"iPod touch",
       @"Macbook", @"Macbook Pro", @"Macbook Air",
       @"iMac", @"Mac mini", @"Mac Pro",
       @"Apple TV", @"iPad", nil];
}

-(void)dealloc {
    [productNames release];
    [super dealloc];
}
@end
```

Be sure to add a declaration of `productNames` in the TableData.h file and also for the function `initProductNames`. Since we're working with constant data, we could use an instance of `NSArray` rather than `NSMutableArray`. However, for a real app, you want to allocate mutable data arrays so that you can buffer and process sensor information or download data over the air into them before displaying their contents.

Now go back to `Table.m`, uncomment the method `viewDidLoad`, and add code to initialize an instance of `TableData` called products and to invoke `initProductNames`. Edit `numberOfSectionsInTableView` to return a value of one, and `numberOfRowsInSection` to return a count value for the number of elements in the `productNames` array. See the example app project `PlainTable` for more detail. If all goes well, the app should build without errors and run. However, nothing appears on the screen.

Final Fixes for Data Display

There are reasons for the lack of data on-screen. First, we haven't done anything about the other required data source method, `cellForRowAtIndexPath`. Search for that method in `Table.m` and you'll find it soon enough. It is mostly stub code, which is why the app builds and executes without errors. This method returns a reference to `UITableViewCell`, which is the class responsible for creating and displaying a cell. (Listing Two.)

Listing Two

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
      dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
```

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

```
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CellIdentifier] autorelease];
    }
/* add these lines of code
    cell.textLabel.text = [products.productNames
      objectAtIndex:indexPath.row];
    cell.accessoryType = UITableViewCellAccessoryNone;
*/
    return cell;
}
```

For performance reasons, the code first attempts to load a cached instance of a cell and reuse it. Otherwise, it generates one. However, this method lacks the means to retrieve and display data content. We'll add our own code to do that.

In `cellForRowAtIndexPath`, add these lines:

```
cell.textLabel.text =
   [products.productNames objectAtIndex:indexPath.row];
cell.accessoryType = UITableViewCellAccessoryNone;
```

The first line uses the row index (provided by the supplied `indexPath` parameter) to access the appropriate string in the `productNames` array. The string is inserted into the cell's `textLabel` field, which then presents it in the content view of the cell. The second line of code suppresses the display of any image in the cell's `accessoryView`. This location is where you can add extra code to access and display content such as optional image, text, or controls.

However, when you build and run the project again, a table still doesn't appear. The second reason is because we have not added and connected the table view controller to the app's main window. Most of these changes are made to the `AppDelegate` class, which, following our example should be named `TestTableAppDelegate`. First go to the header file and add a class declaration for the `Table` class, then a declaration of an instance of it to the interface. Add a property declaration for an instance of `Table`, say, `table`, as well. It's important to specify this property as an `IBOutlet`. This makes a reference to the class appear in Interface Builder (IB). See the header `TestTableAppDelegate.h` for the details.

"The app delegate class generates the instance of the window automatically when it loads, so only one line of code is required"

Go to the `TestTableAppDelegate.m` file and add a synthesize declaration for table. Follow this with a statement that adds the table view to the window:

```
[window addSubview:table.view];
```

The app delegate class generates the instance of the window automatically when it loads, so only the one line of code is required. And now, we add a final step: have IB tie the table into the app's UI.

In Xcode, double-click on the MainWindow.xib file to launch IB. In the MainWindow.xib window, choose Tools > Library to display the library of Cocoa Touch objects. Drag the Table View Controller icon from the library window into MainWindow.xib. Be sure to position the Table View Controller above the Window object, as this can affect the display hierarchy. If you expand the Table View Controller, you'll find the Table View.

Select the table view controller, and choose Tools > Identity Inspector. The Identity Inspector appears. For the Class option, we want to specify our custom table view controller, so click on the menu and

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

choose Table from the list. The name changes in the MainWindow.xib from Table View Controller to Table. Click on TestTable App Delegate object in this window and pick Tools > Connections Inspector. The Connection Inspector appears. You'll see your instance of table as an outlet. Click and drag from the table connection to the Table object in MainWindow.xib. This action has just created the link between table and the app's UI. Save your changes in IB, then build and run the app. Now a list of Apple products appears. As you see, there are a lot of little details to get right when presenting a table. However, now that you understand them, building subsequent tables should be much easier.

The other thing to notice is that the code uses a delegate protocol, `willDisplayCell`. This method allows you to modify a cell's appearance prior to its being displayed, and the code in the example app alternates between two background colors to help the rows stand out.

Sectioning the Data

Now, let's go to the next level and divide the content into sections. To do this, we'll slice the product list into categories and make those into table sections. Modifying the existing `Table` object, as shown in Listing Three, easily does this. Note how the list has been broken into arrays, where each array contains the strings for a particular product family.

Listing Three

```
// Table.m for a sectioned table
#import "TableData.h"

@implementation TableData

@synthesize productNamesPhone;
@synthesize productNamesPlayer;
@synthesize productNamesLaptop;
@synthesize productNamesDesktop;
@synthesize productNamesSettop;
@synthesize productNamesTablet;
@synthesize productSections;

-(void) initProductNames {
    productNamesPhone = [[NSMutableArray alloc]
    initWithObjects:@"iPhone", @"iPhone 3G",
    @"iPhone 3GS", @"iPhone 4", nil];
    productNamesPlayer = [[NSMutableArray alloc]
    initWithObjects: @"iPod classic", @"iPod nano",
    @"iPod shuffle", @"iPod touch", nil];
    productNamesLaptop = [[NSMutableArray alloc]
    initWithObjects:@"Macbook", @"Macbook Pro",
    @"Macbook Air", nil];
    productNamesDesktop = [[NSMutableArray alloc]
    initWithObjects:@"iMac", @"Mac mini", @"Mac Pro",
    nil ];
    productNamesSettop = [[NSMutableArray alloc]
    initWithObjects:@"Apple TV", nil];
    productNamesTablet = [[NSMutableArray alloc]
    initWithObjects:@"iPad", nil];
    productSections = [[NSMutableArray alloc]
    initWithObjects:@"Phone", @"Player", @"Laptop",
    @"Desktop", @"Settop", @"Tablet", nil];
}

-(void)dealloc {

    [productNamesPhone release];
    [productNamesPlayer release];
    [productNamesLaptop release];
    [productNamesDesktop release];
    [productNamesSettop release];
    [productSections release];

    [super dealloc];

}
 @end
```

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Cocoa Touch Class/Protocol	Purpose	Related Documentation
UITableView	Manages table layout and navigation	*UITableView Class Reference*, *Table View Programming Guide for iOS*
UITableViewCell	Presents row of content in a specific style	*UITableViewCell Class Reference*
UITableViewDataSource_Protocol	Gathers information, such as the number of sections and rows required to display a table	*UITableViewDataSource_Protocol Reference*
UITableViewDelegate_Protocol	Used to customize a table's appearance and behaviour	*UITableViewDelegate_Protocol Reference*
UITableViewController	Integrates house-keeping functions and the delegates for table protocols into a view controller object	*UITABleViewController Class Reference*
NSIndexPath	Provides a reference to a specific row in a specific table so data can be displayed through it	*NSIndexPath Class Reference*, *NSIndexPath UIKit Additions*

Table 1: Important classes.

In the table view controller's `viewDidLoad` method, we define another array, `listOfRows`. Into this array we pack the product family arrays. See the second example program, `SectionTable`, for details.

For the `numberOfSectionsInTableView` method, instead of a hard-coded value of one, we now return the count of the product sections, like so:

```
return [products.productSections count];
```

The same is done for `numberOfRowsInSection`, except here we return the count of objects that represent the rows:

```
return [[listOfRows objectAtIndex:section] count];
```

The section parameter references the appropriate product array within `listOfRows`.,

Given the rearrangement of the data model, the code in `cellForRowAtIndexPath` must be modified as well. The code to update the cell's text content becomes:

```
cell.textLabel.text = [[listOfRows
   objectAtIndex:indexPath.section]
   objectAtIndex:indexPath.row];
```

Finally, to improve on the display of rows in the table, we modify the table view's property `separatorColor`, making it a light gray. This is done in the `numberOfSectionsInTableView` method because it is called only once. Now, when the app is built and run, the screen displays the products, partitioned into their families.

These examples only scratch the surface of what's possible with iOS tables. Feel free to edit and tinker with the example apps. Try changing the cell's style. Or with IB, change the table's property from plain to grouped and observe the results. Such experiments can help expand your knowledge of how iOS table design is done.

Table 1 lists important classes and where they're described in the SDK documentation.

— Tom Thompson is a proactive support engineer at Freescale Semiconductor and a frequent contributor to **Dr. Dobb's.**

Comment

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

This Month on DrDobbs.com

Interesting items posted on www.drdobbs.com over the past month that you may have missed

PROJECT OF THE MONTH: COJAC, A NUMERICAL PROBLEM SNIFFER

Cojac's runtime analysis of code identifies risky arithmetical operations both in Java bytecode and in native binaries.

http://drdobbs.com/testing/232601564

HOT-RODDING WINDOWS AND LINUX APP PERFORMANCE WITH CUDA-BASED PLUGINS

Adding GPU capabilities to existing Windows and Linux apps can be done simply using plugins and the built-in support found in CUDA. This easy form of dynamic loading enables CUDA to be used selectively to hugely accelerate individual tasks within a larger application.

http://drdobbs.com/parallel/232601605

ANDROID ON ANDROID

Android Java IDE, AIDE for short, has been released on the Android Market and provides a fully native Android application development experience, running entirely on an Android device. And no, this is not one of those services that zips up your code, ships it off to a build server in the cloud, and has your APK file waiting for download somewhere.

http://drdobbs.com/blogs/tools/232602118

THE CASE AGAINST INT, PART 3: THE ADVANTAGES AND PERILS OF UNSIGNED ARITHMETIC

Integers are often used for counting, and that counting is often better suited to unsigned than to signed arithmetic. To that end, C++ defines an unsigned integer type, `size_t`, that is well-suited to storing array indices, or otherwise counting items that can fit in the computer's memory.

http://drdobbs.com/blogs/cpp/232602212

THE SHRINKING EXPANDING WORLD OF CI

Continuous integration is undergoing somewhat of a renaissance as continuous delivery (and its sidekick, DevOps) begin to find adoption in many enterprises. Simultaneously, the number of viable CI packages is shrinking quickly.

http://drdobbs.com/architecture-and-design/232601132

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

J. D. Hildebrand Editor at Large, Dr. Dobb's
jdhildebrand@drdobbs.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

IN THIS ISSUE

Guest Editorial >>
Mobile UI >>
Chrome >>
iPhone >>
Letters >>
Links >>
Table of Contents >>

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, aleemoll@techweb.com

Director, Client Marketing, Michelle Somers
(516) 562-7928, msomers@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000