Create a professional-looking iOS business app by putting your JavaScript experience to work

Pro iOS and Android Apps for Business

with jQuery Mobile, Node.js, and MongoDB

Frank Zammetti

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Author xv

About the Technical Reviewers xvii

Acknowledgments xix

Introduction xxi

■ Part 1: The Client 1

■ Chapter 1: Designing My Mobile Organizer 3

■ Chapter 2: Introducing jQuery and jQuery Mobile 31

■ Chapter 3: Writing the Application with jQuery Mobile, Part I 53

■ Chapter 4: Writing the Application with jQuery Mobile, Part II 89

■ Part 2: The Server 117

■ Chapter 5: Introducing Node.js 119

■ Chapter 6: Introducing MongoDB 143

■ Chapter 7: Writing the Server with Node.js and MongoDB, Part I 161

■ Chapter 8: Writing the Server with Node.js and MongoDB, Part II 181

■Part 3: Putting It All Together 193

■Chapter 9: Introducing PhoneGap 195

■Chapter 10: The Final Build: Going Mobile with PhoneGap 225

Index 281

Introduction

Anyone who says writing mobile applications is easy probably has never actually done it. Either that or they've been focused on a single platform only. If you're trying to hit multiple platforms, it's a challenge.

Fortunately, we have a ready-made solution at our disposal—web technologies. The combination of HTML, JavaScript, CSS, and HTTP represents a development platform that you can target, and the result is that your work will largely work across multiple platforms. If you already have skills in this area, having to learn a new technology, be it Objective-C, Java, or C/C++, not to mention the myriad of device APIs that are out there to support, isn't a particularly attractive concept.

When your boss says, "Hey, we need an app to run on iPhones and iPads," you can either try to pick up those new technologies (and probably on a tight deadline, as is typical in the business world) or apply the web development skills you already have to get the solution done in record time. That, concisely, is what this book will teach you to do.

The app you write in this way will work on all the Apple-branded devices, plus the vast majority of Android devices and even other platforms that don't quite have the market share of iOS and Android but that nonetheless represent users whom it's better not to ignore if you can help it.

Who This Book Is For

This book is designed for experienced web developers who now need to tackle developing mobile apps for iOS, Android, or both, and perhaps even other platforms such as webOS, Blackberry, and Symbian. This book is for you if you already know HTML5, JavaScript, and CSS reasonably well but you don't want to learn the "native" development model for all the various platforms out there. This book won't teach you the fundamentals of those web development technologies, but you don't necessarily have to be an expert in them either to find value here. You don't need any mobile development experience, though, and that's the key point: it's all about web development in this book. The fact you're creating a mobile app in the end is almost secondary.

How This Book Is Structured

I've split the book into three parts, each covering one broad aspect of writing a single application: My Mobile Organizer. This simple personal information management (PIM) app will demonstrate all of the core concepts you need to produce mobile business applications for iOS and Android.

Part 1 deals with building the client app, that is, the part of the overall solution that runs on the mobile device. In this part, you'll be exposed to jQuery Mobile, a nice front-end JavaScript library based on the ubiquitous jQuery library.

Part 2 covers the server side of the equation. Here, you'll use the popular Node.js framework, coupled with the NoSQL MongoDB product, to build a server that responds to REST requests from the client application and serves as your persistent data store.

Part 3 discusses PhoneGap and PhoneGap Build, the tools you'll use to take your HTML5-based client application and create from it a native application that you can run on iOS, Android, and even other mobile platforms. I'll also talk about getting your app published in the various stores where users expect to find their apps these days.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, `www.apress.com`. You can find a link on the book's information page on the Source Code/Downloads tab. This tab is located underneath the Related Titles section of the page. While you certainly could type in all the code, I make the rather sane assumption that you won't be doing that and that you have the code downloaded and available as you read through the book.

When you download the code, you'll find within the archive two directories: `chapters` and `MyMobileOrganizer`. The directory `chapters` contain code found in the various chapters. (Each chapter has a subdirectory there, although not all chapters have code.) The `MyMobileOrganizer` directory is where you'll find the app that is the subject of this book. Within that directory, you'll find four subdirectories. Naturally, the `client` subdirectory is where the client portion of the app is located, meaning the code you can load into a desktop browser to test the app or that can be built into a true mobile app and deployed to a device, which will be detailed in the chapters to come. The `client_augmented` subdirectory is a version of the client app enhanced with some advanced capabilities, as described in Chapter 10. As I'm sure you have guessed, the `server` subdirectory is where the server portion of My Mobile Organizer is stored. (To start the `server`, you have to execute only the `run.bat` file.) Finally, the `test` subdirectory has some command-line tools for testing the server independently of the client app.

Contacting the Author

If for any reason you'd like to get a hold of me, I'm a pretty easy guy to find online. You can always e-mail me: `fzammetti@etherient.com`. I'm also on "the Twitter," as I hear tell the cool kids call it, as `@fzammetti`, if you're into that sort of thing.

Part 1

The Client

It doesn't stop *being magic just because you know how it works.*

—Terry Pratchett

I am thankful the most important key in history was invented. It's not the key to your house, your car, your boat, your safety deposit box, your bike lock, or your private community. It's the key to order, sanity, and peace of mind. The key is Delete.

—Elayne Boosler

A good programmer is someone who always looks both ways before crossing a one-way street.

—Doug Linder

The only difference between death and taxes is that death doesn't get worse every time Congress meets.

—Will Rogers

When angry, count to four. When very angry, swear.

—Mark Twain

Chapter 1

Designing My Mobile Organizer

Designing a mobile app is no easy task! The wide variety of devices in the world makes it difficult to target them all effectively. Because of this, one of the more popular paradigms for mobilizing an app is to use the same technologies you build websites with: HTML5, JavaScript, and CSS. Doing so is, for most companies and developers nowadays, a good reuse of existing knowledge and skills. However, even after making the seemingly simple decision to follow that path, there are still a bewildering number of technology choices to be made.

Do you write all the code of the app running on the mobile device, or do you use a library to save you time? There are pluses and minuses to both answers. On the server, what technology do you use? Do you go with Java, PHP, or maybe Ruby on Rails?

Moreover, once you make those decisions, how does the client application on the mobile device communicate with the server? Sure, you can simply say "HTTP" and you're probably correct, but there's almost certainly more to it than that, especially if you want a robust and extensible application because, as all professional developers know, applications aren't a steady-state thing. No, they change, grow, and evolve over time, and if they aren't designed to accommodate that change from the start, then they can become a real nightmare to maintain.

It didn't occur to me until I sat down to write this book, but I have in fact been doing mobile development, in one form or another, for more than ten years now. I got in early in the mobile trend. Yet, even with all that experience, I have to admit that all these decision points can be overwhelming still. It *should* be easy, and maybe it will be some day, but it is not this day, to quote Lord Aragorn.[1]

[1]This was a line from the speech given by Lord Aragorn, one of the central characters in *The Lord of the Rings* trilogy. This speech was in front of the black gate of Mordor before the start of the intended battle meant to draw away the forces and attention of Sauron to give Frodo and Sam time to reach Mount Doom and destroy the ring of power. Wow, *now* I see why I had no luck with the ladies in my younger years!

Until that day comes, there is an emerging option that's darned good right now. In this book, you'll learn that there is a popular "full stack" that many developers are using to write mobile business apps, in fact, not just business apps but really *any* type of mobile app—yes, even games in some cases. This stack is flexible, meaning you can swap out any part of it for another. That being said, this stack has become popular for good reason: the underlying pieces make up a powerful and coherent whole that allows you to build mobile apps quickly and effectively, something you know your boss is looking for! At the same time, it allows for that growth we all know will probably occur (well, certainly you *hope* it occurs anyway, as that's a sign of a successful app).

Before we get to discuss the stack and the parts that comprise it, we need to come up with an app that can serve as the vehicle to demonstrate its usage. The app has to be something that is simple enough, frankly, to be presentable in book form while not being overwhelming but still having enough "meat on its bones," so to speak, to be able to demonstrate the stack effectively. I considered many possible app concepts, but the one that jumped out at me is something we all need: the venerable personal information manager.

The Need for Organization

A personal information manager (PIM) is an application where you can store a variety of information and recall it quickly. All mobile devices worth their salt already have this sort of functionality built into them at a fundamental level. You have the option, though, of using any of the numerous third-party apps for this sort of functionality if you prefer. That's exactly what we're going to set out to build here.

A PIM, typically, contains four pieces of information: appointments, contacts, notes, and tasks. While you can probably come up with some other things, as I can, those four are generally the ones that come immediately to anyone's mind, and most other things can actually be fit into one of those four. Those will be the four things, *entities*, as I'll call them,that our app will deal with.

We'll model each of those entities as having a set of data that makes sense for each, but it will be a minimal set. If the intent were to sell this app to others, then the set of data would need to be more expansive or, more likely, extensible to the user in some way. However, as a learning experience, we don't want the app to get overly complex, so we'll keep the data set to a minimum.

Cross-Platform Considerations

As mentioned, one of the big considerations with writing a mobile app is that it should usually be cross-platform. You want this to be true for many reasons, but one that everyone can understand is money! The more users you can reach, the more income you can potentially generate. Even if you're developing a free app and money isn't a concern, you probably still want as many eyeballs as possible on your app, so, again, supporting as many platforms as possible is a good goal.

In terms of market share, Google's Android and Apple's iOS obviously lead the pack and account for the vast majority of modern mobile devices. Therefore, cross-platform, at a minimum, means supporting those two platforms.

However, there are others to be considered: Microsoft's Windows Phone and Blackberry's OS 10, for instance, as well as others that are generally much lower on the market share ladder. Even if you don't specifically target those platforms, I think you'd agree that it's not a bad idea to avoid doing anything that *precludes* you from supporting them at some point in the future. Better still, if you choose the right set of technologies, you can support them out of the gate without doing anything special, so while our direct goal will be support for Android and iOS, we'll also support other devices indirectly by choosing technologies, which we'll discuss shortly in the "Technology Decisions" section, that allow us to do that by design.

Using the Cloud

Another consideration is data storage. That is, where will the data from our PIM be persistently stored? Simply storing it on the device probably isn't the best idea for this sort of application because if the device is lost, stolen, or broken, then that might be a serious inconvenience for the user. Many people can't keep their lives organized at all without such devices and applications anymore.

There's a drive these days to use the cloud for storing data that you want to be "durable" and not be subject to destruction if something bad happens to the mobile device. In simplest terms, storing data in the cloud means storing your data on an Internet-connected server and making it available through some sort of interface that a client application can use.

The term *cloud* actually has a number of meanings and discrete forms besides simple data storage.

- *Software as a Service* (SaaS) means applications hosted on a server that you could access and use almost as if they were installed on your local device. SaaS also to some means web application programming interfaces (APIs) that you can access remotely. Some examples of SaaS include Salesforce (`www.salesforce.com`), a customer relationship management (CRM) service, and GoToMeeting (`www.gotomeeting.com`), an online virtual meeting service.
- *Infrastructure as a Service* (IaaS) means the hosting of real or virtualized systems. If you need a Linux machine, for example, you can have a virtual server built and hosted in the cloud with IaaS, removing the responsibility of building and maintaining the hardware yourself (whether that hardware is real or virtual doesn't really matter in this model; you don't care as the client of IaaS and in fact may not even know). The benefit of this is that most of the responsibility for maintaining servers, worrying about updates, ensuring proper virus protection is in place, and so on, are dealt with by the provider, allowing you to focus on what really matters most to you, namely, your business. Some examples of IaaS include Amazon EC2 (`http://aws.amazon.com/ec2`) and Google App Engine (`http://developers.google.com/appengine`).
- *Platform as a Service* (PaaS) is effectively an extension of IaaS where instead of just a virtualized server you get a virtualized "full stack" including things such as databases, web/app servers, and programming execution environments. Examples of PaaS include IBM's SmartCloud Application Services (`www.ibm.com/cloud-computing/us/en/paas.html`) and VCE's VBLOCK (`www.vce.com/products/vblock/overview`).

- *Network as a Service* (NaaS) includes capabilities such as VPNs and “bandwidth on demand.” Once again, this removes the need to administer the hardware and/or software for your networking capabilities yourself. Any VPN, such as GigaNews’ VyprVPN (`www.giganews.com/vyprvpn`), is an example of NaaS.
- *Storage as a Service* (also abbreviated SaaS) is similar to IaaS but deals specifically with data storage. Sites such as Dropbox, Google Drive, and Microsoft’s SkyDrive are all examples of SaaS.

For the purposes of our little PIM application, we’re going to use the SaaS model (where the first *S* is Software). We’ll build a server-side component to the app that will present an API for our client application running on a mobile device to call on. This API will provide basic CRUD (create, read, update, and delete) operations for our four entity types of appointments, contacts, notes, and tasks. The data will be stored on the server in a database to ensure that if the mobile device needs to be reset, the data will persist.

What About Sunny Days?

Of course, we need to account for the possibility that our cloud-based API, and therefore our PIM data, won’t be available at any given time. Perhaps the server is offline. Maybe we’re hiking in the mountains and don’t have a network connection (I’d question why someone in that situation needs to check when their next dentist appointment is, but who am I to judge?).

Therefore, we will need to do some caching of data on the client so that we can still, at a minimum, look at our existing data, even if we cannot create new items. As with the cross-platform issue, if we choose the correct technologies, we should be able to do this with a minimum of hassle and without writing code specific to each platform we want to support.

A PIM for All Seasons

With a general idea of what we’re looking to build in mind, I present to you the admittedly not very creatively named My Mobile Organizer app!

This app will provide us with a user interface on any Android or iOS mobile device (as well as others) that communicates with a cloud-based API that we’ll build for storing and manipulating our data. It will allow offline access to that data in a read-only fashion as well, and all of this will be done using technologies that allow us to have a single code base for all supported platforms.

Before we get to the technology decisions, let’s look a little more closely at the four entity types My Mobile Organizer will help us organize, including mocking up what the screens for each will look like. We’ll also talk about some overall navigation, a home screen for the app, and some other miscellaneous UI elements that we’ll need to make a complete app.

Home Screen, Header, and Navigational Footer

When the app starts, we need a place for the user to land by default. Sure, we could try somehow to divine what entity they intend to deal with and show the appointment, contact, note, or task screen right away, but really, we don't know their intent going in. Therefore, we need a basic "home screen" instead. This doesn't need to be anything fancy and may be not much more than what you see in Figure 1-1.

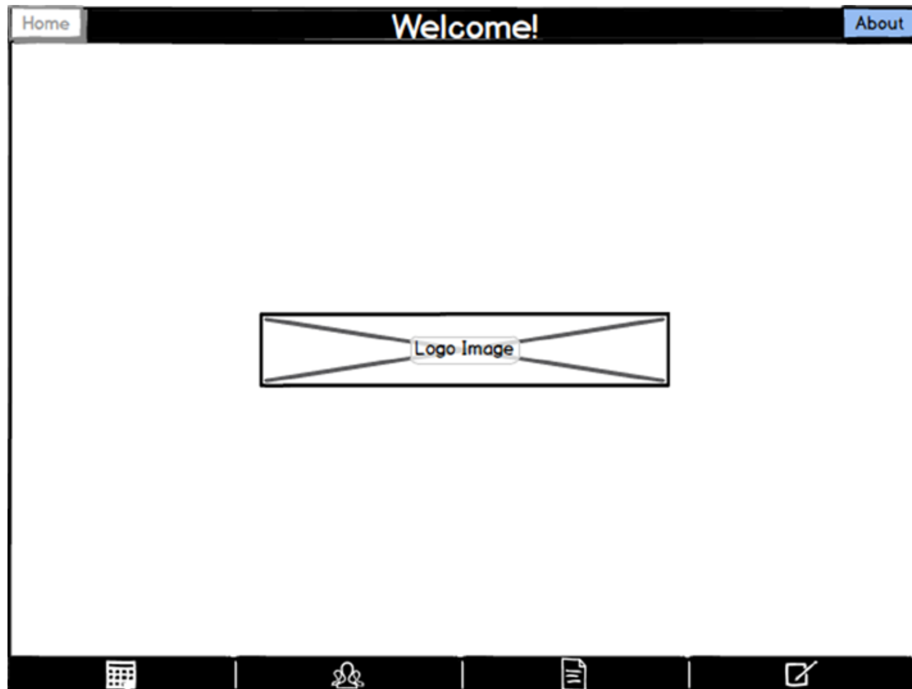

***Figure 1-1.** My Mobile Organizer's home screen*

The basic layout is quite simple, beginning with two elements that will appear on all screens in the app: the header and the navigation bar (or *navbar* for short) at the bottom. The header at the top will always have a button on the left (disabled here of course because we're already there) that returns the user to the home screen and an About button on the right that will show the user a little "about this program" dialog. This button will change on other screens to a menu of functionality specific to the entity the user is dealing with, but on the home screen it's just the About button. In between them is greeting text because, after all, our app should be polite and friendly!

In the center of the screen is a little logo for the app. In keeping with the notion of "low-fidelity" mock-ups, I am using a simple placeholder.

More About Mocking Up Screens

Figure 1-1, as is the case with all of the screen mock-ups that appear in this chapter, was generated with Balsamiq Mockups, which you can find at `www.balsamiq.com`. The nice thing about this program is that, as you can see, it generates "sketches," not proper screens.

Any time you start to mock up a UI, you want to do "low-fidelity" versions first, often drawing them by hand or sometimes using bits of construction paper pinned to a board. This not only allows you to quickly make changes and iterate the design but also keeps the focus where it should be: on navigation, functionality, and overall layout. You don't worry about things such as colors and fonts (for the most part anyway) because those elements aren't important in the first phase of design and obviously won't be final in a low-fidelity mock-up. What's important is figuring out your overall navigation and basic screen layout, where functionality resides, and how the user accesses it. Especially when working with clients, it's easy for nondesigners (yes, which usually includes us developers) to get bogged down in details that shouldn't be the concern up front. Some people prefer doing basic HTML5 work to do such mock-ups, but even that level of "perfection" can distract from what's important because it becomes easy to focus on the color of things, the graphics used, and so on.

Balsamiq Mockups is ideally suited for this because it allows you to do these sorts of low-fidelity mock-ups without having to have any real artistic ability—something very much needed if, like me, you couldn't draw a straight line by hand to save your life! You get the benefit of the low-fidelity look while also having the benefit of being digital, so you get a more word processor type of capability while saving a few trees in the process. This neat little tool is available for multiple platforms too, so you shouldn't have an issue using it no matter what your preferred environment.

At the bottom is our omnipresent navbar. We'll use only icons rather than words (or icons and words together) to save some horizontal space. From left to right the icons are appointments, contacts, notes, and tasks—which, if I've done a reasonable job choosing icons, should be obvious!

About Dialog

When the user clicks the About button on the home screen, they should see some information about the app, as well as one small piece of functionality, for lack of a better place to put it. However, I didn't want displaying this information to require a transition to an entirely new screen. Instead, I wanted it to be shown on the home screen, overlaid on top of it to be more precise, as you see in Figure 1-2.

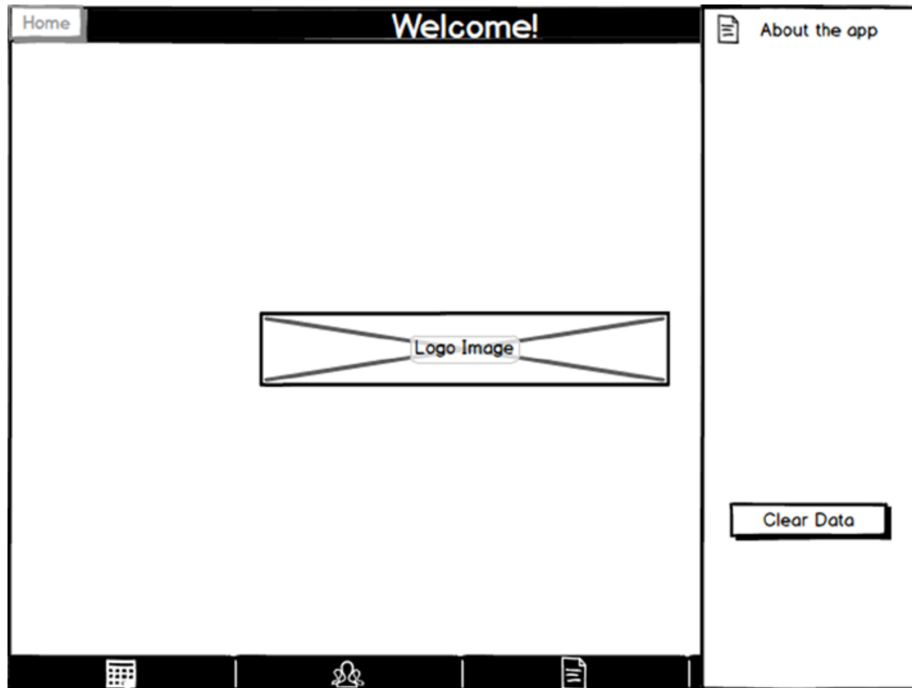

***Figure 1-2.** The About overlay*

This overlay should slide in from the right and then slide away when the user taps anywhere outside of it. Note that I haven't detailed what information should be displayed, and this is by design. Remember, these are intended to be low-fidelity mock-ups, so getting into too much detail about specific content would be counter to that intention. Of course, it's a fine line determining what's too specific and what's not specific enough, and sometimes you might cross it, as you could argue I've done in a few of the mock-ups to come. The point is to put in as little detail as you think makes sense but that still gets the basic design across.

In addition to the "about" information, there will also be a single button that the user can use to clear all of their data. When tapped, the user should see an alert-type dialog, as shown in Figure 1-3.

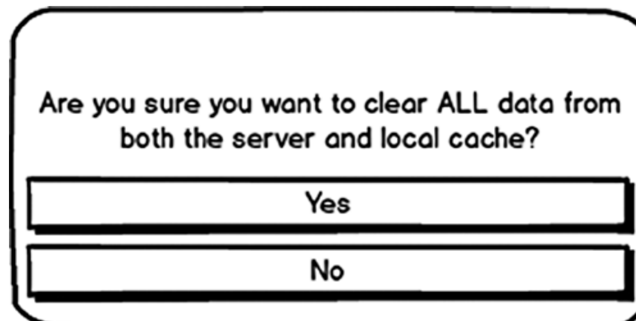

***Figure 1-3.** Verification alert dialog for the Clear Data button*

Not only is this a good feature to provide to the user for security purposes, but it's also quite handy when developing such an application. During the time I was coding this app, I used this function a number of times to reset things to a "clean state" and make it easier to see whether things were working as expected.

Appointments

The first type of data entity that our little PIM application will manage is appointments. Clicking the leftmost button on the navbar, the one with the calendar icon, accesses this functionality.

Data Model

Before we get to the screen mock-ups, let's look at the data model, that is, what information about an appointment we'll store. Figure 1-4 diagrams an `Appointment` object.

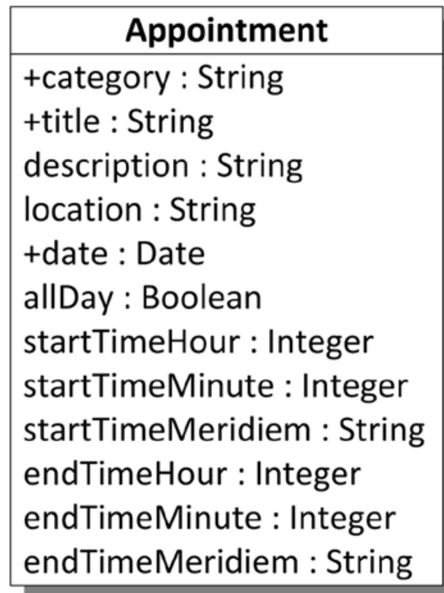

***Figure 1-4.** Appointment data model*

`category`, a field that will be common to all the data models, lets the user determine whether this is a personal appointment or a business-related appointment. The `title` and `description` fields are arbitrary bits of text the user sets to identify their appointment. The `location` field stores where the appointment will occur, and the `date` field tells us when the appointment is. The `allDay` field is a `Boolean` flag that, when true, tells us the appointment lasts for the entire day. When false, the `startTimeHour`, `startTimeMinute`, `startTimeMeridiem`, `endTimeHour`, `endTimeMinute`, and `endTimeMeridiem` fields become relevant because they tell us when the appointment occurs on the specified date.

In this data model, and all the others to come, a plus sign in front of the field name denotes a required field. The client application will be responsible for enforcing these when an item is created.

Note Meridiem, in case you aren't aware (many people aren't, don't feel bad!), is the a.m./p.m. portion of a time.

I decided to break up the times like this more for learning reasons: it allowed me to demonstrate some features of jQuery Mobile, the UI library used to build My Mobile Organizer, which I might not have been able to otherwise. Whether it leads to the best user experience (UX) is debatable.

As you'll see in the chapters to come as we dissect the code, the data models described in this chapter represent a JavaScript Object Notation (JSON) structure that will be stored in a server-side database as well as a client-side data cache. In addition to the fields shown, all entities will therefore wind up having two "hidden" fields: `_id` and __v. The `_id` field is a system-generated ID unique to each object created, and __v is a version indicator that, while unused in this application, would allow us to identify whether a given object was created using a previous version of the database schema and possibly upgrade them if needed. This can be useful when you update the application later and need to change the data models underneath too. I haven't shown them in the data model diagrams, however, as they're a creation of the database we'll store the data in and so not part of our data model per se.

List View Mock-Up

With the data model described, we can now move on to screen mock-ups. Each entity type (appointment, contact, note, and task) will effectively have not one but *two* screens for working with it. The first is the list view, as shown in Figure 1-5, for appointments.

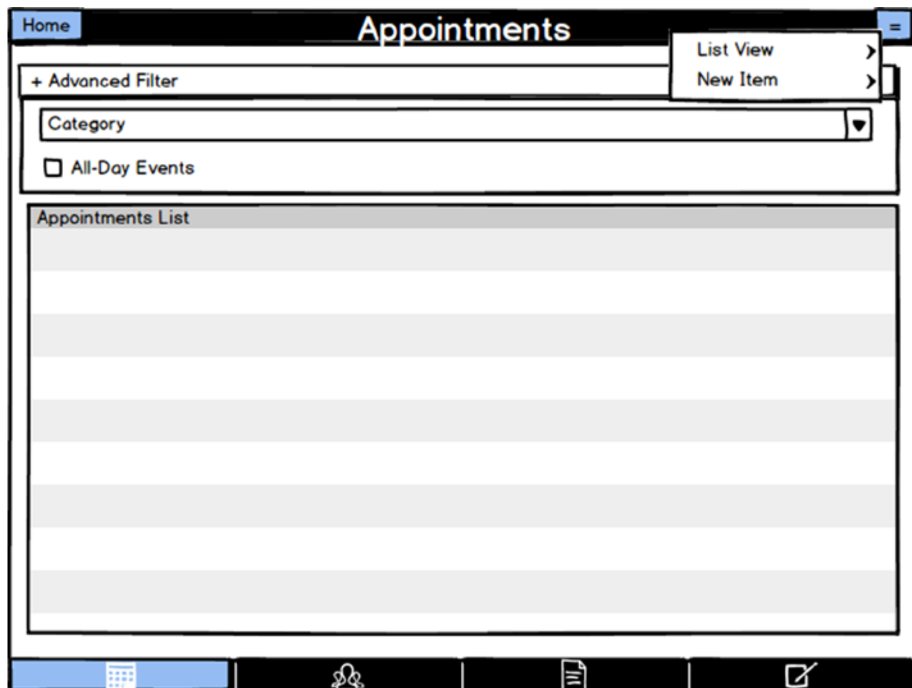

***Figure 1-5.** Appointment list view*

At the top, we have an Advanced Filter section, which is in fact a collapsible section that starts out collapsed, but I've shown it expanded in the mock-up to provide a more complete picture of the screen. In the case of appointments, we can filter the list, which takes up the rest of the screen, by the category, as well as showing only all-day events.

Each of the entity screens has a menu in the upper-right corner, taking the place of the About button on the home screen. When the button is tapped, a menu expands that allows the user to flip back to the list view (which of course makes sense only when they're not on the list view) as well as an option to initiate creating a new appointment. On the mock-up, as with the Advanced Filter section, I've shown that menu as it would appear when the menu button is tapped, but the user would of course not see this until they actually do tap the button.

The list itself is just a simple list of appointment titles, sorted by date. The user can tap any list item to see further details, as well as update it.

Entry View Mock-Up

When the user taps an item in the list, they will be shown the entry view, as in Figure 1-6. This same screen serves double-duty: it allows the user to update an existing item as well as to create a new item.

***Figure 1-6.** Appointment entry view*

All the fields from the data model are represented here. Category is a combo box while Title, Description, and Location are simple text fields. The Date field is a text field as well; however, when it receives focus, it triggers a pop-up, which I've shown overlaid on the mock-up. This allows the user to set the date without actually having to type anything. The start time and end time fields are three combo boxes, one for each element of the time.

Finally, we have two buttons at the bottom, Save and Delete. The Delete button will be disabled, as shown in the mock-up, when creating an item but will be enabled if the user clicks an item from the list. When Save is tapped, one of the two dialogs in Figure 1-7 will be shown, depending on the outcome of the save.

***Figure 1-7.** The possible outcomes of a save (or delete)*

When a delete of an existing item is done, one of these dialogs is shown as well, of course with the appropriate text in the case of success. In any case, the user is returned to the list view, with the list properly updated to reflect the outcome of the save (or delete).

Contacts

Now that you've seen the data model and screen mock-ups for appointments, the rest of the entities should be quick and easy to get through, as they're substantially the same, aside, of course, from the data fields involved. That is entirely by design: the app is designed such that there isn't much separate code for each entity type except in places where there truly *needs* to be. We'll get into how that's accomplished starting in the next chapter, but for now let's continue our journey through the entities and see what differences are present.

Data Model

The data model for contacts, as shown in Figure 1-8, is actually quite simple.

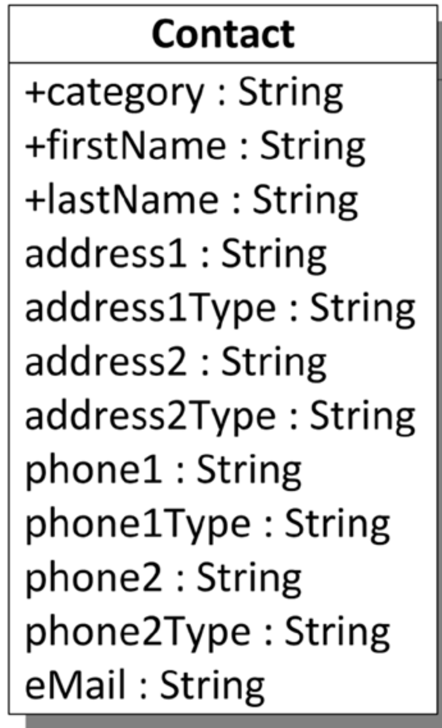

Figure 1-8. *Contact data model*

While there may be a fair number of fields, unlike appointments, we don't even have different data types this time around. Everything in a contact is a simple string. `category`, `firstName`, and `lastName` are all obvious I suspect. For every contact, we allow the user to save two addresses and two phone numbers. Further, they can determine whether each address is a home address, business address, or other address. Similarly, a phone number can be designated a home phone, business phone, cell phone, fax, or other. That's the purpose of the `address1Type`, `address2Type`, `phone1Type`, and `phone2Type` fields. The `address1`, `address2`, `phone1`, and `phone2` fields are where the actual addresses and phone numbers are stored. Finally, we store an `eMail` address too as that's a common piece of information to want about a contact.

List View Mock-Up

The list view mock-up for contacts, shown in Figure 1-9, isn't much different from the one for appointments, but there *are* in fact a few minor differences.

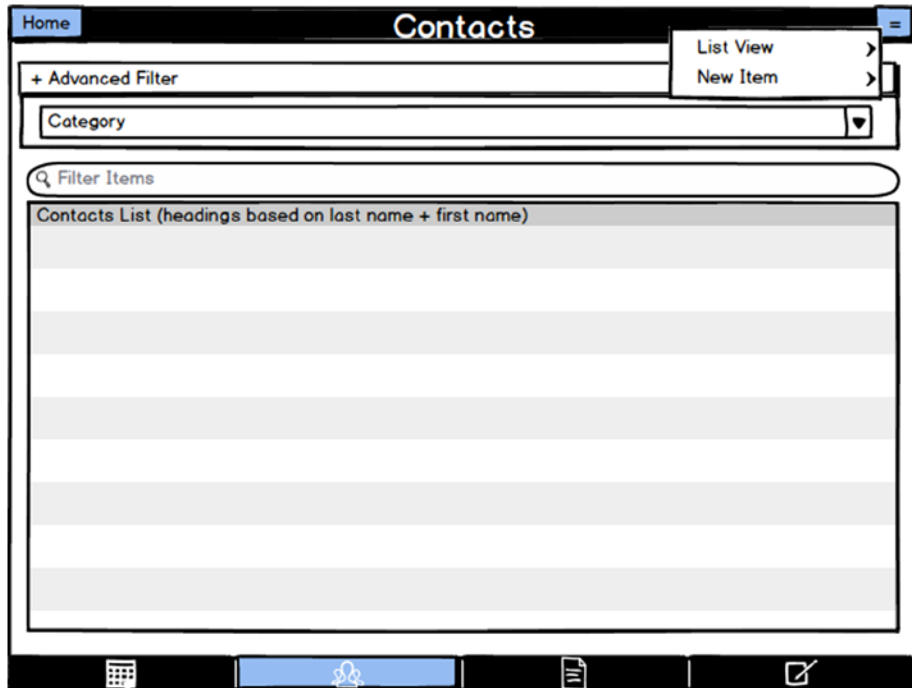

Figure 1-9. Contact list view

First, the Advanced Filter section has only a single filter field this time, Category. However, outside of that section is a Filter Items entry box. This is always present and allows the user to type into this field, and the list will be filtered such that only items matching what the user enters are shown. In addition, the list will have headings between items based on the last name and first name (combined as “last name, first name”) and is alphabetized.

Entry View Mock-Up

When the user chooses to either create a new contact or edit an existing one, the entry view shown in Figure 1-10 is what they see.

Figure 1-10. Contact entry view

For Address 1 and Address 2 we provide text areas to type in, larger than the single-line text fields used for First Name and Last Name. In addition, three buttons are above each section that function similarly to radio buttons in that only one can be selected at any given time. This is true for the two phone numbers as well.

The dialogs for saving and deleting are the same as for appointments but naturally have the appropriate text on them.

Notes

The notes entity is actually the simplest. After all, when you get right down to it, a note isn't much more than some text, right? Still, we'll look at everything for the sake of full disclosure if nothing else!

Data Model

The data model, shown in Figure 1-11, is rather sparse.

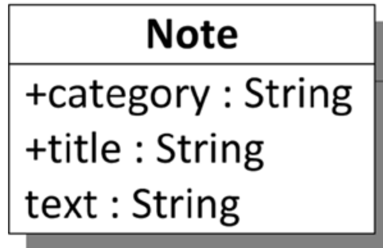

***Figure 1-11.** Note data model*

As with all entities, we have a `category` field, along with a `title` and the actual `text` of the note. That's all there is to a note; it's very simple!

List View Mock-Up

The list view for notes is nearly identical to the one for contacts, except that what's shown is the title of the note. Figure 1-12 is the mock-up of this view.

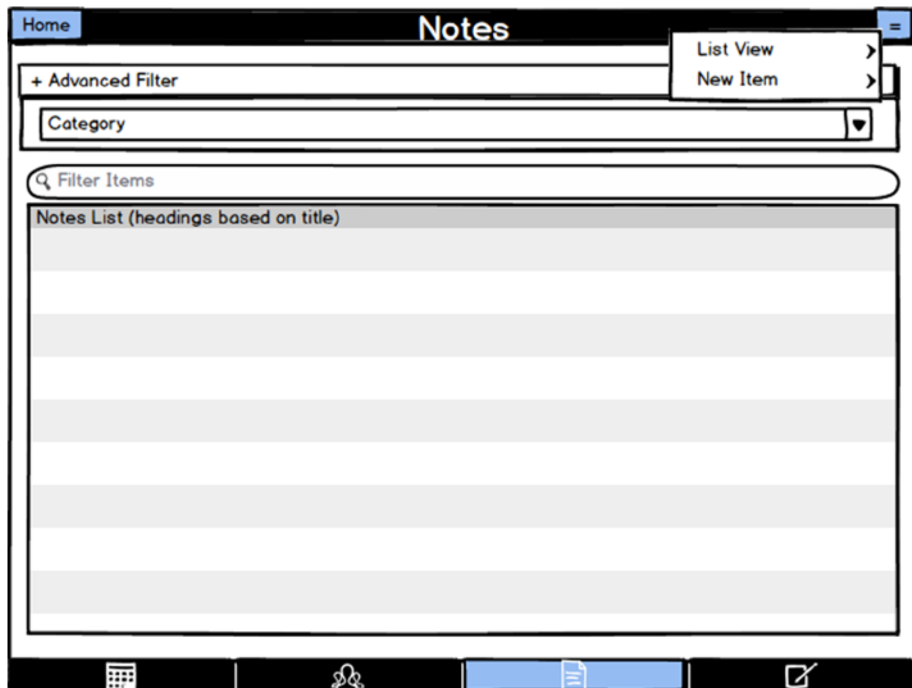

***Figure 1-12.** Note list view*

We can again filter by category as well as type in the Filter Items box to find items with the text we type in the title.

Entry View Mock-Up

The entry view for notes couldn't be simpler, as you can see in Figure 1-13.

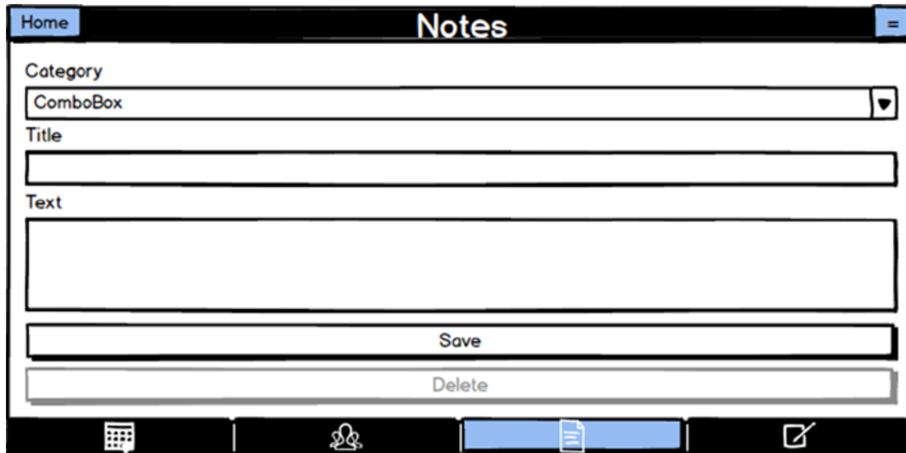

Figure 1-13. Note entry view

It's just the usual category combo box, followed by a text field for the title and a text area to type the text of the note. In addition, in all entry views where we have text areas, we want the text areas to expand vertically as the user adds text. That way, users can type as much as they like.

The Save and Delete buttons are once again present and work the same as for appointments and contacts, including the dialogs that can result from them.

Tasks

The final entity to discuss is tasks, and in terms of complexity, it falls somewhere in the middle of the previous three entities. It's not the simplest but probably not the most complex either.

Data Model

The data model, shown in Figure 1-14, is minimal, not too far off from notes in fact.

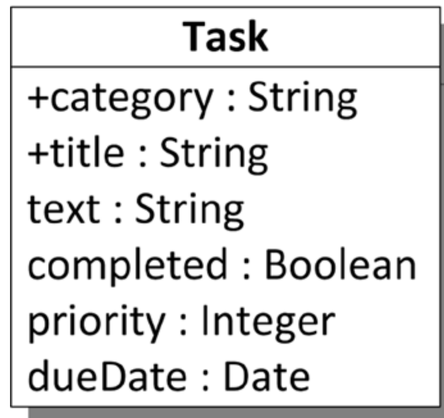

***Figure 1-14.** Task data model*

The `category`, `title`, and `text` fields you're familiar with at this point. For a task, we also want a `Boolean` flag that tells us whether the task has been completed, and that's the purpose of the `completed` field. A task should also have a `priority` we can assign to it, and we'll arbitrarily have a five-level scale of priorities that the user can choose from. In fact, we won't even tell the user whether one is the highest priority or five is; that'll be up to them! Of course, a task needs a `dueDate` as well in most cases, so there's a field for that too.

List View Mock-Up

Figure 1-15 shows the list view for tasks, which should look quite familiar!

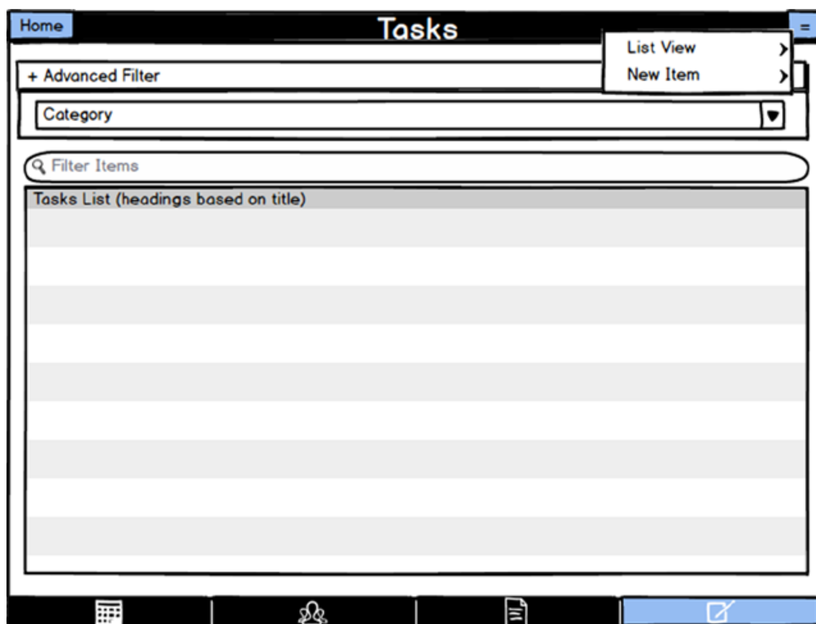

***Figure 1-15.** Task list view*

Yes, this list view is identical to that for notes, including the filtering that is possible.

Entry View Mock-Up

The entry view for tasks, shown in Figure 1-16, is where we finally come to some more substantial differences and some new UI tricks too!

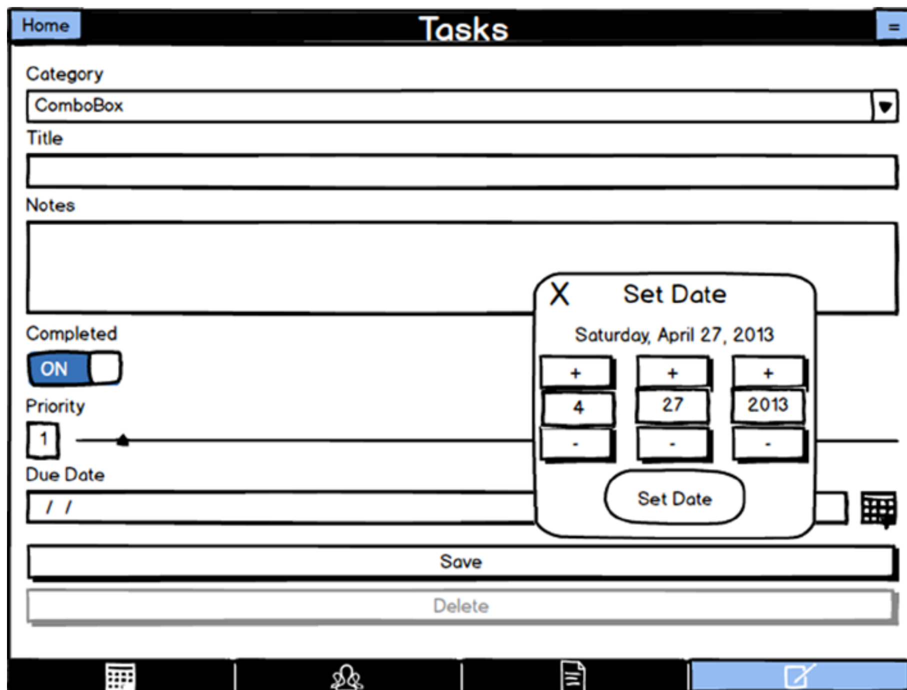

Figure 1-16. Task entry view

After the category, title, and notes, we have an on/off slider to indicate whether the task is completed. Next is the priority slider. This has a text box to the left that shows the priority as a number, but the user will generally just move the slider to set the value (that's up to them, though, because entering a number directly will update the slider as well). Finally, we have the Due Date field, which uses the same pop-up calendar as the appointments entry view.

At the risk of sounding like a broken record, the Save and Delete buttons, and the resultant dialogs, are along for the ride, with tasks just as they were for the other three entities.

… and a Consistent Server-Side API for All

I mentioned earlier that the code for My Mobile Organizer is written in such a way that all four types of entities are largely treated identically. As you'll see in the coming chapters, this reduces the amount of code needed and tends to simplify things a bit. It also allows the code to be written more generically, which is a plus because it means the app can be extended in the future easier

(to add other entity types, for example) and also centralizes things such as error handling to avoid redundant, mostly identical code. There are of course places where writing generic code just isn't possible, but generic code was a guiding principle.

Nowhere is this more evident perhaps than in the server-side API that the client app uses. Simply put, we have four basic CRUD (create, read, update, and delete) operations available for every entity type. That is to say, there is *not* a separate API function for a contact versus a note, for example, to do an update; there's simply an update API call that knows how to handle a contact as well as a note. This is true for all entity types and all API functions.

The function to clear all data falls outside of this model, though, and that's where we see some nongeneric code. Even still, you'll see in later chapters that even it fits nicely within a common model, so even though it's something of a "special function," as compared to the basic CRUD operations, it's not *that* much different either.

How this is all accomplished is a deeper conversation that is presented in Part II of this book, starting with Chapter 5, but for now, you should be aware of this underlying design principle and expect to see it in the server-side API design when we get to that.

Offline Access and Synchronization

One other big consideration of any mobile application is what it does, or doesn't do, when no network connectivity is available. A few years ago, the idea of cloud-based data storage and APIs was ridiculous! The idea that you'd have an always-on Internet connection, especially from a mobile device, was borderline insane. Nowadays, though, it's the norm. Therefore, people expect their apps to continue to work, to at least some degree, even if they can't get on the Net.

My Mobile Organizer will be no different. However, it's not a simple concern because the concepts behind offline access and data synchronization can get hairy in a hurry!

For example, should we allow notes to be created when the user is offline? If so, what happens when connectivity is restored? How do we synchronize the added note with the database on the server? What if that database is also accessible from somewhere else, say a desktop web interface? How do we then synchronize if the user made a change to a contact on the offline mobile device while also making a change to the same contact from the web interface? How do we resolve such conflicts and ensure our data, both on the client and server, is always in a consistent state?

Caching and synchronization are expansive topics that can drive you crazy to get right and nearly always require a good amount of complex code to handle well. Fortunately, some much simpler approaches can largely alleviate the concerns and difficulty involved for you as a developer.

One of the easiest is simply to reduce the functionality of the app somewhat when offline. Some apps simply *can't* work offline under *any* circumstances because of their very nature, but others can, just with a reduction in their capabilities. We'll use that approach for My Mobile Organizer.

Put concisely, the app will go into a read-only mode when offline. The user can still view all their appointments, contacts, notes, and tasks, because we'll cache all that data on the device, but they will be unable to create, update, or delete any entities until connectivity has been restored. That's a fair compromise between an entirely useless app when offline and one that has a ton of complex and potentially brittle code to handle synchronization more robustly. It's not a perfect solution by any stretch, but if we make the simplifying assumption that the app will always have a single user accessing the server component from a single mobile device, then it works out reasonably well.

> **Tip** One other approach that I considered that wouldn't be overly complex is the log approach: every update when offline is logged, meaning all the details of the change are stored locally on the device. Then, when connectivity is restored, the app runs through the log and executes the updates in the order they were originally done. The local copy of the data would already be up-to-date in this model, so only the server-side database would need to be updated. The downside to this is that it does nothing to deal with situation of multiple users or multiple devices; conflicting updates can still arise. Therefore, in order to ensure the complexity of this app didn't balloon too much and become unwieldy as a learning exercise (which, remember, is the point of why we're here!), I decided against that approach in lieu of the "reduced functionality" approach.

One other simplifying assumption we'll make is that if connectivity is available at app start-up, then it's available all the time for the current execution of the app, and likewise, if it's unavailable at start-up, then it's unavailable until the app is restarted and connectivity is checked again.

Technology Decisions

Now that you have a clear picture of *what* we're building and what it'll do, let's get into the questions of *how* to build it. This is the part where we get into those technology decisions I mentioned at the start of the chapter. Let's begin by discussing how we're going to build the client portion of the app.

Native or Mobile Web?

The first major decision point when thinking about creating a mobile app is whether to code it as a native app. In other words, do we write our app in Objective-C for iOS devices and then write it again in Java for Android and then in C# for Windows Mobile devices, and so on, for each platform we want to support, or do we instead use the power of web development to create a "mobile web" app?

It's more than just the programming languages used to consider: the underlying APIs you need to deal with will be different, as well as how you lay out and construct screens, and so on. Yes, if you're smart and architect your code reasonably well, then you'll find you can reuse some of the native code, if not directly, then close enough to lessen the burden of developing for multiple, disparate platforms. Even if you do a good job in this regard, though, you'll still be developing, and don't forget maintaining, multiple code bases going forward; there's just no way around it with true native development.

So, if it's such a hassle, why do people *ever* do native mobile development at all? Certainly, there are some good reasons, chief among them being performance and access to the full capabilities of the devices.

It's hard to argue that the performance of a native version of a given app will usually be superior to that of a mobile web solution. The question is whether that difference will be significant enough to offset the cost of doing that development. To be sure, some types of apps truly do require every ounce of performance and power a mobile device can muster. Games are a prime example. For those types of apps, native development *tends* to be better in terms of performance.

Note In addition to mobile web solutions, there is also the notion of cross-platform "native" development. A number of fantastic cross-platform development tools can very nearly match the performance of native apps while abstracting away from you all those pesky native development issues. I actually wrote a book on one: the Corona SDK (the book is *Learn Corona SDK Game Development*, also published by Apress). While these tools don't provide "true" native development, the abstraction they provide allows a single code base to run at near-native performance on a number of platforms. It usually comes down to what type of app you're writing and what capabilities it truly needs in determining which approach is best, as well as what skill set you already have.

The other concern is that of full access to a device's capabilities. Mobile web development tends to have some limitations in this regard because those capabilities have to be presented in a JavaScript-exposed API that you have access to from your own code, and such APIs aren't always available in mobile web development environments. The situation there is most definitely improving with HTML5 and all it has to offer, but it alone isn't the answer just yet.

Oftentimes, though, the cost of native development, even using a good cross-browser tool, outweighs the performance benefits and device capability access that native development offers. It all comes down, of course, to the margins. How close is mobile web development in terms of performance? How much of the device's capabilities do we have access to from a mobile web app? If it's close enough and our app has some leeway in terms of what it needs, then mobile web development is the way to go.

Why is that? Well, first is reusing existing skills. Especially in the business world, developers, more often than not, are "web" developers. They already know HTML5, JavaScript, CSS, and related technologies. It's to everyone's benefit to use those existing skills. It saves time and money and usually leads to better solutions since there's a lot of experience getting things right to draw from. The knowledge gained over the years developing web applications can be transferred to mobile development, and the quality of the initial solutions will usually be better than they would be if you tried to do native development without the benefit of that experience.

Second, web technologies are by definition designed to be cross-platform. As long as you have a web browser on a given platform, you can target it. Even the best of the near-native cross-platform tools that are available can't usually target as many platforms as the mobile web approach can.

In addition, remember that most web technologies are open standards, not controlled by any one company. That's usually a better position to be in from a business perspective than going with a more proprietary solution.

Certainly, for an app like My Mobile Organizer, a mobile web approach makes a lot of sense, even if it wasn't just a learning exercise in a book! Here are some aspects to consider:

- It doesn't have intensive computational requirements.
- Its screens are, relatively speaking, not all that complex.

- While we of course want it to perform well, it isn't, by its nature, something that requires oodles of performance.
- It's the type of app that could be useful on any mobile device, so ideally we want to target every platform we possibly can.

Overall, it's not a terribly difficult decision to make in this case. Still, I wanted to walk you through some of the thought process and some of the other options because I'm not trying to convince you that a mobile web approach is The One True Approach To Mobile Development™. While a good answer in many cases, perhaps even in most cases, it's not always the best choice, and you should evaluate each project on its own to decide which way to go.

Choosing a Mobile Web Library

Now that we know we're building a mobile web app, the next question to ask is what technologies to use to build the client with. Of course, a mobile web app means we're using HTML5, CSS, and JavaScript. However, that's not all there is to it!

Do we do "naked" HTML5, CSS, and JavaScript? Meaning, do we write all of it by hand ourselves? While I'm an old-fashioned type of developer who likes to write as much of my own code as possible, I'm also a business developer in my day job, which means I can't be spending company money on things that don't truly *need* to be developed by hand.

Take, as a simple example, the navbar in our mock-ups. Writing that by hand wouldn't be the most complex undertaking by any stretch. However, there are aspects to it that could get tricky. For example, how do we maintain state when jumping between pages? How do we anchor it to the bottom of the page regardless of the size of our content? How do we style the buttons dynamically without needing a number of different images (that we'd have to develop ourselves too)? None of that is rocket science to be sure, but even if it takes just an hour or two to write ourselves, aren't there better ways to spend that time?

There's plenty of code we're going to have to write ourselves no matter what, so why not lessen the burden by smartly choosing some libraries to take care of at least some of the more mundane tasks? That's smart development if you ask me. After all, good developers code; great developers steal!

When you start to look at the mobile web libraries available, you'll come across a number of options. There's Sencha Touch (`www.sencha.com`), which is a very good library designed specifically for mobile web development. It allows us to develop screens using JavaScript and provides things such as widgets (grids, buttons, calendars, and so on) as well as a robust data-handling module, among lots of other things.

There's DHTMLX Touch (`www.dhtmlx.com`) that provides a full HTML5-based framework and widget set for developing mobile applications.

There are also some old favorites like Dojo (`www.dojotoolkit.org`) and Yahoo's YUI (`www.yuilibrary.com`) that, while not exclusively mobile-oriented, are capable of helping develop mobile web apps nonetheless.

However, one of the most popular general JavaScript libraries in existence (many polls indicate it is in fact *the* most popular) is jQuery (`www.jquery.com`). As a more general-purpose JavaScript library, however, jQuery isn't focused on mobile development. Although not an all-inclusive description,

probably the main goal of jQuery is to make Document Object Model (DOM) manipulation easier, and it does this exceedingly well. It's extremely fast, lightweight, and, most importantly for the purposes of this discussion, extensible.

That extensibility comes into play with jQuery Mobile (`www.jquerymobile.com`), which is built on top of jQuery. This library is an HTML5-based system for developing mobile UIs. It contains widgets and helper functions for putting such apps together, among other things.

I don't want to get into too much detail about jQuery Mobile here as that's what Chapter 2 is focused on, but suffice it to say it is popular for a reason, as is jQuery underneath it. It's a fine choice for My Mobile Organizer, as you'll see in the coming chapters.

Server-Side Architecture

With the client-side decisions of mobile web app using jQuery Mobile decided, how do we build the server side? Clearly, there's a ton of choices there too.

Do you already know Java (`www.oracle.com`)? Then that might be a good choice. What about PHP (`http://us.php.net`)? Again, there's nothing wrong with PHP in my mind, even though some would argue that it isn't appropriate for "professional" development. I'm not here to pass judgment, though! If you know PHP already, then it's certainly worth considering. Are Microsoft technologies (`www.microsoft.com`) up your alley? If that's a skill set you already have, then they are not a bad choice. Ruby on Rails (`www.rubyonrails.org`) perhaps? Yes, it's worthy of consideration certainly, as are any of a dozen other possible technologies you might come across.

All of these also require potentially significant server infrastructures. Java requires an entire servlet container. PHP is an extension to an existing web server. Microsoft of course requires IIS, its proprietary web server, plus the appropriate extensions. Ruby on Rails is its own server product essentially. All of these also require administration expertise and are therefore somewhat complex to work with, depending on what you might already know.

At the end of the day, there's another option that's fast becoming very popular, and for good reason: node.js (`www.nodejs.org`). One of the key benefits to node.js is that the code you write on the server side is written in JavaScript, just as your client-side code is. Since one of the reasons I stated for going with a mobile web approach is reuse of skills, shouldn't that apply to the server side as well? I think so! The idea of the same language on both sides of the conversation, meaning client side and server side, and assuming performance isn't a problem, is attractive to many people.

The other big benefit of node.js is that it is designed for high performance and concurrency from the start. While neither of these concerns is particularly big for My Mobile Organizer frankly, there's no good reason to use technologies that hamstring us in either regard.

As with jQuery Mobile, we'll be getting intimate with node.js in Chapter 5, but just this little bit of introduction should give you a good foundation to go forward with before then.

What About the Database?

Of course, just deciding on node.js for the server side isn't quite the whole story. There's also data storage to consider. We need a database of some sort too, don't we?

Of course we do!